*Модел. и анализ информ. систем.* Т. **19**, № **4** (2012) 48–58

УДК 004.4'416+004.432.42

# Преобразование хвостовых рекурсий в функционально-потоковых параллельных программах

Легалов А.И., Непомнящий О.В., Матковский И.В., Кропачева М.С. [1]

*Сибирский федеральный университет*

*e-mail: legalov@mail.ru*

*получена 22 апреля 2012*

**Ключевые слова:** функционально-потоковое параллельное программирование, хвостовая рекурсия, преобразование программ, язык программирования Пифагор

Анализируются особенности преобразования функционально-потоковых параллельных программ в программы, использующие ограниченные вычислительные ресурсы. Рассматривается, каким образом влияют на эти преобразования: использование асинхронных списков, возврат задержанных списков, изменение темпа поступления данных по отношению ко времени их обработки. Применение подобных преобразований позволяет порождать различные программы со статическим параллелизмом, используя в качестве основы одну и ту же функционально-потоковую параллельную программу.

## 1. Введение

Разработка параллельных программ в настоящее время характеризуется разнообразием подходов и методов, что во многом связано с ориентацией на различные архитектуры параллельных вычислительных систем (ПВС). Существующие попытки создания языков архитектурно-независимого параллельного программирования пока не принесли весомых результатов. Это обусловливается многими проблемами, которые в основном выливаются в трудности процессов эффективного преобразования архитектурно-независимых параллельных программ в коды реально используемых ПВС.

Одним из языков, ориентированным на архитектурно-независимую разработку программ, является Пифагор [1]. К его специфическим особенностям относятся: ориентация на вычислительные системы с неограниченными ресурсами, управление по готовности данных, параллелизм на уровне базовых операций. Отсутствие

[1]Работа поддержана грантом в рамках ФЦП «Научные и научно-педагогические кадры инновационной России» № 14.A18.21.0396.

процессов, взаимодействующих через общие ресурсы, облегчает отладку и верификацию [2]. Выполнение функционально-потоковых программ обеспечивается событийным процессором [3].

Основным методом представления повторяющихся вычислений в данном языке, как и во всех функциональных языках программирования, является рекурсия. Ее использование позволяет избавиться от ресурсных ограничений обычных циклов, обусловленных возможным повторным использованием уже выделенных ресурсов, которые в потоковых программах могут быть к этому моменту еще заняты. Однако ориентация на рекурсивные вычисления не позволяет использовать разрабатываемые программы в ряде реальных случаев по нескольким причинам. Одной из часто встречающихся ситуаций является использование длительных («бесконечных») повторяющихся вычислений. Для функционально-потокового языка это ведет к постоянным рекурсивным вызовам, порождающим переполнение памяти. Данная ситуация становится еще более критичной, если программу планируется использовать в режиме реального времени, обеспечивая циклическую обработку асинхронно поступающего потока данных.

Вместе с тем, рассматриваемая проблема решена как для функциональных, так и императивных языков программирования, ориентированных на последовательные вычисления. Однако использование управления по готовности данных и параллелизм на уровне отдельных операций вносят свои специфические особенности. Поэтому поиск путей, поддерживающих «бесконечное» выполнение функционально-потоковых параллельных программ в ограниченных вычислительных ресурсах, является актуальной задачей.

В работе рассматриваются подходы, обеспечивающие преобразование функционально-потоковых параллельных программ, написанных на языке Пифагор, в представления, позволяющие выполнять эти программы на ограниченных вычислительных ресурсах.

## 2. Использование задержанных списков

Одним из способов решения описанной проблемы, который возможен при написании программ на рассматриваемом языке, является возврат из вызываемой функции не результата вычислений, производимого в конце цепочки рекурсивного вызова, а задержанного списка с его раскрытием и выполнением требуемых вычислений в вызывающей функции. Данный подход возможен при использовании в программах правой рекурсии, расположенной в конце функции перед возвратом результата. Такие рекурсии легко преобразуются в итерации и в последовательных языках программирования.

В качестве примера рассмотрим вычисление факториала. Ниже представлена обычная функция, использующая правую рекурсию и накопление результата:

```
FactRightRec << funcdef pair {
  acc<< pair:1; N<< pair:2;
  // Проверка значения целочисленного аргумента
  // (формируется четырехэлементный список с одним истинным значением)
```

```
  [((N,0):[<,=], (N,1):[=,>]):?]^
  (
    // Некорректное отрицательное значение
      ("Incorrect negative argument", N),
    // При нуле факториал равен 1
      acc,
    // При единице - то же самое
      acc,
    // Иначе - рекурсивное вычисление
      {((acc,N):*, (N,1):-):FactRightRec}
  ):. >> return
}
```

В том случае, если аргумент больше единицы, осуществляется выбор из списка альтернатив задержанного списка, который раскрывается, порождая перед возвратом из функции очередной рекурсивный вызов.

Использование правой рекурсии требует передачу в представленную функцию аргумента-аккумулятора. Для придания программе законченного вида с одним аргументом необходима еще одна функция, запускающая вычисления:

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

В ходе вычислений программа выполняет определенное число рекурсивных вызовов. После получения результата в последнем из них осуществляется его возврат по этой же многоступенчатой схеме обратно (рис. 1). Глубина рекурсии определяется значением аргумента и при решении многих аналогичных задач может привести к переполнению памяти.

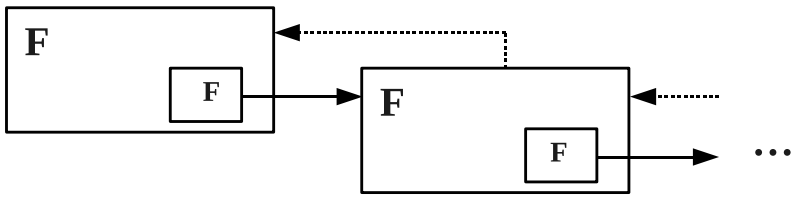


Рис. 1. Сопровождение рекурсивных вызовов захватом ресурсов

Вместе с тем, в языке программирования Пифагор существуют задержанные списки, которые откладывают вычисления размещенных в них операторов до момента раскрытия, определяемого в соответствии с аксиомами и правилами языка. Поэтому концептуально проблемы избавления от вложенных рекурсивных вызовов можно рассматривать как возврат задержанного списка из вызываемой функции в вызывающую с последующим его раскрытием на верхнем уровне. Реализация

функции, осуществляющей возврат задержанного списка, отличается от обычной только отсутствием пустой операции (раскрывающей задержанные списки) перед возвратом из основной функции:

```
FactRightRecDelay << funcdef pair {
  acc<< pair:1; N<< pair:2;
  // Проверка значения целочисленного аргумента
  // (формируется четырехэлементный список с одним истинным значением)
  [((N,0):[<,=], (N,1):[=,>]):?]^
  (
    // Некорректное отрицательное значение
      ("Incorrect negative argument", N),
    // При нуле факториал равен 1
      acc,
    // При единице - то же самое
      acc,
    // Иначе - рекурсивное вычисление
      {((acc,N):*, (N,1):-):FactRightRecDelay}
  ) >> return
}
```

Для того чтобы происходило раскрытие возвращаемого задержанного списка, этот пустой оператор используется в стартовой функции:

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

Аналогичные вычисления можно организовать как для левой, так и для правой рекурсии.

Возвращаемый в вызывающую функцию задержанный список концептуально обеспечивает выход из вызываемой функции, тем самым позволяя избавиться от вложенных рекурсивных вызовов. Ниже представлена демонстрация этого процесса протоколом вычисления факториала для аргумента, равного 3:

```
(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
```

При первоначальном запуске функции `factRDelay` осуществляется обращение к `FactRightRecDelay`, которая возвращает обратно задержанный список. После раскрытия задержанного списка и выполнения подготовительных вычислений осуществляется повторный вызов функции `FactRightRecDelay`, но уже с измененными параметрами. Процесс входа в вызывающую функцию на один уровень и возврат назад происходит до получения окончательного значения.

Однако концептуальное решение задачи уменьшения уровня вложенности рекурсии не позволяет реализовать ее непосредственно. Основной проблемой является необходимость переноса в вызывающую функцию до конца не вычисленного содержимого вызываемой функции. Подобная модификация ведет к расширению размера вызывающей функции и связана с дополнительными накладными расходами на модификацию имеющихся структур, обеспечивающих внутреннее представление функционально-потоковой программы. Более простым являлось бы выполнение всех задержанных операций в контексте вызываемой функции (рис. 2). Однако в этом случае проблема избавления от рекурсивных вызовов не была бы решена, так как в возвращаемых задержанных списках могут встречаться свои рекурсивные вызовы, которые также ведут к возврату вычислений, еще не законченных в вызываемых функциях. Таким образом, прямая реализация возврата задержанных списков ведет к тому, что удаление ресурсов, занимаемых вызываемыми функциями, становится невозможным из-за того, что проводимые в них вычисления еще не завершились.

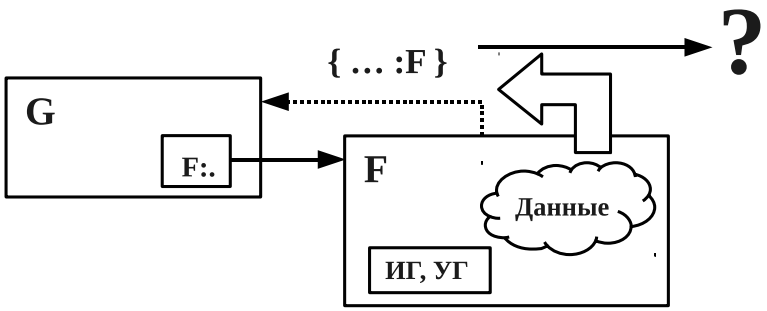


Рис. 2. Возврат задержанных списков не освобождает предыдущие ресурсы

Вместе с тем следует отметить, что в вызывающей функции после точки возврата задержанного списка находится только пустой (сигнальный) оператор, раскрывающий задержанный список. Поэтому, вместо переноса задержанного списка в вызывающую функцию, можно поступить наоборот: перенести этот оператор в вызываемую функцию. Его перенос позволяет передать вызываемой функции также и роль процесса, обеспечивающего завершение вычислений. После этого вызывающую функцию можно уничтожить за ненадобностью. Каждый последующий рекурсивный вызов будет работать по аналогичной схеме:

- осуществляется перенос пустого оператора, раскрывающего задержанный список, в вызываемую функцию;

- затем на эту же функцию переносится фокус возврата;
- после этого вызывающая функция уничтожается.

Условное изображение описанного процесса представлено на рис. 3.

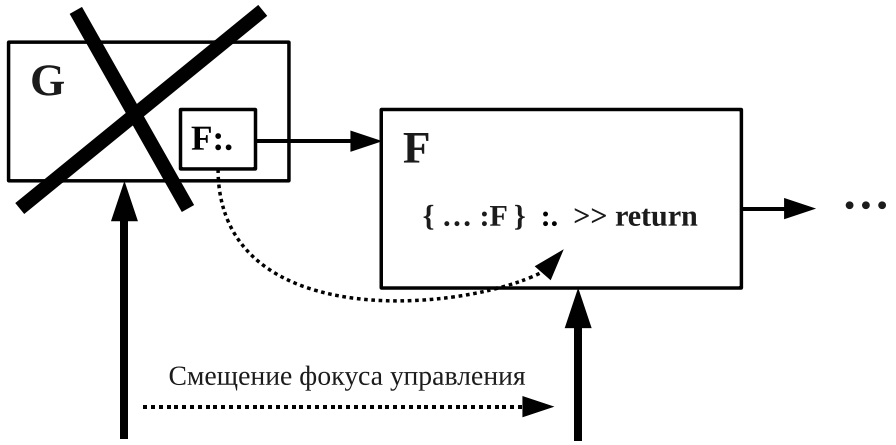


Рис. 3. Динамический перенос кода раскрытия списка в вызываемую функцию

Рассмотренная ситуация обеспечивает эффективное использование вычислительных ресурсов. В тех случаях, когда рекурсивный вызов стоит в конце раскрываемого задержанного списка, то есть тогда, когда код, выполняемый перед возвратом из функции `F`, имеет вид:

```
{ ... F}:. >>return,
```

можно даже не создавать вновь вызываемую функцию, а передать формируемый результат в качестве нового аргумента этой же функции, тем самым заменив процесс порождения ресурсов для новых функций на итеративное использование уже занятых ресурсов.

Возникающая ситуация, связанная с передачей точки раскрытия задержанных списков из вызывающей функции в вызываемую, приводит к тому, что динамически порождаемый процесс управления рекурсивными вызовами ничем не отличается от раскрытия задержанных списков внутри вызываемой функции. Поэтому программа `FactRightRec` может управляться аналогичным способом, так как основной ее особенностью является то, что перед возвратом результата осуществляется следующий правый рекурсивный вызов. Анализ цепочки правых рекурсивных вызовов показывает, что они, в случае функционально-потоковых параллельных программ, формируют возврат результата, перед которым не осуществляется никаких дополнительных действий, кроме раскрытия задержанных списков. То есть формируется цепочка:

```
... :F:.:.:. ... :. >>return,
```

которая по своей сути эквивалентна тем операциям, которые выполняются и при возврате задержанных списков. Это позволяет аналогичным способом реализовать преобразование хвостовых рекурсивных вызовов в циклы и для программ, в которых возврат задержанных списков не происходит.

При нахождении подобных ситуаций транслятор может осуществить соответствующую модификацию кода, заменив цепочку `{ ... :F}:. >>return` на `{ ... :repeat}:. >>return`. При достижении данной операции формируемый результат будет перенаправляться на вход этой же функции. Помимо этого необходимо осуществить освобождение памяти, выделенной для промежуточных данных в результате выполнения предыдущей итерации и сбросить в первоначальное состояние автоматы системы управления выполнением функционально-потоковых параллельных программ, отвечающие за анализ готовности данных. В тех ситуациях, когда формируемый результат совпадает с типом предыдущего аргумента, память, занимаемую промежуточными данными, можно не освобождать, а использовать повторно. Это позволяет повысить эффективность выполнения функций.

## 3. Применение асинхронных списков

Представленные примеры демонстрируют возможность преобразования функционально-потоковых рекурсивных программ в итерационные. Однако они не раскрывают особенности использования этих преобразований в условиях, когда поступающие данные образуют непрерывный входной поток, обрабатываемый в течение длительного времени. Для обработки таких данных в языке используются асинхронные списки [4], позволяющие переходить к выполнению вычислений по готовности хотя бы одного элемента. Программы, использующие асинхронные списки, по своей структуре напоминают программы с правой рекурсией. Однако в них, перед возвратом вычислений, используются различные дополнительные операторы, обеспечивающие требуемые группировки данных в различные списки. Чаще всего это параллельные и асинхронные списки, позволяющие задать взаимодействие нескольких функций в асинхронном режиме.

В качестве примера функции, допускающей преобразование рекурсии в более простые формы, приведена функция, попарно перемножающая элементы двух асинхронных списков:

```
ScalVecMult<< funcdef A {
  x<< A:1:1;          // Поступивший аргумент из первого списка
  y<< A:2:1;          // Поступивший аргумент из второго списка
  tail_x<< A:1:-1;    // Хвост первого асинхронного списка
  tail_y<< A:2:-1;    // Хвост второго асинхронного списка
  v<< (x,y):*;        // Перемножение первой пары
  // Проверка на пустоту остатка (длина хотя бы одного списка =0)
  [(((tail_x:|,tail_y:|):*, 0):[=, !=]):?]^
  (
     v, //  хвост пуст - только одна пара перемножается
     {v, (tail_x, tail_y):ScalVecMult}
```

```
  )  >>return; // возвращаемый список не раскрывается
}
```

Поступающие данные образуют пары, которые перемножаются между собой и выдаются в качестве элементов результирующего параллельного списка, который тоже допускает асинхронную поэлементную обработку. Поступающий на вход аргумент имеет следующий формат:

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

где x1, x2, ..., xn, y1, y2, ..., yn — числа. То есть, при наличии хотя бы по одному числу в каждом из списков, осуществляется их перемножение.

Возвращаемый из функции задержанный список раскрывается ранее описанным способом, образуя, в соответствии с алгеброй преобразований языка, параллельный список. Он содержит внутри себя вложенные параллельные списки, формируемые каждым из рекурсивных вызовов. Такие списки автоматически не раскрываются и остаются вложенными друг в друга до того момента, пока они не окажутся внутри списка данных или асинхронного списка. Поэтому, несмотря на возможность использования механизма освобождения от вызывающих функций, достигаемый эффект, связанный с освобождением ресурсов, будет незначительным. Они будут продолжать выделяться, что, в конце концов, может привести к переполнению памяти.

Вместе с тем, если данная функция планируется для использования в сочетании с другими, например, для перемножения двух векторов с использованием суммирования элементов асинхронного списка [2], она может быть вызвана из функции, преобразующей параллельный список в асинхронный:

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

В этом случае начинают срабатывать правила, определяющие алгебру языка, в соответствии с которыми любые параллельные списки любой вложенности, располагаемые внутри списков данных или асинхронных списков, раскрываются, а их данные расширяют одномерный массив данных соответствующих списков. Так как для асинхронных списков порядок следования элементов определяется только моментом их появления, то представленный механизм раскрытия реализуется во время выполнения без дополнительных затрат.

Раскрытие параллельных списков непосредственно в ходе проводимых вычислений позволяет полностью завершить текущую вызывающую функцию уже при первом появившемся элементе асинхронного списка и передать управление вызывающей функции вместе со ссылкой на этот список по аналогии с ранее рассмотренной схемой. Это обеспечивает освобождение занимаемых ресурсов и позволяет использовать точно такой же подход при наличии хвостовых рекурсивных вызовов. Протокол скалярного перемножения двух асинхронных списков:

```
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)
```

Применение асинхронных списков в сочетании с возможностью эффективного управления вычислительными ресурсами за счет избавления от лишних рекурсивных вызовов обеспечивает поддержку длительных вычислений при взаимодействии нескольких функций через асинхронные списки. В качестве примера можно привести векторное произведение, которое обеспечивается за счет суммирования элементов асинхронного списка, полученных в функции `A_ScalVecMult`. Функция суммирования элементов асинхронного списка:

```
VecSum << funcdef A {
  x1<< A:1;       // Поступивший элемент данных
  tail_1<< A:-1;  // Хвост асинхронного списка
  // Проверка на пустоту остатка списка
  [((tail_1:|, 0):[=, !=]):?]^
  (
     x1, // В списке, только один элемент, определяющий сумму
     {   // Выделение второго аргумента с последующим суммированием
        block {
           x2<< tail_1:1;      // второй аргумент
           s<< (x1,x2):+;      // сумма двух поступивших элементов
           tail_2<< tail_1:-1; // "хвост" от "хвоста"
           // Рекурсивная обработка оставшихся элементов
           [((tail_2:|, 0):[=, !=]):?]^
           (
              s,
              { asynch(tail_2:[], s):VecSum }
           ):. >>break
        }
     }
  ) >>return;
}
```

Возвращаемый из данной функции задержанный список имеет хвостовой рекурсивный вызов функции суммирования. Поэтому использование ранее рассмотренной схемы освобождения от рекурсивных вызовов не вызывает никаких проблем. Объединение данной функции в общую программу со скалярным перемножением двух векторов позволяет получить конвейер из двух одновременно выполняемых параллельных функций без каких-либо дополнительных построений:

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

# 4. Заключение

Асинхронные списки в сочетании с правой рекурсией дают возможность создавать функционально-потоковые параллельные программы, которые могут использоваться в длительных рекуррентных вычислениях, несмотря на описание процессов с применением рекурсивных вызовов. Возникающие при этом эффекты позволяют автоматически организовывать конвейеризацию между независимыми функциями и ведут к построению нового класса алгоритмов с динамическим управлением ресурсами, которые, при определенных условиях их выполнения, допускают эффективное статическое планирование. Это позволяет использовать функционально-потоковую парадигму программирования не только для разработки вычислительных программ, но и при создании программного обеспечения систем, характеризующихся повторяющимися вычислениями в течение длительных периодов времени.

# Список литературы

1. Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ // Вычислительные технологии. 2005. № 1 (10). С. 71–89.

2. Удалова Ю.В., Легалов А.И., Сиротинина Н.Ю. Методы отладки и верификации функционально-потоковых параллельных программ // Журнал Сибирского федерального университета. Серия «Техника и технологии». 2011. Том 4, №2. С. 213–224.

3. Редькин А.В., Легалов А.И. Событийное управление выполнением функционально-потоковых параллельных программ // Научный вестник НГТУ. 2008. №3 (32). С. 111–120.

4. Легалов А.И., Редькин А.В. Расширение асинхронного управления по готовности данных // Труды III Международной конференции «Параллельные вычисления и задачи управления» PACO'2006. М.: Институт проблем управления им. В.А. Трапезникова РАН, 2006. С. 1272–1281. (Электронное издание)

# Tail Recursion Transformation in Functional Dataflow Parallel Programs

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

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

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 some functional dataflow parallel program.

**Сведения об авторах:**
**Легалов Александр Иванович,**
Сибирский федеральный университет, профессор.
**Непомнящий Олег Владимирович,**
Сибирский федеральный университет, профессор.
**Матковский Иван Васильевич,**
Сибирский федеральный университет, аспирант.
**Кропачева Мария Сергеевна,**
Сибирский федеральный университет, аспирант.