ISSN 0146-4116, Automatic Control and Computer Sciences, 2013, Vol. 47, No. 7, pp. 366–372. © Allerton Press, Inc., 2013.

Tail Recursion Transformation in Functional Dataflow Parallel Programs[1]

A. I. Legalov*, O. V. Nepomnyaschy, I. V. Matkovsky, and M. S. Kropacheva

Siberian Federal University

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

Received April 22, 2012

Abstract—The peculiarities of transforming functional dataflow parallel programs into programs with finite resources are analysed. It is considered how these transformations are affected by the usage of asynchronous lists, the return of delayed lists and the variation of the data arrival pace relative to the time of its processing. These transformations allow us to generate multiple programs with static parallelism based on one and the same functional dataflow parallel program.

Keywords: functional dataflow parallel programming, tail recursion, programs transformation, Pifagor programming language

DOI: 10.3103/S0146411613070237

1. INTRODUCTION

The development of parallel programs is currently characterized by a variety of approaches and methods, mainly because of focus on the different architectures of parallel computing systems (PCS). Existing attempts to create a language for architecture-independent parallel programming have not yielded significant results yet. The main reason for this situation is the complexity of the processes for efficient conversion of an architecture-independent program to the executable parallel code for the PCS that is practically used.

Pifagor is an example of the architecture-independent programming language [1]. Its specific features include: focusing on computer systems with unlimited resources, using dataflow principles for the computations control, and parallelism on the level of the basic operations. The absence processes that interact through shared resources, makes debugging and verification much easier [2]. Execution of functional programs is provided by event-based processor [3].

The main method for representing repetitive computations in the considered language, as well as in all functional-oriented programming languages, is recursion. Recursion usage allows getting rid of resource limitations of usual cycles resulting from the reusing already allocated resources, which could be still occupied. However, in some cases, recursive computations are inapplicable. One of the most common situations is using a very long ("infinite") repetitive calculations inherent in the algorithms employed in the real-time control systems. For functional dataflow language that leads to a repetitive recursive calls, which causes memory overflow. In case of working the program in real time mode, when it provides processing asynchronous incoming data flow, the situation becomes even more critical.

The issue under consideration is already solved for both functional and imperative sequential programming languages, but using the functional dataflow parallel language contribute their specific features. Therefore, the search for solutions that support the implementation of "infinite" dataflow functional parallel programs with limited computational resources is an important issue. The article describes approaches for translating a functional dataflow parallel program written in the Pifagor language to the representations, that allow to execute these program with the limited computing resources.

2. USING DELAYED LISTS

One of the possible ways to resolve the above issue for a program development using the Pifagor language is returning from the called function not the result of computations carried out at the end of the

[1] The article is published in the original.

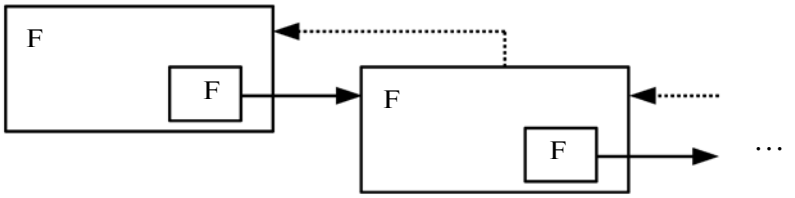

Fig. 1. The recursive calls are followed by the resources capture.

chain of recursive calls, but closed delayed list. After which disclosure of the delayed list is performed and the required computations is executed in the calling function. This approach is applicable in case of the tail recursion which is located at the end of the function before returning the result. Such recursion is easily converted into iterations in the sequential programming languages.

Let us consider factorial as an example. The function below uses typical tail recursion with accumulation of the result:

```
FactRightRec << funcdef pair {
  acc<< pair:1; N<< pair:2;
  // Testing the value of an integer argument
  // (four element list is formed with a single true value)
  [((N, 0) : [<,=], (N,1) : [=,>]) :?]^
  (
    // Incorrect negative value
      ("Incorrect negative argument", N),
    //If value equals 0, the factorial equals 1
      acc,
    // If value equals 1, the factorial equals 1
      acc,
    // Else the recursive computation starts
      {((acc, N):*, (N,1):-):FactRightRec}
  ):. >> return
}
```

In case the argument is greater than one, the function returns the result of processing delayed list, which makes new recursive call after disclosing.

Tail recursion is based on using argument-accumulator. Program would be fully completed with addition of special "start" function, which initiates the computation process:

```
factR << funcdef N {
  (1, N):FactRightRec >> return
}
```

During computations, the program performs a number of recursive calls. After receiving the results in the latter of them it is returned back on the same multi-step way (Fig. 1). The recursion depth is defined by argument value and can cause memory overflow in some cases.

Delayed lists in the Pifagor programming language allow delaying stored operations till the moment of list disclosing, which is defined in accordance with the axioms and rules of the language. The problem of getting rid of nested recursive calls can be considered as a return of the delayed list from the called function to the calling, followed by its disclosing on the upper level. Function below uses this scheme. This function differs from the usual one only in absence of the last empty signal operator:

```
FactRightRecDelay << funcdef pair {
  acc<< pair:1; N<< pair:2;
  // Testing the value of an integer argument
  // (four element list is formed with a single true value)
  [((N,0) : [<,=], (N,1):[=,>]):?]^
  (
```

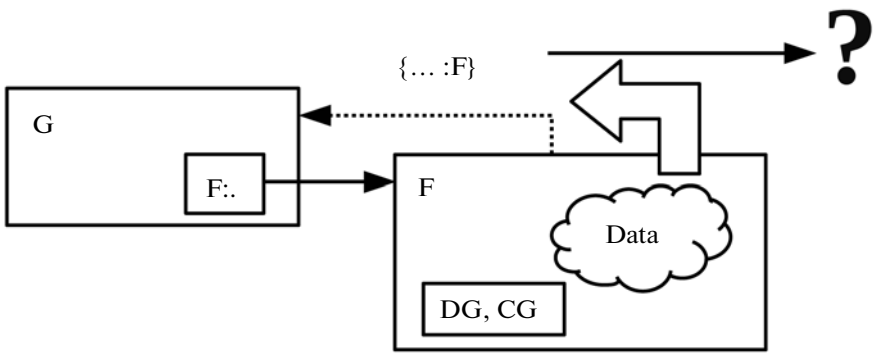

Fig. 2. The return of the delayed list does not release the resources.

```
    // Incorrect negative value
      ("Incorrect negative argument", N),
    // If value equals 0, the factorial equals 1
      acc,
    // If value equals 1, the factorial equals 1
      acc,
    // Else the recursive computation starts
      {((acc,N):*, (N,1):-):FactRightRecDelay}
  ) >> return
}
```

The empty operator is used in the initial function in order to disclose the returned delayed list:

```
factRDelay << funcdef N {
  (1,N) :FactRightRecDelay:. >> return
}
```

The same computations can be used either for head or for tail recursion.

The return of the delayed list to the calling function provides its effective finalization. It allows avoiding nested recursive calls. Logs for 3 factorial computation, which demonstrates this conception, are shown below:

```
(1,3):FactRightRecDelay:.
=> {((acc,N):*, (N,1):-):FactRightRecDelay}:.
=> {((1,3):*, (3,1):-):FactRightRecDelay}:.
=> [((1,3):*, (3,1):-):FactRightRecDelay]:.
=> [(3,2):FactRightRecDelay]:.
=> {((acc,N):*, (N,1):-):FactRightRecDelay}:.
=> {((3,2):*, (2,1):-):FactRightRecDelay}:.
=> [((3,2):*, (2,1):-):FactRightRecDelay]:.
=> [(6,1):FactRightRecDelay]:.
=> 6:.
=> 6
```

The first call of `factRDelay` function continues with the call of `textttFactRightRecDelay`, which returns a delayed list as the result. Disclosing of the delayed list results in a new call of `FactRightRecDelay` with a new argument. The process of entering to the function and returning from it continues till the end of computations.

The implementation of this approach is proven to be hard. The main problem is the necessity to transfer the content of the called function, which is not fully calculated, to the calling function. Such modification leads to an increase in the size of the calling program and the additional overhead for the structure modification to ensure the internal representation of dataflow program. Making all computations in the context of a called function would be much easier (Fig. 2). But this does not solve the problem of getting

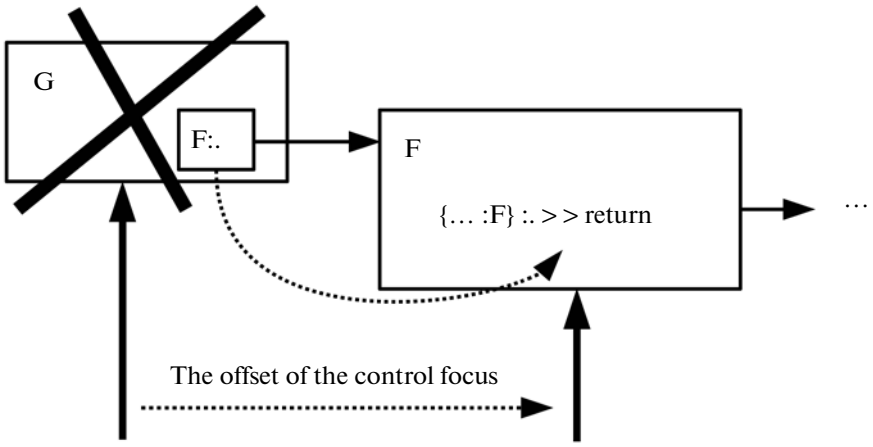

Fig. 3. The dynamical transfer of the list release code to the called function.

rid of recursive calls in whole, since the delayed lists, which were returned, may contain their own recursive calls. Thus the returning of the delayed lists by the straightforward realisation makes impossible the release of the resources occupied by the calling functions, as computations that use these resources are not completed.

It should be noted that the calling function after the point of return of the delayed list contains only an empty (signal) operator, which discloses the delayed list. So instead of moving the delayed list into the calling function, this statement could be moved into the called function. This allows transferring the role of a process which ensures the completion of calculations to the called function. After that, the calling function can be deleted as useless. Each subsequent recursive call will work in a similar way:

- transferring of an empty statement, which discloses the delayed list, into the called function;
- then the return point is moved into the same function;
- after this the calling function is deleted.

This process is schematically represented in the Fig. 3.

The situation considered above enables the effective use of the computational resources. When the recursive call is at the end of the released delayed list, the code is executed before the function `F` return and has the following form:

```
{ … F}: . >>return
```

Instead of making a new function call, it is possible to pass the obtained result as a new argument to the same function, thus iteratively reusing the already allocated resources instead of allocating resources for new function calls.

Transiting a point of the delayed list disclosing from the calling function to the called function leads to the situation when dynamically generated process that control the recursive calls has no difference from the delayed lists in the called function. The program `FactRightRec` can be controlled in the same way, because it based on using tail recursion before returning the result. The analysis of the chain of right recursion calls shows, that in case of functional-dataflow parallel programs they produce result, which does not require nothing but disclosing delayed lists. Thus the following chain is created:

```
… :F:.:.:. … :. >>return
```

This chain is equal to operations, which are executed in case of returning delayed lists. It allows transforming tail recursive calls into the loops in programs, that does not even use delayed list return.

In case of the similar situations, translator can replace `{ … :F}:. >>return` with `{ … :repeat}:. >> return`. When doing this, the generated result is redirected to the input of the same function. In addition to this it is necessary to carry out the deallocation of memory allocated for the intermediate data by the previous iteration, and reset to its initial state the automatons of the control system for functional-dataflow parallel programs execution that are responsible for the analysis of data availability. If the type of the formed result is the same as type of the previous argument, we can avoid deallocation of memory occupied by the intermediate data, and reuse it. This can improve the performance of function execution.

3. USING ASYNCHRONOUS LISTS

Presented examples demonstrate the possibility of converting a functional-dataflow recursive program to an iterative program. However, they did not reveal the particular use of such transformations in case of continuous input data stream that is processed during a long time. Such streams can be processed in Pifagor language using asynchronous lists [4], which allow starting computations under condition of readiness of at least one element of the list. The structure of the programs that use asynchronous lists resembles the structure of the programs with the right recursion. However, they use various additional operators that provide grouping of required data into various lists before return of the computations. Parallel and asynchronous lists, which ensure the interaction of multiple functions in asynchronous mode, are the most commonly used.

The function below shows multiplication of 2 asynchronous lists and can be used as an example of transformation of a recursion into the simpler forms:

```
ScalVecMult<< funcdef A {
  x<< A:1:1;         //An argument incoming from the first list
  y<< A:2:1;         //An argument incoming from the second list
  tail_x<< A:1:-1;  // The tail of the first asynchronous list
  tail_y<< A:2:-1;  // The tail of the second asynchronous list
  v<< (x,y):*;       // First pair multiplication
  // Emptyness test for the tail (at least one list length is equal to zero)
  [(((tail_x:|, tail_y:|):*, 0):[=, !=]):?]^
  (
    v, // if the tail is empty only one pair is multiplied
    {v, (tail_x, tail_y):ScalVecMult}
  ) >>return; // the returned list is not released
}
```

The elements of each produced pair are multiplied, and the product enters in the parallel list, which also can be processed asynchronously, element-by-element.

An argument that are fed to the input has the following format:

```
(asynch(x1, x2, …, xn), asynch(y1, y2, …, yn)),
```

where `x1, x2, …, xn, y1, y2, …, yn` are numbers. Multiplication starts when each list contains at least one element.

The delayed list which is returned by the function is disclosed in the previously described manner, in accordance with the algebra of transformations of the language, and forms a parallel list. It contains within itself nested parallel lists generated by each of the recursive calls. Those lists can't be disclosed automatically until they got into data list or asynchronous list. Therefore, despite the possibility of using the mechanism of the release from the calling functions, the positive effect associated with the deallocation of resources would be negligible. Resources continue to be allocated, and this process at the end may result in a memory overflow.

If a function is supposed to work with others (for example, in vectors multiplication subroutine [2]) then it can be called from another function, which transforms a parallel list into an asynchronous one:

```
A_ScalVecMult << funcdef A {
  asynch(A:ScalVecMult:.) >> return
}
```

In this case, the rules, that define the algebra language, are working. According to them, any parallel lists with any nesting, which are placed within the data lists or asynchronous lists, are disclosed and their data extend the one-dimensional array of data of corresponding lists. Since the order of the elements in an asynchronous list is determined only by the moment of their appearance, the presented mechanism for the list disclosing is implemented at runtime without the additional overhead.

Disclosure of parallel lists that is executed directly during the calculations allow us to complete the current calling function already at the moment when the first element of the list is becoming available, and to transfer control to the calling function with a reference to the list by analogy with the previously considered scheme. It provides an exemption of the allocated resources and allows to use exactly the same approach in the case of tail recursive calls. Logs of scalar multiplication of two asynchronous lists are shown below:

```
asynch(((1,2),(4,5)):ScalVecMult:.) =>
asynch({v,(tail_x, tail_y):AD_ScalVecMult}:.) =>
asynch([v,(tail_x, tail_y):AD_ScalVecMult]:.) =>
asynch([4:., ((2),(5)):AD_ScalVecMult:.]) =>
asynch([4:., ((2),(5)):AD_ScalVecMult:.]) =>
asynch(4, 10:.) =>
asynch(4, 10)
```

The use of asynchronous lists, combined with the ability to manage computing resources effectively by getting rid of unnecessary recursive calls, provides the support for long computations when multiple functions communicate via asynchronous lists. Let us consider the vector product as an example. It is based on summation of elements of asynchronous lists made by `A_ScalVecMult`. The function code is shown below:

```
VecSum << funcdef A {
  x1<< A:1;       // An incomming data element
  tail_1<< A:-1;  // A tail of the asynchronous list
  // Testing the list tail to be empty
  [((tail_1:1, 0):[=, !=]):?]^
  (
     x1, // The list has only one element
     {   // The extraction of the second list argument and summation
        block {
           x2<< tail_1: 1;      // the second argument
           s<< (x1,x2):+;       // the sum of two incoming arguments
           tail_2<< tail_1:-1;  // tail to tail
           // The recursion over the rest elements
           [((tail_2: |,0): [=, !=]):?]^
           (
             s,
             { asynch(tail_2:[], s):VecSum }
        ): . >>break
      }
    }
  ) >>return;
}
```

The delayed list, returned from this function, uses tail recursion for summation. Therefore the use of the previously discussed schemes of exemption from the recursive calls does not cause any problems. This function can work in parallel with scalar vector multiplication without any extra transformation:

```
A_VecMult << funcdef A {
  A:A_ScalVecMult:VecSum:. >> return
}
```

4. CONCLUSIONS

Asynchronous lists in conjunction with the tail recursion allow creating of functional-dataflow parallel programs that can be used in long recurrence calculation, despite the usage of recursive calls. Emerging effects allow automatically organize pipelining between independent functions and led to development of a new class of algorithms with dynamic resource management, which, under certain conditions, can be effectively static-scheduled. This allows using of functional dataflow programming paradigm not only for the development of computer programs, but also for the creation of software systems characterized by repetitive calculations over long periods of time.

ACKNOWLEDGMENTS

This work was supported by the Federal Targeted Program "Scientific and Scientific-Pedagogical Personnel of Innovative Russia" under the Grant no. 14.A18.21.0396.

REFERENCES

1. Legalov, A.I., The functional programming language for creating architecture-independent parallel programs, *Computational Technologies*, vol. 10, no 1, pp. 71–89; Novosibirsk: Institute of Computational Technologies SB HAS, 2005.
2. Udalova, U.V., Legalov, A.I., and Sirotinina, N.U., Debug and verification of function-stream parallel programs, *Journal of Siberian Federal University. Engineering and Technologies*, 2011, vol. 4, no. 2, pp. 213–224.
3. Redkin, A.V. and Leglaov A.I., Event-based control of computations for functional-dataflow programming, *Scientific Bulletin of Novosibirsk State Technical University*, 2008, vol. 32, no. 3, pp. 111–120.
4. Leglaov, A.I. and Redkin, A.V., Expansion of asynchronous control using data readiness, *The Third International Conference on Parallel Computations and Control Problems (PACO'2006), ISBN 5-201-14990-1*, 2006, pp. 1272–1281.