Computing, Telecommunication and Control, 2023, Vol. 16, No. 1, Pp. 46–59.
Информатика, телекоммуникации и управление. 2023. Том 16, № 1. С. 46–59.

Software of Computer, Telecommunications and Control Systems

Программное обеспечение вычислительных, телекоммуникационных и управляющих систем

Research article
DOI: https://doi.org/10.18721/JCSTCS.16104
UDC 004.43

FLOVVER: A GRAPHICAL FUNCTIONAL LANGUAGE WITH A COMPILER FOCUSED ON RECURSION OPTIMIZATION

A.A. Zavyalov[1], *S.M. Staroletov*[2] ✉

[1] Novosibirsk State University, Novosibirsk, Russian Federation;
[2] Polzunov Altai State Technical University, Barnaul, Russian Federation
✉ serg_soft@mail.ru

Abstract. Visual languages reflect many parts of textual programming languages, however, the existing visual programming solutions lack higher-order functions and recursion concepts. The article introduces the design of a visual language Flovver, which implements the concepts of graphical functional programming. We propose a programming language that supports higher-order and recursive computations. The language accepts programs in a specially designed notation with semantics which we explain in this paper using the lambda calculus. The syntactic unit of such a program is a function that can be combined in a specific way with other functions. We present a fixpoint combinator that helps to specify a recursive behavior in the graphical functional language. To obtain calculate-effective programs, we design and implement a compiler for it, which is capable to optimize recursive programs. We also discuss code generation to JavaScript using the static single assignment (SSA) form. Finally, we propose a sketch of graphical integrated environment to design programs in Flovver using pre-defined blocks, and we present the generated SSA-like code in the paper. The approach is demonstrated on well-known Factorial and Fibonacci recursive programs.

Keywords: programming language, graphical language, functional language, optimizing compiler

Citation: Zavyalov A.A., Staroletov S.M. Flovver: A graphical functional language with a compiler focused on recursion optimization. Computing, Telecommunications and Control, 2023, Vol. 16, No. 1, Pp. 46–59. DOI: 10.18721/JCSTCS.16104

Научная статья
DOI: https://doi.org/10.18721/JCSTCS.16104
УДК 004.43

FLOVVER: ГРАФИЧЕСКИЙ ФУНКЦИОНАЛЬНЫЙ ЯЗЫК С ОРИЕНТИРОВАННЫМ НА ОПТИМИЗАЦИЮ РЕКУРСИИ КОМПИЛЯТОРОМ

А.А. Завьялов[1], *С.М. Старолетов*[2] ✉

[1] Новосибирский государственный университет, Новосибирск, Российская Федерация;

[2] Алтайский государственный технический университет им. И.И. Ползунова, Барнаул, Российская Федерация

✉ serg_soft@mail.ru

Аннотация. Визуальные языки отражают многие черты текстовых языков программирования, однако в существующих решениях визуального программирования не хватает функций высшего порядка и рекурсии. В статье описан дизайн визуального языка Flovver, реализующего концепции графического функционального программирования. Предложен язык программирования, поддерживающий рекурсивные вычисления более высокого порядка. Язык принимает программы в специально разработанной нотации с семантикой, объясняемой с использованием лямбда-исчисления. Основной синтаксической единицей такой программы является функция, которая может определенным образом комбинироваться с другими функциями. Представлен комбинатор неподвижной точки, помогающий определить рекурсивное поведение в данном графическом функциональном языке. С целью получения вычислительно-эффективных программ разработан и реализован компилятор, способный оптимизировать рекурсивные программы. Рассмотрена генерация кода в программу на JavaScript с использованием формы статического одиночного присваивания (SSA). Предложен эскиз графической интегрированной среды для разработки программ во Flovver с использованием заранее определенных блоков и представлен сгенерированный SSA-подобный код. Подход демонстрируется на известных рекурсивных программах вычисления факториала и последовательности Фибоначчи.

Ключевые слова: язык программирования, графический язык, функциональный язык, оптимизирующий компилятор

Для цитирования: Zavyalov A.A., Staroletov S.M. Flovver: A graphical functional language with a compiler focused on recursion optimization // Computing, Telecommunications and Control. 2023. Т. 16, № 1. С. 46–59. DOI: 10.18721/JCSTCS.16104

Introduction

With the involvement of more people in the process of software development, graphical or visual programming languages are beginning to regain popularity. This takes us back to 1970s when Alan Kay was developing the *Dynabook* project [1], with the aim of involving children in programming, in particular by manipulating graphic objects to construct a program, due to the fact that visual information is easier to remember. In the last decade, MIT has cultivated the *Scratch* and *App Inventor* languages, which allow users to combine blocks or graphical elements of programs that include variables, loops, conditions, and so on [2, 3] in a web interface. Accordingly, such blocks can include other blocks, enabling nested programs organization. The latter language can be even used as the initial programming language to teach children how to create mobile applications, and has been successfully applied in the development of computational thinking [4].

If we try to define a visual programming language, then it can be noted that such a language contains graphical elements as syntactic units or primitives, and allows the developer to create programs by ma-

nipulating such elements instead of specifying them in the text [5]. A. Repenning has been analyzing the experience of using existing graphical programming languages over the past twenty years [6] and noted that such languages make programming more accessible to a wide range of people without extensive programming experience. The use of graphical languages helps the developers at three levels:

• At the syntax level: instead of a boundless text, the elements of visual languages are conveniently represented in the form of icons, blocks and diagrams, which eliminates the possibility of syntactical errors in the program.

• At the level of semantics: graphical representation of language objects can visually show the purpose of program primitives and ensure control of their connections only with compatible elements, which means reducing the time of learning.

• At the application level: visual languages enable programming languages researchers to get a certain representation based on a program to explore or prove its properties.

All of the above corresponds to the modern *No-code* or *Low-code* paradigms, which implies the refusal (partial or complete) of writing textual code when building software systems. This approach also correlates with the Model-Driven Development concept, where the program construction starts with some model and the code is only a by-product.

In his 1977 Alan Turing Award lecture [7], a programming language researcher John Backus delivered a lecture "Can programming be freed from the von Neumann paradigm?" [8]. In this speech, he proposed functional languages as an alternative to traditional or imperative languages, and also presented the algebra of functional programs as a formal system of functional programming.

The use of functional languages is especially relevant in the modern era of big data since the execution process in such languages involves the calculation of functions without data dependencies; therefore, it can be parallelized without synchronization overheads, and even dynamically replaced during the calculation if necessary [9].

Creating specifically a graphical visual language is a challenge for us. There has been a long history of work in this area that has led to the design of graphical functional languages (one can mention, for example, such pioneering work as [10, 11]). However, some important questions remain regarding the construction of (i) a formalized syntax for a graphical functional language, as well as the implementation of a full-featured graphical environment, including (ii) a compiler from a graphical language to an internal representation (iii) an optimizer, and (iv) a launcher for running resulting programs and handling their interaction with graphical input-output elements. In this work, we are addressing these issues.

In conducting the present research, we focus on some key factors. The first is the design of an efficient architecture of the graphical environment, where we use the Elm [12, 13] approach, which implements an architecture for creating web-oriented functional languages to generate web applications and games. However, the design of the environment is not a subject of the present paper. The second factor is the implementation of an optimizing compiler for recursive calls. It should be noted here that functional programming is closely related to recursion, which can be used both for organizing simple loops and for solving enumeration problems of practical value.

However, in many cases, it is possible to eliminate recursive calls when generating the resulting program code [14]. In this paper, we discuss means to optimize both tail-recursive calls [15] and general recursive schemes using the memoization technique [16, 17], as applied to graphical functional language programs. Due to the native graph structure of the programs, it is easy to get an internal representation for such optimizations. The third and crucial factor is the ability to study a formal treatment of the graphical language, where the λ-calculus and fix point combinators are useful for us.

Our work is mainly inspired by classic pioneering approaches on graphical languages that were proposed in the 1980s. The thing is that at that time, the graphical interface just began to appear and a large number of researchers started to develop their own graphical languages, including functional ones. However, later interest in graphical languages faded; we attribute this to the dominant paradigms of the time, which led

large programs poorly expressed in graphical languages. Nevertheless, we can state that now interest in graphical languages has begun to grow again, since by now, almost all algorithms have been written and are available as components, and the code turns simply into manipulating them. Such programs can just be well implemented in graphical languages, which is exploited by the mentioned systems like MIT App Inventor.

Therefore, in the existing work, we set the goal of creating a sketch of a visual functional language, which is intended primarily for teaching the basic concepts of functional languages and lambda calculus. It was a challenge for us to develop a fully functional graphical IDE that allows the user to create, run and view the results of programs in the browser. We designed a software so that the components (standard functions) can be extended in the future. For our purposes, it is advisable to generate an SSA (Static Single Assignment) representation of graphical programs in JavaScript: such generation makes it possible both to show the user a text representation of his graphical program in its original and optimized form, and also to interpret the graphical program directly in the browser.

We understand that the examples of programs for calculating the factorial and the Fibonacci sequence considered in the work are very speculative, since both cases are best examples not to use recursion at all. However, in this case we have two different types of recursion (tail and general), and it is possible to demonstrate compiler optimization methods on it.

Syntax and semantics of the proposed Flovver language

In this section, we propose the syntax of the developed *Flovver* language in the form of elements of a graphical diagram. As for its semantics, we denote language units as λ-calculus terms.

Representation of functions. Flovver belongs to a class of applicative languages (like, for example, LISP in its original design [18]) that is, it assumes a sequence of evaluations of a function with a given number of arguments and passes the result of such an evaluation to another function. For a discussion of the semantics of an applicative language, see [19]. Therefore, at the syntax level in the Flovver language, there is only one object: a function.

A function converts from 1 to N values of the given input types into one value of the output type (the variant of constant functions with 0 inputs is also possible). From a mathematical point of view, a function is a mapping of a domain set A to a range set B [20]:

$$f : A \to B.$$

Since a datatype in a language is a set of values that have the general structure or form [21], then by introducing $A = t_{i_1} \times t_{i_2} \ldots \times t_{i_N}$ and $B = t_o$ where t_{i_1}, $t_{i_2}, ... \times t_{i_N}$, $t_o \in T$ and T is the set of input and output datatypes, we define the function $f : T^n \to T$ in terms of the programming language, which has the signature $f : t_{i_1} \to t_{i_2} \to \ldots t_{i_N} \to t_o$.

In the Flovver language, elementary objects are functions or terms $\lambda x_1 \ldots x_n f\left(x_1, \ldots, x_n\right)$ that are represented by diagrams of the form shown in Fig. 1.

Here f is some function of type *input* $1 \to$ *input* $N \to$ *output*.The left side of the block is the inputs of the function, while the right side of the block is the output of the function.

Composition of functions. On the right side of the function block, there is an arc that can be connected to the input of another function (Fig. 2). The semantics of this construction for input values $v_1 ... v_n$ (see an example of composition for λ-calculus in [22]) can be explained as:

$$f' = \left(\left(\left(\left(\lambda x_1 \ldots x_n . f\left(x_1, \ldots, x_n\right)\right) v_1\right) \ldots\right) v_n\right), \quad g' = \left(\left(\lambda x . g\left(x\right)\right) f'\right).$$

Here, the λ-term f is applied to all of its (given) arguments, after which the λ-term g is applied to the result.

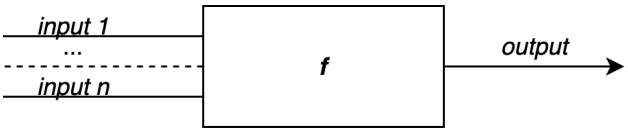

Fig. 1. Function representation in Flovver

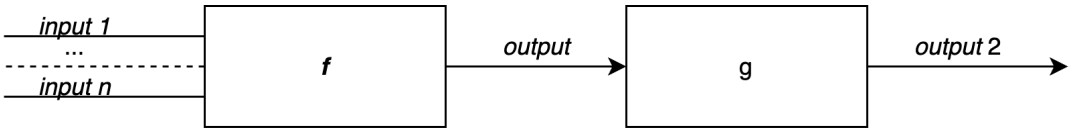

Fig. 2. Function composition in Flovver

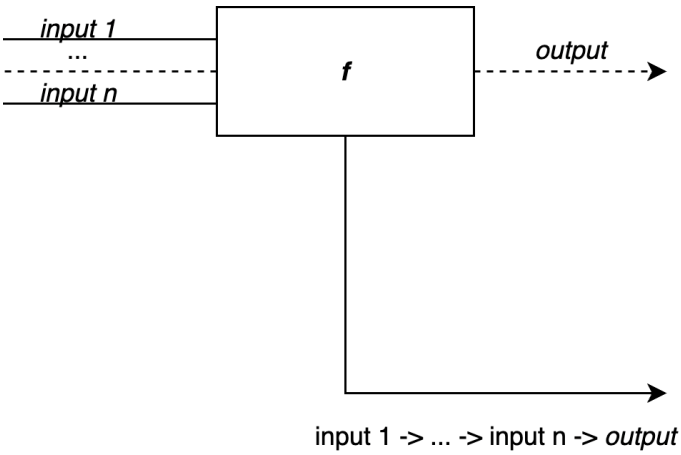

Fig. 3. Partial function application syntax

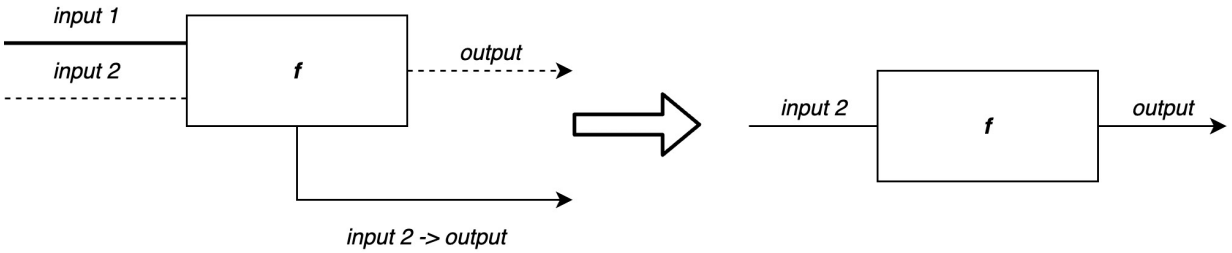

Fig. 4. An example of the partial function application

If not all arguments are passed to the input, the function is considered to be underdefined, and the output arc from *f* to *g* cannot be created in our visual editor.

Partial application of functions. A function and passed arguments can be partially applied by drawing an arc from the bottom of their block to a point of use (Fig. 3).

As a result, we get a function from a (non-strictly) smaller number of arguments, and the previously passed arguments will be fixed (Fig. 4).

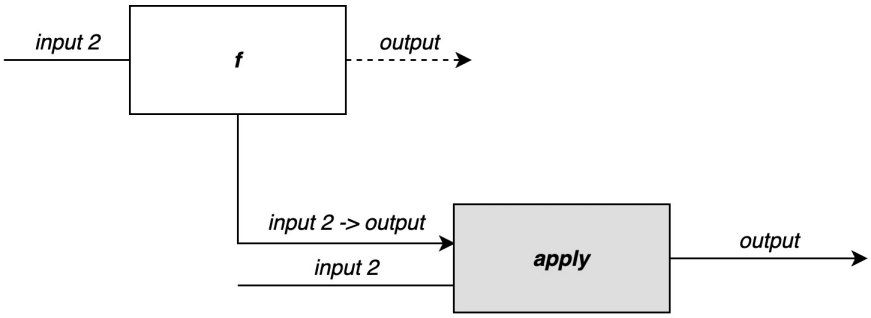

Fig. 5. An example of using the special function *apply*

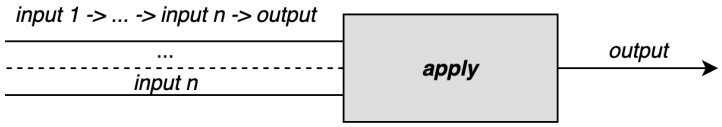

Fig. 6. Syntax of the *apply* function

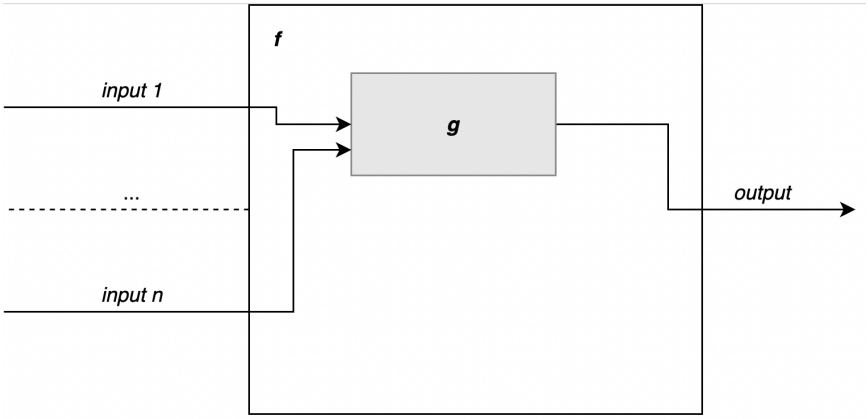

Fig. 7. Building a new function

To calculate a function passed as a value [23], a graphical language developer can use the special function *apply* (Fig. 5).

The *apply* function takes a function of N arguments as its first parameter; the next $2...N + 1$ parameters are the arguments passed to the parameter function (Fig. 6).

Compound functions. The construction of new functions from the given ones is shown in Fig. 7. Here we define a logically separate function block f with its own inputs and outputs. The left side of the block is the inputs, and the right side is the output. Inside the block, there is a function g, to which the inputs f are applied; the result of the function g is passed to the output f.

Thus, the semantics of the construction presented in Fig. 7 is defined as:

$$f := \lambda x_1 \ldots x_n . g\left(x_1, \ldots x_n\right).$$

In general, for an arbitrary function block fun, the semantics looks like

$$fun := \lambda x_1 ... x_n . T,$$

where T is a term dependent on $x_1 ... x_n$, and the dependence is determined as a result of graphic connections within the block.

The *Self* operator to support recursion. To support the declaration of recursive functions inside a functional block, we propose the creation of a special *self* block, which is a link to the function that is being declared. Functions with a self block can be considered applied to the fixed point combinatory [24]. Such a combinator (also known as the Y-combinator [25]) is a special higher-order function that calculates the fixed point of another function according to the rules [26]:

$$fix := \lambda f . (\lambda y . yy)(\lambda z . f(zz)),$$
$$combine\ \ self\ \ f := fix(\lambda self . \lambda arg . f).$$

The practical value of such a function lies in the ability to use recursion for anonymous functions without having to define a name for them.

In this case, the *Factorial* and *Fibonacci* functions can be expressed as following (we use LISPish parenthesized prefix notation to describe functions here):

$$fac := combine\ \ self\, x\ \left(if\,(=x0)\,1\,(*x\,(self\,(-x1)))\right),$$
$$fib := combine\ \ self\, x\ \left(if\,(<x2)\,x\,(+\,(self\,(-x1))\,(self\,(-x2)))\right).$$

Schematic example for the Factorial function. In Fig. 8, we show a function block for the *Factorial* function: $N \to N$! Since the elementary syntactic unit in Flovver is a function, in this diagram, the block consists of connected function nodes (including special cases as constant functions, for this example, these are functions that return 1).

There are also *eq*?: $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$ function that returns the result of comparing two arguments; *mul*: $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ returning the result of the product; minus1: $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ subtracting 1 from the argument; *if*: $\mathbb{B} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ verifies the first argument, and evaluates and returns the second otherwise the third; and finally, the *self* operator described earlier. This assumes that there is a set (a palette) of standard functions that a functional language developer can use. This scheme operates entirely in accordance with the rule specified in the previous section.

Building an optimizing compiler

On the internal representation. To address further issues of optimization and code generation, it is necessary to consider the internal representation (IR) of the compiler of a graphical functional language into code in a language suitable for execution (in this case, in a browser). A good discussion of IR is given in the lectures by Xavier Leroy [27].

Due to the initially chosen graphical form of programs, Flovver can use graph IR natively, following the ideas from [28], i.e.:

- there are two sources of data: nodes and their connections;
- the connections can be internal and external "by value" and "by name";
- vertexes reflect applications, function definitions, and recursive calls.

Methods for recursion optimization. After IR is determined, some optimizations can be made on it, in this case, we describe the optimizations of recursive calls. First, consider an algorithm for optimizing tail-recursive calls [29]. It occurs where a recursive call is the last operation before the call from the function. In this case, there is no need to call the function and save the execution context in the stack since the

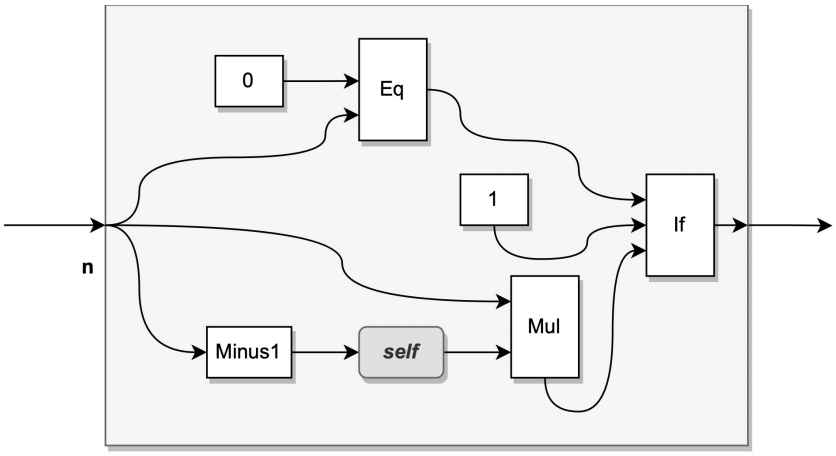

Fig. 8. A scheme for the Factorial function

parameters will not be used and the return address is already on the stack. Therefore, we can substitute passed parameters instead of function arguments, and rather than calling, go to the beginning of the function, organizing a loop.

An optimizing compiler algorithm might look like this:

1. Mark all functions that return a special tail recursion pattern with a special flag.
2. At the code generation stage, for such functions, generate the *while (...) {...}* construct for their body, in which we change the parameter and the accumulator variable, but do not generate a recursive call.

The next question arises: how to define such a tail recursion pattern? The valid tail call notation was formalized by William Clinger [30] and specified in the documents (R6RS: 5.11, 11.20) for the Scheme language [31]. The grammar of such a tail expression can be defined as:

```
<almost tail expr> ::= <rec. call> | <if> | <expr>
<if> ::= if <expr> <almost tail expr> <almost tail expr>
```

Therefore, we define *a tail recursion context* as the place where a recursive call is guaranteed to be a tail recursion call. In our work, we are interested in two contexts: (1) the end of the function and (2) the conditional expression at the end of the function.

Secondly, we consider issues of general recursion optimization. As in the previous case, for each specific recursive scheme, one can search for the corresponding context. However, we decided to optimize the general form of recursion using the memoization concept (caching previous calculations using a hash table with a key according to the passed parameters).

It is possible to memoize any calculation in Flovver since the language is purely functional [32]. In this case, we can monitor the growth of the table for potentially non-terminating functions and report this to the user before the stack overflow program crashes.

In Fig. 9, white color indicates direct calculation of values of the *Fibonacci* sequence Fib with memorization, and gray color shows getting values from memo tables. The computational complexity at the first run was reduced from a value comparable to $O(Fib(n))$ to $O(n)$, thereby approaching the complexity of the iterative algorithm for calculating the Fibonacci function. However, the memoizable version requires $O(n)$ memory for memo tables, while the iterative algorithm with two intermediate values in a loop uses just $O(1)$ memory, which leads to the conclusion that such an optimization is universal, but not completely, optimal.

Code generation. To easily emit code for a target platform, we need our intermediate representation to be transformed in a specific way. We decided to translate the Flovver programs to textual languages supporting higher-order functions and lexical closures, such as JavaScript, Scheme or Python (Haskell or Elm

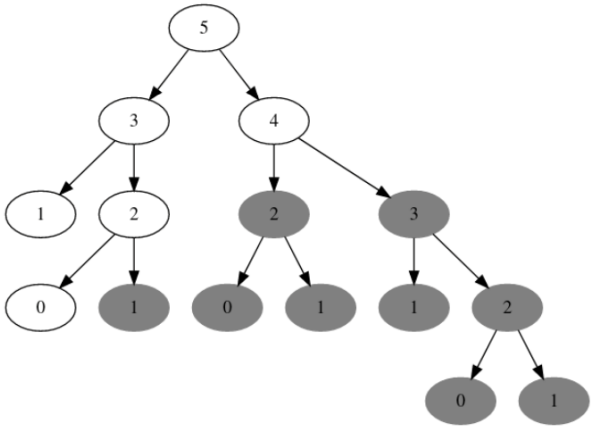

Fig. 9. Memoization for Fib (5)

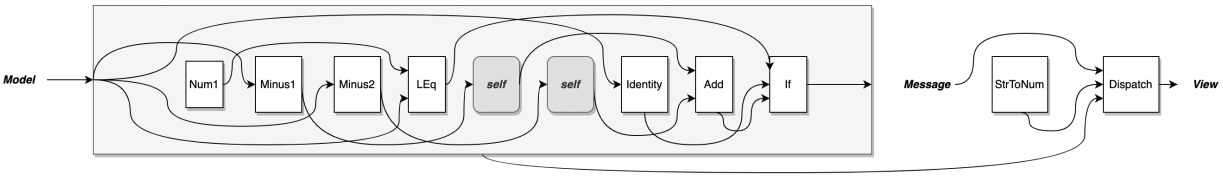

Fig. 10. A topologically sorted DAG of the Fibonacci program

would be fine too), to compile partially applied functions and function blocks easily, without reasoning about such concepts as "closure conversion" and "lambda lifting" [33].

To simplify the target code generation, it is convenient to structure the program as the chain of variable definitions in which each variable is assigned a value only once, and its identifier is used in the following part of the program. Similar ways to structure programs presented in the concepts of the Static Single Assignment (SSA) form [34, 35] used mainly in imperative programming languages, and the Administrative Normal form (A-normal form or ANF) [36] that leverages let-style of ML family languages [37].

We need to place a variable definition before its use. It can be done by reordering the vertices of the program graph since an IR graph is a dependency graph in which links represent value dependencies between objects and there are no circular dependencies in it. Thus our IR is a directed acyclic graph (DAG, see an example in Fig. 10). For a DAG, we can arrange the vertices in the following order: $source \rightarrow \rightarrow v_1 \rightarrow ... \rightarrow sink$ by performing a topological sorting [38]. Finally, the ordered graph will be fairly easy to translate into the target platform code. So, it takes three steps to convert our IR into code:

Step 1. The program graph is ordered by the topological sorting.

Step 2. Each node is mapped to the variable definition by the following rule in sort order:

1. Obtain a unique identifier for the variable.
2. Specialize the code generation for a node:

• the node is a block without input parameters → obtain a simple value (e.g. integer, string);

• the node is a fully-connected block with all inputs and output specified → generate an application of function, corresponding to the block, to its inputs;

• the node is a partial application → generate an anonymous function with unbound inputs of block used as its parameters;

• the node is a function block or a partial application → replace each use of bound input of the block with the unique name and generate code for function body.

3. Replace each use of the original node with an identifier of the corresponding variable.

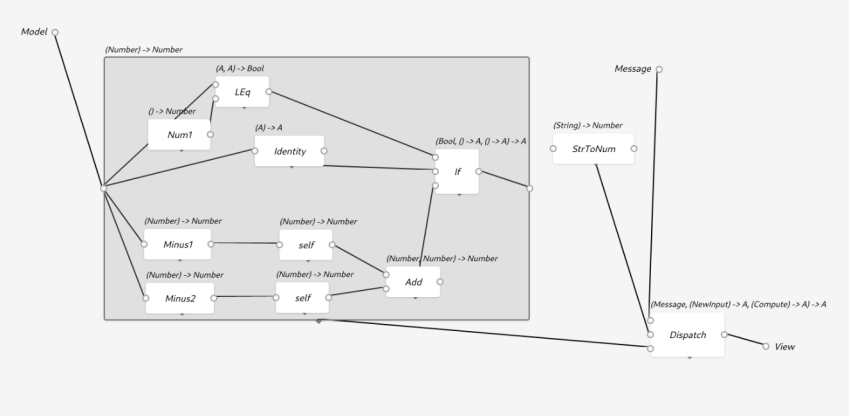

Fig. 11. F_n in Flovver environment

Step 3. Using the variable names, it is relatively easy to generate code for the target platform (in the case of Flovver, in JavaScript).

The *Fibonacci* program design, code generation and optimization

Figure 11 shows a program created in the Flovver interactive environment to calculate the Fibonacci function as F(0) = 0, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ for $n > 1$.

The implementation of our approach was already discussed in Fig. 8, and the reader can see the differences for the Fibonacci function in this case. Function signatures and their purposes are mostly clear, and we can also note the *StrToNum* function, which is passed to the *Dispatch* input, receives a *Message* from the GUI environment with a value of N and returns the result to *View*. It all implements the Elm architecture.

With the optimizer flags disabled, the code in Listing 1 is generated for the discussed graphic program. The code starts at line 18, after which a sequence of SSA calls is defined that implements the calculation scheme.

Listing 1. Non-optimized generated code for the Fibonacci function

```
const update = (model, message) => {
  const fsa_1 = () => Num1();
  const fsa_2 = (fsa_2_arg_0) => StrToNum(fsa_2_arg_0);
  const fsa_6 = () => {
    const fsa_6_r = (fsa_6_arg_0) => {
      const fsa_7 = () => Minus2(fsa_6_arg_0);
      const fsa_8 = () => fsa_6_r(fsa_7());
      const fsa_9 = () => Minus1(fsa_6_arg_0);
      const fsa_10 = () => fsa_6_r(fsa_9());
      const fsa_11 = () => Add(fsa_10(), fsa_8());
      const fsa_12 = () => Identity(fsa_6_arg_0);
      const fsa_13 = () => LEq(fsa_6_arg_0, fsa_1());
      const fsa_14 = () => If(fsa_13(), fsa_12, fsa_11);
      return fsa_14();
    }
    return fsa_6_r(model);
  }
 const fsa_15 = () => Dispatch(message, fsa_2, fsa_6);
  return fsa_15();
}
```

With the memoization flag set, the code in Listing 2 is generated for the discussed graphics program. This code works similarly to the previous one, but additionally, a hash table is defined at line 6, which is used at lines 18 and 19.

Listing 2. Optimized generated code for the Fibonacci function

```
const update = (model, message) => {
  const fsa_1 = () => Num1();
  const fsa_2 = (fsa_2_arg_0) => StrToNum(fsa_2_arg_0);
  const fsa_6 = () => {
    const fsa_6_r = (() => {
      const fsa_6_st = {};
      const fsa_6_w = (fsa_6_arg_0) => {
        const fsa_7 = () => Minus2(fsa_6_arg_0);
        const fsa_8 = () => fsa_6_r(fsa_7());
        const fsa_9 = () => Minus1(fsa_6_arg_0);
        const fsa_10 = () => fsa_6_r(fsa_9());
        const fsa_11 = () => Add(fsa_10(), fsa_8());
        const fsa_12 = () => Identity(fsa_6_arg_0);
        const fsa_13 = () => LEq(fsa_6_arg_0, fsa_1());
        const fsa_14 = () => If(fsa_13(), fsa_12, fsa_11);
        return fsa_14();
      }
    return (fsa_6_arg_0) => fsa_6_st[[fsa_6_arg_0]] =
    fsa_6_st[[fsa_6_arg_0]] || fsa_6_w(fsa_6_arg_0);})();
    return fsa_6_r(model);
  }
  const fsa_15 = () => Dispatch(message, fsa_2, fsa_6);
  return fsa_15();
}
```

Related work

There have been years of research behind the visual programming languages since Goldstine and von Neumann proposed to represent machine-aided calculations as flow diagrams [39, 40]. The approach was firmly rooted in software modeling but was considered ineffective and unimplementable relatively to computers of those times. With the growth of computer power, this approach was abandoned in favor of a well-known textual approach to programming popularized by FORTRAN and ALGOL. However, the interest in the visual approach to program construction has begun to return back since the '70–80s with the development of declarative and applicative programming paradigms, and logical/functional programming. There is a variety of languages developed back in the '80s and '90s that present ideas similar to our work. So, one example is the Prograph language [10], in which programs were organized as a "prographs" (Prolog graphs). Prograph also supported the structuring of programs into procedures. However, Prograph provided iterations via imperative FOR, WHILE and REPEAT blocks.

There were a few visual languages based on the applicative and functional paradigms. For instance, in Viz [41] there were mechanisms to represent mathematical functions and λ-abstraction to organize a program with combinators; the discussed Backus's FP system was implemented in Pagan's graphical FP language [42].

However, Viz offers manipulation with arcs in the flowchart to organize cycles and conditionals, whereas in our work we rely on combinators. Pagan's graphical FP language, in turn, puts forward space-par-

titioning based syntax, which we consider impractical compared to flow diagrams. Modern ideas of the usages of graphical functional languages include their application in data science, focusing on visibility and explainability (see, for example, the Enso language [43]).

We have observed that there is a lack of syntax and semantics formalization in this area, seems it is not uncommon in mathematics to use diagrammatic reasoning. The area where it can be used is the category theory. The concept of string diagrams has attracted a lot of attention as a formal foundation for reasoning using graphical notation [44]. They allow for formal conversion between the topological point of view (boxes and wires) and algebraic (certain categorical constructions), and such diagrammatic syntax could be used to give precise control over resources. It is already presented in our work by driving wires to duplicate values of variables. Another vision on wired dataflow programming is presented in the work [45].

Conclusion

As a result of this work, we designed the visual language Flovver and developed a visual programming environment to create and run programs in this language. It includes a multi-pass visual language compiler with the ability to eliminate tail recursion, as well as to optimize general recursion through memoization. The generated code can be executed in the browser and the result of its execution is obtained in the associated controls. Therefore, the environment is self-contained but currently includes a palette of elementary blocks only for the Factorial and Fibonacci functions. In the implementation, the Scala language, Jetty server, Scalatra and Svelte frameworks were used. To provide interaction with GUI that send messages to and receive responses from a graphical program, we follow the Elm approach [13] and Model-View-Update architecture. This project is completely open and available on GitHub [46].

Preliminary information about the described approaches was published in [47], discussed at the ruSTEP seminar and defended in the form of a qualifying work at the Department of Applied Mathematics of AltSTU. Finally, the tool was demonstrated at the SEIM'22 conference.

Future research directions may include: support for reciprocal recursion; formalization of the language from the point of view of the theory of graphical and functional languages; introduction of static typing and type inference mechanism; the study of common recursive patterns by analyzing the structure of large software systems using real functional languages.

REFERENCES

1. **Kay A., Goldberg A.** Personal dynamic media. *Computer*, 1977, Vol. 10, Pp. 31–41. DOI: 10.1109/C-M.1977.217672

2. **Pokress S.C., Veiga J.J.D.** MIT app inventor: Enabling personal mobile computing. 2013. Available: *https://appinventor.mit.edu/explore/resources/personal_mobile_computing* (Accessed 17.02.2023).

3. **Patton E.W., Tissenbaum M., Harunani F.** *MIT app inventor: Objectives, design, and development, computational thinking education*, 2019, Pp. 31–49. DOI: 10.1007/978-981-13-6528-7_3

4. **Tissenbaum M., Sheldon J., Sherman M.A., et al.** The state of the field in computational thinking assessment. *13th International Conference of the Learning Sciences* (*ICLS*), 2018, Pp. 1304–1311. DOI: 10.22318/cscl2018.1304

5. **Rémi D.** The maturity of visual programming. 2015. Available: *https://web.archive.org/web/20210119062636/https://www.craft.ai/blog/the-maturity-of-visual-programming/* (Accessed 17.02.2023).

6. **Repenning A.** Moving beyond syntax: Lessons from 20 years of blocks programing in agentsheets. *J. Vis. Lang. Sentient Syst.*, 2007, Vol. 3, Pp. 68–91. DOI: 10.18293/VLSS2017-010

7. ACM, Chronological listing of Turing award winners. 2020. Available: *https://amturing.acm.org/byyear.cfm* (Accessed 17.02.2023).

8. **Backus J.** Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 1978, Vol. 21, Pp. 613–641. DOI: 10.1145/359576.359579

9. **Staroletov S.M.** Functional languages for distributed systems (in Russian). 2019, 215 p. Available: *https://www.elibrary.ru/item.asp?id=41291912* (Accessed 17.02.2023). EDN VIRQBH

10. **Cox P., Mulligan I.** Compiling the graphical functional language PROGRAPH. *Proc. of the 1985 ACM SIGSMALL Symposium on Small Systems*, 1985, Pp. 34–41. DOI: 10.1145/317164.317169

11. **Cordy J.R., Graham T.N.** GVL: A graphical, functional language for the specification of output in programming languages. *Proc. 1990 International Conference on Computer Languages*, 1990, Pp. 11–12. DOI: 10.1109/ICCL.1990.63756

12. **Czaplicki E.** Elm: Concurrent FRP for functional GUIs, Senior thesis. Harvard University. 2012, 44 p. Available: *https://elm-lang.org/assets/papers/concurrent-frp.pdf* (Accessed 17.02.2023)

13. **Czaplicki E.** The Elm architecture. 2021. Available: *https://guide.elm-lang.org/architecture/* (Accessed 17.02.2023).

14. **Shilov N.V.** Etude on recursion elimination. *Modeling and Analysis of Information Systems*, 2018, Vol. 25, Pp. 549–560. DOI: 10.18255/1818-1015-549-560

15. **Debray S.K.** Optimizing almost-tail-recursive Prolog programs. *Conference on Functional Programming Languages and Computer Architecture*, 1985, Pp. 204–219. DOI: 10.1007/3-540-15975-4_38

16. **Michie D.** "Memo" functions and machine learning. *Nature*, 1968, Vol. 218, Pp. 19–22. DOI: 10.1038/218019a0

17. **Norvig, P.** Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 1991, Vol. 17, Pp. 91–98.

18. **McCarthy J., Levin M.I., Abrahams P.W., Edwards D.J., Hart T.P.** LISP 1.5 programmer's manual. 1965. DOI: 10.5555/1096473

19. **Turner D.A.** The semantic elegance of applicative languages. *Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture*, 1981, Pp. 85–92. DOI: 10.1145/800223.806766

20. **Kirkinskiy A.S.** Mathematical analysis (in Russian). 2006. Available: *https://www.elibrary.ru/item.asp?id=25759717* (Accessed 17.02.2023). EDN VRSEXL

21. **Amadio R.M., Cardelli L.** Subtyping recursive types. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*), 1993, Vol. 15, Pp. 575–631. DOI: 10.1145/155183.155231

22. **Fischer M.J.** Lambda calculus schemata. *ACM SIGPLAN Notices*, 1972, Vol. 7, Pp. 104–109. DOI: 10.1145/942578.807077

23. **Plotkin G.D.** Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1975, Vol. 1, Pp. 125–159. DOI: 10.1016/0304-3975(75)90017-1

24. **Pierce B.C., Benjamin C.** *Types and programming languages*. 2002. DOI: 10.5555/509043

25. **Park D.** *The Y-combinator in Scott's lambda-calculus models*. 1976. DOI: 10.5555/901310

26. Rosetta code, Y combinator / Scheme. 2022. Available: *https://rosettacode.org/wiki/Y_combinator#Scheme* (Accessed 17.02.2023).

27. **Leroy X.** Functional programming languages Part V: Functional intermediate representations. 2017. URL: *https://xavierleroy.org/mpri/2-4/fir.2up.pdf* (Accessed 17.02.2023).

28. **Leißa R., Köster M., Hack S.** A graph-based higher-order intermediate representation. *2015 IEEE/ACM International Symposium on Code Generation and Optimization* (*CGO*), 2015, Pp. 202–212. DOI: 10.1109/CGO.2015.7054200

29. **Abelson H., Sussman G.J.** Structure and interpretation of computer programs.1996. Available: *http://library.oapen.org/handle/20.500.12657/26092* (Accessed 17.02.2023).

30. **Clinger W.D.** Proper tail recursion and space efficiency. *Proc. of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998, Pp. 174–185. DOI: 10.1145/277650.277719

31. **Dybvig R.K.** *The Scheme programming language*. 2009. DOI: 10.5555/525334

32. **Sabry A.** What is a purely functional language? *Journal of Functional Programming*, 1998, Vol. 8, Pp. 1–22. DOI: 10.1017/S0956796897002943

33. **Johnsson T.** Lambda lifting: Transforming programs to recursive equations. *Conference on Functional Programming Languages and Computer Architecture*, 1985, Pp. 190–203. DOI: 10.1007/3-540-15975-4_37

34. **Kelsey R.A.** A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 1995, Vol. 30, Pp. 13–22. DOI: 10.1145/202530.202532

35. **Cooper K., Torczon L.** *Engineering a compiler*. 2011. DOI: 10.5555/1526330

36. **Flanagan C., Sabry A., Duba B., Felleisen M.** The essence of compiling with continuations. *Proc. of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 1993, Pp. 237–247. DOI: 10.1145/155090.155113

37. **Leroy X., Doligez D., Frisch A., Garrigue J., Rémy D., Vouillon J.** The OCaml system release 4.11: Documentation and user's manual. 2020. Available: *https://ocaml.org/releases/4.11/ocaml-4.11-refman.pdf* (Accessed 17.02.2023).

38. **Cormen T.H., Leiserson C.E., Rivest R.L., Stein C.** *Introduction to algorithms*. 2009. DOI: 10.5555/500824

39. **Morris S.J., Gotel O.** Flow diagrams: Rise and fall of the first software engineering notation. *International Conference on Theory and Application of Diagrams*, 2006, Pp. 130–144. DOI: 10.1007/11783183_17

40. **Glimm J.G., Impagliazzo J., Singer I.** *The legacy of John von Neumann*. 2006, 344 p. DOI: 10.5555/549689

41. **Holt C.M.** Viz: A visual language based on functions. *Proc. of the 1990 IEEE Workshop on Visual Languages*, 1990, Pp. 221–226. DOI: 10.1109/WVL.1990.128410

42. **Pagan F.G.** A graphical FP language. *ACM SIG-PLAN Notices*, 1987. Vol. 22, Pp. 21–39. DOI: 10.1145/24697.24699

43. New Byte Order Inc., Enso Language Syntax. 2021. Available: *https://enso.org/docs/syntax* (Accessed 17.02.2023).

44. **Bonchi F., Pavlovic D., Sobocinski P.** Functorial semantics for relational theories. 2017. Available: *https://arxiv.org/pdf/1711.08699.pdf* (Accessed 17.02.2023).

45. **Nilsson H., Courtney A., Peterson J.** Functional reactive programming continued. *Proc. of ACM SIGPLAN Workshop on Haskell*, 2002, Pp. 51–64. DOI: 10.1145/581690.581695

46. **Zavyalov A.** Flovver (WIP). 2021. Available: *http://github.com/flovver/* (Accessed 17.02.2023).

47. **Zavyalov A.A.** Designing a visual functional language with recursion description capabilities (in Russian). Science and youth. 2021, Pp. 173–176. Available: *https://www.elibrary.ru/item.asp?id=46680361* (Accessed 17.02.2023). EDN VWVNOC.

INFORMATION ABOUT AUTHORS / СВЕДЕНИЯ ОБ АВТОРАХ

Завьялов Антон Алексеевич
Anton A. Zavyalov
E-mail: a.zavyalov.98@yandex.ru

Старолетов Сергей Михайлович
Sergey M. Staroletov
E-mail: serg_soft@mail.ru

Submitted: 18.02.2023; Approved: 03.04.2023; Accepted: 17.05.2023.
Поступила: 18.02.2023; Одобрена: 03.04.2023; Принята: 17.05.2023.