ISSN 0146-4116, Automatic Control and Computer Sciences, 2018, Vol. 52, No. 7, pp. 843–849. © Allerton Press, Inc., 2018.
Original Russian Text © V.S. Vasilev, A.I. Legalov, 2018, published in Modelirovanie i Analiz Informatsionnykh Sistem, 2018, No. 4, pp. 347–357.

Loop-invariant Optimization in the Pifagor Language[1]

V. S. Vasilev[a], * and A. I. Legalov[a], **

[a]*Siberian Federal University, Institute of Space and Information Technology, Krasnoyarsk, 660074 Russia*

**e-mail: vsvasilev@sfu-kras.ru*

***e-mail: legalov@mail.ru*

Received March 15, 2018

Abstract—The paper considers methods of program transformation equivalent to optimizing the cycle invariant, applied to the functional data-flow model implemented in the Pifagor programming language. Optimization of the cycle invariant in imperative programming languages is reduced to a displacement from the cycle of computations that do not depend on variables that are changes in the loop. A feature of the functional data flow parallel programming language Pifagor is the absence of explicitly specified cyclic computations (the loop operator). However, recurring calculations in this language can be specified recursively or by applying specific language constructs (parallel lists). Both mechanisms provide the possibility of parallel execution. In the case of optimizing a recursive function, repeated calculations are carried out into an auxiliary function, the main function performing only the calculation of the invariant. When optimizing the invariant in computations over parallel lists, the calculation of the invariant moves from the function that executes over the list items to the function containing the call. The paper provides a definition of "invariant" applied to the Pifagor language, algorithms for its optimization, and examples of program source codes, their graph representations (the program dependence graph) before and after optimization. The algorithm shown for computations over parallel lists is applicable only to the Pifagor language, because it rests upon specific data structures and the computational model of this language. However, the algorithm for transforming recursive functions may be applied to other programming languages.

Keywords: data driven functional parallel programming, Pifagor programming language, code optimization, loop optimization, invariant optimization, program dependence graph

DOI: 10.3103/S0146411618070295

INTRODUCTION

Optimization is a process of equivalent conversion that eliminates redundant calculations, the execution of which does not affect the program execution path. Such conversions improve the required characteristics of the program. One such transformation, widely used in imperative programming, is the loop-invariant optimization [1]. A calculation is called a loop-invariant if for a certain group of operators it gives the same result regardless of how many times the body of the cycle is executed. When performing loop-invariant code motion, such calculations are removed from the loop body and placed in front of him, thereby reducing the amount of calculations [2].

Features of loop-invariant optimization in imperative languages, you can consider the following simple example on the C++ programming language.

```
int inv (int n, int p) {
  int s = 0;
  int i = 0;



  while (i < n-2) {
    s += p*3;
```

```
int inv (int n, int p) {
  int s = 0;
  int i = 0;
  int inv_n = n-2;
  int inv_p = p*3;
  while (i < inv_n) {
    s += inv_p;
```

[1] The article was translated by the authors.

```
    ++i;                                    ++i;
  }                                       }
  return s;                               return s;
}                                       }
```

The part of code on the left side contains the loop, that at each iteration calculates $n - 2$ and $p*3$. It is possible to remove these operations from the loop, because the variables n and p do not change inside the loop. The optimized version of the program is shown in the right side.

The Pifagor language is developed as a tool for writing architecture-independent parallel programs. It is based on the data-flow computing principle [3]. The program in this language consists of functions, each of which uniquely mapped in the dataflow graph (DFG) which reflects data dependencies between operators [4].

The DFG is acyclic because the Pifagor uses the principle of the single use of computational resources, which essentially reinforces the principle of single assignment [5]. This graph can be used to perform various optimizing transformations that ensure the generation of more efficient executable code. Specificity underlying computing language model, allows to generate recurring calculations, either recursively, or through parallel lists operations.

1. INVARIANT OPTIMIZATION IN A RECURSIVE FUNCTION

An invariant of the recursive function, by analogy with the loop invariant, we will consider calculations, the result of which will not change during the repeated execution inside recursive calls. Such calculations depend on constants and values that cannot be changed between recursive calls.

Below is a function in the Pifagor language, which describes calculations similar to the C++ example described above. Figure 1 shows the DFG of this function. The argument passed to the function is a list containing four values, which are named similarly to the variables from the previous example.

```
rec_inv << funcdef X {
  i << X:1;
  n << X:2;
  s << X:3;
  p << X:4;
  [((i, (n,2):-):[=>, <]):?]^(
    s,
    {
      block {
       next_s << (s, (p,3):*):+;
       break << ((i,1):+, n, next_s, p)
         :rec_inv;
      }
    }
  ):. >> return;
}
```

In this example:

• the immutable arguments are "*n*" and "*p*" (the second and fourth elements of the list "*X*", respectively);

• invariants are calculations "(*n*,2)*:*-", "(*p*,3)*:**", as well as operations inside the parallel list "[=>, <]". From Fig. 1 it can be seen that these calculations do not depend on changing arguments of a recursive function ("*i*" and "*s*").

Forming a parallel list "[=>, <]" is formally an invariant, but it cannot be optimized, since this value is passed to the function as an additional argument, but according to the transformation algebra, the parallel list disclosured when performing such operation [5].

To detect invariant of recursive function, it is necessary:

• build a set *Args* of nodes that are function arguments;

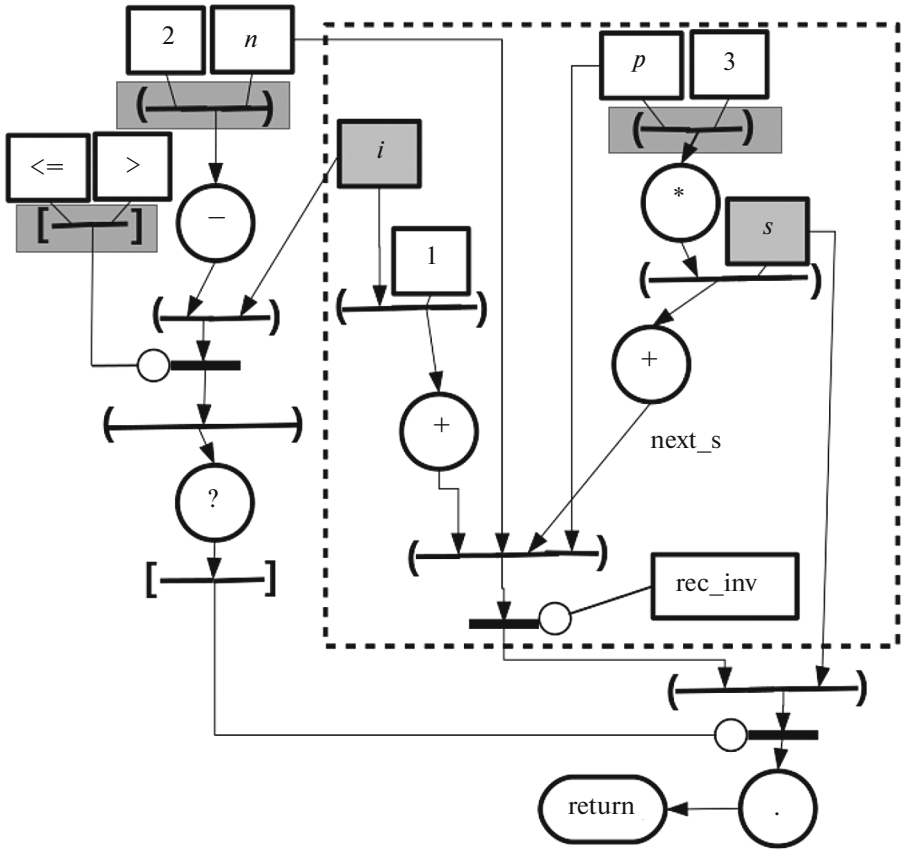

Fig. 1. The DFG of a recursive function with invariant.

• select a subset *ConstArgs* of immutable arguments from *Args*;

• select the invariant from the set of program operators(set *Invs*), using the following method: a graph node is an element of this set if it does not perform the operation of forming a parallel list and has data dependencies only on constants that are elements of *ConstArgs* and other elements of *Invs*.

To optimize the function *F*, containing invariant performed to creation of auxiliary function *G*. Calculations of invariants from the *F* function are transferred to *G*. Also in the function *G* is added to the call of *F*, herewith the results of the invariant calculation are passed ton *F* as additional arguments. In the *F* function, use values of the invariants are replaced by the use of appropriate arguments. Figure 2 shows the DFG considered above function after optimization. The following listing is the code in Pifagor after optimization.

```
rec_inv_opt_h << funcdef X {
  i << X:1;
  n << X:2;
  s << X:3;
  p << X:4;
  inv_n << X:5;
  inv_p << X:6;
  [((i, inv_n):[=>, <]):?]^(
    s,
    {
      block {
        next_s << (s, inv_p):+;
        break << ((i,1):+, n, next_s, p, inv_n, inv_p)
          :rec_inv_opt_h;
```

```
      }
    }
  ):. >> return;
}
rec_inv << funcdef X {
  n << X:2;
  p << X:4;
  inv_n << (n,2):-;
  inv_p << (p,3):*;
  return << (X:[], inv_n, inv_p)
    :rec_inv_optimization_h;
}
```

It should be noted that the optimization results are formed in the intermediate representation without generating the source code. In this article, the source code of the functions is provided only for a visual demonstration of the results of the optimization carried out.

2. OPTIMIZATION OF INVARIANT IN CALCULATIONS WITH PARALLEL LISTS

In some cases, it is possible to replace recursive calculations in the Pifagor language with parallel lists operations, which allow you to specify the simultaneous processing of all data by a single function. In this way possible to achieve higher efficiency of paralleling than using recursion, in particular, it is possible to translate such constructions into parallel loops of some languages and libraries of parallel programming.

In accordance with the transformation algebra of the data-flow model, applying a function to a parallel list transforms into many parallel operations, such that this function is applied in parallel to each element of this list:

$$[x_1, x_2, \ldots x_n] : f \rightarrow [x_1 : f, x_2 : f, \ldots x_n : f].$$

A list item can be both an atom and a data list. Duplicate calculations with parallel lists can be seen in the following example. Suppose there is a data list $XList = (x_1, x_2, \ldots x_n)$, each element must be multiplied by a value Y. To perform such operation as described above, it is necessary to create a parallel list of pairs (x_i, Y) and apply the multiplication operation to it. This situation arises frequently. At the same time,

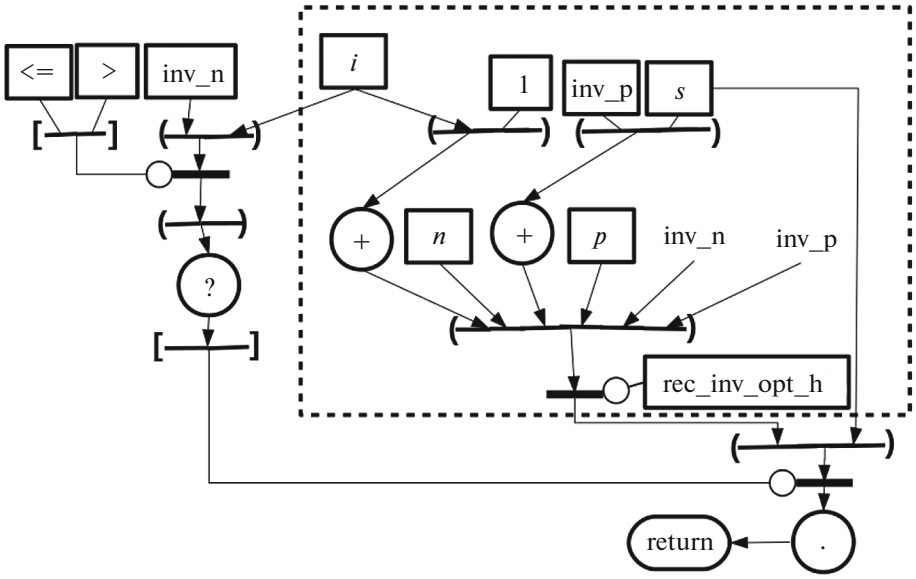

Fig. 2. The DFG of the recursive function after optimization of the invariant.

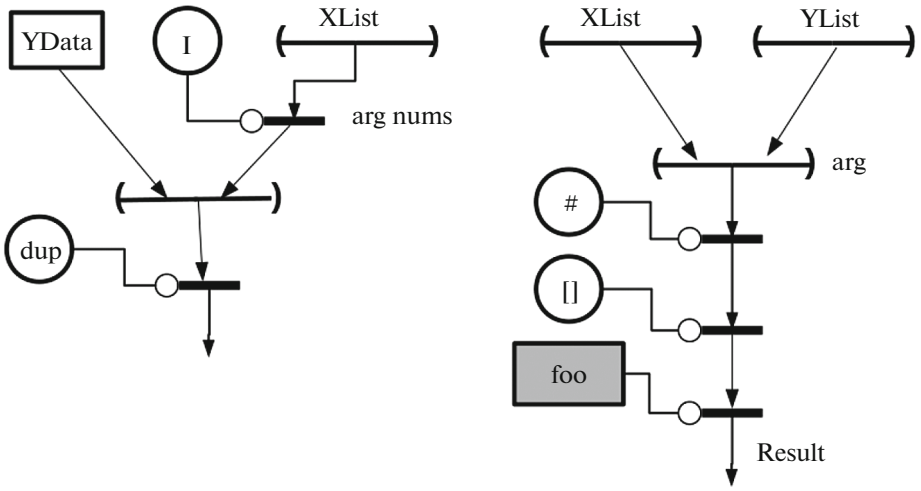

Fig. 3. A fragment of the graph that specifies repeating computations on lists.

instead of a multiply operation, another function can be used, for example, *foo*. The generalized source code for this case will look like this:

```
Len << XList:|;
YList << (Y, Len):dup;
(XList, YList):#:[]:foo;
```

This example calculates the length *Len* of the list *XList*, which elements are processed. The *Len* value is used when performing the duplicate operation (*dup*) that forms the list of data *YList*. This list consists of *Len* duplicates of *Y* values. The *XList* and *YList* are placed in another list (tuple). A transposition operation (#) is applied to this tuple, the result of which is the desired list of pairs. Then, the list of pairs is converted into a parallel list by the *[ ]* operator, to which the *foo* function is applied. Each element of the *YList* is equal to *Y*, this value will be the second argument of the *foo* function. Thus, calculations depending on Y and constants will be an invariant and can be optimized.

For the proposed method of optimization is crucial that part of the data is duplicated (operation *dup* is performed). Duplicate data are constant parameters. In this case, the invariant is the calculation in the *foo* function, depending only on constants, constant parameters and other invariants. Such an invariant can be moved from the *foo* function to the calling function.

To identify constant parameters in the function that implements the calculations on the parallel list, the optimizer searches in the DFG fragments, the scheme of which is shown in Fig. 3. *YList* contains duplicate data, in the general case there can be several such lists.

The search for invariants inside the *foo* function is performed in the same way as for recursive functions, only the method of defining immutable arguments differs (*ConstArgs*). The invariant is moved to the calling function, and the result of its calculation is passed to *foo* as an additional argument.

As an example, consider the rotation function around the axis of a three-dimensional shape given by a list of points. The function takes two parameters – the shape (*Figure*) and the angle in radians (*Alpha_rad*), which is required to rotate. To rotate the point, use the *x_rotate* function. To apply it to all points, the number of points (*PointsCount*) in the *Figure* is determined and a list of corners (*Angles*) is created, the length of *Angles* is *PointsCount*. To rotate a point, the matrix-row containing the coordinates of the point is multiplied by the corresponding rotation matrix.

```
figure_rotate << funcdef X {
  Figure << X:1;
  Alpha_rad << X:2;
  PointsCount << Figure:|;
  Angles << (Alpha_rad, PointsCount):dup;
  return << (Figure, Angles):#:[]:x_rotate;
}
// rotation of point X in three-dimensional space
```

```
// around the x-axis on alpha_rad
x_rotate << funcdef X {
  Point3d << X:1;
  Alpha_rad << X:2;
  cosA << Alpha_rad:cos;
  sinA << Alpha_rad:sin;
  rotMartix << (
    (1, 0, 0),
      (0, cosA, sinA:-),
    (0, sinA, cosA)
  );
  return << ((Point3d), rotMartix):matrix_mul;
}
```

As a result of the optimization of the shape rotation function, construct a graph equivalent to the following code fragment:

```
1 figure_rotate_inv_opt << funcdef X {
2  Figure << X:1;
3  Alpha_rad << X:2;
4
5  PointsCount << Figure:|;
6  Angles << (Alpha_rad, PointsCount):dup;
7
8  cosA << Alpha_rad:cos;
9  sinA << Alpha_rad:sin;
10
11  rotMartix << (
12    (1, 0, 0),
13    (0, cosA, sinA:-),
14    (0, sinA, cosA)
15  );
16
17  rotMatrices << (rotMartix, PointsCount):dup;
18
19  return << (Figure, Angles, rotMatrices):#:[]:x_rotate_inv_opt;
20 }
21
22 x_rotate_inv_opt << funcdef X {
23  Point3d << X:1;
24  Alpha_rad << X:2;
25  rotMartix << X:3;
26
27  return << ((Point3d), rotMartix):matrix_mul;
28 }
```

The invariant was the computation of sine, cosine of an angle, and the operation of forming a rotation matrix. In the optimized code, all calculations are performed only once, regardless of the number of points, but the resulting matrix is duplicated (the list *rotMatrices*), since they are included in the *figure_rotate_inv_opt* function (lines *5–17*).

Thus, if the *F* function contains a call to the *foo* function on a parallel list and there is an invariant, then the optimization can be performed using the following algorithm.

(1) In the *F* function, find the node of the function call, whose argument is such a list that one or more of its elements is the result of the *dup* operation.

(2) Create a list *ConstArgNums* indices of constant arguments of the *F*.

(3) In the *func* function, find and save invariants in the list *Invs*. Taking into account that arguments with indices from the list textit ConstArgNums are constants. A *Inv* node belongs to the set *Invs* if it:

(4) is an actor;

- is in the zero delayed list;
- depends by data only on constants, arguments (*ConstArgNums*) and other invariants;
- not a grouping operation in a parallel list.

(4) Move invariants from *func* to *F*.

(5) Find the use of moved invariants in the *func*; create a list *UsingInvs* from nodes whose values are missing in the *func* function. For the above example, the invariant will be, for example, the list grouping operation (*0, cosA, sinA:-*), however, this value is not directly used in the *x_rotate_inv_opt* function therefore it is not added in the *UsingInvs*.

(6) In function F, add the results of calculating the elements of *UsingInvs* as parameters of the *func* function. For this, duplicate by the built-in function *dup*. In the above example, add into *figure_rotate_-inv_opt*:

```
rotMatrices << (rotMartix, PointsCount):dup;
```

(7) In the function func add the code for extracting the calculated values of the invariants from the argument list of the function. In particular, for the rotation function, add to *x_rotate_inv_opt*

```
rotMartix << X:3;
```

3. CONCLUSIONS

The article for the Pifagor language shows the possibility of carrying out transformations that are equivalent to optimizing the cycle invariant for two cases: recursive function and parallel lists. Invariant-optimization of the recursive function may be applicable to other programming languages, however, such optimization for computations on parallel lists is based on specific language constructs and can be used only in the Pifagor. In both variants of optimization we consider the function, which body is executed multiple times. The optimizer searches for fragments in the body of this function, the result of which will be the same for all function calls. These fragments are moved out of function and calculated once (and passed to the function as additional arguments) – therefore, the transformation reduces the amount of computation. The presented optimization methods are carried out after the translation of the source code of functions, thereby providing an architecturally independent analysis of the program code. Other transformations of data-flow parallel programs, including verification, testing, debugging, as well as conversion to architecture-dependent forms, can be carried out after applying the proposed optimization methods.

ACKNOWLEDGMENTS

The research is supported by RFBR (research project no. 17-07-00288).

REFERENCES

1. Aho, A.V., Lam, M.S., Ravi Sethi, and Ullman, J.D., *Compilers: Principles, Techniques, and Tools,* 2006, Addison Wesley, 2nd ed.
2. Dortman, P.A., Program optimization in SFP, in *Programmnye sredstva i matematicheskie osnovy informatiki* (Software Tools and Mathematical Foundations of Informatics), Novosibirsk, 2004, pp. 43–49.
3. Legalov, A.I., Matkovsky, I.V., Kropacheva, M.S., Udalova, Y.V., and Vasilev, V.M., Technological aspects of creating, converting and executing functional data-flow parallel programs, *Scientific Service on the Internet: All Facets of Parallelism: Proceedings of the International Supercomputer Conference,* Moscow, 2013, pp. 443–447.
4. Legalov, A.I., Vasilyev, V.S., Matkovskii, I.V., and Ushakova, M.S., Support tools for creation and transformation of functional-dataflow parallel programs, *Tr. Inst. Sist. Progr. Ross. Akad. Nauk,* 2017, vol. 29, no. 5, pp. 165–184.
5. Legalov, A.I., Functional language for creating of architectural independent parallel programs *Comput. Technol.,* 2005, vol. 10, no. 1, pp. 71–89.