Data Abstraction in a Language of Multilevel Computations Based on Pattern Matching[⋆]

Igor B. Shchenkov

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia

Abstract. It is well-known that data abstraction badly coexists with pattern matching. Pattern matching is applicable in the domain with "transparent" structure of symbolical data while the traditional data abstraction supposes hiding data from the user. In this paper, it is shown how it is possible to provide data abstraction at reception of all completeness of convenience of it by simple means within the framework of the system of symbolic manipulations based on pattern matching. However, the problem is not put to hide abstract data from the programmer completely.
In our case data abstraction is based on possibility of obtaining of values of functions being inversed to functions-constructors that in turn is based on possibility of dynamic computation of patterns. Used way of pattern matching is based on multilevel computations.
The examples of data abstraction given in this paper are based on the subsystem of algebraic computations that has been built in the system of symbolic manipulations.

Keywords: Refal, multilevel computations, pattern matching, data abstraction, algebraic computations.

1 Introduction

The purpose of our project is such principle of computations when by means (by a set of operators) G of a programming language some operator H can be constructed: = G (M), where M — some material for making of operator H; this operator will be used then in the program for transformation generally intermediate data D in intermediate result R: R = H (D). Such process of computations combines as making of operator H, and application of this operator to some data as a result for the purpose of obtaining a new data R which can be both intermediate, and final. The combination of program code construction and its execution is the subject of multilevel computations.

Generally operator H can contain a set of arbitrary operators of the source language. In particular case operator H can contain the limited set of own operators of language, such case just represents a subject of consideration of this paper.

[⋆] Supported by Russian Foundation for Basic Research projects No. 06-01-00574-a.

Proceeding from the object in view, symbolic manipulations language should be the language potentially providing both making of operators, and their application. The opportunity of application of the constructed operators during initial program execution also is a subject of the present development. Some examples are given below.

Refal was taken [24,25,1,27,2,5], as a prototype of the programming language for our purposes as the language with the developed mechanism of symbolic manipulations. At the same time the language is compact enough and laconic in its expressive possibilities, this feature is of great importance for programmer.

Multilevelness of computations, widely used in our approach and described below, is based on the construction of a program code (program fragments) and its execution in single process of program execution. It means, that the program is changing itself during its execution, i.e. it is not static. At the same time all existing implementations of Refal are based on the compilation which demands static character and invariance of a program code. Therefore, an interpreting way of program execution has been chosen for the implementation of the given project. It assumes separate implementation.

It is expedient, besides, to build functional language as functionality is important property for the further works on automatic updating, optimization and supercompilation [26] of programs written in such language. All the more so, as the chosen prototype of building language, — Refal, — possesses functionality.

Let's add that language of multilevel computations can be easily expanded by the means providing convenient performance of algebraic computations likely it takes place at Refal using. In papers [9,10,6] such expansion of Refal is reached by various ways: as simple addition of library of the corresponding functions, allowing to carry out those or other transformations of algebraic objects, including the convenient form of input/output, and additional to library the special source language of algebraic computations, programs in which are translated into Refal by preprocessing.

Created language of multilevel computations is named Santra 3 (SANTRA — Symbol-ANalytic TRAnsformations; digit 3 means following generation in relation to language and system Santra 2 [15,16,17,18,19,20]). In comparison with the system Santra 2 — Santra 3 is represented completely altered implementation constructed on the basis of last dialects of Refal (Refal-5 [27], Refal-6 [2], Refal Plus [5]) in comparison with Refal 2 [25,1] which is used as base of Santra 2 language. It is supposed also that, besides altering language means of actually symbolic manipulations, new language means of algebraic computations should be anew implemented. In implementation aspect the library of algebraic functions should be altered also. The sketch of language Santra 3 is given in paper [8]. In this paper additional properties of language will be considered.

It would be desirable to note one feature of the given approach which is the tendency in programming in general. In our case at first from the Refal a language of multilevel computations was build actually, means of such computations were entered into Refal by insignificant expansion and updating of its syntax and semantics. Then the constructed language has been expanded by additional means of algebraic computations so the language Santra 3 was built in result. So we obtained conceptual hierarchy of languages in a direction of expansion of their possibilities. The similar hierarchy is well looked through in Refal family which has at least 4 versions, not considering the language Flac [6] and the present language Santra 3. It has been supposing further use of the language Santra 3 as the base for the creation of hierarchy of languages of the algebraic computations having or universal character, but more simple on syntax and semantics in respect of habitualness of the mathematician using it, or having this or that problem-oriented character in various areas of applied mathematics. The problem-oriented language Dislan intended for difference schemes construction [11,12] can serve as an example. It was built on over language Santra [13,14] which was the predecessor of language Santra 2.

It is possible to say, that the environment of family of languages of Refal type, including language Flac and family of languages Santra, is the environment for building domain-specific languages (DSL).

2 Related Work on Multilevel and Multistage Languages

Works on multilevel languages and multistage programming can be classified into two ways: for languages without static typing and languages with static typing. (The represented project concerns the first class of languages). In works for not typed languages, i.e. languages without static typing, substantial programming problems of building of metalanguages for the description of specialized constructions and languages are solved basically. And researches for the typed languages are devoted basically to a problem as, overcoming typing restriction, to build multilevel languages with safe system of typing. From our point of view, at the given stage of development of these methods typing too much complicates and blocks up the metalanguage building though in distant prospect it can really lead to occurrence of reliable and convenient means of programming.

2.1 Typed Languages

Historically the earliest line of works on languages with means of dynamic generation of programs goes back to the language Lisp and its "descendant" - to the language Scheme [21]. Occurrence of the language Scheme was in the late seventies connected with many "unneatnesses" of the language Lisp, preventing generation of programs. The language Scheme is nice for the "hygienic" macrosystem in which there is no mess in levels of the local variables, taking place in the language Lisp. However, a macrosystem in Scheme as all macrosystems, "tuned" for definition only one level over the programming language. Therefore and syntax of "hygienic" macrodefinitions in Scheme is adjusted for such narrow application.

Though among users of Lisp and Scheme style of programming with dynamic generation of programs, on the basis of these languages, was always popular, as far as we know, except two-level "hygienic" macro means, means of multilevel programming have not been developed. From our point of view, it is connected with too “low level” languages Lisp and Scheme. Our experience of working out of multilevel language shows that use of language of higher level of Refal type [27] is required.

Among the modern languages without static typing “languages of scenarios” (scripting languages), such as Tcl, Phyton, Ruby, etc are popular. As a rule, they allow to generate and execute dynamically programs, but — using for programs code forming simple operations over lines and not offering means for multilevel programming.

Thus, among the not typed languages till now there are still no languages supporting means of fully-featured multilevel programming though, from our point of view, such languages are just convenient for the decision of the given problem. In the present project the step on filling of this gap is made.

2.2 Untyped Languages

A more formalized stage in the development of ideas of multilevel languages has begun with the end of 80th years [7] on the basis of the ideas of partial computations and the automatic analysis of the program for its division into execution stages which have appeared by then. The concept of only two-level languages [7] was first studied and the problem of development of multilevel programming was not put.

The following stage in the development of multilevel languages has been summed up in the late nineties by publications [22,23]. Authors have developed the typed language MetaML in the frameworks of which the programmer can write down multilevel programs. The basic problem which was solved by the authors of MetaML — the making of an adequate system of types.

From the recent publications on multilevel languages the language ReFLect [4] is of interest. It is the universal functional typed language, but first of all intended for special practical application in systems of modeling of integrated schemes logic (in the Intel Company). Multilevel language is used for the generation of schemes from compact descriptions.

In these works the basic difficulties of language’s building are connected with their typing. These problems are not present in our approach.

In logic programming the idea of multilevelness for support of self-application of logic theories and programs also was investigated. For example, in the paper [3] the expansion of the Prologue language for the processing of mathematical theories by multilevel means is offered. However, the metalogic programming language Alloy defined in this paper was not developed further, probably because of its high complexity and narrow orientation on the processing of logic systems. It sharply differs from our purposes of working out the universal multilevel functional language of symbolic manipulation.

3 Multilevel Expression

Language Santra 3 differs from Refal mainly in the aspect that in Refal computations are determined by functions calls only while in language Santra 3 concept of computations is expanded by construction and execution of program fragments in addition to functions calls. For this aim the concept of structural expression is entered, such expressions syntactically are embraced by parentheses. Basic elements of structural expressions in the language Santra 3 are, besides traditional for Refal calls of functions, — names of functions preceded by sign `#`, numbers, structural expressions in parentheses and variables. Functions calls are expanded by arithmetic expressions also. Computation of structural expressions in parentheses is initiated by angular brackets: `<(`*`structural expression`*`)>`. As an example of the of concrete structural expression computation divining can serve the following: `<(55 (#Alfa) e1 '123')>`. Here `55` and `Alfa` — number and the name of function translated in internal representation of the computer, `e1` — a variable instead of which its value must be substituted, and `'123'` — the literally given text. This text is not subject for computation that is pointed out by means of inverted commas, however, in the course of computation of structural expression inverted commas are evacuated, and the number `123` becomes a set of digits already without inverted commas. Thus, inverted commas are means of a delay of computations while angular brackets are means of computations initiation. Besides, calculated values of self-defined elements, such as `55` and `Alfa` from an example above, are not subject to the further computation. However, they can be a part of structural expression so that to participate in the further computations. For example, `Alfa` can be a name of calling function.

The values of structural expressions are new structural expressions, in particular case these values represent new fragments of the program or computed values in essence. On this in addition to Refal 6 [2] all computations are based in language Santra 3.

Traditional compilation imposes a condition of static character of program fragments, operators and functions owing to what it is inapplicable for multilevel computations. Therefore in this case just interpretation is chosen which is that in essence.

4 Multilevel Computations

Structural expressions are the basis of multilevel computations and represent the special form of coding (giving) of the program. The coding essence consists in a marking, what parts of the program need to be left invariable and what need to be transformed. The invariance of parts of fragments is provided by a delay of their transformation (computation) up to a certain stage while other parts of the program should be transformed by the general rules. Thus, for multilevel computations the way of marking is necessary, first, computation of what program parts needs to be delayed and, secondly, at what stage of the program execution these parts should be computed, as there can be some stages of program forming. Such marking is provided by means of a so-called metacode.

Metacode using is known in the supercompiler [26], however, there it is used for a marking of the program for the purpose of differentiation of the status of variables for further program transformation. In our case the metacode is used, the main thing, for a marking, what fragments need to be calculated, and what computation needs to be delayed. Such regular use of a special metacode for traditional program executions is of interest for programming.

Thus, structural expressions are the program fragments given in a metacode.

5 Metacode

Main principle of a metacode as means for supporting of multilevel computations is the way of program making at which parts of the program where computation demands a delay, are given in the literal form. The degree of a delay is determined by the level of a nesting of literals. As a result at the program execution at each stage of literal use there is a text of a corresponding level of a nesting that allows to use it as a fragment of the program at a corresponding step of it performance.

The concept of multilevelness also includes the performance of partial computations the results of which can be used in further computations. The examples are transformation of numbers and functions names to their internal computer representation. Besides, rather complex computations can be executed and values can be obtained, and these values can be included in building program. Repeated use of such values can essentially raise the efficiency of computations as a whole.

At the program execution including its dynamic construction, the transformed fragments can be transferred further for the subsequent transformation or execution by means of variables or arguments of functions; also the generated fragments can be executed in place for what angular brackets are used, it has been mentioned above. Angular brackets are used as well for the invocation of the transferred fragments and also in other places of the program. Difference from a traditional way of use of the calculated values by means of variables consists in an opportunity of building of the program fragments including just variables values with absence of variables themselves.

For the giving of the literal structures the essential nesting of which is typical for the mapping of multilevel computations, a pair of inverted commas is used. It is demonstrable, convenient and besides the length of the text at a nesting grows as 2*n where n is the depth of a nesting. In connection with a special role of "literals" for the considered way of programming of multilevel computations and importance of their essential nesting, the special term — 'multitext' is entered for them. This term is entered also for the reflection of potentially consecutive, multistage stile of transformation of structural expressions and "literals" entrances into them. We will give a multitext example.

The text, its multitext, the multitext from the multitext:

```
ABC‘123 --> ‘ABC\‘123’ --> ‘‘ABC\‘123’’
```

Let's give an example of dynamic generation of fragments of the program and their execution. Let, for example, it is necessary to calculate value of some expression and to increase it by `1`. And let this expression, for example, `-1+3*5` be assigned in a symbolical kind as a value to the variable eX. Then for execution of the given actions in the language Santra 3 it is necessary to write down:

```
    ‘-1+3*5’: eX, <1 + <(eX)>>
```

As a result number 15 will be obtained as the value of the given expression.

The main difference from Refal 6 [2] in this example is the use of multitexts instead of literals and the use of angular brackets for arithmetic expression giving and performance of actions over expressions in brackets.

In this example at first the multitext

```
    ‘-1+3*5’
```

will be transformed into the text

```
    -1+3*5
```

which will be assigned as a value to the variable eX. Then the program fragment `<(eX)>` will be generated and the text `-1+3*5` will be its value. Further this text taking into account a sign `-` at its beginning will be inserted in resulting expression `<1 +...>` instead of dots. It will lead to that the new fragment of the program will be built which will be already finally executed.

The given example does not represent *essentially* anything new in comparison with Refal, it is demonstrated with the purpose to make it easier to accept syntax of the language Santra 3.

6 Functions-constructors in Patterns

It is interesting to use in patterns functions-constructors which arguments contain variables. In this case the value of function-constructor, containing variables of the pattern without change, will enter in the pattern. Therefore, the set of the objects defined by the pattern as a whole will depend on the set of the objects defined by the values of function-constructor also.

This function when used entirely as the pattern will define in this case not a concrete matrix, but a set of matrixes of the given dimension if a number of elements of these matrixes is fixed, and if variables stand instead of other elements. Let’s look at an example. Let function FORMMA be a function forming a matrix of the given dimension. It is a function-constructor in relation to the arguments that give matrix elements. Its arguments are the dimension of a matrix given by the numbers of rows and columns, and the elements listed in rows. For example, for forming of a matrix 2*2

```
    1 2
    3 4
```

it is necessarv to write down

```
<#FORMMA (<2>) (<2>)
                    (<1>) (<2>)
                    (<3>) (<4>)>
```

This function can be used also for giving the set of matrixes of the given dimension on which main diagonal arbitrary elements stand, as follows:

```
: <#FORMMA (<2>) (<2>)
                    (e11) (<2>)
                    (<3>) (e22)> =
```

Here instead of the elements standing on the main diagonal, variables `e11` and `e22` are used. The sign `=` in the end of text is one of the separators of the operators of language, it is similar to how it takes place in language Refal 6 [2].

If now these two examples will be concatenated as a program fragment, at its execution, the concrete matrix will be compared with the pattern of a matrix of the same dimension, so the value number `1` will be assigned to variable `e11`, and the value number `4` will be assigned to variable `e22`. Furthermore, numbers `<2>` and `<3>`, which stand not on the main diagonal of an initial matrix and a matrix-pattern, should coincide, and that, naturally, takes place. Thus, the values of a function to be inverted to function-constructor `FORMMA` in relation to elements of the main diagonal of a matrix will be obtained.

7 Functions

Let's give an example of the function calculating a factorial of the integer non-negative number and described in the mathematics by a recurrent relations

```
FACT (1) = 1
FACT (N) = N * FACT (N-1)
```

In language Santra 3 this function is described by means of special function `FDEF` as follows:

```
<#FDEF ('FACT') ('
   {
    (<1>) = <1>;
    (eN) = <(eN) * #FACT (<(eN)-1>)>;
   }
')> =
```

Here function `FDEF` is the system metafunction intended for defining of new functions. Its call, as well as a call of any function, is pointed by prefix sign `#`, and its arguments are the name of defining function `FACT` and the text of the program corresponding to mathematical definition, given by the multitext. Without going into other syntactic and semantic details, we would like to notice that giving this program text as the multitext allows in the process of the execution of the initial program to prepare it for the execution and to delay the generated function till the moment of its call. Thus one program is generating dynamically in the process of execution of another program.

The moment of execution of the given program is defined by the call of the function representing this program, by its name, for example:

```
<#FACT (<100>)>
```

The calculated value, naturally, should be a part of the expression in which it is used, properly.

8 Abstract Data Types

Another function-constructor is the function of assigning type to the object. It is named `CLASS` as the word “class” covers a wide sphere of subjects and phenomena. Its first parameter is the classname, and the second is classified (typified) expression. For example, for the classification of structure of the matrix which is the value of a variable `eM` as an abstract type “matrix”, it is necessary to write: `<#CLASS (‘M’) (eM)>` or `<#CLASS (‘M’, eM)>`, which is the same. Here letter `M` is using as a class name.

Function CLASS is the function-constructor in relation to the second parameter. This function is used, for example, in the description of the function `FORMMU` forming a matrix of an abstract type and having a format of the function `FORMMA`, in the following way:

```
<#FDEF (‘FORMMU’)
       (‘eA = <’#CLASS(‘M’)‘(<<(’#FORMMA‘eA)>>)>
’)>=
```

Here the function `FORMMA` — mentioned above function directly forming matrix structure. By a call of the function `FORMMU` the function `FORMMA` is called first then the type “matrix” is assigned to the generated by the function `FORMMA` structure what is carried out by the function `CLASS` call. All parameters of the function `FORMMU` are passed to the function `FORMMA` entirely by means of the only variable `eA`. (By the way, in the example of finding the elements of a matrix given above instead of the function `FORMMA` it would be possible to use the function `FORMMU`). Syntax of a call of the function `FORMMA` has also some specifics that we will not mention here.

Now for actions with typified matrixes it is necessary to describe operations so that it would be possible to call corresponding functions for actions with matrixes in its pure content, i.e. without attributes of typing. For example, arithmetic operation `+` occurrence in arithmetic expression causes the function `ADDU`, intended for addition of objects of various types of data, call accordingly. This function should distinguish the types of data and call the corresponding function which is carrying out the addition of objects in the pure content.

The example of the description of the function ADDU in special case for recognition only objects of matrix type and a call of corresponding function performing addition of matrixes in its pure content is the following:

```
<#FDEF (‘ADDU’) (‘
   {
    (<’#CLASS(‘M’)‘(eM1)>, <’#CLASS(‘M’)‘(eM2)>) =
     <’#CLASS(‘M’)‘(<’#ADDM‘(eM1, eM2)>)>;
   }
’)>=
```

Let’s look at this in more details.

The definition of the function ADDU is given by the function FDEF, and the function CLASS serves for giving the “matrix” data type. The sign , is used for the separation of the two added parameters, and the sign = is used for the separation of matrixes recognition and their assigning as values to variables eM1 and eM2 from the following action: actually addition of matrixes in its pure content by the function ADDM and result classification as matrixes of abstract type by the function CLASS. Here the delay of computations is actively used; it is possible to illustrate it by displaying a delay of computations by a lower level on a vertical instead of the use of inverted commas as follows:

```
<#FDEF(    )(   #CLASS( )         #CLASS( )         #CLASS( )   #ADDM               )>
       ADDU  {(<       M (eM1)>, <       M (eM2)>)=<        M (<     (eM1, eM2)>)>;}
```

Here at the bottom level the elements which should be passed to the functions FDEF without change are located, i.e. their computations concerning preparation of parameters should be delayed, for what they have been presented by multitexts. Furthermore, type of matrix M is simply a letter which then is used by the function CLASS as a classname. Similarly, the name of the defined function ADDU is also a set of letters similar. These elements essentially are the literals given by the means of a of pair inverted commas.

Elements, the computation of which should be delayed in essence, are also entered by the means of multitexts. Two aspects of a delay of computations in this case take place with the aim of providing of: 1) abstract data type recognition and extraction of matrixes structures in the pure content 2) actual addition of matrixes and typing the result as a matrix. Abstract data types providing consists of 1.1) pattern variables forming and 1.2) call of the function CLASS with these variables as arguments. Each of these stages would be more logically represented by the separate level of the multitext the depth of which would point out the order of it fulfilling. Accordingly, the number of levels should be 3, including the external level of the call of the function FDEF and the transformation of the names of the functions CLASS and ADDM into references to functions. The second aspect does not demand splitting on sublevels and can be left, as shown above. Let’s list the contents of these levels integrally:

0. the function FDEF call, names of the functions FDEF, CLASS and ADDM which must be transformed in references to the functions, plus corresponding parenthesis:

1. the classname of matrixes, the variables which must be formed with corresponding parenthesis, the syntactic signs of statements and functions, which form result, calls with the corresponding parenthesis and the angle brackets;
2. the syntactic signs of the function `CLASS` call for the purpose of pattern forming, including the necessary parenthesis, and just a call, indicated by angle brackets.

It is possible to illustrate it as follows:

```
<#FDEF(    )(    #CLASS( )         #CLASS( )           #CLASS( )   #ADDM               )>
       ADDU   {         M (eM1)            M (eM2)  = <       M (<      (eM1, eM2)>)>;}
               (<              >, <               >)
```

In essence, at the top zero level of the multitexts the names of the functions `FDEF`, `CLASS` and `ADDM` are transformed into the references to functions and the call of the function `FDEF` is being executed. Transition to the following levels is caused by the necessity to pass the parameters of the function `FDEF` in its original invariable kind already for the substantial transformation. Furthermore, actually just the function `FDEF` forms the statement into which calls of the functions `CLASS` and `ADDM`, brackets, a semicolon and variables must be entered properly. The sequence of actions for the correct forming of a name of defined function and actually the statement representing its body is set by corresponding levels of multitexts. Furthermore, at second level there are only the angular brackets which provide the function `CLASS` calls and parenthesis, containing them (the comma relates to the parenthesis structure), for the purpose of data abstraction providing. However, it turned out to be that the step of the data abstraction providing, caused by the act of the function-constructor `CLASS` call, does not depend on the step caused by the act of pattern variables forming. In this connection it appeared possible to join together two bottom levels of providing of abstract data types due to their independence which takes place in essence and to avoid the occurrence of the second level by that as it was shown in the example given in the beginning.

9 Pattern Matching and Data Abstraction

In this paper it is shown how it is possible to provide data abstraction by the means of only one function intended for data types setting within the framework of the system of symbolic manipulations based on the pattern matching. Such opportunity follows from a special way of building of patterns which is so taken that to provide a possibility to obtain values of the inversed functions. As a result we get the opportunity of obtaining a value of function being inversed to function which set the types of data. Obtaining of such value allows both to distinguish data by their types, and to take this data itself for the performance of operations above them, that is a necessary condition of supporting the data abstraction. The necessary way of building the patterns is provided on the basis of multilevel computations. The noted opportunity of data abstraction appears entirely within the framework of the mechanism of symbolic manipulations based on the pattern matching principle at providing of multilevel computations in a special manner.

The paper by Philip Wadler [28] also is devoted to a problem of interrelation of pattern matching and abstract data types; Wadler offers a way of reconciling pattern matching with data abstraction for the purpose of possibility of their fitting together, while both the mechanism of pattern matching and the mechanism of data abstraction are self-defined and independent mechanisms. In his paper possibility of valuable qualities of a combination of two these mechanisms is shown.

Thus, the approach offered by us and Wadler's approach seem to be absolutely different. First, Wadler considers two mechanisms and, secondly, these mechanisms are independent, while in our case only one mechanism of symbolic manipulations on the basis of pattern matching takes place. Nevertheless, the doubtless conceptual likeness of the two approaches takes place, despite the essential distinction of objects considered. We will look at it in more details.

According to Wadler, the mechanism of reconciling pattern matching with data abstraction offered by him could allow to use all possibilities of pattern matching during the use of data abstraction. Initial Wadler's position consists in that the pattern matching is the convenient and effective mechanism of the data transformation for such purposes as the proof of correctness of programs or their transformation; however, the presence of abstract data brings these valuable qualities to nothing. The problem of reconciling pattern matching with data abstraction is caused by especially various ways of data representation and their processing: for the systems based on pattern matching the representation providing the convenient review of objects structure is used and for data abstraction systems the representation is approached to machine one for the purpose of peak efficiency reaching and compactness. As a result data of abstract types appear to be hidden from direct visualization by means of pattern matching. For the purpose of such visualization Wadler suggests to enter special functions, separately for each concrete data type. Besides such functions entering it is necessary to enter inversed functions so for transformation of the obtained results by pattern matching means into abstract type. The term "visualization" is used just as a reflection of such qualities as clearness, obviousness, transparency and convenience taking place in pattern matching systems.

Here two things are looking through at once: 1) at us all actions are carried out exclusively within the framework of system of symbolic manipulations, and Wadler connects absolutely various two mechanisms: the mechanism of symbolic manipulations and the data abstraction mechanism and 2) at us it is entered the only one function, at Wadler — two mutually inversed functions. It is interesting that at visible distinction there is one similarity: Wadler in addition to the abstract data visualization function suggests to enter inversed to it function while at us unique function of data type setting is entered, and computation of value of inversed to it function is carried out automatically on the basis of certain manner of performance of symbolic manipulations. Thus, at us two functions, direct and inversed take place also. However, inversed function is virtual, and it does not need to be defined unlike Wadler. Thus, despite formal distinction of two approaches, they conceptually have much in common in view of a generality of properties both symbolic manipulations, and data abstraction without dependence from their implementations.

It is necessary to notice also, that Wadler offers not a concrete implementation, but only the interesting idea which is subject to the further development. We offer the concrete implementation, automatically giving the possibility to use the data abstraction during the performance of symbolic manipulations, in particular, for algebraic computations. Despite all that, undoubtedly, the ideas of Wadler deserve attention.

10 Conclusion

In this paper a number of properties is shown which were received in the particular case of multilevel computations with the use of the symbolic manipulations language on the basis of the pattern matching of Refal type during the inclusion of a possibility to compute the patterns in the language. These properties are the following:

- the possibility to find value of functions being inversed to the functions-constructors at their use as a part of the patterns and
- on this basis — directly following possibility of data abstraction.

Let's notice that the specified possibilities automatically become the possibilities as well of the subsystems of algebraic computations in full at its implementation into the system of the symbolic manipulations constructed on the basis of the provided ideas. Implementation itself conceptually does not concern the mechanism of symbolic manipulations, and changes syntax of the constructed language a bit. These changes consist, mainly, in the insignificant expansion of the language of symbolic manipulations with new possibilities.

The expansion of Refal by possibility to compute patterns leads to interesting and valuable qualities. It is necessary to note the naturalness of inclusion in the language of this possibility which practically does not change the syntax of the source language, but only expands it possibilities, due to new semantic properties. Additional inclusion in the language of the means of algebraic computations appeared to have little effect on the syntax of the language which shows the expediency of expansion. The possibilities of algebraic computations were automatically added with the means of data abstraction and, besides, were essentially expanded by the full volume of symbolic manipulations in the widest sense, including symbolical making of new program fragments and their execution that can appear important for algebraic computations in essence.

11 Acknowledgments

I would like to express my sincere gratitude to Andrei V. Klimov for permanent attention to my work and help without which the present work would be hardly accomplished. I am deeply grateful to Georgy B. Efimov for his interest to my work and moral support from the beginning of my scientific activity. I would also like to express gratitude to Gleb V.Fomin for his support and assistance.

References

1. And.V. Klimov, Ark.V. Klimov, A.G. Krasovsky, S.A. Romanenko, E.V. Travkina, V.F. Turchin, V.F. Khoroshevsky, I.B. Shchenkov. *Basic REFAL and Its Implementation on Computers*. CNIPIASS, V-40, Moscow, 1977. (In Russian).
2. Ark.V. Klimov. *Refal-6*. `http://refal.ru/~arklimov/refal6`
3. J. Barklund, K. Boberg, and P. Dell'Acqua. A Basis for a Multilevel Metalogic Programming Language. In L. Fribourg and F. Turini (eds.), *Logic Program Synthesis and Transformation — Meta Programming in Logic, LNCS* 883, Springer, 1994, pages 262–275.
4. Jim Grundy, Tom Melham, and John O'Leary. A Reflective Functional Language for Hardware Design and Theorem Proving. *Journal of Functional Programming*, 16 (2):157–196, March 2006.
5. R.F. Gurin, S.A. Romanenko. *The Programming Language Refal Plus*. Intertekh, Moscow, 1991. (In Russian).
6. V.L. Kistlerov. *Principles of Building of Language of Algebraic Computations FLAC*. Preprint In. Prob. Contr., Moscow, 1987. (In Russian).
7. F. Nielson, and H.R. Nielson. Two-level Semantics and Code Generation. *Theoretical Computer Science* 56, 1 (Jan. 1988): 59–133.
8. A.W. Niukkanen, I.B. Shchenkov, G.B. Efimov. *A Project of a Globally Universal Interactive Program of Formula Derivation Based on Operator Factorization Method*. Keldysh Inst. of Appl. Math. RAS, preprint No 82, 2003.
9. L.V. Provorov, V.S. Shtarkman. *ALKOR: System of Analytical Manipulations. Release 1. The Source Language Description*. Keldysh Inst. of Appl. Math. RAS, preprint No 61. Moscow, 1982. (In Russian).
10. L.V. Provorov, V.S. Shtarkman. *ALKOR: System of Analytical Manipulations. Release 2. Operations on Power Series, New Possibilities of System*. Keldysh Inst. of Appl. Math. RAS, preprint No 166. Moscow, 1982. (In Russian).
11. M.J. Shashkov, I.B. Shchenkov. Use of Symbolic Manipulations for Construction and Research Differences Schemes. *Proceedings of All-Union Conference of System for Analytical Transformations to the Mechanic*, GSU, Gorkiy, 1984. (In Russian).
12. M.J. Shashkov, I.B. Shchenkov. *System DISLAN*. Keldysh Inst. of Appl. Math. RAS, preprint No 23, 1985. (In Russian).
13. I.B. Shchenkov. System SANTRA. In *Works of the International Meeting on Analytical Manipulations on the COMPUTER and to their Application in the Theoretical Physics*, JINR, Dubna, 1985, pages 39–44. (In Russian).
14. I.B. Shchenkov. *System for Symbol-Analytical Transformations SANTRA. The Source Language (the Informal Description)*. Keldysh Inst. of Appl. Math. RAS, preprint No 19, 1987. (In Russian).
15. I.B. Shchenkov. *System for Symbol-Analytical Transformations SANTRA-2. The Description of a Formal Part of the Source Language*. Keldysh Inst. of Appl. Math. RAS, preprint No 1, 1989. (In Russian).
16. I.B. Shchenkov. *System for Symbol-Analytical Transformations SANTRA-2. The Description of Dynamic Functions*. Keldysh Inst. of Appl. Math. RAS, preprint No 7, 1989. (In Russian).

17. I.B. Shchenkov. *System for Symbol-Analytical Transformations SANTRA-2. The Description of the Functions Providing Nonalgebraic Operations.* Keldysh Inst. of Appl. Math. RAS, preprint No 21, 1989. (In Russian).
18. I.B. Shchenkov. *System for Symbol-Analytical Transformations SANTRA-2. Operations over Expressions of the Basic Classes.* Keldysh Inst. of Appl. Math. RAS, preprint No 14, 1991. (In Russian).
19. I.B. Shchenkov. *System for Symbol-Analytical Transformations SANTRA-2. Operations over Matrixes.* Keldysh Inst. of Appl. Math. RAS, preprint No 15, 1991.
20. I.B. Shchenkov. *System for Symbol-Analytical Transformations SANTRA-2. Preparation of Programs and a Debugging Facility.* Keldysh Inst. of Appl. Math. RAS, preprint No 1, 1993.
21. Guy Lewis Steele, Jr. and Gerald Jay Sussman. *The Revised Report on Scheme, a Dialect of Lisp.* Massachusetts Institute of Technology. MIT AI Memo 452. January 1978.
22. Walid Taha. *Multi-stage Programming: Its Theory and Applications.* PhD dissertation. Oregon Graduate Institute of Science and Technology, USA. 1999.
23. Walid Taha, Tim Sheard. MetaML and Multi-stage Programming with Explicit Annotations. *Theor. Comput. Sci.* 248 (1–2: 211–242) (2000).
24. V.F. Turchin. *The Algorithmic Language of Recursive Functions (Refal).* Keldysh Inst. of Appl. Math. AS USSR, 1968. (In Russian).
25. V.F. Turchin. *Programming in Language Refal.* Keldysh Inst. of Appl. Math. AS USSR, 1971, preprints N 41, N 43, N 44, N 48, N 49. (In Russian).
26. Valentin F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 8, Issue 3. MIT Press, 1986. Pages: 292–325.
27. V.F. Turchin. *Refal-5, Programming Guide and Reference Manual.* New England Publishing Co., Holyoke, 1989.
28. Philip Wadler. Views. A Way for Pattern Matching to Cohabit with Data Abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1987), Munich, Germany, January 1987*, pages: 307–313.