MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 2, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

The System for Transforming the Code of Dataflow Programs into Imperative

V. S. Vasilev[1], A. I. Legalov[2], S. V. Zykov[2]

DOI: 10.18255/1818-1015-2021-2-198-214

[1]Siberian Federal University, 79 Svobodny Ave., Krasnoyarsk 660041, Russia.
[2]Higher school of Economics, National research University, 20 Myasnitskaya str., Moscow 101000, Russia.

MSC2020: 68N15, 68Q10
Research article
Full text in Russian

Received May 7, 2021
After revision May 31, 2021
Accepted June 2, 2021

Functional dataflow programming languages are designed to create parallel portable programs. The source code of such programs is translated into a set of graphs that reflect information and control dependencies. The main way of their execution is interpretation, which does not allow to perform calculations efficiently on real parallel computing systems and leads to poor performance. To run programs directly on existing computing systems, you need to use specific optimization and transformation methods that take into account the features of both the programming language and the architecture of the system. Currently, the most common is the Von Neumann architecture, however, parallel programming for it in most cases is carried out using imperative languages with a static type system. For different architectures of parallel computing systems, there are various approaches to writing parallel programs. The transformation of dataflow parallel programs into imperative programs allows to form a framework of imperative code fragments that directly display sequential calculations. In the future, this framework can be adapted to a specific parallel architecture. The paper considers an approach to performing this type of transformation, which consists in allocating fragments of dataflow parallel programs as templates, which are subsequently replaced by equivalent fragments of imperative languages. The proposed transformation methods allow generating program code, to which various optimizing transformations can be applied in the future, including parallelization taking into account the target architecture.

Keywords: transformation of programs; dataflow parallel programming; program analysis; typing; intermediate program representations

INFORMATION ABOUT THE AUTHORS

Vladimir S. Vasilev correspondence author	orcid.org/0000-0002-3340-6678. E-mail: vsvasilev@sfu-kras.ru Senior lecturer.
Alexander I. Legalov	orcid.org/0000-0002-5487-0699. E-mail: alegalov@hse.ru Doctor of Science, Professor.
Sergey V. Zykov	orcid.org/0000-0002-2115-5461. E-mail: szykov@hse.ru Doctor of Science, Professor.

For citation: V. S. Vasilev, A. I. Legalov, and S. V. Zykov, "The System for Transforming the Code of Dataflow Programs into Imperative", *Modeling and analysis of information systems*, vol. 28, no. 2, pp. 198-214, 2021.

© Vasilev V. S., Legalov A. I., Zykov S. V., 2021
This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/).

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 2, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Трансформация функционально-потоковых параллельных программ в императивные

В. С. Васильев[1], А. И. Легалов[2], С. В. Зыков[2]

DOI: 10.18255/1818-1015-2021-2-198-214

[1]Сибирский федеральный университет, пр. Свободный, д. 82, г. Красноярск, 660041 Россия.
[2]Национальный исследовательский университет «Высшая школа экономики», ул. Мясницкая, д. 20, г. Москва, 101000 Россия.

УДК 004.4'42
Научная статья
Полный текст на русском языке

Получена 7 мая 2021 г.
После доработки 31 мая 2021 г.
Принята к публикации 2 июня 2021 г.

Функционально-потоковая парадигма параллельного программирования ориентирована на разработку параллельных переносимых программ. Исходный код функционально-потоковых программ транслируется в набор графов, отражающих информационные и управляющие зависимости. Основным способом их исполнения является интерпретация, что не позволяет эффективно выполнять вычисления на реальных параллельных вычислительных системах и ведет к низкой производительности. Для непосредственного выполнения программ на существующих вычислительных системах требуется использование специфических методов оптимизации и трансформации, учитывающих особенности как языка программирования, так и архитектуры исполнителя. В настоящее время наиболее распространенной является архитектура Фон-Неймана, параллельное программирование для которой в большинстве случаев осуществляется с использованием языков, поддерживающих императивный стиль и ориентированных на статическую систему типов. Для различных архитектур параллельных вычислительных систем существуют разнообразные подходы к написанию параллельных программ. Трансформация функционально-потоковых параллельных программ в императивные позволяет сформировать общий каркас из фрагментов императивного кода, непосредственно отображающих последовательные вычисления, который в дальнейшем может быть адаптирован к конкретной параллельной архитектуре. В работе рассматривается подход к выполнению такого типа трансформации, заключающийся в выделении фрагментов функционально-потоковых параллельных программ в качестве шаблонов, заменяемых впоследствии на эквивалентные фрагменты императивных языков. Предлагаемые методы трансформации позволяют порождать программный код, к которому в дальнейшем можно применять различные оптимизирующие преобразования, включая распараллеливание с учетом целевой архитектуры.

Ключевые слова: трансформация программ; функционально-потоковое параллельное программирование; анализ программ; типизация; промежуточные представления программ

ИНФОРМАЦИЯ ОБ АВТОРАХ

Владимир Сергеевич Васильев
автор для корреспонденции
orcid.org/0000-0002-3340-6678. E-mail: vsvasilev@sfu-kras.ru
старший преподаватель.

Александр Иванович Легалов
orcid.org/0000-0002-5487-0699. E-mail: alegalov@hse.ru
доктор технических наук, профессор.

Сергей Викторович Зыков
orcid.org/0000-0002-2115-5461. E-mail: szykov@hse.ru
доктор технических наук, профессор.

Для цитирования: V. S. Vasilev, A. I. Legalov, and S. V. Zykov, "The System for Transforming the Code of Dataflow Programs into Imperative", *Modeling and analysis of information systems*, vol. 28, no. 2, pp. 198-214, 2021.

© Васильев В. С., Легалов А. И., Зыков С. В., 2021
Эта статья открытого доступа под лицензией CC BY license (https://creativecommons.org/licenses/by/4.0/).

Введение

Основным направлением в параллельном программировании является написание кода, ориентированного на целевую архитектуру независимо от того, насколько эффективно система программирования отражает специфику предметной области. В большинстве случаев это связано со стремлением получить эффективный код в ущерб эффективности и надежности процесса разработки. Постоянное совершенствование и изменение архитектур параллельных вычислительных систем (ПВС) ведет к тому, что ранее написанные параллельные программы приходится перерабатывать, зачастую достаточно сильно, для адаптации к новым условиям эксплуатации. За относительно короткий срок сменилось несколько разновидностей архитектур высокопроизводительных систем. В настоящее время встречаются различные комбинированные решения, в которых, зачастую одновременно, используются механизм передачи сообщений, многопоточность, графические ускорители, решения на уровне систем на кристалле [1].

Разнообразие ПВС ведет к поиску подходов, которые могли бы поддержать архитектурно-независимую разработку параллельных программ и обеспечить их эффективную трансформацию нужную в целевую архитектуру. Можно выделить ряд таких подходов, ориентированных на исходное создание программ для абстрактных параллельных вычислителей и последующую трансформацию в реальные архитектуры. Концепция ресурсно-независимого параллельного программирования реализована в языке COLAMO, ориентированного на разработку систем на кристалле [2, 3]. Создание универсальных языков, напрямую не связанных с архитектурными ограничениями, можно проследить на примере функциональных языков параллельного программирования Sisal [4] и Пифагор [5]. В языке Set@l программа представляет собой описания алгоритма в виде архитектурно-независимого информационного и набора архитектурно-зависимых аспектов, что позволяет переносить код между различными параллельными архитектурами без изменения алгоритма [6].

В языке функционально-потокового параллельного программирования Пифагор реализована концепция единственного использования вычислительных ресурсов, что позволяет формировать параллельные программы, ориентированные на архитектурную независимость. Для анализа возможностей языка разработаны инструментальные средства, поддерживающие процесс создания, преобразования и выполнения функционально-потоковых параллельных программ [7]. Показано, что написанные программы могут быть оптимизированы [8], отлажены [9] и использованы для формальной верификации [10] программ еще до того, как они будут выполняться на конкретной вычислительной системе.

Несмотря на многоэтапный процесс компиляции и формирование промежуточных представлений, позволяющих порождать выходное представление на любом императивном языке, на текущий момент была реализована только интерпретация, базирующаяся на этих промежуточных представлениях. В связи с этим актуальной является задача трансформации программы в выходные императивные представления, что позволяет выполнять код, после дополнительных компиляций, на современных вычислительных архитектурах без дополнительных накладных расходов. К таким императивным языкам относятся, например, C, C++, Фортран.

Трансформация функционально-потоковых параллельных (ФПП) программ в языки со статической типизацией затруднена из-за того, что в языке Пифагор используется динамическая типизация данных. Однако можно отметить, что существует ряд общих принципов, связанных с процессом трансформации, которые напрямую не связаны с системой типов и могут быть реализованы независимо от нее. Можно также отметить императивные языки, в которых реализована динамическая типизация данных, например, Python. И хотя Python не используется в высокопроизводительных параллельных вычислениях, существуют предметные области, в которых на нем разрабатываются многопоточные параллельные программы. Поэтому трансформация функционально-потоковых параллельных программ в императивные может представлять интерес и для таких языков.

Вместе с тем для функциональных языков программирования, к которым относится и язык Пифагор, проработана концепция вывода типов [11] из поступающих на вход аргументов, что, в принципе, позволяет применить данную концепцию для трансформации ФПП программ. Наряду с этим также можно отметить, что на базе функционально-потоковой парадигмы параллельного программирования предложен статически типизированный язык Smile [12, 13], при разработке компилятора которого планируется использовать рассматриваемые решения. Другим вариантом использования, также связанного с ФПП программированием, является введение явной типизации в реверсивный информационный граф, формируемый в результате компиляции программ, написанных на Пифагоре. Этот подход предлагается в работе.

1. Особенности трансформации функционально-потоковых параллельных программ

К особенностям языка ФПП программирования Пифагор, определяющим специфику трансформации в императивный код следует отнести:

1) Использование динамической типизации, при которой одна и та же функция может получать в качестве аргумента данные различного типа, что порождает результат, тип которого также может быт различным.
2) В языке имеются специфические операторы, отсутствующие в императивных языках, например: задержанный список, параллельный список. Они используются специфическим образом, что предопределяет особый подход к их анализу и трансформации.
3) В ходе выполнения, вычисления некоторых ветвей алгоритма могут приводить к ошибкам, которые не влияют на работу алгоритма если их результаты не используются.
4) Используется управление вычислениями по готовности данных.

В связи с этим трансформация исходных программ в статически типизированные программы для повышения эффективности связана с наложением определенных ограничений.

Трансформация программ осуществляется на основе анализа реверсивного информационного графа (РИГ), являющего результатом компиляции исходных текстов программы на языке Пифагор [7]. Данный граф сохраняет всю информацию и при этом отображает ее в форме, удобной для дальнейшего анализа. В ходе анализа можно выделить следующие основные фазы, ориентированные на анализ и трансформацию различных комбинаций программных объектов:

- списков данных, обеспечивающих структурирование;
- предопределенных функций общего вида;
- специализированных предопределенных функций;
- функций, разработанных пользователем;
- параллельных списков, определяющих массовые вычисления;
- задержанных списков, ориентированных на альтернативное выполнение операторов.

При анализе РИГ и формировании корректной последовательности операций императивной программы учитывается зависимость по данным. Помимо этого, для формирования различных конструкций императивной программы, узлы РИГ формируются в группы (шаблоны), образующие конструкции, необходимые для преобразования. Эти шаблоны в тексте поясняются соответствующими эквивалентными фрагментами кода на языке Пифагор. Порождаемые в ходе трансформации императивные конструкции иллюстрируются на языке программирования C++, который используется для формирования выходного представления.

2. Использование статической типизации для трансформации функционально-потоковых параллельных программ

Основной областью применения параллельного программирования являются высокопроизводительные вычисления. Применяемые при этом языки программирования являются статически

типизированными, что позволяет избавиться от динамической проверки типов данных во время вычислений и повышает производительность параллельных программ. Исходя из этого основной акцент в работе сделан на трансформацию в императивные языки, поддерживающие статическую типизацию. Однако следует отметить, что в языке ФПП программирования Пифагор используется динамическая типизация данных. В следующем варианте языка ФПП программирования Smile [12, 13] применяется статическая типизация, облегчающая процесс трансформации. Переход в данном языке к статической типизации ведет также к изменению аксиоматики языка. В большинстве случаев это связано с упрощением семантики операторов и предопределенных функций, что обуславливается стремлением повысить эффективность процесса трансформации. Однако для данного языка еще не разработан комплект инструментальных средств, обеспечивающих компиляцию исходных программ в промежуточное представление, позволяющее проводить дальнейший анализ и компиляцию. Поэтому для отработки методов трансформации принято решение использовать реверсивный информационный граф, порождаемый компилятором языка Пифагор с последующим явным добавлением к нему информации о типах данных. Подобные решения уже использовались при трансформации ФПП программ в программируемые логически интегральные схемы (ПЛИС) [14, 15], а также при разработке методов формальной верификации [16]. Планируется, что полученные результаты будут учтены при генерации промежуточного представления компилятором языка Smile. Для этого анализу и дальнейшему преобразованию в императивный код подвергаются не все конструкции языка Пифагор. В ряде случаев на преобразуемые конструкции накладываются дополнительные ограничения, что позволяет выстроить более эффективные методы трансформации.

Тип результата для каждого узла РИГ, определяющего оператор, может быть вычислен если известен тип входного аргумента функции, а также типы результатов, возвращаемых всеми используемыми внешними функциями. Исходя из этого, сделав соответствующую начальную разметку этих узлов, можно полностью сформировать статически типизированное представление РИГ. Результатом работы компилятора ФПП программ, является набор РИГ. Для добавления статической типизации предложено ввести дополнительное описание, связанное с каждой вершиной, в следующем формате:

```
<Тип функции> ::= <Тип аргумента> -> <Тип возвращаемого значения>.
<Тип кортежа> ::= @{<Элементы кортежа>}
<Элементы кортежа> ::= <пусто> | <Тип> | <Тип> [,<Элементы кортежа>]
<Тип> ::= @char | @int | @double | <Тип списка> | <Тип функции>
<Тип списка> ::= @(<Тип>) | @[<Тип>]
<Тип возвращаемого значения> ::= <Тип>
```

Функция на языке Пифагор всегда имеет один аргумент, который, однако, может задаваться списком (кортежем), содержащим несколько элементов различных типов.

Семантика модели вычислений гарантирует, что на вход не будет подан параллельный список. Он может быть сформирован в результате выполнения функции. Поэтому в описании возможно как задание списка данных *@(<Tun>)*, так и параллельного списка *@[<Tun>]*. Описание типов сопоставляется с каждым узлом РИГ и размещается в отдельном файле, что позволяет использовать немодифицированный РИГ в ранее разработанных инструментальных средствах. Данные в дополнительный файл в настоящий момент вносятся вручную. Однако промежуточные типы, как и в других языках функционального программирования [11], могут автоматически выводиться из информации об аргументах текущей функции, а также известных сигнатур функций, используемых в РИГ.

Table 1. Example of converting binary and unary arithmetic operators

Таблица 1. Пример преобразования бинарных и унарных арифметических операторов

Фрагмент кода на Пифагор	Эквивалентный фрагмент на C++

`// @{@int} -> @int`
`fact << funcdef X {`
`  X1 << (X, 1);`
`  [(X1:[<=,>]):?]^(`
`  { X },`
`  { (X, X1:-:fact ):*  }`
`  ):. >> return;`
`}`
	`int fact (int arg_0 ){`
	`  int var1 = 1;`
	`  int var2 = 1;`
	`  int var3;`
	`  if (arg_0 <= var1){`
	`  var3 = arg_0;`
	`  }`
	`  if (arg_0 > var1){`
	`  int var4 = arg_0 - var2;`
	`  int var5 = fact(var4);`
	`  int var6 = arg_0 * var5;`
	`  var3 = var6;`
	`  }`
	`  return var3;`
	`}`

В таблице 1 приведена функция вычисления факториала на языке Пифагор и результат ее преобразования в императивную программу на C++. В первой строке исходного кода на Пифагор приведено описание типа аргумента функции, на основе которого выведены остальные типы.

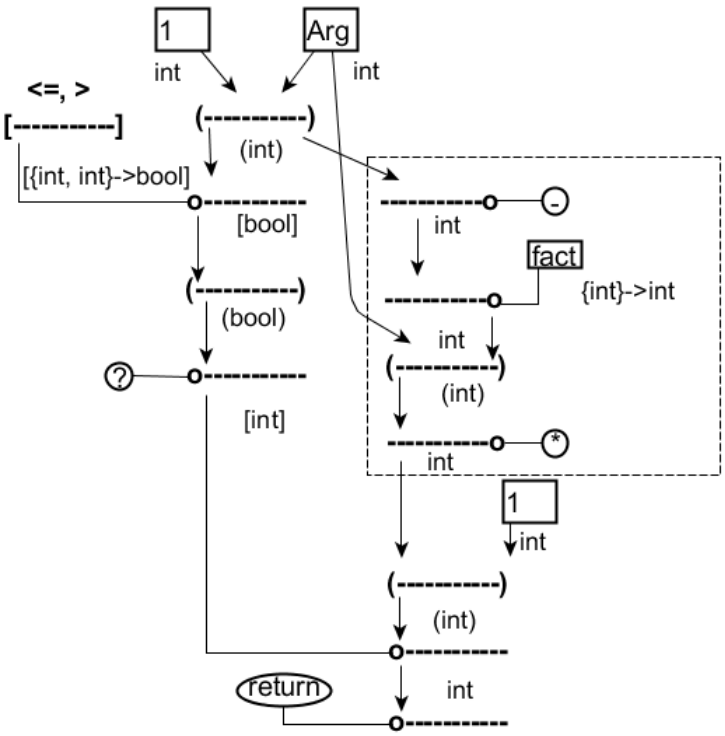

Fig. 1. Example of marking up a reverse information graph with types

Рис. 1. Пример разметки реверсивного информационного графа типами

В таблице 2 приведено текстовое представление РИГ, генерируемое системой трансляции. Показанные типы выставляются при первичной разметке графа, затем они пересчитываются для ряда узлов с учетом правил эквивалентных преобразований [16]. Например, узел 11 – группировка в параллельный список – добавлен в результате трансляции задержанного списка, так как семантика модели вычислений допускает описания в задержанном списке параллельных вычислений. Однако так как задержанный список в данном случае используется лишь для организации ветвления, то в ходе преобразования в императивную форму он распознается как «шаблон», и узел 11 устраняется. Кроме того, узел 6 в программе используется для задания необходимых приоритетов выполнения операций – перед преобразованием в императивную форму такие узлы удаляются.

Table 2. Example of marking up a reverse information graph with types

Таблица 2. Пример разметки реверсивного информационного графа типами

Текстовое представление РИГ	Тип данных узла
External 0 fact Local 0 1 1 {2}11 id delay operation links 0 0 arg 1 0 (---) 0 loc:0 2 0 [---] <= > 3 0 : 1 2 4 0 (---) 3 5 0 : 4 ? 6 0 [---] 5 7 2 : 1 - 8 2 : 7 ext:0 9 2 (---) 0 8 10 2 : 9 * 11 2 [---] 10 12 0 (---) loc:0 loc:1 13 0 : 12 6 14 0 : 13 . 15 0 return 14	------------------ {int} -> int ------------------ int {[int]} ------------------ int (int) [{int, int} -> bool] [bool] (bool) int [int] int int (int) int [int] (int) int int int

На рисунке 1 размеченный РИГ приведен в сокращенном графическом формате (задержанные списки показаны рамкой), при этом убраны узлы, не участвующие в процессе трансформации.

3. Особенности трансформации различных конструкций языка ФПП программирования

Операторы языка ФПП программирования образуют различные комбинации, которые могут рассматриваться при трансформации как группы, определяющие различные шаблоны. Каждый из этих шаблонов может порождать на выходе различные конструкции императивных языков программирования. Ниже рассмотрены типичные ситуации, связанные с проводимыми трансформациями.

3.1. Трансформация списков данных

В языке с динамической типизацией списки данных в общем случае представляют иерархически вложенные коллекции, содержащие данные различного типа. Они также играют различные роли как при использовании в качестве аргументов функций, так и внутри функций. На данный вид списков наложены следующие ограничения.

1) Списки данных могут использоваться для формирования только одномерных массивов, элементы которых принадлежат к одному типу. Учитывая то, что в языке отсутствует описание типов, элементы списка могут иметь только предопределенный (базовый) тип. К базовым типам относятся целые и действительные числа, булевские величины, символы. Для таких ограниченных списков задается описание размера, что позволяет рассматривать их как одномерные массивы.
2) Список данных может являться кортежем, состоящим из данных разного типа. В таком виде он может использоваться для задания описаний аргументов функций, а также в качестве списка фактических параметров, поступающих на вход вызываемых функций. Тип элементов в данном случае также ограничен имеющимися предопределенными типами данных или указателями на предопределенные типы. Указатели позволяют передавать массивы.

3.2. Предопределенные функции

В функционально-потоковой модели параллельных вычислений (ФПМПВ) определен ряд функций, которые не имеют аналогов в императивных языках, но используются для организации вычислений. Их трансформация связана с заменой на специально написанные императивные функции. Следует также отметить, что не все функции данной модели поддаются эффективной трансформации. Поэтому их преобразование не имеет смысла. Подобный подход также принят при создании статически типизированного языка ФПП программирования Smile, в котором ряд функций, используемых в языке Пифагор, отсутствует. При оставшихся функций могут вноситься ограничения по семантике, что связано с их изначально универсальностью, которая не всегда является необходимой в статически типизированных программах.

Существует набор предопределенных функций, которые при наличии дополнительной разметки типов, сопоставляемой вершинам РИГ, достаточно просто трансформируются в операции императивного языка программирования. В частности, к ним относятся арифметические и логические функции, функции сравнения, ряд вспомогательных функций, манипулирующих контейнерными типами и атомарными данными.

Функция «|» возвращает длину списка данных. Поэтому его легко сопоставить с функциями, осуществляющими вычисление длины. В стандартной библиотеке многих императивных языков имеются структуры данных, поддерживающие соответствующую операцию. Например, в C++ это могут быть контейнеры типа vector или list.

Арифметические и логические функции, функции сравнения могут быть непосредственно сопоставлены с соответствующими операциями. В ходе трансформации нужно учесть, что для ряда операторов существуют унарные и бинарные формы. Кроме того, такие операторы языка Пифагор могут обрабатывать значения различных типов. Решение задачи сопоставления операций

осуществляется на основе анализа типов данных. В таблице 3 приведен пример выполнения подобных преобразований над функциями определения модуля числа и сравнения двух дробных чисел с погрешностью.

Table 3. Example of converting binary and unary arithmetic operators

Таблица 3. Пример преобразования бинарных и унарных арифметических операторов

Фрагмент кода на Пифагор

```
//@{@double}->@double
d_abs << funcdef X {
  [((X, 0.0):[<, >=]):?]^(
    {X:-},
    {X}
  ):. >> return;
}

//@{@double,@double,@double}->@bool
is_close << funcdef X {
  A << X:1;
  B << X:2;
  Eps << X:3;
  return << ((A,B):-:d_abs,Eps):<;
}
```

Эквивалентный фрагмент на C++

```
double d_abs (double arg_0 ) {
  double var1 = 0.0;
  double var2;
  if (arg_0 < var1) {
    double var3 = -arg_0;
    var2 = var3;
  }
  if (arg_0 >= var1) {
    var2 = arg_0;
  }
  return var2;
}

double is_close (double arg_0,
                 double arg_1 ,double arg_2 ) {
  double var1 = arg_0 - arg_1;
  double var2 = d_abs(var1);
  bool var3 = var2 < arg_2;
  return var3;
}
```

В ряде ситуаций предопределенные функции языка программирования Пифагор используются в типовых комбинациях, которые можно рассматривать совместно. Неэффективная реализация

отдельных функций в этом случае может быть заменена на более эффективное решение в формируемом на языке C++ выходном представлении. Выделение фрагментов, образующих шаблоны, ведется на основе анализа реверсивного информационного графа. В частности это касается функций транспонирования и формирования дубликатов.

Функция транспонирования «#» выполняет преобразование для вложенных списков данных, аналогичное транспонированию матриц и чаще всего используется для построения пар аргументов, которые в дальнейшем вычисляются параллельно одной и той же функцией. Примером может служить попарное сложение элементов двух векторов. В сочетании с формированием на выходе параллельного списка данная функция производит одновременное сложение всех пар элементов векторов A и B, получая на выходе вектор C. Анализируя данные фрагменты кода можно заменить их на более эффективные императивные решения, которые вместо транспонирования используют прямое попарное сложение элементов массивов. Результат такой трансформации приведен в таблице 4.

Table 4. Transformation of a template fragment containing a predefined transpose function

Таблица 4. Трансформация шаблонного фрагмента, содержащего предопределенную функцию транспонирования

Фрагмент кода на Пифагор

```
//@{@(@int),@(@double)}->@(@double)
vec_sum << funcdef vecPair {
  A << vecPair:1;
  B << vecPair:2;
  C << (A,B):#:[]:+; // Трансформируемый фрагмент
  return << C;
}
```

Эквивалентный фрагмент на C++

```
vector<double > vec_sum (vector<double > arg_0 ,vector<double > arg_1 ){
  size_t var1_size = arg_0.size();
  vector<double > var1(var1_size);
  for (size_t var1_iter = 0; var1_iter < var1_size; ++var1_iter){
    var1[var1_iter] = arg_0[var1_iter] + arg_1[var1_iter];
  }
  return var1;
}
```

Функция «*dup*» выполняет дублирование данных и используется чаще всего для организации параллельных вычислений, когда один из аргументов остается неизменным. Данное дублирование в последовательном императивном программировании является избыточным, так как легко заменяется на повторное обращение к одной и той же переменной. Поэтому непосредственная реализация функции вряд ли имеет смысл. Целесообразнее на основе анализа более крупных фрагментов рассматривать эту функцию как часть общих преобразований, в которых происходит многократное обращение к одной и той же ячейке памяти.

Функция «..» является генератором последовательности данных, образующих ряд целых или действительных чисел с заданным шагом. Для его порождения используется задание начала интервала, его конца и шаг. На выходе формируется параллельный список чисел в заданном диапазоне и с заданным шагом. Результаты работы генератора чаще всего используются в циклах как значение

счетчика. Данную функцию также можно рассматривать в составе соответствующих шаблонов, что позволяет интегрировать ее в операторы цикла императивных программ. Пример выделения такого шаблона и порождения на его основе фрагмента императивного кода представлен в таблице 5.

Table 5. Converting number sequence generators to imperative form

Таблица 5. Преобразование генераторов последовательности чисел в императивную форму

Фрагмент кода на Пифагор

```
//@{@(@double)}->@(@double)
ex18 << funcdef X {
  Len << X:|;
  Generator << (2,Len,3):..;
  Sub << X:Generator;
  return << Sub;
}

//@{@(@double)}->@(@double)
ex19 << funcdef X {
  len << X:|;
  return << (X,(1,len):..):#:[]:ex5;
}
```

Эквивалентный фрагмент на C++

```
vector<double > ex18 (vector<double > arg_0 ){
  int var1 = 2;
  int var2 = 3;
  int var3 = arg_0.size();
  size_t var4_size = (var3 - var1)/var2;
  vector<double > var4(var4_size);
  for (int var4_iter_in = 0, var4_iter_ex = var1;
       var4_iter_ex < var3;
       ++var4_iter_in, var4_iter_ex += var2) {
    var4[var4_iter_in] = arg_0[var4_iter_ex];
  }
  return var4;
}

vector<double > ex19 (vector<double > arg_0 ){
  int var1 = 1;
  size_t var2_size = arg_0.size();
  vector<double> var2(var2_size);
  for (size_t var2_iter = 0; var2_iter < var2_size; ++var2_iter){
    var2[var2_iter] = ex5(arg_0[var2_iter], [var2_iter]);
  }
  return var2;
}
```

3.3. Задержанные списки

Операторы, вложенные внутрь задержанного списка, не начинают выполняться даже при готовности всех своих аргументов до тех пор, пока список не будет «раскрыт». Задержанные списки в функционально-потоковом программировании используются, в том числе для организации ветвления, при этом:

- фрагменты кода, выполняемые условно, помещаются в различные задержанные списки, которые, в свою очередь, вкладываются в список данных;
- в зависимости от результата выполнения «условия» из списка данных выбирается один из элементов — задержанный список;
- к выбранному задержанному списку применяется оператор интерпретации, выполняющий «раскрытие списка», приводящее к выполнению всех вложенных операторов.

При преобразовании в императивную форму, операторы разных задержанных списков, представляющих собой альтернативные ветви алгоритма, списка необходимо разместить в разных блоках оператора ветвления, следовательно система преобразования должна учитывать не только зависимости по данным между узлами, но и вложенность узлов в задержанные списки. Преобразование задержанных списков, задающих ветвление, в императивную форму представлено в таблице 3 на примере трансформации функции **d_abs**.

3.4. Списки данных как аргументы функций

Списки данных являются одной из основных структур, используемых в ФПП программах. В текущей версии интерпретатора они обрабатываются как массивы [17]. Однако в ряде случаев списки данных используются как элементы синтаксических конструкций. При этом в императивном коде такие списки присутствовать не должны, так как преобразуются в другие программные объекты. Например, список, построенный из задержанных списков, в таблице 3 служит для организации ветвления.

Если функция принимает несколько аргументов, то перед вызовом они группируются в кортеж. В императивных языках аргументы целесообразнее передавать в виде списка параметров. Вместе с тем для обращения к конкретному аргументу функции в языке Пифагор выполняется обращение к элементу списка данных по индексу. Такие обращения при преобразовании должны заменяться на обращения к соответствующим аргументам функции, как показано в таблице 4.

3.5. Параллельные списки

Параллельные списки отличаются от списков данных тем, что применяемая к ним операция применяется к каждому вложенному элементу. Наряду со списками данных они зачастую служат вспомогательными нетрансформируемыми понятиями в составных конструкциях. РИГ, приведенный в таблице 1, содержит три задержанных списка: первый используется для задания приоритета операций, второй – для группировки операторов условия, третий был добавлен в результате трансляции задержанного списка. Ни один из них в явном виде не появится в императивном коде.

В связи с ориентацией на статическую типизацию предполагается, что параллельные списки анализируемой программы не имеют вложений и состоят только из элементов, имеющих одинаковый предопределенный тип (как и списки данных). Поэтому для их трансформации возможно использовать класс *vector* из стандартной библиотеки языка программирования C++.

Основное отличие процесса трансформации параллельных списков от списков данных заключается в контексте функций выполняемых над этим списком. На основе анализа этого контекста выделяется соответствующий шаблон, который помещает в цикл все функции, осуществляющие обработку всех элементов формируемого вектора. В ходе трансформации параллельные списки раскрываются, формируя циклические конструкции в соответствии с правилом ФПП-модели

вычислений:

$$[d_1, d_2, \ldots, d_n] : f \longrightarrow [d_1 : f, d_2 : f, \ldots, d_n : f].$$

В данной ситуации использование функции f с параллельным списком $[d_1, d_2, \ldots, d_n]$ приводит к ее применению к каждому элементу списка, то есть повторяющемуся действию. Однако параллельные списки зачастую описываются в программе неявно, а получаются в результате других вычислений. В примере преобразования, показанном в таблице 4, формируется параллельный список, состоящий из пар соответствующих элементов списков данных *vecPair:1* и *vecPair:2*.

3.6. Функции высшего порядка

Передача функции в качестве аргумента другой функции часто используется для повышения гибкости программ. Язык Пифагор имеет поддержку функций высшего порядка и они часто используются в программах. В качестве аргумента может передаваться не только функция, но также оператор и даже произвольный фрагмент кода, помещенный в задержанный список. Такие конструкции крайне сложно перенести на императивный язык в виду следующих причин:

- на вход арифметических и логических операторов могут быть поданы аргументы различных типов данных, но для преобразования необходимо указать типы явно;
- операторы **+** и **-** могут быть унарными;
- фрагмент кода, помещенный в задержанный список может иметь зависимости от контекста функции, в которой он был создан и содержать частично вычисленные значения.

В связи с этим, в программах, предназначенных для преобразования, не должны использоваться указанные выше синтаксические возможности ФПМПВ, а для передачи кода в качестве аргумента функции необходимо явно описывать вспомогательные функции – обертки и типизировать их, как показано в таблицах 6 и 2.

4. Инструментальная поддержка трансформации функционально-потоковых параллельных программ

В процессе преобразования осуществляется следующая трансформация типов данных ФПП программы:

- типы ***int, double*** и ***char*** заменяются на соответствующие типы языка C++;
- числовые списки данных и параллельные списки преобразуются в тип (***std::vector***), а списки символов в строки типа (***std::string***);
- в качестве аргументов функций высшего порядка используется тип ***std::function***.

В функционально-потоковом параллельном программировании используется принцип единственного присваивания, поэтому реверсивный информационный граф функции, по сути, задает граф программных зависимостей (ГПЗ) [18]. Такая форма используется как для внутреннего представления программы в системе преобразования.

Алгоритм преобразования применяется отдельно к каждой функции, при этом он сводится к тому, что выделяет в графе описанные выше шаблоны, для которых задан порядок преобразования. Реверсивный информационный граф обрабатывается в порядке распространения потока данных: от аргумента функции к оператору ***return***.

Выбранное внутреннее представление программы в системе преобразования использует ярусно-параллельную форму (ЯПФ), за счет этого отражает структуру ФПП-программы и позволет удобно и эффективно получать нужную для обработки информацию [19]. Преобразование выполняется, начиная с верхних ярусов ЯПФ, однако отдельно выделяются задержанные списки. В коде ФПП-программ такие списки чаще всего задают альтернативные блоки операторов. Несмотря на то, что узлы одного задержанного списка в ЯПФ располагаются на разных ярусах – в коде на императивном языке они размещаются последовательно друг за другом.

Table 6. Using auxiliary functions to pass operators as arguments to a function

Таблица 6. Использование вспомогательных функций для передачи операторов в качестве аргументов функции

Фрагмент кода на Пифагор

```
d_plus << funcdef X { //@{@double, @double}->@double
  return << (X:1, X:2):+;
}
//@{@(@double),@(@double), @{@double,@double}->@double}->@[@double]
map3_d_d << funcdef X {
  VectorA << X:1;
  VectorB << X:2;
  Operation << X:3;

  [((VectorA:|, VectorB:|):[!=, =]):?]^(
    {"badVectorSize":error},
    {(VectorA , VectorB):#:[]:Operation}
  ):. >> return;
}
// скалярное произведение векторов
//@{@(@double),@(@double)}->@(@double)
vec_mul_d_d << funcdef X {
  ((X:1, X:2, d_plus):map3_d_d) >> return;
}
```

Эквивалентный фрагмент на C++

```
double d_plus (double arg_0 ,double arg_1 ){
  double var1 = arg_0 + arg_1;
  return var1;
}
vector<double > map3_d_d (vector<double > arg_0 ,vector<double > arg_1 ,
function<double(double,double)> arg_2 ){
  int var1 = arg_0.size();
  int var2 = arg_1.size();
  vector<double > var3;
  if (var1 != var2){
    throw std::runtime_error("badVectorSize");
  }
  if (var1 == var2){
    size_t var4_size = arg_0.size();
    vector<double> var4(var4_size);
    for (size_t var4_iter = 0; var4_iter < var4_size; ++var4_iter){
      var4[var4_iter] = arg_2(arg_0[var4_iter], arg_1[var4_iter]);
    }
    var3 = var4;
  }
  return var3;
}
vector<double > vec_mul_d_d (vector<double > arg_0 ,vector<double > arg_1 ){
  vector<double > var1 = map3_d_d(arg_0, arg_1, d_plus);
  return var1;
}
```

На рисунке 2 окружностями обозначены узлы информационного графа, штриховой линией выделены задержанные списки, сверху в ромбе приведен номер задержанного списка, в восьмиугольниках записаны номера ярусов.

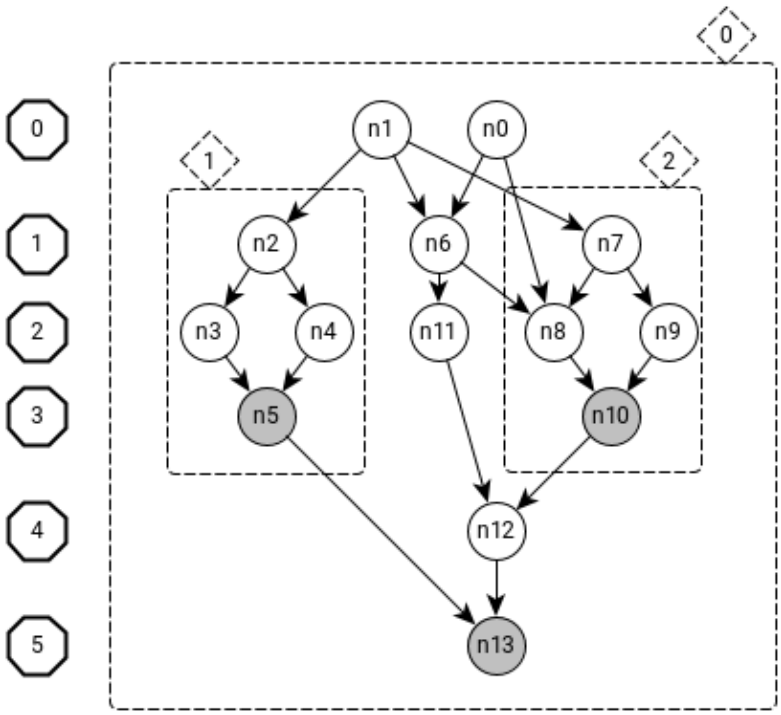

Fig. 2. Information graph with highlighted delayed lists and tiers

Рис. 2. Информационный граф с выделенными задержанными списками и ярусами

Алгоритм трансляции состоит из ряда этапов:

1) Разметка узлов типами на основе типа аргумента функции и правил модели вычислений.
2) Разметка узлов именами переменных, соответствующих узлам ГПЗ в генерируемом коде на императивном языке программирования.
3) Начальная разметка узлов флагами трансляции. Используются следующие флаги:
   - ***NOT_TRANSLATED*** – необработанный узел;
   - ***NO_TRANSLATABLE*** – узел, не требующий преобразования;
   - ***TRANSLATED*** – обработанный узел.
   - ***SWITCH_RESULT*** – метка для узлов, являющихся результатами вычисления ветвей алгоритма при трансляции условных операторов.
4) Подготовка узлов к трансляции. При этом в ГПЗ выделяются и размечаются подграфы, задающие определенные синтаксические конструкции языка C++.
5) Трансляция констант.
6) Трансляция выделенных подграфов для синтаксических конструкций по описанным выше шаблонам.

Заключение

В работе описаны проблемы и особенности преобразования функционально-потоковых программ на основе реверсивных информационных графов, задающих максимальный параллелизм, в императивную форму. Приведены некоторые детали реализации такой системы, преобразующей программы языка Пифагор в язык C++. Предложены:

1) формат описания типов функций функционально-потокового языка программирования;
2) алгоритм преобразования, основанный на поиске в реверсивных информационных графах конструкций, соответствующих заданному множеству шаблонов;
3) структуры данных, обеспечивающие эффективную реализацию такого алгоритма.

Разработанная система позволяет преобразовывать достаточно широкий класс программ и является основой для дальнейшего анализа конструкций ФПП-программ, которые имеет смысл выполнять параллельно на конкретной целевой архитектуре.

References

[1] K. Vivek, *Parallel Computing Architectures and APIs: IoT Big Data Stream Processing*. Dec. 2019, ISBN: 9781351029223. DOI: 10.1201/9781351029223.

[2] I. Levin, A. I. Dordopulo, and V. A. Gudkov, "Programming of reconfigurable computing nodes in the COLAMO language", *Training manual. Taganrog: Publishing house of TTI SFEDU. In Russian*, 2011.

[3] A. I. Dordopulo and I. I. Levin, "Resource-independent programming of hybrid reconfigurable computing systems", in *Russian Supercomputing Days. In Russian*, 2017, pp. 714–723.

[4] V. Kasyanov, "Sisal 3.2: functional language for scientific parallel programming", *Enterprise Information Systems*, vol. 7, no. 2, pp. 227–236, 2013.

[5] A. I. Legalov, "Functional language for creating architecturally independent parallel programs", *Computing technologies*, vol. 10, no. 1, 2005.

[6] I. I. Levin, A. I. Dordopulo, I. V. Pisarenko, and A. K. Melnikov, "An approach to architecture-independent programming of computing systems based on the aspect-oriented Set@ language", *Proceedings of the Southern Federal University. Technical sciences. In Russian*, vol. 197, no. 3, 2018.

[7] A. I. Legalov, V. S. Vasilev, I. V. Matkovskii, and M. S. Ushakova, "A toolkit for the development of data-driven functional parallel programmes", in *International Conference on Parallel Computational Technologies*, Springer, 2018, pp. 16–30.

[8] V. S. Vasilev and A. I. Legalov, "Loop-invariant Optimization in the Pifagor Language", *Automatic Control and Computer Sciences*, vol. 52, no. 7, pp. 843–849, 2018.

[9] U. V. Udalova, A. I. Legalov, and N. U. Sirotinina, "Methods for debugging and verifying functional-stream parallel programs", *Journal of the Siberian Federal University. Equipment and technologies*, vol. 4, no. 2, 2011.

[10] M. S. Ushakova and A. I. Legalov, "Verification of Programs with Mutual Recursion in Pifagor Language", *Automatic Control and Computer Sciences*, vol. 52, no. 7, pp. 850–866, 2018.

[11] S. V. Zykov, *Fundamentals of Modern Programming. Development of heterogeneous systems in an Internet-oriented environment*. IPR Media, 2017. DOI: 10.23682/62072.

[12] A. I. Legalov, I. A. Legalov, and I. V. Matkovsky, "Specifics of semantics of a statically typed language of functional and dataflow parallel programming", in *Scientific Conference Scientific Service on the Internet*, 2019, pp. 489–500.

[13] A. I. Legalov, I. V. Matkovsky, M. S. Ushakova, and D. S. Romanova, "Dynamically Changing Parallelism with the Asynchronous Sequential Data Flows", *Modeling and analysis of information systems*, vol. 27, no. 2, pp. 164–179, 2020.

[14] O. V. Nepomnyashchiy, I. N. Ryzhenko, and A. I. Legalov, "Method of architecture-independent high-level synthesis of VLSI", *Proceedings of the Southern Federal University. Technical sciences*, vol. 202, no. 8, 2018.

[15] O. V. Nepomnyashchiy, I. N. Ryzhenko, and A. I. Legalov, "Methods, algorithms, and software tools for architecturally independent high-level synthesis of single-chip digital systems", in *Supercomputing Technologies (SCT-2018)*, 2018, pp. 104–109.

[16] M. S. Ushakova, "Data type semantics of the dataflow parallel programming language Pifagor", *Educational resources and technologies*, vol. 14, no. 2, 2016.

[17] I. V. Matkovsky and A. I. Legalov, "Instrumental support for the translation and execution of functional-stream parallel programs", *Polzunovsky vestnik*, no. 2, pp. 49–52, 2013.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[19] V. M. Bakanov, "Software tools for analyzing the information structure of algorithms based on their information graphs", in *Parallel Computing Technologies (PaVT ' 2016)*, 2016, pp. 432–441.