UNIWERSYTET TECHNOLOGICZNO-PRZYRODNICZY
IM. JANA I JĘDRZEJA ŚNIADECKICH W BYDGOSZCZY
ZESZYTY NAUKOWE NR 267
TELEKOMUNIKACJA I ELEKTRONIKA 20 (2017) 25-33

SYSTEM APPROACH TO BUILDING FUNCTIONAL PROGRAMS

Vasyl Zaiats[1], Jacek Majewski[1], Beata Marciniak[1], Marii Zaiats[2]

[1]UTP University of Science and Technology, Faculty Telecommunication, Computer Science and Electrical Engineering, Al. prof. S. Kaliskiego 7, 85-785 Bydgoszcz, Poland,
e-mail: zvm01@rambler.ru, vasyl.zaiats@utp.edu.pl ,
[2]Lviv National Polytechnic University, ISM Department, st. S. Bandera 12, 79013 Lviv, Ukraine,
e-mail: zayats.mariya@gmail.com

Summary. The Basic approaches for constructions of the functional programs are considered in the article. The main methods of optimization of the new functions and of the functional are determined. The authors proposed a system approach to building and optimizing the functionality of applications that can be used in the creation of the system of recognizing objects and phenomena. The essence of the approach in a gradual improvement software through software optimization. The developed software is always open and its can be modified, improved or refilled. Each of methods is illustrated by the examples of realization in the environment of Lisp**.**

Keywords: functional program, declarative language, functional defined, Lisp environment, accumulation parameters.

1. INTRODUCTION

We consider approaches to the construction and optimization of newly functional programs to improve the efficiency and reliability of their work. The authors proposed a system approach to creating software for the functional language. The essence of the approach is the constant improvement of the developed software product, its renewal and expansion of functional capabilities with insignificant algorithmic complications, and in some cases be simplification.

The feasibility of using this approach is illustrated by specific application developments are implemented in an environment of standard Lisp [10], which refers to declarative programming languages.

Construction of modern programming languages today is far from perfect. Each of the known languages have their advantages and disadvantages. To determine the usefulness of a language should take into account such considerations:

a) Clarity, simplicity and consistency of language concepts. Obviously, we must to avoid subtle and tricky language restrictions. These restrictions must not be too ambiguous. Semantic clarity of language – is what determines its value.
b) Clarity structure of the program. This requirement must provide syntactic clarity programs written in that language. It should be such as to design other than

semantics, and syntactic records which differed. It is essential that the structure of the program reflects the structure of the algorithm that allowed the development programs the principles of structured programming, software design hierarchy - top to bottom. In this approach, the structure of the program is readily available for the diagnosis, modification and optimization.

c) Naturalness use. This should provide most successful in solving the problem of data structure, to make operations, control structures and easily understandable syntax. This greatly simplifies the creation of software in a given field of knowledge or technical applications.
d) Ease of expansion. Software that are created in that language, can be seen as an extension language. In fact, most programming languages, provides the programmer mechanisms for creating routines. However, the properties of the language itself can facilitate or complicate their use. This ease of expansion the most pronounced in programming languages that have identical presentation of data and applications.
e) External software. This is one aspect that affects the efficiency of the use of language. If we have powerful testing tools, editing, storage, software modifications, then can be made weak language convenient to use, than it without a strong technical support.
f) He effectiveness of creating, testing, transmission, implementation, modification and practical using application programs.

2. ADVANTAGE OF THE FUNCTIONAL LANGUAGES TO RELATION PROCEDURAL

One of the fundamental properties of programming languages, which enables clear calculation described in that language – simple semantics. The great advantage of functional languages like logical that there are some basic concepts, each of which has a simple semantics. In particular, the semantics of functional languages understood in terms of the values that are expressions, but not in terms of action sequences and their use. But from a practical point of view it would be fairer to conclude that a strictly functional language is very elementary and some of its expansion to significantly increased efficiency and clarity of certain classes of deductions. Obviously, it is necessary to distinguish between purely syntactic extensions the semantics requires caution, because it is difficult to understand (clear) already debugged functional programs.

Today, developed hundreds of different programming languages. Even in 1969. Jean E. Sammet [6, 10] gives a list of 120 languages that are quite widely used. This amazing number of programming languages contradicts that most programmers in your practice uses several programming languages, and many of them – one or two programming languages. There is the question for what come to the development of different languages at the unlikely possibility of their implementation. However, if not limited to superficial acquaintance with the language, and have a deep understanding of the concepts underlying the design of programs in that language, then no doubt you can verify the feasibility of the development of various programming languages based on the following considerations:

a) Due to different programming languages study improved understanding of a particular language, its concepts and basic methods and techniques that it uses.

A typical example is the recursion. With proper use it can be elegant, effective program and its application to a simple algorithm could lead to an astronomical increase in time costs. On the other hand, lack of use of recursion in languages such as Fortran, Cobol and understanding of the basic principles and methods of implementation of recursion can clarify the limitations of language, which at first glance is false,

b) The value of programming languages expanding stock of useful programming structures and promotes thinking. Working with data structures of one language, and produce appropriate structure of thinking. By studying other languages and design methods to implement, expand programming thesaurus.
c) Knowledge of several languages allows reasonably choose a programming language for solving a particular problem,
d) The development of a new programming language, like natural language of human communication is always easier if there are several well-known languages,
e) Knowledge of principles of construction of various programming languages facilitates the development of a new programming language.

Construction of modern programming languages today is far from perfect. Each of the known languages have their advantages and disadvantages.

Most modern programming languages are universal, since they give to record any algorithm that language, if not impose restrictions on runtime and capacity of memory, algorithmic complexity, etc. If anyone will offer a new programming language, it is likely to be universal if ignored limits on memory or time. Comparing different programming languages should not proceed with the proportion of what they can do and qualitative differences that define elegance (briefness and clarity), lightness (transparency) and efficiency (speed and hardware) programming them. This comparison should be done in the context of specific applications.

Traditional (algorithmic) programming language versus declarative (descriptive) is quite large and bulky, because do not allow:

a) maximize the capabilities of modern computer technology to ensure the effectiveness of software;
b) clearly visualize algorithms and programs to provide easy inspection and modification of the latter.

Declarative programming languages, which include strict functional programming languages such as S-Lisp [10], R-Lisp [7], Reduce [2], Common Lisp and Auto-Lisp [1] is quite simple and only provide it high enough the severity of programs compared with traditional languages. Several functional programs can effectively run on modern computers, however not as efficient as relevant programs with the assignment operator. This is due to the structure of the architecture of modern computers. In addition, the choice of a slightly different structure view, than is usual in Lisp, give tool for provides a more clear representation of programs and increase their efficiency using modern computers of old architecture.

On the one hand, modern programming languages should effectively use modern machines, but from the other – to give the algorithms clearly express software to facilitate verification of the latter. Strictly functional language has simple in structure, shows a higher severity compared with traditional languages, where there is the assignment operator. This is due, largely, with the way we present of data structures.

Something the choice of data structures of can provide increased efficiency and functional applications on modern computers. In particular, is a problem using the features of the components of the result, because it is important to have elegant solution.

Ones from fundamental properties of programming languages, which enables calculation, clearly describe the language - it is simplest semantics. The great advantage of functional languages is that there are some basic concepts, each of which has a simple semantics. In particular, the semantics of our language understood in terms of the values that are expressions, but not in terms of action sequences and their use. But from a practical point of view it would be fair to conclude that a strictly functional language is very elementary and some of its expansion would significantly increased the efficiency and clarity of certain classes of calculations. Obviously, it is necessary to distinguish between purely syntactic extensions and extensions that require change semantics. Changing the semantics requires caution, because it difficult then to understand already debugged functional programs.

3. STRICTLY FUNCTIONAL LANGUAGE

When it comes to building a strictly functional language, new functions or functional are based on some basic set of primitive features or functions [12]. These include functions:

CAR (X) – it selecting the first element of the list **X**,
CDR (X) – return the remaining elements of list **X** without the first element,
CONS (X, Y) – construction of a new list, where the parameter X is the first element in the list **Y**,
EQ (X, Y) – the predicate is true in the case of equivalent atoms **X** and **Y**.

The presence of these primitive functions and presentation arithmetic operations as functions give the possibility of setting recursive (repeat) functional calls or appeals functions to itself (principle of function composition) can build a fairly substantial functional program [5, 8]. Demonstrate this procedure we can on example building of function the **CONNECTION (X, Y)**, what generates a new list, which will list all the elements of **X** and **Y**. In addition, each argument can be either an atom as a list of arbitrary length. Direct analysis of all possible cases in list X leads to such a functional definition:

CONNECTION (X, Y) = if EQ (X, NIL) then if EQ (Y, NIL) then NIL
another if Y EQ (Y, NIL) then X another
CONS (CAR (X), CONNECTION (CDR (X), Y))

Thus, if X = NIL connect (X, Y) = Y regardless of the value of Y, the definition can be rewritten more optimal way:

CONNECTION (X, Y) = if EQ (X, NIL) then Y
then If EQ (Y, NIL) then X another
CONS (CAR (Y), connection (CDR (X), Y))

If we take into account, that when X = NIL regardless from kind of function **Y** we have **CONS (CAR (X), connect (CDR (X), Y)**, then checking's for **Y** can also be put down and we can write functional definition as:

CONNECTION (X, Y) = if EQ (X, NIL) then Y else
CONS (CAR (X), CONNECTION (CDR (X), Y))

All three versions mean equivalent functions and provide the same result. The first version can be preferred because it clearly lists all possible cases. The third version - is its shortness. But the second and third versions of this tool for connect have different efficiency. The second version avoids calculations in the case **of Y = NIL** by redundant checks when **Y <> NIL**. The third version avoids revisions to Y, but requires rebuild X even when **Y = NI**. Thus, this definition can be done more optimal, particularly using additional sub functions.

Thus, we came to systematically important conclusions after considered constructions of function **CONNECTION(X, Y)**:

a) no other way to solve the problem optimally than continuous improvement achieved,
b) necessary technical instrument for continuous save given knowledge's, them change and improvement.

4. THE USE ADDITIONAL UNDER-FUNCTIONS

The often for building optimal features should be introduced additional functions [13]. Thus, to optimize definition of the function **CONNECTION (X, Y)** previous could be used intermediate function such as **CONNECT(X, Y)**, which connects **X** and **Y** assuming that **Y <> NIL**. Then we arrive at the definition:

CONNECTION (X, Y) = if EQ (X, NIL) then X another CONECT ((Y, NIL)
CONECT (X, Y) = if EQ (X, NIL) then Y
another CONS (CAR (X), CONECT (CDR (X), Y))

Note, that in the previous section we got advantages combined second and third versions thanks of the use function connect

It is generally welcome in a functional programming when the program of building and identifying the main new features are defined in terms of the old. Thus, the functional program consists of sets of sub-functions that are defined through the second one. This feature or functionality that is the purpose of the calculations it is the main program, the root cause of the other, and all other functions are sub-programs.

The choice of sub-functions in the development of the main features is the central problem of structuring programs. Sometimes the standard sub-functions are in themselves, but more frequent occasions when a good selection sub-functions special purpose simplifies the structure of the set of functions in general. To build a well-structured program, we can give one piece of advice: try to constantly improve what is already done. Actually, this is one of the principles of optimal results not only in a functional programming, but also in any other field of knowledge.

5. USE OF ACCUMULATION PARAMETERS

The idea of the method parameters of accumulation is to determine the function of supporting an additional parameter that is used to accumulate the desired result [13].

To illustrate the essence of this programming method to wrap function **ROTATION (X),** this is responsible for the list's elements, possibly empty. For this, we introduce an additional function **ROTATE (X)**, where **X** – a list that is subject to rotation, and **Y** – an additional parameter that accumulates reverse list. We give the following definition:

ROTATION (X) ≡ IF EQ (X, NIL) then Y
ROTATE (CDR (X), CONS (CAR (X), Y))

Because of this feature can determine the function **ROTATION (X))**:

ROTATION (X) = ROTATE (X, NIL)

In beginning we explain how it works, and then describe the algorithm for its construction. When called the function **REVERSE (X, Y)**, y is a list of all accumulated a list of items considered to be the rotated. So if **X** is **NIL**, then y contains all the reverse list, and if x is not NIL, then we can accumulate y **CAR (X)** and then recursively call the sample for processing **CDR (X).** Let us give a table of successive calls function **REVERSE (X, Y)**, that first time by help of functions **ROTION (X)** drawn to the **LIST (A B C D)**.

Table 1. Table of successive calls function REVERSE (X, Y)

X	Y
(A B C D)	NIL
(B C D)	(A)
(C D)	(B A)
(D)	(C B A)
NIL	(D C B A)

In fact, we recorded guessing definition of this function, and then described how this works. In order to apply the conventional method to construct the functions is necessary prove in appearance of results ROTATE(x, y) at an arbitrary y. Let the result of the function rotate (x, y), where x and y are lists, possibly empty, a list of all elements x, are taken in reverse order and are complemented by all elements y in their original order. Thus

ROTATE (X, Y) = CONNECTION (ROTATION(X), Y)

Although this definition is not entirely appropriate because the unknown is the function ROTATE. Yet now we can write the algorithm for constructing functions:

case (1): X = NIL ROTATE (X, Y) = Y
case (2): X = NIL
Let ROTATE(CDR (X), Z) = CONECTION (ROTATION CDR ((X), Z)
then
ROTATE (x, y) = CONECTION (ROTATION (X), Y) =
CONECTION (ROTATION (CDR (x)), CONS (CAR (x), y)) =
= ROTATE (CDR (X), CONS (CAR (X), Y))

The algorithm is faster proved justice above the designated function than the method of its construction, so in the second case, the transformation is quite complex.

The problem of building effective functional programs can be successfully solved by under-function successful recruitment of additional options. Indeed, if we count the number of calls to the designer last version function **ROTATION**, they will be exactly n, if the length of the list is x n. If you compare this number of calls the constructor with a number of n × (n-1) / 2 in the case of the function **ROTATION** (X) without parameters of accumulation, we have significant economy of machine resources.

6. DIRECTIONS OF OPTIMIZATION FUNCTIONAL LANGUAGES

The principal feature of modern computers is that they store calculated values in the memory cells, occasionally replacing their contents or overwriting them. This property is reflected in the design assignment, which is typical for traditional programming languages. There is another aspect of the real machine, through which the traditional language more effective compared to functional programming languages. This concept of access to data structures using indexing. In this connection, functional programs cannot achieve the same effect on modern computers as a program algorithmic programming languages. We can specify at least three outing from this situation:

a) in the unwillingness to put up with some loss of efficiency (in gains in quality and elegance program) refuse functional programs,
b) to develop methods building of functional programs so that their expressiveness and clarity combined with efficiency, comparable with programs written in traditional programming languages,
c) restructure modern computers so as to meet the objectives of interpretation of the functional programs.

From a certain point of view, each of these approaches has the right to exist. Operation appropriation is closely associated with binding value for variable in a functional language. It can be shown that any application where there is a transaction assigning certain variable values can be transformed into functional program. At this functional program with restrictions on its structure can be compiled into the program with the assignment. Thus, one could argue that functional languages do not preclude the effective use of modern computers. But this is only part of the problem because the assignment transaction value of the array type

A[I]:=1
A[I+1]:=A[I]+1

significantly different from operation assignment value of variables. In a strictly functional language is no concept of subtraction sequence of actions and we to behave with arrays as with arrays of integer values. The basic operations on arrays are index values for components and constructing a new array from of old values. If a – values of the array, and – the index, X – value, the record

UPDATE (A, I, X)

means maintaining of array values a and change of **I** - th components on the value of **X**.

And the expression

UPDATE (A I, A [J]), J, A [I])

means replace places of i-th and j-th array elements.

Therefore, the problem of correct interpretation of function correct update is quite complicated taking into account that one and the same value array can have different names. For the operation updating values in the array requires multiple copy these values. The question is: can choose a data structure to fully or partially avoid these copies? Problems arise when used for this purpose simple function as cons, as structures that are components of the cons are not copied, but saved only pointers to them in the structure resulting from cons. But the main advantage is lost today's computers - the possibility of indexing. Indeed, the linear sequential access lists and access time-element is proportional to the number of items and. However, in the array of memory cells simultaneously is access to any item, regardless of position. Arrays can be represented as a binary tree; access time proportional value $\ln_2 n$. However it is incompatible with access time to the unit, that is needed for such elementary statements **a [i]: = x**. Functional language performance on modern computers cannot compete with traditional languages, if efficiency is an absolute requirement. However, applications written with by operators assignment structured values are programs low level. These require considerable effort for their construction and therefore are difficult to correct interpretation.

Increasing the speed and capacity of computer memory nowadays makes available a number of important functional programs applied nature. But qualitative leap in the use of functional languages can be expected only with the advent of new computer architecture that is focused not on the use only of new technology, but her is tied with peculiarities of the language itself, built based on the natural needs of the user.

7. CONCLUSIONS

With using discussed ways and suggested herein of system approach to building applications to optimize complex structured functional programs we can improve of performance, the technical means of realization and form of entry to avoid repetition logical outcome, which is important when working with large volumes of data and database processing of symbolic of information, creation of interpreters other programming languages and recognition of objects complex dynamic structures.

This approach (permanent improvements, from simple to complex, from obvious to unlikely) should be used as in the process of development of new information so and in teaching that will promote a better understanding of material and its efficient use. Creating applications with presentation of new material advisable to implement regardless of content using universal modeling language (umm) [3, 6, 11], that have advanced functional capabilities for creating, storing, permanent updating and presentation of product information.

BIBLIOGRAPHY

[1] Badaev Yu. I., 1999. Theory of functional programming. Movi Common Lisp is Auto Lisp. Kiev, 150.
[2] Edneral V.F., Kryukov A.P., Rodionov A.Ya., 1984. Language of analytical calculations. Publishing House of Moscow State University, 176.
[3] Gamma E., Helm R., Johnson R., Vlissides J., 2010. Wzorce projektowe. Helion Gliwice.
[4] Henderson P., 1983. The functional programming. Application and implementatio. Peace Moscow.
[5] Hüvenen P.E., Slepyan J., 1990. The World of Lisp. In two vol. T.1. Introduction to Lisp language and functional programming. Trans.: Myr, 447.
[6] Jean E. Sammet J.E., 1969. Programming Languages: History and Fundamentals. Prentice-Hall New Yersey.
[7] Kryukov A.P., Rodyonov A.Y., Taranov E.M., Shablyhyn E.M., 1991. Programming language for R-Lisp, Radio and communication, 192.
[8] Maurer W.U., 1972. The programmer's introduction to LISP. London: Macdonald; New York: American Elsevier.
[9] McAllister J., 1987. Artificial Intelligence and PROLOG for Microcomputers. Hodder Arnold London.
[10] Mccarthy J., 1960. Recursive functions of symbolic expressions and their computation by machine, Comm. ACM Vol. 3, P.184-195.
[11] Wrycza S., 2006. UML 2.1, ćwiczenia. Praca zbiorowa, Helion Gliwice.
[12] Zayats V.M., 1999. The summary of the lecture in the course "Functional programming". Lviv, 55.
[13] Zaiats V.M., Zaiats M.M., 2006. Logical and functional programming. Training manual. Publisher Beskyd Bit Lviv, 352.

PODEJŚCIE SYSTEMATYCZNE DO BUDOWANIA PROGRAMÓW FUNKCJONALNYCH

Streszczenie

W pracy są rozważane podstawowe założenia konstrukcji programów funkcjonalnych. Określono główne metody optymalizacji nowych funkcji i funkcjonalności. Autorzy zaproponowali podejście systemowe do budowania i optymalizacji funkcjonalności aplikacji, które mogą być wykorzystane przy tworzeniu systemów rozpoznawania obiektów i zjawisk. Istotą proponowanego sposobu na optymalne rozwiązanie problemu jest ciągła poprawa, bazująca na optymalizacji funkcjonalnej elementów składowych oprogramowania. Rozwinięte oprogramowanie jest zawsze otwarte i może być modyfikowane, udoskonalane lub uzupełniane. Każda z proponowanych w pracy metod, przedstawiona jest jako przykład realizacji w środowisku Lisp.

Słowa kluczowe: program funkcjonalny, język deklaratywny, oprogramowanie zdefiniowane funkcjonalnie, środowisko Lisp, parametry akumulacji