МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Параллельных вычислительных технологий

(полное название кафедры)

Дубовик Антон Сергеевич
(И., О., фамилия студента – автора работы)

Разработка алгоритмов компиляции и реализации массовых фрагментов
(полное название темы магистерской диссертации)
данных и вычислений в системе фрагментированного программирования
LuNA

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению высшего профессионального образования

010500 – Прикладная математика и информатика
(код и наименование направления подготовки магистра)

факультет прикладной математики и информатики
(факультет)

Тема диссертации утверждена приказом по НГТУ № 1614/2 от «28» марта 2012 г.

Руководитель

Малышкин В.Э.
(фамилия, И., О.)
д.т.н., профессор
(уч. степень, уч. звание)

Новосибирск, 2012 г.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Параллельных вычислительных технологий

(полное название кафедры)

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.

(фамилия, И., О.)

(подпись, дата)

ЗАДАНИЕ
на магистерскую диссертацию

студенту Дубовику Антону Сергеевичу

(фамилия, имя, отчество)

факультета прикладной математики и информатики

Направление подготовки 010500 – Прикладная математика и информатика
(код и наименование направления подготовки магистра)

Магистерская программа Параллельные вычислительные технологии
(наименование программы)

Тема Алгоритмы компиляции и реализации массовых фрагментов данных
(полное название темы)
и вычислений в системе фрагментированного программирования LuNA

Цели работы Разработать и реализовать алгоритмы поддержки массовых фрагментов данных и вычислений для расширения круга задач численного моделирования, реализуемых в системе фрагментированного программирования LuNA. Протестировать работоспособность реализованных алгоритмов.

Руководитель

Малышкин В.Э.
(фамилия, И., О.)

д.т.н., профессор
(уч. степень, уч. звание)

(подпись, дата)

Содержание

ВВЕДЕНИЕ 4

Базовые принципы технология фрагментированного программирования 6
Модель фрагментированной программы 7
Индексированный способ описания фрагментированных программ 8
Формулировка проблемы 9
Обзор существующих подходов к решению проблемы 10
Спецификация программ с помощью систем рекуррентных соотношений 10
Языки описания графов 11
Функциональные языки программирования 12
Язык НОРМА 13
Язык SETL 14
Цель работы 14
Апробация 16
Тезисы 16
Доклады 16

1. ИНДЕКСИРОВАННОЕ ПРЕДСТАВЛЕНИЯ ФРАГМЕНТИРОВАННОЙ ПРОГРАММЫ 18

1.1. Анализ требований к индексированному представлению фрагментированной программы 21
1.2. Предлагаемые алгоритмы поддержки индексированного представления фрагментированной программы 25
1.2.1 Алгоритм выполнения запроса №2 25
1.2.2 Алгоритмы выполнения запросов №1 и №3 31
1.2.3 Алгоритм выполнения запроса №4 31

2. РЕАЛИЗАЦИЯ ПОДДЕРЖКИ ИНДЕКСИРОВАННОГО ОПИСАНИЯ ФРАГМЕНТИРОВАННОЙ ПРОГРАММЫ В СИСТЕМЕ LUNA 33

2.1. Язык описания фрагментированных программ LuNA 33
2.2. Особенности реализации компилятора 36
2.3. Реализация индексированного представления фрагментированной программы в исполнительной системе 38
2.4. Механизм порционной выдачи результатов запросов 40
2.5. Ограничения реализации 43
2.5.1. Сложность поддержки новых расширений языка LuNA 43
2.5.2. Высокая вычислительная сложность запросов к модулю IndexedFP 44
2.5.3. Фиксированный порядок выдачи результатов запроса 44
2.6. Пути развития 45

3. ТЕСТИРОВАНИЕ 47

3.1. Компилятор 47
3.2. Методы класса *IndexedFP* 49

ЗАКЛЮЧЕНИЕ 52

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 53

ПРИЛОЖЕНИЕ А. ГРАММАТИКА ЯЗЫКА LUNA 57

ПРИЛОЖЕНИЕ В. МОДУЛЬ INDEXEDFP 61

ПРИЛОЖЕНИЕ С. КОМПИЛЯТОР ЯЗЫКА LUNA 81

Введение

Параллельные вычисления находят широкое применение в науке и индустрии [1]. В различных прикладных и научных дисциплинах необходимо их использование для решения сверхбольших задач численного моделирования. Параллельные вычисления находят все более широкое применение, выходя за границы научных лабораторий и отдельных областей в промышленности. Увеличение производительности персональных компьютеров происходит в большей степени за счет увеличения числа вычислителей – ядер или процессоров, нежели увеличения тактовой частоты процессора [2].

Современные программисты все чаще сталкиваются с разработкой параллельных программ и алгоритмов, так как появление доступных многоядерных процессоров и графических ускорителей поставило вопрос об эффективном параллельном программировании для них. В будущем разработка параллельных программ станет данностью.

За 60-летнюю историю программирования, основная часть которой приходится на «эпоху последовательного программирования», было разработано масса программных средств, облегчающих разработку программ (IDE, статические анализаторы кода, инспекторы утечек памяти и т.д.). Однако, к началу «эры параллельного программирования» программисты пришли безоружными. За последние десять лет в научных кругах наблюдается повышение интереса к направлениям, связанным с автоматизацией параллельного программирования на основе использования языков высокого уровня, в частности, к системами параллельного программирования.

Система параллельного программирования – это система, позволяющая программисту абстрагироваться (целиком или частично) от реализации параллелизма (порождения и слияния вычислительных процессов, синхронизации потоков, передачи сообщения и т.д.) в параллельной программе и, тем самым, сосредоточиться на описании параллельного алгоритма [3]. Как следствие, система параллельного программирования в той или иной мере должна решать задачи балансировки нагрузки на вычислительные узлы,

настройки на все имеющиеся аппаратные ресурсы и обеспечения прочих динамических свойств параллельной программы [4].

Задача разработки качественной системы параллельного программирования сложна и активно исследуется. Среди существующих средств параллельного программирования стоит выделить системы Charm++ [5], ProActive Parallel Suite [6], SMP Superscalar [7], Uintah [8], библиотеки Intel Threading Building Blocks [9], PLASMA [10], DPLASMA [11] и языки параллельного программирования (concurrency oriented programming languages) Erlang [12] и Go [13].

В Институте Вычислительной Математики и Математической Геофизики СО РАН в отделе Математического обеспечения высокопроизводительных вычислительных систем ведется разработка системы параллельного программирования LuNA [14,15]. Теоретической основой системы служит метод синтеза параллельных программ на вычислительных моделях, разработанный В.А. Вальковским и В.Э. Малышкиным [3].

Работа является частью большого проекта по разработке системы параллельного программирования для распределенных вычислений LuNA. Отличительная особенность LuNA – неизменность представления прикладных алгоритмов с использованием индексных выражений для всех типов вычислительных архитектур. Система позволяет автоматических конструировать программу, реализующую описанный алгоритм, при этом, производится автоматическая настройка на архитектуру вычислителя – топологию сети, размеры кэш-памяти вычислительных узлов и т.д. Т.е., для разных архитектур генерируются разные программы.

Программа в системе LuNA представляется в виде множества агрегированных переменных множественного присваивания, множества операций над этими переменными и отношения порядка, определяющего последовательность исполнения операций. Эти множества описываются на языке LuNA не в поэлементном виде, а в виде компактных правил, описывающих сразу множество операций или значений. Размер правила при

этом не зависит от размера описываемого множества. Это достигается за счет использования индексированных операций и значений, например, запись

```
A[i,j] | i=1..10, j=1..5
```

описывает сразу 50 значений A с индексами из множества $\{(i, j) | 1 \le i \le 10, 1 \le j \le 5\}$.

Программа, представленная в виде множества подобных индексированных записей, называется *программой в индексированном виде* (или просто индексированной программой). Текст программы компилируется и передается исполнительной системе (ИС), которая выполняет все операции в определенном порядке.

В системе LuNA в модулях компилятора и исполнительной системы должна быть реализована поддержка индексированных программ и представлена:

1. средствами трансляции индексированной программы на языке LuNA в структуру данных, с которой будет непосредственно работать исполнительная система;
2. алгоритмами извлечения информации из структуры данных, хранящей программу.

Работа посвящена вопросам реализации описанной поддержки. Таким образом, результаты магистерской диссертации входят в систему LuNA везде, где требуется обрабатывать индексированные выражения.

Базовые принципы технология фрагментированного программирования

В технологии фрагментированного программирования [16] программа представляется в фрагментированном виде, т.е. в виде множества переменных и операций над этими переменными, а также некоторого минимального частичного порядка на множестве операций, определяющего порядок их исполнения.

Фрагментированная программа компилируется в промежуточное представление, при этом компилятор может выполнить определенные статические оптимизации кода программы (понятие фрагментированной программы в дальнейшем будет сокращаться как ФП, однако, мы оставляем за собой право использовать расшифровку там, где акроним неудобен).

Промежуточное представление принимает на вход исполнительная система (ИС), которая выполняет все операции в порядке, не нарушающем заданный частичный порядок. Фрагментированная природа программы позволяет ИС автоматически обеспечивать выполнение динамических свойств параллельной программы за счет равномерного распределения операций между вычислительными узлами, минимизации межузловых коммуникаций, которая достигается с помощью анализа информационных зависимостей между операциями и т.д.

Модель фрагментированной программы

Фрагментированная программа [14] – это набор $\langle CF, DF, \rho, In, Out \rangle$, где

1. CF – множество фрагментов вычислений (ФВ) – агрегированных операций множественного срабатывания,
2. DF – множество фрагментов данных (ФД) – агрегированных переменных множественного присваивания,
3. $\rho \subseteq CF \times CF$ – отношения строгого частичного порядка на множестве CF, определяющее порядок исполнения ФВ,
4. $In, Out : CF \to 2^{DF}$ – функции, сопоставляющие каждому фрагменту вычислений множество входных и выходных фрагментов данных, соответственно.

Если $(A, B) \in \rho$, значит ФВ A должен завершить свое исполнение до начала выполнения ФВ B. Если $(A, B) \notin \rho$, то ограничения на порядок исполнения A и B относительно друг друга нет, в каком порядке выполнить эти ФВ,

последовательно или параллельно, решает ИС в ходе выполнения фрагментированной программы.

Введем понятия:

1. граф ρ – это пара $\langle CF, \rho \rangle$;
2. двудольный граф IO – граф с множеством вершин $DF \cup CF$ и множеством ребер, которое определяется функциями In, Out;
3. граф ФП – это объединение графов ρ и IO.

Здесь и далее под понятием *фрагмент* будет пониматься фрагмент данных или фрагмент вычислений.

Каждому фрагменту вычислений cf сопоставляется процедура (т.н., фрагмент кода) на языке высокого уровня (*C/C++*, *Fortran*), которая реализует алгоритм вычисления функции $Out(cf)$ из $In(cf)$. В свою очередь, с каждым фрагментом данных в ходе исполнения ФП связывается адрес в памяти мультикомпьютера с определенным количеством выделенной памяти по этому адресу.

Программист, собирающийся разработать программу согласно фрагментированному подходу, должен предоставить описание ФП системе фрагментированного программирования, т.е. дать определение вышеперечисленным компонентам ФП. Разработанный фрагментированный алгоритм, записанный в виде текста, компилируется в промежуточное представление, и выполняется исполнительной системой (ИС), которая исполняет все ФВ в порядке, не противоречащем отношению ρ.

Индексированный способ описания фрагментированных программ

Важным свойством алгоритмов, описываемых в фрагментированном представлении, является свойство массовости [17], что значит их параметризацию относительно размера решаемой задачи. Например, фрагментированная программа умножения матриц не должна умножать матрицы фиксированного, заранее определенного размера. Желательно, чтобы

ФП содержала ряд параметров, обозначающих размеры матриц. Такую программу можно использовать для умножения матриц любого размера, а не только заранее определенного. Таким образом, за счет параметризации множество фрагментов данных и вычислений может быть потенциально бесконечным, а значит, его невозможно задать в поэлементном виде.

Один из возможных способов описания множеств ФД и ФВ в тексте фрагментированной программы является их описание с помощью конечного числа *параметризированных (индексированных) выражений*, например:

$$\rho\left(cf1_{i,j}, cf2_{i+j}\right),\ 1 \le i \le j \le N, \quad (1)$$

что значит, ФВ $cf1_{i,j}$ должен завершить исполнение до начала исполнения ФВ $cf2_{i+j}$ для всех i, j таких, что $1 \le i \le j \le N$. Таким образом, каждый конкретный ФД и ФВ характеризуется строковым идентификатором и конечным (возможно пустым) набором целочисленных индексов. Такого рода индексированное представление ФП, содержит дополнительную информацию о структуре ФП, которой может воспользоваться компилятор или ИС для оптимизации исполнения программы.

Значения некоторых параметров фрагментированной программы, определяющих множества CF, DF и ρ, на стадии компиляции могут быть неизвестны, например, величина N из (1), что обеспечивает свойство массовости ФП.

Формулировка проблемы

Вопрос о представлении ФП в ИС сводится к двум подходам:

a. поэлементное представление графа ФП,
b. индексированное представление, т.е. представление в виде индексированных выражений (перечисляющих алгоритмов), подобных выражению (1).

Использование первого варианта, в силу того, что множества CF и DF потенциально бесконечны, может потребовать для хранения графа ФП

неограниченно большей памяти, чем хранение просто набора индексированных выражений, поэтому поэлементный подход в большинстве случаев неприемлем.

Чтобы выполнить фрагментированную программу, исполнительной системе требуется, в частности, следующая информация:

1. в каком порядке должны запускаться на исполнение ФВ, что определяется отношением ρ;
2. какие фрагменты данных нужны для исполнения конкретного ФВ – это свойство определяется графом IO.

Работа посвящена вопросам разработки и реализации эффективных алгоритмов, позволяющих извлекать информацию типа 1. и 2. из индексированного представления ФП, соответствующего подходу b.

Обзор существующих подходов к решению проблемы

Для решения поставленной проблемы нам могут быть полезны результаты из области языков описания потенциально бесконечных множеств и графов, а также алгоритмы и программные средства, позволяющие ими манипулировать.

Спецификация программ с помощью систем рекуррентных соотношений

В статье [18] и в развивающей её работе [19] рассматриваются системы уравнений вида:

$$
\begin{aligned}
&A_1\left[I_1,\dots,I_{d_1}\right] = F_1\left(\dots I_j,\dots,A_1\left[E_1^l,\dots,E_{d_1}^l\right],\dots,A_m\left[E_1^s,\dots,E_{d_m}^s\right]\right)\\
&\vdots\\
&A_m\left[I_1,\dots,I_{d_m}\right] = F_m\left(\dots I_l,\dots,A_1\left[E_1^t,\dots,E_{d_1}^t\right],\dots,A_m\left[E_1^u,\dots,E_{d_m}^u\right]\right)
\end{aligned}
$$

где $I_1,\dots,I_{d_k}$ - индексные переменные, называемые *свободными индексами*, принимающие неотрицательные целочисленные значения. E_j^i – *индексные выражения*, которые могут иметь две формы:

1. $I_l \pm c,\ c \in \mathbb{N}$;

2. $g\left(I_1,\ldots,I_{d_t},A_1\left[E_1^l,\ldots,E_{d_1}^l\right],\ldots,A_m\left[E_1^s,\ldots,E_{d_m}^s\right]\right)$, где g – функциональный символ, обозначающий функцию, принимающая целочисленные значения.

F_j – функциональный символ для функции над множеством D – областью определения значений массивов $A_1,\ldots,A_m$.

Такая система определяет значения потенциально бесконечных массивов $A_1,\ldots,A_m$.

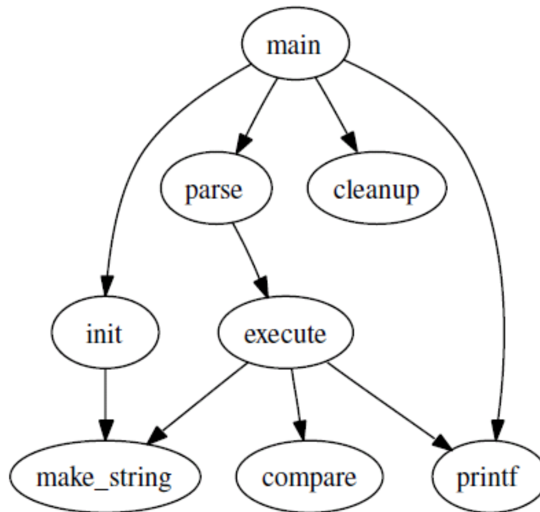
Главный результат работы [19] состоит в алгоритме определения возможности и собственно перевода подобных систем уравнений в программу, состоящую только из циклов типа *FOR* и операторов присваивания значений элементам массивов $A_1,\ldots,A_m$, позволяющую вычислять значения массивов $A_1,\ldots,A_m$.

Такой алгоритм может быть использован для определения ацикличности отношения информационных зависимостей и отношения ρ в фрагментированной программе.

Языки описания графов

Существует ряд языков (форматов) описания графов, таких как DOT [20], GraphML [21], TGF (Trivial Graph Format) [22].

В перечисленных языках графы представляются в поэлементном виде как множество вершин и ребер. Например, в формате DOT граф



представляется следующим образом:

```
digraph G {
 main -> parse -> execute;
 main -> init;
 main -> cleanup;
 execute -> make_string;
 execute -> printf
 init -> make_string;
 main -> printf;
 execute -> compare;
}
```

Ни один из рассмотренных языков не предназначен для описания потенциально бесконечных графов.

Функциональные языки программирования

Функциональные языки программирования, поддерживающие ленивые вычисления, такие, например, как Haskell [23] и Scheme [24], позволяют представлять бесконечные структуры данных, структурные составляющие которых вычисляются по мере необходимости за счет механизма ленивых (отложенных) вычислений.

Например, на языке Haskell можно определить функцию `primes`, возвращающую неограниченно длинный список простых чисел в порядке возрастания, начиная с 2:

```
ghci> let primes' (x:xs) = x:[p | p <- primes' xs, p `mod` x /= 0]
ghci> let primes = primes' [2..]
```

Элементы этого списка будут вычисляться по требованию, поэтому можно взять первые десять элементов этого списка, не опасаясь бесконечно долгого вычисления элементов бесконечного списка:

```
ghci> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

Однако, библиотек для приведенных языков, позволяющих представить потенциально бесконечные множества в индексированной форме, не существует.

Язык НОРМА

НОРМА – декларативный язык для спецификации задач вычислительного характера, разрабатываемый прикладными математиками ИПМ им. М.В.Келдыша РАН [25].

В записи программы на языке НОРМА не требуется никакой информации о порядке счета, способах организации вычислительных (циклических) процессов. Порядок предложений языка может быть произвольным – информационные взаимосвязи выявляются и учитываются транслятором при организации вычислительного процесса. Программа на языке Норма может рассматриваться как описание запроса на вычисление, а реализация этого запроса с учетом архитектуры ЭВМ и возможностей выходного языка – то есть синтез выходной программы – возлагается на транслятор.

Программа на языке НОРМА описывается в декларативном виде, похожем на систему индексированных рекуррентных уравнений из предыдущего параграфа. Компилятор языка НОРМА анализирует программу и генерирует код вычислительно эквивалентной параллельной программы на языке высокого уровня с реализацией параллелизма средствами OpenMP и/или MPI.

Вид рекуррентных соотношений, специфицирующих алгоритм, на языке НОРМА ограничен так, что возможен синтез параллельной программы,

реализующей алгоритм. Алгоритмы синтеза представлены в работе И.Б. Задыхайло [18] и развиты в диссертации А.Н. Андрианова [26].

Компилятор языка НОРМА транслирует программу на язык высокого уровня, которая потом выполняется в рамках исполнительных систем MPI или OpenMP. Таким образом, работа с индексными выражениями на этапе исполнения не производится.

Язык SETL

Отдельного рассмотрения заслуживает язык SETL [27], который основан на математической теории множеств. SETL позволяет записывать формулы исчисления предикатов, т.е. использовать в выражениях языка кванторы всеобщности и существования, например, программа печатающая список простых чисел до N:

```
print([n in [2..N] | forall m in {2..n – 1} | n mod m > 0]);
```

Однако, в языке SETL отсутствуют средства для работы с индексированными множествами.

Анализ особенностей приведенных языков и программных продуктов показал, что ни один из них не применим для решения поставленной проблемы.

Цель работы

Текст ФП компилируется в промежуточное представление, которое используется ИС для выполнения ФП (рисунок 1).

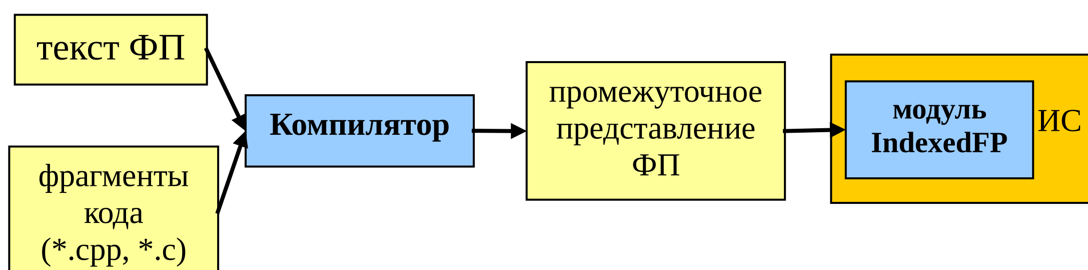


Рисунок 1. Схема исполнения ФП

ИС получает всю необходимую информацию о фрагментированной программе с помощью программного модуля поддержки индексированного представления (далее просто модуль *IndexedFP*) (рисунок 2).

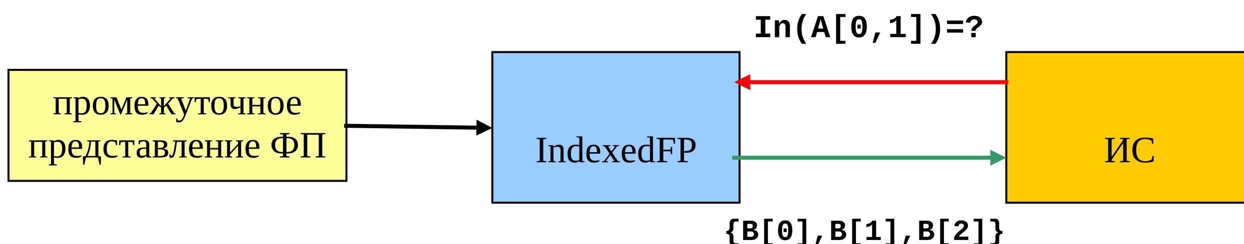


Рисунок 2. Пример взаимодействия ИС с модулем IndexedFP

Целью работы является:

1. разработка индексированного представления ФП и алгоритмов извлечения из этого представления информации, которая необходима ИС для исполнения фрагментированной программы;
2. реализация компилятора, транслирующего исходный код ФП в промежуточное индексированное представление;
3. реализация алгоритмов из пункта 1. в виде программного модуля *IndexedFP* для ИС, осуществляющего поддержку индексированного представления.

Научная новизна работы состоит в разработке индексированного представления фрагментированной программы и алгоритмов, позволяющих извлекать необходимую информацию о ФП (например, вычисление окрестности определенного фрагмента в графе ФП) из индексированного представления ФП.

Актуальность работы заключается в необходимости использования индексированного представления ФП в задачах численного моделирования для обеспечения свойства массовости. Кроме того, работа позволяет расширить круг реализуемых в системе LuNA задач численного моделирования, т.к.

использование индексированного представления ФП позволяет исполнять ФП с произвольным количеством фрагментов. Это особенно актуально, т.к. алгоритмы решения сверхбольших задач численного моделирования, на реализацию которых ориентирована технология фрагментированного программирования, имеют по большей части регулярную структуру. Такие алгоритмы нужно описывать компактно, в виде набора индексированных выражений, задающих способ вычисления одних величин из других. К регулярным алгоритмам относятся, например, конечно-разностные схемы, матрично-векторные операции, метод частиц в ячейках (PIC-метод).

Результаты работы позволяют реализовать систему параллельного программирования LuNA, в чем состоит ее **практическая ценность**.

Апробация

Тезисы

Дубовик А.С., Массовое представление фрагментов данных и вычислений в системе фрагментированного программирования LuNA : тезисы доклада / А.С. Дубовик ; науч. рук. В. М. Малышкин // Материалы XLIX международной научной студенческой конференции «Студент и научно-технический прогресс». – Новосибирск, 2011. – Информационные технологии. – С. 220

Доклады

1. **Дни студенческой науки НГТУ** (март 2011 г.) «Массовое представление фрагментов данных и вычислений в системе фрагментированного программирования LuNA.».

   Получен диплом 1-ой степени в соответствующей секции.
2. **Конференция молодых ученых ИВиМГ СО РАН** (апрель 2011 г.) «Представление множеств перечисляющими алгоритмами в системе фрагментированного программирования LuNA.»

3. **Конкурс на соискание студенческих научных грантов НГТУ** (21 апреля 2011 г.) «Представление множеств перечисляющими алгоритмами в системе фрагментированного программирования LuNA.»

   Грант получен, № 034 – НГС – 11.

4. **XLIX международная научная студенческая конференция «Студент и научно-технический прогресс»**. – Новосибирск, 2011. «Массовое представление фрагментов данных и вычислений в системе фрагментированного программирования LuNA.»

   Получен диплом 3-ей степени в секции «Информационные технологии» подсекции «Архитектура информационных систем для параллельных вычислений».

1. Индексированное представления фрагментированной программы

Определим индексированный способ представления ФП, который используется в языке LuNA и который сохраняется в исполнительной системе. Фрагментированная программа описывается в виде множества индексированных правил 3-х типов. Каждое такое правило состоит из

- зависящего от набора целочисленных индексных переменных утверждения, которое описывает некоторое множество вершин и ребер графа ФП,
- множества значений, которые могут принимать индексные переменные.

Использование понятия *правило* имеет смысл, т.к. правило, примененное к определенным значениям индексных переменных, продуцирует конкретное множество вершин и ребер графа ФП.

I. Определение фрагмента вычислений cf и соответствующих ему значений $In(cf)$, $Out(cf)$:

$$
\begin{aligned}
& cf\left[h_1(\vec{i}),\ldots,h_d(\vec{i})\right] = \\
& in: \\
& \qquad df_1\left[g_1^1(\vec{i}),\ldots,g_{d_1}^1(\vec{i})\right], \\
& \qquad \ldots \\
& \qquad df_p\left[g_1^p(\vec{i}),\ldots,g_{d_p}^p(\vec{i})\right] \\
& out: \\
& \qquad df'_1\left[h_1^1(\vec{i}),\ldots,h_{d'_1}^1(\vec{i})\right], \\
& \qquad \ldots \\
& \qquad df'_{p'}\left[h_1^{p'}(\vec{i}),\ldots,h_{d'_{p'}}^{p'}(\vec{i})\right]
\end{aligned}
$$

Здесь и далее набор из индексных переменных $i_1,i_2,\ldots,i_k$ обозначается как $\vec{i}$. На значения индексов $i_1,i_2,\ldots,i_k$ наложены ограничения:

$$
\begin{array}{l}
L_1 \le i_1 \le U_1 \\
L_2(i_1) \le i_2 \le U_2(i_1) \\
L_3(i_1, i_2) \le i_3 \le U_3(i_1, i_2) \\
\ldots \\
L_k(i_1, i_2, \ldots, i_{k-1}) \le i_k \le U_k(i_1, i_2, \ldots, i_{k-1})
\end{array}
$$

h_i, g_i^j, L_i, U_i – целочисленные функции от значений параметров $i_1, \ldots, i_k$. Ограничения на область определения $\vec{i}$ заданы таким образом, что если известны значения $i_1, \ldots, i_l$, то можно вычислить ограничения на i_{l+1}, что позволяет вычислить всё допустимое множество значений индексов и целиком развернуть правило, получив множество определений для конкретных ФВ.

II. Определение порядка ρ:

$$
\rho\left(cf_{before}\left[h_1(\vec{i}), \ldots, h_d(\vec{i}) \right],\ cf_{after}\left[g_1(\vec{i}), \ldots, g_p(\vec{i}) \right] \right)
$$

при этом, как и ранее

$$
\begin{array}{l}
L_1 \le i_1 \le U_1 \\
L_2(i_1) \le i_2 \le U_2(i_1) \\
L_3(i_1, i_2) \le i_3 \le U_3(i_1, i_2) \\
\ldots \\
L_k(i_1, i_2, \ldots, i_{k-1}) \le i_k \le U_k(i_1, i_2, \ldots, i_{k-1})
\end{array}
$$

В силу транзитивности отношения ρ нет необходимости описывать его целиком, достаточно дать описание отношения $R \subseteq \rho$ такого, что его транзитивное замыкание есть ρ, т.е. $R^+ = \rho$. Таким образом, в тексте программы описывается R – транзитивная редукция ρ.

III. Определение множества ФД с одним идентификатором и заданным размером в байтах:

$$
dfName\left[h_1(\vec{i}), \ldots, h_d(\vec{i}) \right] = block\left(g(\vec{i}) \right),
$$

$$
\begin{array}{l}
L_1 \le i_1 \le U_1 \\
L_2(i_1) \le i_2 \le U_2(i_1) \\
L_3(i_1, i_2) \le i_3 \le U_3(i_1, i_2) \\
\ldots \\
L_k(i_1, i_2, \ldots, i_{k-1}) \le i_k \le U_k(i_1, i_2, \ldots, i_{k-1})
\end{array}
$$

Опишем индексированное представление ФП, реализующей умножение матрицы на вектор. Пусть A – матрица размера $N \times N$, а B, C – вектор-столбцы размера N. Тогда произведение $A \times B = C$ с помощью блочного алгоритма будет вычисляться следующим образом:

$$C_i = \sum_{k=1}^{N/M} A_{i,k} \times B_k \ , \ 1 \le i \le K \ ,$$

где $K = N/M$, $A_{i,j}$, $1 \le i, j \le K$ – подматрицы матрицы A размером $M \times M$, B_i, C_i, $1 \le i \le N/M$ – подвекторы векторов B, C размером M.

ФП, реализующую умножение матрицы на вектор можно представить в виде набора правил I-III. Схематическое изображение ФП представлено на рисунке 3.

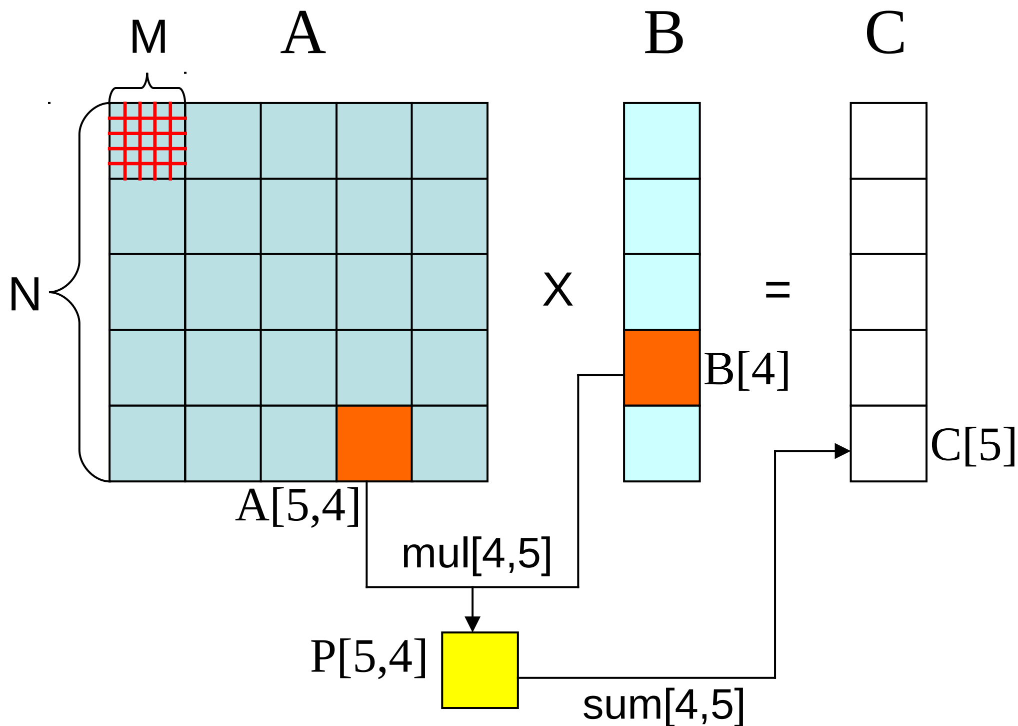


Рисунок 3. Схема фрагментированной программы блочного умножения матрицы на вектор для K=5

I. Определение ФД, хранящих матрицы и векторы:

$A[i, j] = block(4 \times M \times M), 1 \le i \le K, 1 \le j \le K$,

$P[i, j] = block(4 \times M), 1 \le i \le K, 1 \le j \le K$,

$B[i] = block(4 \times M), 1 \le i \le K$,

$C[i] = block(4 \times M), 1 \le i \le K$.

где P - фрагменты данных, в которых будут храниться произведения блоков матрицы A и вектора B.

II. Определение множества фрагментов вычислений для начальной инициализации A, B и обнуления вектора C.

$initA[i, j] = out : A[i, j]$, $1 \le i \le K, 1 \le j \le K$,

$initB[i] = out : B[i]$, $1 \le i \le K$,

$zeroC[i] = out : C[i]$, $1 \le i \le K$.

ФВ, реализующие вычисление промежуточных произведений $P[i, j]$:

$$prod[i,k] = \frac{in : A[i,k], B[k]}{out : P[i,k]},\ 1 \le i \le K, 1 \le k \le K,$$

и суммирующие ФД $P[i, j]$:

$$add[i,k] = \frac{in : P[i,k], C[i]}{out : C[i]},\ 1 \le i \le K, 1 \le k \le K \cdot$$

III. Определение порядка выполнения ФВ.

Перед тем, как вычислять частичные произведения, необходимо инициализировать A и B:

$\rho(initA[i,k], prod[i,k])$, $1 \le i \le K, 1 \le k \le K$,

$\rho(initB[i], prod[i,k])$, $1 \le i \le K, 1 \le k \le K$.

Аккумулирование результата в C возможно, только после обнуления C и вычисления промежуточного произведения в P:

$\rho(initC[i], sum[i,k])$, $1 \le i \le K, 1 \le k \le K$,

$\rho(prod[i,k], sum[i,k])$, $1 \le i \le K, 1 \le k \le K$.

1.1. Анализ требований к индексированному представлению фрагментированной программы

Упрощенно порядок исполнения ФП можно описать следующим образом.

Исполнительная система выполняет ФВ, так чтобы не нарушить отношение порядка ρ. С каждым ФВ связан счетчик числа ФВ, которые должны исполниться до него. Это число определяется отношением ρ – для некоторого ФВ A значение счетчика будет равно мощности множества $\{x|\rho(x,A)\}$. Множество кандидатов на исполнение среди ФВ состоит из ФВ, у которых значение счетчика равно 0.

Исполнительная система выбирает одного из кандидатов, обозначим его как B, и исполняет его на одном из вычислительных узлов. После завершения исполнения B для всех ФВ, которые должны были исполниться после него, т.е. ФВ из множества $\{x|\rho(B,x)\}$, уменьшается значение счетчика на единицу. После этого обновляется список кандидатов и так продолжается до тех пор, пока все ФВ не будут исполнены. Это произойдет, только если отношение ρ ациклично, что всегда верно, т.к. ρ – отношение строгого частичного порядка.

Таким образом, для выполнения конкретного ФВ исполнительной системе необходимо знать:

- в каких ФД он нуждается, т.е. его входные и выходные ФД;
- множество ФВ, которые должны исполниться непосредственно до него для того, чтобы определить, остались ли зависимости по порядку исполнения;
- множество ФВ, которые должны исполниться непосредственно после него, чтобы сообщить им о завершении его исполнения.

Представленный ход работы ИС позволяет выделить набор запросов, который должен быть реализован в модуле индексированного представления ФП. Такой набор запросов является необходимым минимумом, который требуется исполнительной системе для организации исполнения фрагментированной программы.

1. Множество ФД, необходимых для исполнения ФВ: $DF_{need}(cf) = In(cf) \cup Out(cf)$ (рисунок 4)

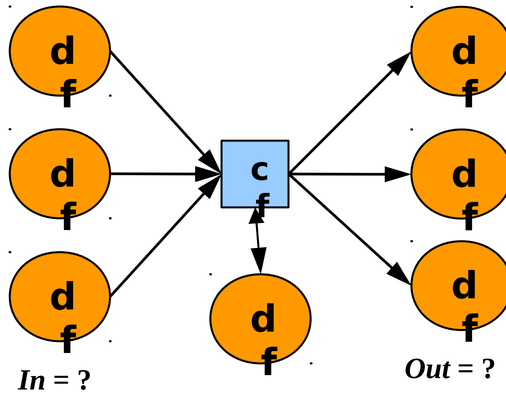


Рисунок 4. Множество $DF_{need}(cf)$

или что то же, 1-окрестность в графе IO для данного ФВ.

2. Множество ФВ, для исполнения которых необходим текущего ФД $CF_{need}(df) = \{ cf \mid df \in DF_{need}(cf) \}$ (рисунок 5)

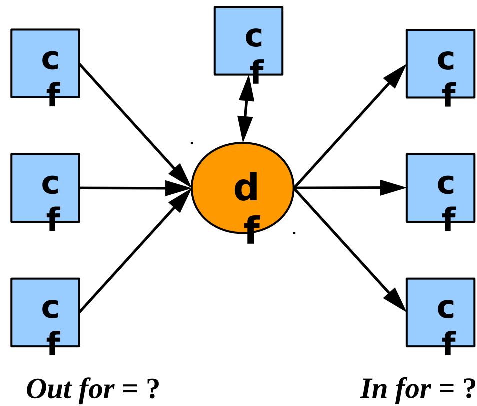


Рисунок 5. Множество $CF_{need}(df)$

иными словами, 1-окрестность в графе IO для рассматриваемого ФД.

3. Множество ФВ, которые должны быть исполнены непосредственно до (после) исполнения данного ФВ в смысле отношения ρ (рисунок 6).
$CF_{before}(cf) = \{ cf' \mid \rho(cf', cf) \}$,

$CF_{after}(cf) = \{ cf' \mid \rho(cf, cf') \}$.

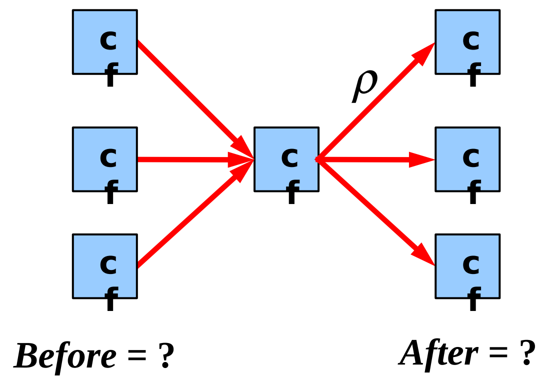


Рисунок 6. Множества $CF_{before}(cf)$ и $CF_{after}(cf)$

или что то же, множество родительских (дочерних) вершин в графе R для данного ФВ.

4. Начальное множество ФВ и ФД – множество тех ФВ, которые могут быть исполнены первыми в смысле ρ, и те ФД, которые необходимы для вычисления начального множества ФВ (рисунок 7).

$$CF_{start} = \{ cf \mid CF_{before}(cf) = \varnothing \},$$

$$DF_{start} = \bigcup_{cf \in CF_{start}} DF_{need}(cf).$$

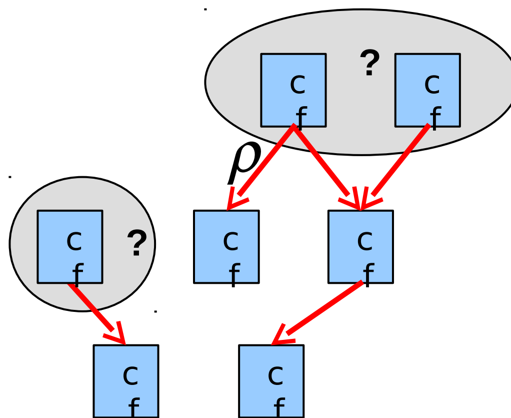


Рисунок 7. Множество CF_{start}

Тривиальные запросы, типа определения является ли ФД df входным, выходным или ни тем, ни другим для ФВ cf, и подобные им здесь не рассматриваются.

Понятно, что нахождение образов и прообразов для функций типа $f : F \to \mathbb{R}$, $f : F \to 2^F$, $f : F \to F^n$ $(F = CF \cup DF)$ и отношений вида $P \subseteq F \times F$, описываемых правилами подобными II и III, осуществляется аналогично тому, как это делается для In, Out и ρ в запросах №1-4.

1.2. Предлагаемые алгоритмы поддержки индексированного представления фрагментированной программы

Рассмотрим алгоритмы выполнения запросов №1-4, использующие представление ФП в виде набора правил вида I-III.

1.2.1 Алгоритм выполнения запроса №2

Начнем с второго запроса – необходимо вычислить множество $CF_{need}(df) = \{cf \mid df \in DF_{need}(cf)\}$ для рассматриваемого ФД df, который характеризуется определенным идентификатором $dfName$ и набором целочисленных индексов $j_1, j_2, \ldots, j_k$, который в дальнейшем будем обозначать просто $\vec{j}$.

В рамках всей ФП задача разбивается на множество подзадач, заключающихся в нахождении $CF_{need}(df)$ для каждого правила в отдельности, и объединении полученных множеств в одно, которое будет результатом запроса. В свою очередь, решение задачи в пределах одного правила сводится к объединению решений относительно каждого вхождения массового ФД с идентификатором $dfName$ в данное правило.

Опишем элементарную подзадачу, которую необходимо решить.

Дано:

1. правило типа II, описывающее ФВ и его входные, выходные ФД:

$$
\begin{aligned}
& cfName\left[l_1(\vec{i}),\ldots,l_n(\vec{i}),nl_1(\vec{i}),\ldots,nl_m(\vec{i})\right] = \\
& in: \\
& \qquad \ldots \\
& \qquad dfName\left[l'_1(\vec{i}),\ldots,l'_{n'}(\vec{i}),nl'_1(\vec{i}),\ldots,nl'_{m'}(\vec{i})\right], \\
& \qquad \ldots \\
& out: \\
& \qquad \ldots
\end{aligned}
$$

область определения $\vec{i}$ задается неравенствами:

$$
\begin{aligned}
& L_1 \le i_1 \le U_1 \\
& L_2(i_1) \le i_2 \le U_2(i_1) \\
& L_3(i_1,i_2) \le i_3 \le U_3(i_1,i_2) \\
& \ldots \\
& L_p(i_1,i_2,\ldots,i_{p-1}) \le i_p \le U_p(i_1,i_2,\ldots,i_{p-1})
\end{aligned}
\qquad (2)
$$

где l_i, l'_i – линейные функции от своих аргументов, nl_i, nl'_i – нелинейные функции, $\vec{i} = i_1,\ldots,i_p$ – набор индексных переменных (или просто индексов), U_i, L_i – целочисленные функции, которые могут быть как линейными, так и нелинейными.

Под *линейной функцией (выражением)* l понимается целочисленная функция $l(x_1,\ldots,x_n) = a_1x_1 + \ldots + a_nx_n + a_0$, где $a_0,a_1,\ldots,a_n \in \mathbb{Z}$. Под *нелинейной функцией* – все остальные целочисленные функции, например, $l(x,y,z) = xy + z$ или $l(x,y,z) = x\%y + z$, где $\%$ – операция деления по модулю.

2. определенный ФД $df = dfName[j_1, j_2,\ldots,j_k]$, где $k = n' + m'$.

Требуется вычислить множество ФВ cf' таких, что df является входным ФД для cf', причем данное отношение между cf' и df определено рассматриваемым правилом.

Понятно, что значения индексов df и значения индексов в ФД с именем $dfName$, описанном в правиле, должны совпадать, т.е.

$$
\begin{aligned}
&l'_1(\vec{i}) = j_1\\
&\ldots\\
&l'_{n'}(\vec{i}) = j_{n'}\\
&nl'_1(\vec{i}) = j_{n'+1}\\
&\ldots\\
&nl'_{m'}(\vec{i}) = j_k
\end{aligned}
\qquad (3)
$$

Таким образом, необходимо:

1. найти все решения системы (2) – векторы $\vec{i}$, которые кроме того должны удовлетворять ограничениям (3);
2. для каждого найденного решения вычислить индексы в выражении $cfName\left[l_1(\vec{i}), \ldots, l_n(\vec{i}), nl_1(\vec{i}), \ldots, nl_m(\vec{i})\right]$. Множество найденных ФВ будет решением данной подзадачи.

Алгоритм выполнения второго пункта очевиден и заключается в вычислении известных функций с целочисленными аргументами, при этом значения функций должны быть целыми, так как, они являются значениями индексов фрагмента вычислений.

Рассмотрим алгоритм выполнения первого пункта – нахождение целочисленных решений системы (3) в условиях ограничений (2).

Составим СЛАУ из линейных уравнений системы (3):

$$
\begin{aligned}
&a_{1,1}i_p + a_{1,2}i_{p-1} + \ldots + a_{1,p}i_1 = j_1\\
&a_{2,1}i_p + a_{2,2}i_{p-1} + \ldots + a_{2,p}i_1 = j_2\\
&\ldots\\
&a_{n',1}i_p + a_{n',2}i_{p-1} + \ldots + a_{n',p}i_1 = j_{n'}
\end{aligned}
\qquad (4)
$$

Воспользуемся методом Гаусса, чтобы привести матрицу СЛАУ (4) к верхнетреугольному виду. В результате, для каждой индексной переменной $i_t,\ 1 \le t \le p$ будет найдена линейная функция, выражающая его через индексные переменные с меньшим номером

$$
i_t = t_t\left(i_{t-1}, i_{t-2}, \ldots, i_1\right), \qquad (5)
$$

либо значение индексной переменной останется неизвестным, назовем такие индексы *свободными индексами*.

Теперь необходимо перебрать все возможные наборы $\vec{i}$, такие, что выполняются:

1. равенства (3), а именно часть равенств с нелинейной правой частью,
2. ограничения (2),
3. найденные в результате решения СЛАУ соотношения между индексами (5).

Реализуем перебор значений индексных переменных, начиная с наиболее независимых от остальных индексов и заканчивая наиболее зависимыми, т.е. в направлении увеличения номера индексной переменной. Если индекс свободен, то изменяем его значение в пределах заданных ограничениями (2), иначе вычисляем его значение через значения уже известных индексов, проверяем, чтобы это значение удовлетворяло ограничению (2) и рекурсивно запускаем процедуру нахождения значений для следующих индексных переменных.

Когда все значения индексов вычислены, проверяем, выполняются ли равенства с нелинейной левой частью из системы (3). Стоит заметить, что проверку на выполнение равенств с нелинейной левой часть из (3) можно осуществлять не тогда, когда известны *все* значения параметров, а тогда, когда известны все *необходимые* для вычисления выражения индексы.

Пусть *sol* – результат работы данного алгоритма, т.е. набор значений индексных переменных $\vec{i}$, тогда процедуру поиска всех целочисленных решений (3) в ограничениях (2) можно записать следующим образом:

Поиск(*index*)

 # $1 \le index \le p$ – номер индекса, значение которого перебирается

 # на момент вызова процедуры известны значения индексов $i_1, i_2, \ldots, i_{index-1}$

 # если известны значения всех индексов, проверяется корректность решения

Если $(index > p)$

Если $\forall t \in [1, N_{m'}] : nl'_t(\vec{i}) = j_{n'+t}$

Добавляем $\vec{i}$ в *sol*

Выход из процедуры

 # вычисляется нижняя и верхняя граница изменения index –ого индекса

low = $L_{index}(i_1, i_2, \ldots, i_{index-1})$

up = $U_{index}(i_1, i_2, \ldots, i_{index-1})$

Если (индекс i_{index} свободен)

Для всех целых $j \in [low, up]$

$i_{index} = j$

Поиск(*index*+1)

Иначе

$i_{index} = t_{index}(i_{index-1}, i_{index-2}, \ldots, i_1)$

Если ($i_{index} \in Z \wedge low \le i_{index} \le up$)

Поиск(*index*+1)

Набор решений сохраняется в списке *sol*. Начальный запуск процедуры – **Поиск(1)**.

Предложенный алгоритм решает задачу удовлетворения ограничений (*constraint satisfaction problem* [28,29]). Алгоритмы решения таких задач реализованы в многочисленных специализированных библиотеках (*constraint satisfaction solvers*), таких как Gecode [30], Minion [31] и специализированных языках программирования – Oz [32], AMPL [33] и др.

Одно из требований к реализации описанного выше алгоритма является порционная выдача результатов – возможность получать требуемые множества по частям в разные моменты времени, а не всё единовременно. Ни одна из существующих библиотек решения задачи удовлетворения ограничений не предоставляет такой возможности, что означает их неприменимость для решения проблемы. Использование языков программирования в ограничениях

(*constraint programming*) неприемлемо, т.к. частое обращение ИС к системе программирования в ограничениях неэффективно.

Рассмотренный запрос выполняется не более одного раза для каждого ФД на одном вычислительном узле. Достигается это за счет того, что исполнительная система сохраняет результат запроса до тех пор, пока он нужен.

Проиллюстрируем алгоритм на простом примере.

Дано правило типа II:

$$\begin{matrix} A[i,k,j] = \\ in: \\ out: B[k+i+j,j] \end{matrix}, \begin{matrix} 1 \le i \le 5 \\ 1 \le j \le i \\ 1 \le k \le j \end{matrix}$$

и ФД $B[5,2]$. Необходимо вычислить множество ФВ $A[i,k,j]$, для которых $B[5,2]$ является выходным ФД.

Составим СЛАУ (4):

$$\begin{matrix} k+j+i=5 \\ j \quad =2 \end{matrix}$$

Заметим, что для перебора значений индекса i значения других индексов не нужны, т.е. область определения i не зависит от других индексов. Чтобы перебрать значения j, должна быть известна величина i, а для перебора k необходимы значения j, а значит и i. Поэтому, индексы в СЛАУ расположены в порядке топологической сортировки относительно зависимости значений индексов друг от друга. Сначала берутся более зависимые переменные (k), потом менее (j, а после – i).

Решив СЛАУ, получаем:

1. i – свободная переменная
2. $j=2$
3. $k=3-i$

Перебор значений индексов i,j,k схематически изображен на рисунке 8.

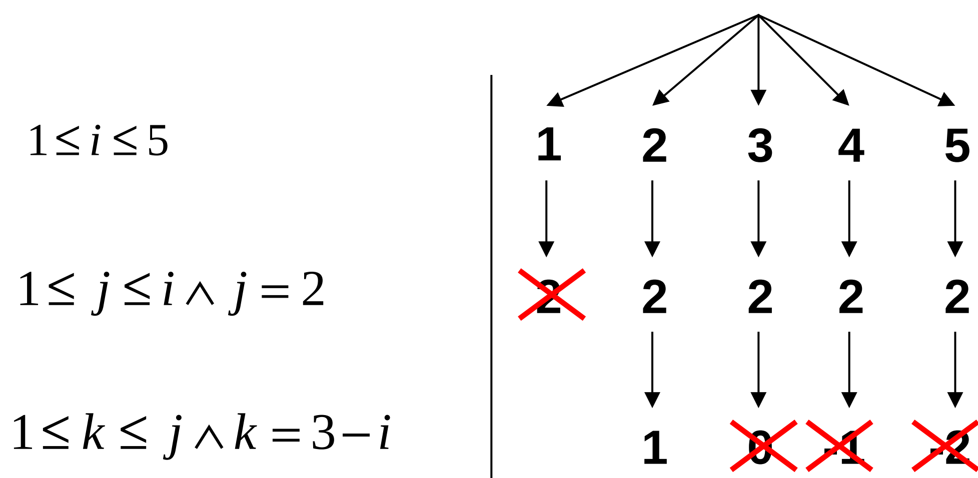


Рисунок 8. Перебор значений i, j, k

Ветвь перебора с индексами $i=1, j=2$ отбрасывается, т.к. не выполняется неравенство $j \le i$. Тройки значений индексов $(3,2,0)$, $(4,2,-1)$, $(5,2,-2)$ отвергаются, т.к. нарушено неравенство $1 \le k$. Единственное корректное означивание индексов $i=2, j=2, k=1$. Следовательно, результатом запроса будет ФВ $A[2,1,2]$.

1.2.2 Алгоритмы выполнения запросов №1 и №3

Запросы №1 и №3 выполняются аналогично запросу №2.

1.2.3 Алгоритм выполнения запроса №4

Рассмотрим алгоритм выполнения четвертого запроса: нахождение начальных ФВ и ФД. Этот запрос имеет наибольшую вычислительную трудоемкость, но выполняется всего один раз – непосредственно в начале исполнения ФП. Начальное множество ФВ и ФД вычисляется в три шага:

1. находится всё множество CF, путем перебора возможных значений индексных переменных для каждого правила, описывающего ФВ;
2. для каждого cf из CF определяется, существует ли cf', который должен исполниться непосредственно до cf, т.е. $\rho(cf', cf)$. Данная задача решается с помощью проверки истинности утверждения $\left|CF_{before}(cf)\right| \ne 0$, что осуществляется с помощью 3-его запроса. Те cf, для которых не существует такого cf', добавляются в начальное множество ФВ;

3. Для каждого ФВ из начального множества по уже известным алгоритмам находится множество ФД, в которых он нуждается. Множество всех таких ФД составляет начальное множество ФД.

2. Реализация поддержки индексированного описания фрагментированной программы в системе LuNA

Система фрагментированного программирования LuNA состоит из компилятора и исполнительной системы (run-time системы). Задача компилятора состоит в том, чтобы преобразовать текстовое описание фрагментированной программы на языке LuNA в промежуточное представление ФП, с которым непосредственно будет работать ИС. Задача исполнительной системы заключается в исполнении ФП, заданной в промежуточном представлении, с помощью модуля поддержки индексированного представления ФП.

Было принято решение представлять ФП в ИС в виде экземпляра класса *IndexedFP*, члены которого есть списки правил вида I-III, а методы разделены на две группы:

1. добавляющие индексированные правила в экземпляр класса,
2. реализующие запросы №1-4.

Таким образом, промежуточным представлением индексированной ФП является процедура, которая инициализирует экземпляр класса *IndexedFP* всеми теми правилами, которые присутствуют в исходном коде ФП.

2.1. Язык описания фрагментированных программ LuNA

Язык фрагментированного программирования LuNA (**L**ang**u**age for **N**umeric **A**lgorithms) является декларативным языком, служащим для описания фрагментированных программ. Каждая строка в программе на языке LuNA представляется в виде:

```
E | <индекс1>=IntE..IntE, ..., <индексn>=IntE..IntE;
```

где *E* – выражение, зависящее от индексных переменных $<индекс_1>$,..., $<индекс_n>$, *IntE* – выражение, которое принимает целые значения и может

содержать другие индексы. После черты описывается множество значений, которые могут принимать индексные переменные. Таким индексированным образом описывается сразу множество сущностей типа *E*, которые отличаются друг от друга только значениями индексов. *E* может и не зависеть ни от каких индексных переменных, тогда используется такая запись:

```
E ;
```

Всего рассмотренным способом в языке LuNA можно описать 4 сущности:

1. константы – индексированные и не индексированные, определенные на этапе компиляции и не определенные, например:

```
const N = 100;
const M = 2*N;
const K;
cont P[i] = i*i | i=1..N;
```

Если значение константы не известно заранее, то оно может быть определено непосредственно перед исполнением ФП, например, прочитано из файла.

2. фрагменты данных с указанием количества требуемых байт памяти, что соответствует правилу III:

```
df NORM := block(4);
df SQ[i] := block(4) | i=1..N;
df A[i] := block(4) | i=1..N;
df B[i,j] := block(4) | i=1..N, j=i..N;
df SUM := block(4);
```

3. фрагменты вычислений (правило I):

```
cf init_by_zero := InitByZero(out: SUM);
cf norm := Sqrt(in: SUM; out: NORM);
cf sum[i] := Sum(in: SQ[i], SUM; out: SUM) | i=1..N;
cf square[i] := Square(in: A[i]; out: SQ[i]) | i=1..N;
```

где `InitByZero, Sqrt, Sum, Square` – фрагменты кода, которые реализуют выполнение ФВ и определяются с помощью процедур на языке высокого уровня. Сигнатура таких процедур определяется количеством входных и выходных ФД для рассматриваемого ФВ.

Например, процедура *Sum*, будучи реализованной на языке *C++* имеет сигнатуру

```
void Sum(void* in1, void* in2, void* out1)
```

При выполнении процедуры указатель `in1` будет указывать на ФД `SQ[i]` для некоторого `1≤i≤N`, а `in2` и `out1` на `SUM`.

4. отношение порядка (правило II):

```
init_by_zero < sum[i] | i=1..N;
square[i] < sum[i] | i=1..N;
sum[i] < norm | i=1..N;
```

где первый пример эквивалентен в математической записи выражению $\forall i:1\le i\le N\Rightarrow\rho\left(init_by_zero,sum\left[i\right]\right)$.

Пример ФП умножения матриц:

```
1 const N=5;
2 const B=5;
3 df X[i,j] := block(8*B*B) | i=..N, j=..N;
4 df Y[i,j] := block(8*B*B) | i=..N, j=..N;
5 df Z[i,j,k] := block(8*B*B) | i=..N, j=..N, k=..N;
6 df C[i,j] := block(8*B*B) | i=..N,j=..N;
7 cf ini[i,j] := ini<B,0>(out: C[i,j]) | i=..N, j=..N;
8 cf iniX[i,j] := ini<B,1>(out: X[i,j]) | i=..N, j=..N;
9 cf iniY[i,j] := ini<B,2>(out: Y[i,j]) | i=..N, j=..N;
10 cf mul[i,j,k] := mul<B,k>(in: X[i,k], Y[k,j]; out:
Z[i,j,k]) | i=..N, j=..N, k=..N;
11 cf sum[i,j,k] := sum<B,k>(in: C[i,j], Z[i,j,k]; out:
C[i,j]) | i=..N, j=..N, k=..N;
12 iniX[i,k] < mul[i,j,k] | i=..N, j=..N, k=..N;
13 iniY[k,j] < mul[i,j,k] | i=..N, j=..N, k=..N;
14 mul[i,j,k] < sum[i,j,k] | i=..N, j=..N, k=..N;
15 ini[i,j] < sum[i,j,k] | i=..N, j=..N, k=..N;
```

Программа умножает две квадратные матрицы, представленные фрагментами данных с идентификаторами *X* и *Y*, и записывает результат умножения в фрагменты *C* (рисунок 9). Каждая матрица разбивается `N*N` на блоков, каждый из которых представляет квадратную матрицу размера `B*B` с 8-байтными элементами (строки 3-6). Фрагменты вычислений `ini[i,j]`, `iniX[i,j]`, `iniY[i,j]` реализуют начальную инициализацию фрагментов

данных `C[i,j]`, `X[i,j]` и `Y[i,j]` соответственно (8-10). ФВ `mul[i,j,k]` вычисляет произведение матриц, соответствующих ФД `X[i,k]` и `Y[k,j]`, которое потом прибавляется к матрице `C[i,j]` с помощью ФВ `sum[i,j,k]` (10-11). Отношение порядка выражает информационные зависимости (12-15).

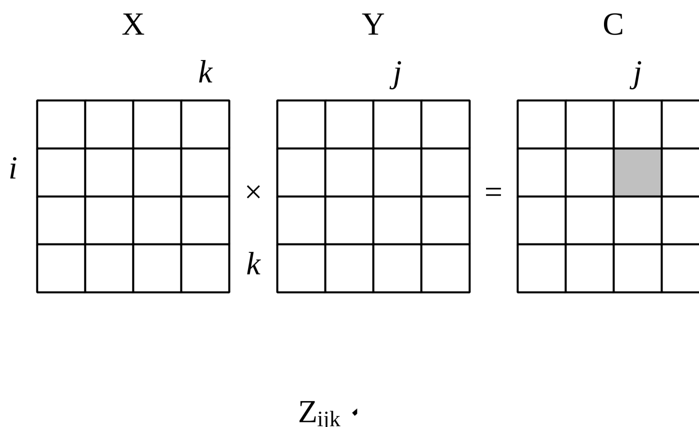


Рисунок 9. Схема фрагментированной программы

2.2. Особенности реализации компилятора

Компилятор выполняет следующие действия:

1. читает файл с описанием ФП на языке LuNA
2. производит лексический и синтаксический разбор в соответствии с грамматикой языка LuNA (**приложение А**)
3. строит абстрактное синтаксическое дерево, которое затем переводит в промежуточное представление – процедуру, инициализирующую экземпляр класса *IndexedFP*.

Было принято решение реализовать компилятор на языке Haskell и осуществлять лексический и синтаксический разборы средствами библиотеки монадических комбинаторных парсеров Parsec [34,35]. Данное проектное решение обусловлено следующими факторами.

1. Алгебраических типов данных (АТД) в языке Haskell позволяют удобно описывать рекурсивные типы данных, в частности, абстрактные синтаксические деревья (АСД), которые являются результатом работы синтаксического анализатора. Пример АТД для простого языка арифметических

выражений с целочисленными константами и операциями сложения и вычитания:

```
data Expr = Const Int | Sum Expr Expr | Mul Expr Expr
```

Тогда АСД, соответствующее выражению *2*(1+5)*, будет

```
Sum (Const 2) (Mul (Const 1) (Const 5))
```

Механизм сопоставления с образцом (pattern matching) позволяет деконструировать значения типа `Expr` на составляющие части, что необходимо для осуществления преобразований или анализа АСД, представленных типом `Expr`. Пример функции, вычисляющей значение АСД:

```
eval :: Expr -> Int
eval (Sum a b) = eval a + eval b
eval (Mul a b) = eval a * eval b
eval (Const x) = x
```

Тогда

```
ghci> eval (Sum (Const 2) (Mul (Const 1) (Const 5)))
12
```

2. Библиотека Parsec позволяет осуществлять разбор контекстно-свободных грамматик. При этом код при использовании Parsec лаконичен, выразителен и похож на описание реализуемой грамматики в форме Бэкуса-Наура. Кроме того, Parsec позволяет совместить в коде фазу лексического и синтаксического анализа, что удобно для программиста.

Приведем пример использования Parsec для разбора языка арифметических выражений из пункта 1, грамматика которого:

```
exp -> exp '+' exp
exp -> exp '*' exp
exp -> '(' exp ')'
exp -> integer
```

Реализация парсера, который возвращает АСД типа `Expr`:

```
exp :: Parser Expr
exp = add <|> mul <|> parens <|> int
```

add :: Parser Expr
add = do
     a <- exp
     char '+'
     b <- exp
     return (Sum a b)
	mul :: Parser Expr
	mul = do
	     a <- exp
	     char '*'
	     b <- exp
	     return (Mul a b)
		parens :: Parser Expr
		parens = do
		     char '('
		     e <- exp
		     char ')'
		     return e

```
int :: Parser Int
int = do
 n <- many1 digit
 return (Const (read n))
```

С помощью описанных парсеров можно разобрать строку «*2*(1+5)*»:

```
ghci> parse exp "" "2*(1+5)"
Right (Sum (Const 2) (Mul (Const 1) (Const 5)))
```

2.3. Реализация индексированного представления фрагментированной программы в исполнительной системе

Запросы №1-4 реализованы в виде методов класса *IndexedFP* (**приложение B**):

```
class IndexedFP
{
...
public:
 ...
 DfByCfIterator GetDfByCf(const fInstance& cf);
 CfByDfIterator GetCfByDf(const fInstance& df);
 CfByCfIterator GetCfByCf(const fInstance& cf, cfRelation rel);
 AllCfIterator GetAllCf();
 StartFragmentIterator GetStartFragments();
};
```

Функция `GetDfByCf` реализует запрос №1. Функция возвращает итератор типа `DfByCfIterator`, который позволяет перечислить входные и выходные ФД для `cf` – первого аргумента функции. `cf` имеет тип `fInstance`, который

позволяет описывать фрагменты данных или вычислений, а именно, он хранит строковый идентификатор фрагмента name, набор целых индексов index.

```
class fInstance
{
private:
 dfType    m_dfType;
 fType     m_fType;
public:
 vInt      index;
 string    name;
};
```

Поле m_fType в структуре fInstance указывает на тип фрагмента – ФД или ФВ.

```
enum fType {
 UNDEF_FTYPE = 0,
 CF = 1,
 DF = 2
};
```

Поле m_dfType обозначает тип ФД – входной и/или выходной.

```
enum dfType
{
 UNDEF_DFTYPE = 0,
 IN = 1,
 OUT = 2,
 IN_OUT = (2|1)
};
```

Функция GetCfByDf реализует запрос №2 – для определенного ФД df возвращает итератор типа CfByDfIterator, перечисляющий фрагменты вычислений, для которых df является входным или выходным.

Функция GetCfByCf реализует запрос №3 – для данного ФВ cf возвращает множество ФВ, которые должны исполниться непосредственно до или после cf, что определяется вторым аргументом rel типа cfRelation.

```
enum cfRelation
{
 BEFORE = 0,
 AFTER
```

```
};
```

Функция `GetStartFragments` реализует запрос №4 – получение множества всех ФВ, которые могут исполниться первыми и множества ФД, которые нужны этим ФВ. Функция `GetAllCf` позволяет перечислить все фрагменты вычислений в ФП, что используется в запросе №4.

2.4. Механизм порционной выдачи результатов запросов

Результаты запросов №1-4, выполняемых в процессе исполнения ФП, потенциально бесконечны, поэтому целесообразно получать результирующее множество ФВ (ФД) не целиком, но по частям, содержащим определенное конечное число фрагментов, передаваемое вместе с запросом.

Реализовать порционную выдачу результатов возможно было бы с помощью оператора *yield*, реализованного в том или ином виде как часть языка в Ruby, Python, C#, F#, JavaScript [37] и др. Когда поток исполнения достигает оператора *yield*, исполнение функции прерывается и исполнение возвращается к вызывающей функции. В этом отношении *yield* похож на обыкновенный возврат из функции – *return*. Однако, при использовании *yield*, контекст (стек) прерванной функции сохраняется и ее исполнение может быть возобновлено с места прерывания.

В языке C++ оператор *yield* не представлен, поэтому было принято решение реализовать порционную выдачу результатов вручную с помощью паттерна *Итератор*, что значит, искусственное хранение контекста функции в итераторе и его восстановление при необходимости.

Таким образом, для каждого фрагмента, относительно которого совершался запрос и не был целиком возвращен, хранится множество параметров $i_1, i_2, \ldots, i_p$, на котором прервался перебор в процедуре **Поиск**, что позволяет при повторном запросе продолжить перебор с того момента, когда он закончился в предыдущий раз.

Модифицированный алгоритм **Поиск**II изменяет значения следующих глобальных переменных:

- *blockSize* – переменная изначально задает, сколько решений требуется возвратить – размер «порции», если значение blockSize в конце работы процедуры больше нуля, значит все решения исчерпаны;
- *state* – это состояние итератора (контекст алгоритма **ПоискII)**, содержащее множество параметров $i_1, i_2, \ldots, i_k$, на котором прервался перебор ранее;
- *down* – булева переменная, которая позволяет отслеживать в каких пределах нужно перебирать текущий индекс, если down Истина, то начиная со значения, записанного в контексте, иначе с нижнего предела изменения индекса;
- *sol* – вектор, в который добавляются найденные решения.

ПоискII(*index*)

 # $1 \le index \le p$ – номер индекса, значение которого перебирается

Если ($index > p$)

Если (down==True)

down = False

Выход из процедуры

Иначе

Если $\forall t \in [1, N_{m'}] : nl'_t(\overbrace{state}) = j_{n'+t}$

blockSize = blockSize – 1

Добавить state в sol

Иначе

up = $U_{index}(\overbrace{state})$

low = $L_{index}(\overbrace{state})$

Если (down==True)

low = $state_{index}$

Если (*index–ый* индекс свободен)

Для всех целых $j \in [low, up]$ **пока** blockSize > 0

$state_{index} = j$

ПоискII(*index*+1)

Иначе

$state_{index} = t_{index}\left(\overrightarrow{state}\right)$

Если ($state_{index} \in Z \wedge low \le state_{index} \le up$)

ПоискII(*index*+1)

Пример использования алгоритма:

blockSize = 10

state = вектор из k элементов

down = False

Повторять

sol = []

ПоискII(1)

печать sol

Пока blockSize == 0

В классе *IndexedFP* механизм порционной выдачи результатов для запросов, реализован с помощью паттерна проектирования *Итератор*. Например, метод `GetDfByCf` класса *IndexedFP* возвращает итератор типа `DfByCfIterator`, который определен следующим образом:

```
class DfByCfIterator
{
...
public:
     vector<fInstance> GetNext(int blockSize);
     vector<fInstance> GetRest() { return GetNext(maxInt); }
};
```

Данный класс содержит метод `GetNext`, принимающий количество фрагментов, которое требуется возвратить и возвращает вектор из этих фрагментов. Метод `GetRest` возвращает все оставшиеся фрагменты.

2.5. Ограничения реализации

2.5.1. Сложность поддержки новых расширений языка LuNA

При добавлении новых конструкций языка сложно модифицировать модуль *IndexedFP* – требуется реализовывать множество новых запросов и итераторов им соответствующих. Эту работу можно автоматизировать с помощью создания специального языка запросов к структуре данных ФП и транслятора запросов в C++ код для модуля *IndexedFP*.

2.5.2. Высокая вычислительная сложность запросов к модулю IndexedFP

Алгоритмы **Поиск**, **ПоискII** рекурсивны, т.к. глубина рекурсии не константа и для разных запросов может быть разной. Кроме того, алгоритмы при каждом запросе заново осуществляются вычисления, которые общи для множества запросов, а именно приведение матрицы к верхнедиагональному виду.

Оба недостатка можно исправить, если возложить на компилятор задачу по генерации специализованной версий алгоритмов для каждого типа запросов. В таком случае, рекурсию можно развернуть в циклы (т.к. глубина рекурсии будет известна для конкретного типа запросов), а матрицу выразить прямо в коде запроса, а не интерпретировать ее, как было прежде. Такого рода оптимизация, переносит часть вычислений из времени исполнения в код программ за счет его увеличения. Описанный подход носит название *смешанных вычислений* [38].

2.5.3. Фиксированный порядок выдачи результатов запроса

Результаты запросов выдаются с порядке, фиксированном в алгоритме **ПоискII**. Однако, для эффективного исполнения ФП может потребоваться специальный порядок:

- в запросах можно выдавать первыми фрагменты с наивысшим приоритетом (который задается пользователем в тексте ФП) или те, что наиболее близки (в смысле расстояния на графе ФП) к исполняющимся в данный момент фрагментам вычислений;
- если начали исполняться ФВ из группы[1], то остальные ФВ из этой же группы должны быть назначены на исполнение ранее других, а следовательно быть первыми в результатах запросов.

2.6. Пути развития

[1] Группа – это множество ФВ и ФД, относительно независимое от остальных фрагментов ФП, в котором сосредоточены информационные зависимости и для которой существует простой способ исполнения – блочно-последовательный или конвейерный. Группу выгодно исполнять целиком, так что промежуточные ФД в группе производятся и сразу потребляются. Группу удобно рассматривать как единицу при распределении ресурсов. Задача выявления групп в ФП сложна и на данный момент не решена.

В дальнейшие планы входит устранение имеющихся ограничений реализации, а также воплощение ряда предложений.

Спекулятивное выполнение запросов, основанное на принципе локальности запросов по аналогии с кэш-памятью. При выполнении запросов относительно определенного ФВ *A* велика вероятность, что следующие запросы будут относительно фрагментов из некоторой окрестности А в графе ФП. Следовательно, можно заранее обработать ряд запросов еще до того, как они потребуются исполнительной системе.

Оптимизация выполнения запроса на получение множества начальных ФВ.

Алгоритм выполнения запроса 4. наиболее ресурсоемок, он требует вычисление всего множества CF. Предлагается следующий алгоритм:

1. $CF_{st} = CF$
2. берется очередное правило, определяющее отношение ρ. Для него вычисляется множество ФВ, для которых есть ФВ – родитель в смысле графа ρ. Можно надеяться, что данное множество выразимо в индексированном виде. Обозначим его как CF'.
3. $CF_{st} = CF_{st} / CF'$
4. переход на шаг b., если множество правил, определяющих ρ не исчерпано

В итоге результат запроса 4. есть множество CF_{st}, определенное в индексированном виде. Алгоритм применим, если вид арифметических выражений в тексте ФП ограничен линейными выражениями от индексных переменных.

Проверка существования целочисленного решения СЛАУ при выполнении запросов

Еще до решения СЛАУ можно проверить существует ли целочисленное решение СЛАУ, удовлетворяющее ограничениям, представленных линейными функциями, из (5). Для этого можно применить ряд простых приближенных тестов, использующихся в задаче определения зависимостей по данным, таких

как, НОД-тест, обобщенный тест Банержи, которые обладают линейной вычислительной сложностью [3]. Такие тесты особенно полезны при выполнении запроса 4. на нахождение множества начальных ФВ, где нужно определить существуют ли ФВ, связанные с данным отношением ρ, т.е. определить существует ли целочисленное ограниченное решение системы (6).

Оптимизация алгоритма Поиск для нелинейных выражений с операцией деления по модулю.

Поддержка нелинейных выражений в индексах фрагментов добавлена в первую очередь из-за удобства использования операции деления по модулю при описании ФП в ряде задач.

В таких задачах индексные выражения обычно имеют вид *<линейное функция>%<константа>*, где «%» – операция взятия по модулю. Пусть необходимо получить множество ФВ $B[i,j]$, которые нуждаются в ФД $A[4]$:

```
cf B[i,j] := codeB(in: A[(i+j)%N]) | i=1..M, j=1..M;
```

В текущей реализации алгоритма **Поиск** решение задачи сводится к перебору все пар индексов из множества $\{(i,j) | 1 \le i,j \le M \wedge i \in \mathbb{Z} \wedge j \in \mathbb{Z}\}$ и отбору тех пар, для которых выполняется $(i+j)\%N = 4$. Однако, перебор можно оптимизировать, если учесть, что $i+j$ может принимать значения $4, 4+N, 4+2N, \ldots, 4+kN, \ldots$.Таким образом, исходная задача разбирается на ряд подзадач относительно ФД $A[4]$, $A[4+N]$, $A[4+2N]$ и т.д., при этом нелинейное выражение $(i+j)\%N$ заменяется на линейное – $i+j$. Множество перебора сокращается в N раз.

3. Тестирование

Было проведено тестирование работоспособности компилятора и программного модуля поддержки индексированного представления ФП на множестве имеющихся фрагментированных программ.

3.1. Компилятор

Продемонстрируем работу компилятора (**приложение С**) на примере ФП:

```
const N=10;
df a[i] := block(4) | i=1..N;
df c := block(4);
cf b[i] := echo(in: a[i]; out: c) | i=1..N;
b[i] < b[i+1] | i=1..N-1;
```

Компилятор переводит каждую строку программы LuNA в вызовы методов класс *IndexedFP*, которые добавляют ту или иную сущность в экземпляр этого класса.

```
bool InitFP_noname(IndexedFP& fp)
{
 {
 //добавление в индексированное описание ФП fp
      //правила для константы
      Constant& entity =
      fp.AddConstant("N", vExpr(), vExpr() <<
Expression(fp,10), STATIC);
 }
 {
 //добавление правил, описывающих ФД
      Property& entity =
           fp.AddProperty("df_size", "a",
                vExpr() << Expression(fp,"i", 0),
                vExpr() << Expression(fp,4));
      entity.AddRange("i",0,make_pair(Expression(fp,1),
                     Expression(fp,Variable("N"))));
 }
 {
      Property& entity =
           fp.AddProperty("df_size","c",vExpr(),
                vExpr() << Expression(fp,4));
 }
 {
      //добавление правила для ФВ
```

```
        RuleCf& entity = fp.AddRuleCf();
        //задание идентификатора ФВ и его индексных переменных
        entity.SetCf("b", vExpr() << Expression(fp,"i", 0));

        //входные и выходные ФД
        entity.AddDf(IN, "a", vExpr() << Expression(fp,"i", 0));
        entity.AddDf(OUT, "c", vExpr());

        //задание фрагмента кода – имя процедуры, реализующей ФВ
        entity.SetCodeName("echo");
        entity.SetParameter(vExpr());
        //пределы изменения индексных переменных
        entity.AddRange("i",0,make_pair(Expression(fp,1),
                       Expression(fp,Variable("N"))));
    }
    {
        //задание отношения порядка
        Relation& entity = fp.AddRuleOrder();
        //идентификатор и набор индексных
        //переменных для первого ФВ
        entity.SetFirst("b", vExpr() << Expression(fp,"i", 0));
        //идентификатор и набор индексных переменных для
        //второго ФВ, который может исполниться только после
        //завершения исполнения первого ФВ
        entity.SetSecond("b", vExpr() << Expression(fp,
                 Expression(fp,"i", 0), Add, Expression(fp,1)));
        //пределы изменения индексных переменных
        entity.AddRange("i",0,make_pair(Expression(fp,1),

        Expression(fp,Expression(fp,Variable("N")), Sub,
Expression(fp,1))));
    }
    return true;
   }
```

3.2. Методы класса *IndexedFP*

Покажем работоспособность методов *IndexedFP*, реализующих запросы №1-4, на примере фрагментированной программы умножения матриц, которая представлена в параграфе «2.1. Язык описания фрагментированных программ LuNA».

Для тестирования работоспособности методов *IndexedFP*, отвечающих за реализацию запросов №1-4, разработана интерактивная программа. Ниже представлена сессия данной программы, в которую загружена ФП умножения матриц, с пользователем.

```
Выберите действие:
:: [0] Выход
:: [1] Получить ФВ, нуждающиеся в определенном ФД
:: [2] Получить входные и выходные ФД для определенного ФД
:: [3] Получить ФВ, которые должны выполниться непосредственно до [0] или после [1] определенного ФВ
:: [4] Получить начальное множество ФВ
> 1
Введите фрагмент данных.
Опишите фрагмент в виде: <имя фрагмента> <индекс_1> <индекс_2> ... <индекс_n>
> C 1 1
Введите количество требуемых фрагментов: сразу все [-1], отмена [0], определенное количество [натуральное число]
> -1
[sum[1, 1, 0], sum[1, 1, 1], sum[1, 1, 2], sum[1, 1, 3], sum[1, 1, 4], ini[1, 1]]
Выберите действие
> 2
Введите фрагмент вычислений.
Опишите фрагмент в виде: <имя фрагмента> <индекс_1> <индекс_2> ... <индекс_n>
> mul 3 2 1
[in: X[3, 1], in: Y[1, 2], out: Z[3, 2, 1]]
Выберите действие
> 3
Введите фрагмент вычислений.
Опишите фрагмент в виде: <имя фрагмента> <индекс_1> <индекс_2> ... <индекс_n>
> ini 2 1
Какие ФВ требуется получить, те что должны исполниться ДО[0] данного или ПОСЛЕ[1]?
> 0
Введите количество требуемых фрагментов: сразу все [-1], отмена [0], определенное количество [натуральное число]
> -1
[sum[2, 1, 0], sum[2, 1, 1], sum[2, 1, 2], sum[2, 1, 3], sum[2, 1, 4]]
Выберите действие
> 4
Введите количество ФВ, которое вы хотите получить: сразу все [-1], отмена [0], определенное количество [натуральное число].
> -1
cf = [iniY[0, 0], iniY[0, 1], iniY[0, 2], iniY[0, 3], iniY[0, 4], iniY[1, 0], ..., ini[3, 1], ini[3, 2], ini[3, 3], ini[3, 4], ini[4, 0], ini[4, 1], ini[4, 2], ini[4, 3], ini[4, 4]]
```

```
df = [out: C[0, 0], out: C[0, 1], out: C[0, 2], out: C[0, 3], out: C[0, 4], out: C[1, 0], out: C[1, 1], out: C[1, 2], out: C[1, 3], out: C[1, 4], ..., out: Y[3, 1], out: Y[3, 2], out: Y[3, 3], out: Y[3, 4], out: Y[4, 0], out: Y[4, 1], out: Y[4, 2], out: Y[4, 3], out: Y[4, 4]]
Выберите действие
> 0
```

Заключение

В работе рассмотрена проблема поддержки индексированного представления фрагментированных программ в системе LuNA. Сделан обзор возможных решений данной проблемы.

Разработаны алгоритмы извлечения требуемой для исполнения ФП информации из ее индексированного представления.

Реализован компилятор и программный модуль *IndexedFP* для исполнительной системы LuNA, реализующий предложенные алгоритмы. Тестирование показало их работоспособность.

Основными защищаемыми положениями работы являются:

1. Система требований к индексированному представлению ФП в ИС и компиляторе;
2. Алгоритмы реализации запросов к индексированному представлению ФП;
3. Компилятор с языка LuNA и модуль *IndexedFP* в ИС.

Список использованных источников

1. Grama A. Introduction to Parallel Computing / A. Grama, G. Karypis, V. Kumar, A. Gupta. – Second Edition. – Addison Wesley, 2003. – 856 с.
2. Sutter H. «The free lunch is over: A fundamental turn toward concurrency in software» // Dr. Dobb's Journal. – 2005. – Vol. 30, Issue 3. – P. 16-22
3. Вальковский В.А. Синтез параллельных программ и систем на вычислительных моделях. // В.А. Вальковский, В.Э. Малышкин. – Новосибирск : Наука, 1988. – 128 с.
4. Малышкин В.Э. Параллельное программирование мультикомпьютеров. / В.Э. Малышкин. – Новосибирск : Изд-во НГТУ, 2006. – 296 с.
5. Charm++ [Электронный ресурс]. – Режим доступа: http://charm.cs.uiuc.edu
6. ProActive – Professional Open Source Middleware for Parallel, Distributed, Multi-core Programming [Электронный ресурс]. – Режим доступа: http://proactive.inria.fr/
7. SMP Superscalar [Электронный ресурс]. – Режим доступа: http://www.bsc.es/smpsuperscalar/
8. Berzins M. Uintah: A Scalable Framework for Hazard Analysis / M. Berzins, J. Luitjens, Q. Meng, T. Harman, C.A. Wight, J.R. Peterson // Proc. of the Teragrid 2010 Conference, Pittsburgh, PA, USA —2 – 5 August 2010. – No. 3.
9. Intel® Threading Building Blocks [Электронный ресурс]. – Режим доступа: http://threadingbuildingblocks.org/
10. PLASMA Library [Электронный ресурс]. – Режим доступа: http://icl.cs.utk.edu/plasma/
11. Bosilca G. Distibuted Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA / G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra // University of Tennessee Computer Science Technical Report, UT-CS-10-660, Sept. 15, 2010.

12. Armstrong J. Concurrent Programming in Erlang / J. Armstrong, R. Virding, C. Wikström, M. Williams // 2nd Edition Prentice Hall, 1996, ISBN 0-13-508301-X
13. The Go Programming Language [Электронный ресурс]. – Режим доступа: http://golang.org
14. Malyshkin V.E. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem / V.E. Malyshkin, V.A. Perepelkin // Proceedings of the 11th Conference on Parallel Computing Technologies.– Springer, 2011. – Vol. 6873 of Lecture Notes in Computer Science. – P. 53-61
15. Kireev S. The LuNA Library of Parallel Numerical Fragmented Subroutines / S. Kireev, V. Malyshkin, H. Fujita // Proceedings of the 11th Conference on Parallel Computing Technologies.– Springer, 2011. – Vol. 6873 of Lecture Notes in Computer Science. – P. 290-301
16. Malyshkin V. E. Run-time system for parallel execution of fragmented subroutines. / V. E. Malyshkin [и др.] // Proceedings of the 9th International conference on Parallel Computing Technologies. – Springer, 2007. – Vol. 4671 of Lecture Notes in Computer Science. – P. 544-552.
17. Мальцев А.И. Алгоритмы и рекурсивные функции / А.И. Мальцев. – 2-е изд. М.: Наука, 1986. – 368с.
18. Задыхайло И.Б. Составление циклов по параметрическим записям специального вида / И.Б. Задыхайло // Журнал вычислительной математики и математической физики, 1963 – Т. 3, №.2. – С. 337-357
19. Pnueli A. Realizing an Equational Specification / A. Pnueli, R. Zarhi // Proceedings of the 8th International Colloquium on Automata, Languages, and Programming (ICALP 1981). – Springer-Verlag, 1981. – Vol. 115 of Lecture Notes in Computer Science. – P. 459-478.
20. The DOT Language [Электронный ресурс]. – Режим доступа: http://www.graphviz.org/content/dot-language
21. The GraphML File Format [Электронный ресурс]. – Режим доступа: http://graphml.graphdrawing.org/specification.html

22. Trivial Graph Format [Электронный ресурс]. – Режим доступа: http://en.wikipedia.org/wiki/Trivial_Graph_Format
23. The Haskell Programming Language [Электронный ресурс]. – Режим доступа: http://www.haskell.org/haskellwiki/Haskell
24. Acyclic lists and innumerable trees in Scheme [Электронный ресурс]. – Режим доступа: http://okmij.org/ftp/Computation/uncountable-sets.html
25. Андрианов А.Н. НОРМА. Описание языка. Рабочий стандарт. / А.Н. Андрианов, А.Б. Бугеря, К.Н. Ефимкин, И.Б. Задыхайло – 52 с. – (Препринт ИПМ им. М.В. Келдыша РАН, 1995, № 120)
26. Андрианов А.Н. Синтез параллельных и векторных программ по непроцедурной записи в языке Норма : дис. канд. физ.-мат. наук. – М., 1990. – 144 с.
27. SETL Documentation [Электронный ресурс]. – Режим доступа: http://cs.nyu.edu/bacon/setl-doc.html
28. Brailsford S. Constraint satisfaction problems: Algorithms and applications / S. Brailsford, C. Potts, B. Smith // European Journal of Operational Research. – 1999. – Vol. 119. – P. 557-581.
29. Russell S. Artificial Intelligence – A Modern Approach / S. Russell, P. Norvig // Prentice Hall, 3rd edition, 2009. – p. 1152.
30. Gecode – Generic constraint development environment [Электронный ресурс]. – Режим доступа: http://www.gecode.org/
31. Minion – Fast, Scalable Constraint Solver [Электронный ресурс]. – Режим доступа: http://minion.sourceforge.net/
32. The Mozart Programming System [Электронный ресурс]. – Режим доступа: http://www.mozart-oz.org/
33. AMPL – A Modeling Language for Mathematical Programming [Электронный ресурс]. – Режим доступа: http://www.ampl.com/
34. The Parsec package [Электронный ресурс]. – Режим доступа: http://hackage.haskell.org/package/parsec-3.1.1

35. Hutton G. Monadic parser combinators / G. Hutton, E. Meijer // Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
36. Евстигнеев В.А. Анализ зависимостей: основные тесты на зависимость по данным / В.А. Евстигнеев, Р.Н. Арапбаев, Р.А. Осмонов // Сиб. журн. вычисл. математики СО РАН. – Новосибирск, 2007. – Т. 10 №3. – С 247-265.
37. James R. Yield: Mainstream Delimited Continuations / R. James, A. Sabry // Informal proceedings of the 2011 International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011). – Novi Sad, Serbia, 29–30 May 2011. – P. 20-32.
38. Ершов А.П. Организация смешанных вычислений для рекурсивных программ // Докл. АН СССР. — 1979. — Т. 245 — № 5. — С. 1041–1044.

Приложение А. Грамматика языка LuNA

```
KW_CF          = cf
KW_DF          = df
KW_IN          = in
KW_OUT         = out
KW_BLOCK       = block
KW_CONST       = const
KW_NEIGHBOURS  = neighbours
KW_PROGRAM     = program

ASSIGN    = ":="
EQ        = "="
LT        = "<"
GT        = ">"
SCOLON    = ";"
PERCENT   = "%"
COLON     = ":"
QMARK     = "?"
COMMA     = ","
DOT  = "."
DIAP = ".."
LB   = "("
RB   = ")"
LSB  = "["
RSB  = "]"
PLUS = "+"
MINUS= "-"
MUL  = "*"
DIV  = "/"
PIPE = "|"
DBLEQ= "=="
LEQ  = "<="
GEQ  = ">="
NEQ  = "!="
```

```
INT  ~ [0-9]+
REAL ~ [0-9]+\.[0-9]+
NAME ~ [A-Za-z_][A-Za-z0-9_]*

named_program ->
     KW_PROGRAM NAME SCOLON program |
     KW_PROGRAM NAME glob_args SCOLON program |
     program
glob_args ->
     LB arg_list RB
program ->
     program_item |
     program program_item
program_item ->
     SHARP NAME EQ NAME |
     simple_program_item SCOLON |
     simple_program_item PIPE subst_list SCOLON
subst_list ->
     subst |
     subst_list COMMA subst
subst ->
     NAME EQ expr DIAP expr |
     NAME EQ DIAP expr |
     NAME EQ expr
simple_program_item ->
     const | data_fg | comp_fg |
     order | neighbours | property |
     g_in |    g_out |
g_in ->
     KW_IN COLON name_list
g_out ->
     KW_OUT COLON name_list
neighbours ->
     KW_NEIGHBOURS id COMMA id
property ->
```

```
     id DOT NAME EQ expr
const ->
     KW_CONST id EQ expr
data_fg ->
     KW_DF id ASSIGN KW_BLOCK LB expr RB
comp_fg ->
     KW_CF id ASSIGN NAME opt_param_list LB arg_list RB
opt_param_list ->
     LT expr_list GT |
     epsilon
order ->
     id LT id
arg_list:
     KW_IN COLON name_list SCOLON KW_OUT COLON name_list |
     KW_IN COLON name_list |
     KW_OUT COLON name_list |
     epsilon
name_list ->
     id | name_list COMMA id
id ->
     NAME |
     NAME LSB RSB |
     NAME LSB expr_list RSB
expr_list ->
     expr |
     expr_list COMMA expr
expr ->
     INT |
     NAME |
     NAME LSB expr_list RSB |
     NAME DOT NAME |
     NAME LSB expr_list RSB DOT NAME |
     expr PLUS expr |
     expr MINUS expr |
     expr MUL expr |
     expr DIV expr |
```

```
expr PERCENT expr |
LB expr RB |
LB expr LT expr RB |
LB expr GT expr RB |
LB expr DBLEQ expr RB |
LB expr LEQ expr RB |
LB expr GEQ expr RB |
LB expr NEQ expr RB |
LB expr QMARK expr COLON expr RB
```

Приложение B. Модуль IndexedFP

Файл IndexedFP.h

```
enum cfRelation
{
      BEFORE = 0,
      AFTER
};

enum RelationPosition
{
      FIRST = 0,
      SECOND,
      FIRST_SECOND
};

//базовый класс для всех индексированных сущностей в ФП – правил, описывающих
отношение порядка на множестве ФВ; правил, описывающих ФД и ФВ и пр.
//в классе реализованы алгоритм ПоискII в виде метода GetIndex, который получает
информацию о том, относительно каких выражений, набора индексов и его области
определения нужно решить задачу перебора. Этот метод возвращает итератор,
позволяющий перебирать результаты запроса.
class GeneralizedDependency
{
private:
      vector<vExpr>          object;          //set of objects, object[i] –
it's object is characterized by
                                                    //set of expressions i.e.
object[i] = indexes of i-th object
      vector<string>         varName;    //set of all variables in `range`, they
all in `object` expressions must be
                                                    //all of them must be
included in range
      bool                   permRange;  //have we already rearranged `range`?

protected:
      int                          dummyIndex; //dummy object – object without
indexes at all
      vector<rangeExpr>      range;
```

```
public:

      bool checkRanges(intDict& varValues);

      class GeneralizedDependencyIterator
      {
      private:
            vector<vExpr>                 receiverList;
            //ConstraintSolver – класс, который непосредственно реализует
алгоритм перебора ПоискII
            ConstraintSolver        solver;

      public:
            GeneralizedDependencyIterator() {}

            GeneralizedDependencyIterator(     GeneralizedDependency* gD, int
senderInd, const vInt& senderIndex,
                                                            const vInt&
receiverInd, bool delDuplicates)
            {
                  solver = ConstraintSolver(gD->varName, gD->range);
                  solver.Init(gD->object[senderInd], senderIndex);

                  for(int i=0, n=receiverInd.size(); i<n; ++i)
                        receiverList.push_back(gD->object[receiverInd[i]]);
            }

            vector< vector<vInt> > GetNext(int blockSize)
            {
                  assert(blockSize >= 0);
                  return solver.Solve(receiverList, blockSize);
            }

            void Reset()
            {
                  solver.Reset();
            }

            vector< vector<vInt> > GetRest()
            { return GetNext(maxInt); }
      };

      typedef GeneralizedDependencyIterator iterator;
```

```
public:
	GeneralizedDependency();

	int   AddObject(const vExpr& expr);
	void  AddRange(const string& name, const rangeExpr& wExpr);

	void  Simplify();
	const       vExpr& GetObject(int index) const;

protected:
	GeneralizedDependencyIterator
	GetIndex(int senderInd, const vInt& senderIndex,
	            const vInt& receiverInd, bool delDuplicates);

	//get all possible realizations of `receiverInd` objects
	GeneralizedDependencyIterator GetIndex(const vInt& receiverInd);

	string PrintRange() const;
};

//класс, реализующий правило для описания отношеиня порядка, типа A[i]<B[i] |
i=1..10;
class Relation : public GeneralizedDependency
{
private:
	string                  relName;    //execute `order`, `neighbour` f.e.
	vector<string>    name;       //cfName.size()==cfIndex.size()==2 – for
right and left cf in order
	vInt              index;            //specify `object`s indexes
corresponding to `name`

	bool              infixPrint; //print relation like `X Relation Y` or
`Relation X Y`

public:
	class RelationIterator
	{
	private:
		int                                       ind;
		bool                                      nextIter;
		Relation*                                 parent;
		fInstance                                 fragment;
```

```
		RelationPosition			position;
		GeneralizedDependency::iterator iter;
		vInt					index;

	public:
		RelationIterator() {}
		RelationIterator(Relation* relation, const fInstance& fr,
RelationPosition pos);

		vector<fInstance> GetNext(int blockSize);
		vector<fInstance> GetRest() { return GetNext(maxInt); }
	};

	typedef RelationIterator outer_iterator;

public:
	Relation(const string& rName, bool infix = false);

	void SetFirst(const string& objName, const vExpr& expr);
	void SetSecond(const string& objName, const vExpr& expr);

	string GetFirstName();
	string GetSecondName();

	RelationIterator GetFragment(const fInstance& fragment, RelationPosition
position);
	RelationIterator GetFragment(const fInstance& fragment);

	virtual string ToString() const;
};

//класс, реализующий индексированное правило, описывающее ФВ и множество его
входных и выходных ФД
class RuleCf : public GeneralizedDependency
{
private:
	vInt			cfIndex;	//mark cf object in common `object`
list, usually `cfIndex.size()==1`
	string			cfName;

	string			codeName;	//name of code fragment
```

```
      vInt                    dfIndex;    //mark df object in common `object`
list
      vector<string>          dfName;          //names of df
      vector<dfType>          dfIOType;   //or IN, or OUT, or IN_OUT

      vInt                    paramIndex; //mark code fragment parameters in
common `object` list

      bool                    purity;          //true if cf declared like
cf[i,j,...,k] = ...
                                                     //false if cf[i+k,j-
1,...,2*k] = ...

public:
      class RuleCfAllCfIterator
      {
      private:
            RuleCf*                                   parent;
            GeneralizedDependency::iterator iter;

      public:
            RuleCfAllCfIterator() {}
            RuleCfAllCfIterator(RuleCf* ruleCf);

            vector<fInstance> GetNext(int blockSize);
            vector<fInstance> GetRest() { return GetNext(maxInt); }
      };

      class RuleCfCfByDfIterator
      {
      private:
            RuleCf*                                   parent;
            GeneralizedDependency::iterator iter;
            bool                                      nextIter;
            int                                       ind;
            fInstance                                 df;
            vector<vExpr>                             df_index;

      public:
            RuleCfCfByDfIterator() {}
            RuleCfCfByDfIterator(RuleCf* ruleCf, const fInstance& dataFragment);

            vector<fInstance> GetNext(int blockSize);
```

```
		vector<fInstance> GetRest() { return GetNext(maxInt); }
	};

	class RuleCfDfByCfIterator
	{
	private:
		RuleCf*                                 parent;
		GeneralizedDependency::iterator iter;
		fInstance                               cf;
		bool                                    END;

	public:
		RuleCfDfByCfIterator() {}
		RuleCfDfByCfIterator(RuleCf* ruleCf, const fInstance& cFragment);

		vector<fInstance> GetPure();
		vector<fInstance> GetNext(int blockSize);
		vector<fInstance> GetRest() { return GetNext(maxInt); }
	};

	typedef RuleCfAllCfIterator   all_cf_iterator;
	typedef RuleCfCfByDfIterator  cf_by_df_iterator;
	typedef RuleCfDfByCfIterator  df_by_cf_iterator;

public:
	RuleCf();

	void SetCf(const string& name, const vExpr& expr);
	void SetCodeName(const string& name);
	string GetCfName() { return cfName; }
	void AddDf(dfType type, const string& name, const vExpr& expr);
	void SetParameter(const vExpr& expr);

	void CheckForPurity();
	bool IsPure();

public:
	RuleCfAllCfIterator GetAllCf();                                //get all
cf those need `df`
	RuleCfCfByDfIterator GetCfByDf(const fInstance& df);
	RuleCfDfByCfIterator GetDfByCf(const fInstance& cf); //get all df those
need for `cf`
```

```
	bool GetParameter(const fInstance& cf, vInt& parameter);
	string GetCodeName();

	string ToString() const;
};

/** @class IndexedFP
 * @brief Class that contain all information about FP in indexed form.
 *
 * Look at `Interactive.h` to get insight about how this all can be used at
practice.
 */
class IndexedFP: public Informator
{
private:
	///Mark about did we eliminate all constants in FP or not
	bool			constEliminated;
	string			FPName;

public:
	typedef map<string, vector<Relation> >	relDict;

	///Collection of all CF rules in FP.
	///
	///Rules like: \<cf_name>[\<set of index expressions>] :=
	///\<code_fragment_name>[\<df_name>[\<set of df index exprs>]...] |
\<var_name>=\<up_limit>..\<down_limit>, ...
	vector<RuleCf>	ruleCf;

	///Collection of relations like cf order, df/cf neighbourhood e.t.c
	relDict			relation;

	///Collection of constants
	vector<Constant> constant;

public:
	/** @name Typedef for collection of getter iterators
	 * All iterators have same set of methods:
	 * – vector\<fInstance> GetNext(int blockSize) – Return collection of
fragment instances with size not greater then blockSize
	 * – vector\<fInstance> GetRest() – Return all possible fragment instances
at once
	 */
```

```
	//@{
	/** @brief Iterator that needs for get information about relation */
	class RelationIterator
	{
	private:
		IndexedFP*		parent;
		string			relName;
		fInstance		fragment;
		RelationPosition	position;
		int			ind;
		bool			nextIter;
		Relation::outer_iterator iter;

	public:
		RelationIterator(IndexedFP* indFP, const string& relationName,
					const fInstance& frag, RelationPosition
pos);

		vector<fInstance> GetNext(int blockSize);
		vector<fInstance> GetRest() { return GetNext(maxInt); }
	};

	/** @brief Iterator that needs for get information about cf connected by
order relation */
	class CfByCfIterator
	{
	private:
		IndexedFP*		parent;
		fInstance		cf;
		cfRelation		rel;
		int			ind;
		bool			nextIter;
		Relation::outer_iterator iter;
	public:
		CfByCfIterator(IndexedFP* indFP, const fInstance& cFrag, cfRelation
relation);

		vector<fInstance> GetNext(int blockSize);
		vector<fInstance> GetRest() { return GetNext(maxInt); }
	};

public:
	IndexedFP();
```

```
	~IndexedFP();

	void		SetFPName(const string& name);
	string			GetName() const;

	/** @brief Constant elimination – substitution of all constants in all FP
expressions to their concrete calculated values.
	 * @param[in] constDict Dictionary with values of constants. Can be
omitted.
	 */
	void		ConstantElimination();

	/** @name Modifying functions for IndexedFP object. */
	//@{
	Relation&	AddRuleOrder();
	RuleCf&			AddRuleCf();

	/** @brief Add relation.
	 *
	 * For now there are relations with names: cf order – "<", neighbourhood
relation – "neighbours".
	 * @param[in] relName Relation name
	 * @return Reference to Relation object we just created.
	 */
	Relation&	AddRelation(const string& relName, bool infix = false);

	/** @brief Getting collection of relation pairs for concrete fragment.
	 * @param[in] relName Relation name like "<" for cf order, "neighbours"
	 * @param[in] fragment Instance of fragment for which we want to get
collection of fragments
	 * @param[in] pos Position in relation of `fragment`
	 * – if `pos` == FIRST we will get collection \f$ \left \{ b | R(a,b)
\right \} \f$, where a = `fragment`, R = `relName`
	 * – if `pos` == SECOND -> \f$ \left \{ b | R(b,a) \right \} \f$
	 * – if `pos` == FIRST_SECOND -> \f$ \left \{ b | R(a,b) \right \} \cup
\left \{ b | R(b,a) \right \} \f$
	 */
	RelationIterator GetRelation(const string& relName, const fInstance&
fragment, RelationPosition pos = FIRST_SECOND);

	/** @brief Getting collection of cf which connected with current cf by cf
order relation
	 *
```

```
        * Work just like IndexedFP::GetRelation() but with "<" relation
        * @param[in] cf Instance of computational fragment for which we want to
get collection of his "cf order partners"
        * @param[in] rel `cf` relation in cf order. It's can be:
        * – AFTER – can be interpretated like SECOND from
IndexedFP::GetRelation()
        * – BEFORE – like FIRST from IndexedFP::GetRelation()
        *
        * Example:\n
        * if we have cf order rule: cfA[i,j] < cfB[i+j] | i=1..4, j=1..4 \n
        * then
        * – fp.GetCfByCf(fInstance(CF,"cfB", vInt() << 3)), AFTER) -> [cfA[1,2],
cfA[2,1]]
        * – fp.GetCfByCf(fInstance(CF,"cfB", vInt() << 3)), BEFORE) -> [] – empty
vector
        * – fp.GetCfByCf(fInstance(CF,"cfA", vInt() << 3 << 3)), BEFORE) ->
[cfB[6]]
        *
        * @see IndexedFP::GetRelation()
        */
      CfByCfIterator GetCfByCf(const fInstance& cf, cfRelation rel);

      /** @brief Getting collection of all cf in fragmented program
        */
      AllCfIterator GetAllCf();

      /** @brief Getting type of cf<->df access
        * @return dfType – IN|OUT|IN_OUT or UNDEF_DFTYPE if cf and df not
connected in any way
        */
      dfType GetAccessType(const fInstance& cf, const fInstance& df);

      /** @brief Getting collection of data fragment those needs for concrete
computational fragment
        * @param[in] cf Instance of computational fragment for which we want
        * to get collection of data fragment those needs for his execution
        *
        * Example: \n
        * if we have cf rule: cfA[i,j] = codeF(in: dfA[i,j], dfB[i]) | i=1..4,
j=1..4 \n
        * then
        * – fp.GetDfByCf(fInstance(CF,"cfA", vInt() << 3 << 4))) -> [dfA[3,4],
dfB[3]]
```

```
        */
       DfByCfIterator GetDfByCf(const fInstance& cf);

       /** @brief Getting collection of computational fragment which execution
depends on concrete data fragment
        * @param[in] df Instance of data fragment for which we want to get
collection of computational fragment those needs in `df`
        * There can by dublicates.
        * However, some cases of dublicating can be detected.
        * – If to df in cf rule have set of indexes with not equal set of index
variables, like this: \n
        * cf = code<>(in: df[i+j]; out: df[j]) | ... \n
        * then this two df is different, we'll generate cf for each of them
        * – If to df have exact same set of index expressions, e.g. \n
        * cf = code<>(in: df[i+j-10][j*k]; out: df[i+j-10][j*k]) | ... \n
        * then this two df considered as equal, we'll generate cf only for one of
them \n
        * @note If df have equivalent set of index expression, but not in the
same form, then we'll get cf dublicate, e.g. \n
        * cf = code<>(in: df[(i+j)*(1<0) + (i+j)*(1>=0)]; out: df[i+j]) | ...
        * @see IndexedFP::GetCfByDfCount()
        */
       CfByDfIterator GetCfByDf(const fInstance& df);

       /** @brief Getting count of computational fragment which execution depends
on concrete data fragment
        * @param[in] df Obvious
        * @param[in] blockSize Implementation of this function uses repeating
calls of IndexedFP::GetCfByDf().GetNext(blockSize)
        * @return Size of set of computational fragments
        * Complexity – O(n^2), where n is number of fragment returned by
IndexedFP::GetCfByDf(df)
        * @see IndexedFP::GetCfByDf()
        */
       int GetCfByDfCount(const fInstance& df);

       /** @brief Getting all initial computational and data fragment
        *
        * Return InitCf – collection of computational fragments for which there
are not
        * computational fragments that are must executed BEFORE them \n
        * Moreover, return collection of data fragment wich needs for execution
of fragments from InitCf.
```

```
	 */
	StartFragmentIterator GetStartFragments();

	int GetDfSize(const fInstance& cf);
};
```

Файл ConstraintSolver.h

```
//Класс, реализующий алгоритм ПоискII
class ConstraintSolver
{
private:
	vector<string>	varName;
	vector<rangeExpr> range;

	//В классе SLAE реализован решатель СЛАУ
	SLAE		slae;
	vExpr		nonLinearExpr;	//set of nonlinear expressions
	vInt		nonLinearVal;	//right side of equations for nonlinear
expressions

	bool		NoSolution, Initialized;
	int			SolveCalls;

public:
	ConstraintSolver() {}

	ConstraintSolver(vector<string>& _varName, vector<rangeExpr>& _range)
		: varName(_varName), range(_range), Initialized(false),
SolveCalls(0)
	{}

private:
	bool IterateRange(int deep, bool& goDeeper, int& blockSize, vector<vInt>&
ret);

public:
	//it's like solving system of equations: A[i] = b[i], for each i,
	//where for each variable varName[j] there is constraint: range[j].first
<= varName[j] <= range[j].second
	//return all possible solutions of this system
```

```
	//[NOTE]: there is no move back, only forward to new solutions block by block
	//[NOTE]: `blockSize == -1` mean getting all solutions at once
	vector<vInt> Solve(int blockSize = maxInt);

	void Reset();

	//same function as `Solve` above, but `toFit` contain expression that we need to calc. using values of variables
	//those we get solving slae, so we return values of `toFit` exprs
	vector<vInt> Solve(vExpr& toFit, int blockSize = maxInt);
	vector<vector<vInt> > Solve(vector<vExpr>& toFit, int blockSize = maxInt);

	//[NOTE]: only once you can init solver
	//but many times call `Solve`
	void Init(vExpr& A, const vInt& b);
};
```

Файл ConstraintSolver.cpp

```
#include "ConstraintSolver.h"

//return true if we must continue calculations according to `blockSize` value
bool ConstraintSolver::IterateRange(int deep, bool& goDeeper, int& blockSize, vector<vInt>& ret)
{
	intDict varValPartial = slae.GetDict();

	//end of recursion – we went through all ranges
	if (deep < 0)
	{
		//and of going down process
		if (goDeeper)
		{
			goDeeper = false;
			return true;
		}

		//check with nonlinear expressions
		//[TODO]: we can check nonlinear equations more earlier
		//i.e. check nonlinear at those moment, when we know all valiable values
```

```
            //that we need for calculate nonlinear expression
            for(int i=0,n=nonLinearExpr.size(); i<n; ++i)
            {
                  int value; nonLinearExpr[i].Calculate(value,varValPartial);
                  if (value != nonLinearVal[i]) return true;
            }

            //add values of variables to ret
            //according to order set by `varName`
            ret.push_back(slae.GetAnswer());
            blockSize--;

            return (blockSize != 0);
      }

      //low and up limits of variable values
      int a; range[deep].first.Calculate(a,varValPartial);
      int b; range[deep].second.Calculate(b,varValPartial);

      //make the cycle if current variable is free
      string name = varName[deep];
      if (slae.IsFree(name))
      {
            int start = a;    //us usual
            if (goDeeper)     //continue from the place we end process
                  start = slae.GetVar(name);

            for(int val = start; val <= b; ++val)
            {
                  slae.SetVar(name, val);
                        if (!IterateRange(deep-1, goDeeper, blockSize, ret))
                              return false;     //go out, if `blockSize`
solutions we get, leave `slae` with set var `name`!
            }
      }
      //else calculate variable using previous calculated variables
      else
      {
            //check for integrality of variable value
            if (!slae.CalculateVar(name)) return true;
            int val = slae.GetVar(name);
            if (a <= val && val <= b)
                  if (!IterateRange(deep-1, goDeeper, blockSize, ret))
```

```
                        return false;     //go out, if `blockSize` solutions we
get, leave `slae` with set var `name`!
      }

      return true;
}

//it's like solving system of equations: A[i]*x = b[i], for each i,
//where for each variable varName[j] there is constraint: range[j].first <=
varName[j] <= range[j].second
//return all possible solutions of this system
void ConstraintSolver::Init(vExpr& A, const vInt& b)
{
      if (Initialized)
            ERR << "ConstraintSolver already initialized" << BANG;

      Initialized = true;

      if (b.size() != A.size())
            ERR << "Incorrect SLAE in ConstraintSolver; " << "matrix size = " <<
A.size() <<
                        " right vector size = " << b.size() << BANG;

      if (range.size() != varName.size())
            ERR << "Incorrect number of variables; in range " << range.size() <<
                        "; in variable name vector " << varName.size() << BANG;

      //=================== Making SLAE based on linear expressions in `A` expr.
list ===================//
      //slae
      vector<vInt>      matrix;
      vInt              rightVec;

      int n = A.size();                   //number of equations
      int m = varName.size();             //number of variables

      //for each expression in `A`
      for(int i=0, j=0; i<n; ++i)
      {
            Expression expr = A[i];

            //if linear
            if (expr.CheckForLinear())
```

```
        {
            //then add information to slae
            rightVec.push_back(b[i] – expr.GetFreeMember());

            matrix.push_back(vInt());
            //for each variable get his coeff. from linear expression and
set cell of matrix
            for(int k=0; k<m; ++k)
                matrix[j].push_back(expr.GetCoeffAt(varName[k]));

            j++;
        }
        else
        {
            nonLinearExpr.push_back(expr);
            nonLinearVal.push_back(b[i]);
        }
    }

    //=================== Making,solving SLAE ===================//
    slae = SLAE(matrix, rightVec, varName);
    NoSolution = !slae.Solve();  //[WARNING]: `slae.Solve()` return
true/false – telling there is solution or not, make attention
}

vector<vInt> ConstraintSolver::Solve(int blockSize/* = maxInt*/)
{
    if (!Initialized || NoSolution)
        return vector<vInt>();

    int             m               = varName.size();           //number
of variables
    bool            goDeeper    = (SolveCalls > 0);          //if it's first
`Solve` call, then `goDeeper=true` we don't need
    int             block_size  = blockSize;
    vector<vInt>    ret;
    SolveCalls++;

    IterateRange(m-1, goDeeper, block_size, ret);

    return ret;
}
```

```
void ConstraintSolver::Reset()
{
	SolveCalls = 0;

	if (Initialized)
		slae.ResetVarValues();
}

//same function as `Solve` above, but `toFit` contain expresion that we need to
calc. using values of variables
//those we get solving slae, so we return values of `toFit` exprs
vector<vInt> ConstraintSolver::Solve(vExpr& toFit, int blockSize/* = maxInt*/)
{
	vector<vInt> varValue = Solve(blockSize);
	return Expression::FitValuesToExpressions(varValue, varName, toFit);
}

vector<vector<vInt> > ConstraintSolver::Solve(vector<vExpr>& toFit, int
blockSize/* = maxInt*/)
{
	vector<vInt> varValue = Solve(blockSize);

	vector<vector<vInt> > ret;
	for(int i=0,n=toFit.size(); i<n; ++i)
		ret.push_back(Expression::FitValuesToExpressions(varValue, varName,
toFit[i]));
	return ret;
}
```

Файл Expr.h

```
enum Operator
{
	//Look at `Expression::CreateOperatorAlgebraicMap()` for clarification
	Add, Sub,
	Mul,
	Div, Rem,
	Lt, Gt,
	Eq,
	Geq, Leq,
	Question, Colon,
	Identity,
```

```
	Function1,
	Undef
};

class Expression;

typedef vector<Expression>			vExpr;
typedef pair<Expression, Expression>	rangeExpr;

enum exprType
{
	INDEX,		//index variable like i,j,k e.t.c.
	VARIABLE,	//property of indexed/element static/dynamic constant
	VALUE,		//elementary type – integer
	FUNCTION	//obviously!
};

enum variableType
{
	CONSTANT,
	PROPERTY
};

class Variable
{
private:
	variableType	type;
	string			name;
	vExpr			index;
	string			propName;

public:
	Variable();
	Variable(const string& _name, const vExpr& _index = vExpr(), const string&
_propName = "");

	string GetName() const
	{ return name; }

	//pair.X – get info or not
	//pair.Y – in the case of Dynamic constant true if not yet fixed constant,
else false
	bool		Simplify(int& value, Informator* info);
```

```
	wBool			Calculate(int& value, Informator* info, const intDict& iDict = intDict(), bool strong = true);
	friend ostream& operator<<(ostream& os, const Variable& o);
	friend bool	operator==(const Variable& o1, const Variable& o2);
	string			toCode() const;
};

//Класс, реализующий арифметические выражения, например, "1*3-5" или "x+y+1", где x,y – индексные переменные
class Expression
{
private:
	exprType	type;

	//--------FUNCTION stuff--------//
	Operator	oper;
	vExpr		operand;

	//--------VALUE stuff-----------//
	int		iVal;

	//--------INDEX stuff-----------//
	string		vName;

	//--------VARIABLE stuff--------//
	Variable	variable;

	//--------Common stuff----------//
	bool		isLinear, isLinearChecked;
	intDict		coeff;		//multipliers as int at variables as string
	int		constVar;	//just const in linear composition

	//object which give information about values of constant e.t.c.
	Informator* info;

	static map<Operator, string> operToStr;
	static map<Operator, string> operToAlgebraic;

private:
	static map<Operator, string> CreateOperatorAlgebraicMap();
	static map<Operator, string> CreateOperatorMap();

	void copy(const Expression& op);
```

```
public:
                        Expression(): info(NULL) {}
                        Expression(Informator& informator, const Expression&
op1, Operator op, const Expression& op2);
      explicit    Expression(Informator& informator, int val = 0);
      explicit    Expression(Informator& informator, const string& val);
                        Expression(Informator& informator, const Variable& var);

      static vector<vInt> FitValuesToExpressions(const vector<vInt>& varValue,
const vector<string>& varName, vExpr& toFit);

      wBool                   Calculate(int& value, const intDict& iDict =
intDict(), bool strong = true);
      wBool                   Calculate(int& value, const vector<string>&
varName, const vInt& varValue, bool strong = true);

      void              ToValue(int value);

      bool              Simplify();
      bool              Simplify(int& value);

      bool              IsPure() const;
      vector<string>    GetIndexNames() const;
      int               GetCoeffAt(const string& varName) const;
      int               GetFreeMember() const;
      bool              CheckForLinear();
      bool              IsVariable() const;
      string                  GetVariable() const;
      int               GetValue() const;

      string toCode(bool noNew = false) const;
      string toLinearView();
      string toAlgebraic(bool noBrackets = false) const;
};
```

Приложение C. Компилятор языка LuNA

Файл Parser.hs, содержащий парсеры, преобразующий текст на языке LuNA в абстрактное синтаксическое дерево.

```
{-# LANGUAGE RankNTypes #-}

module Parser where

import FProgram

import Text.ParserCombinators.Parsec
import qualified Text.ParserCombinators.Parsec.Token as T
import Text.ParserCombinators.Parsec.Language

import Data.Char
import Data.List
import Data.Function
import Data.Monoid
import qualified Data.Map as Map
import System.IO
import Control.Monad.Trans.Error

-- | Lexer & language stuff

fp_language :: LanguageDef ()
fp_language = LanguageDef {
      commentStart      = "/*",
      commentEnd        = "*/",
      commentLine       = "//",
      nestedComments    = False,
      identStart        = letter <|> char '_',
      identLetter       = alphaNum <|> char '_',
      opStart                 = oneOf "!?:+-/*=><%.|",
      opLetter          = oneOf "=.",
      reservedNames     =
["cf","df","block","const","static","dynamic","in","out","inout","program"],
      reservedOpNames = [],
      caseSensitive     = True
}
```

```
lexer       = T.makeTokenParser fp_language
lexeme            = T.lexeme lexer

parens            = T.parens lexer
braces            = T.braces lexer
angles            = T.angles lexer
brackets    = T.brackets lexer
integer           = lexeme $ T.integer lexer
identifier  = lexeme $ T.identifier lexer
semi        = T.semi lexer
comma       = T.comma lexer
colon       = T.colon lexer
dot               = T.dot lexer
stringL           = T.symbol lexer
charL       = lexeme . char
kw                = T.reserved lexer

semiSep           = T.semiSep lexer
commaSep    = T.commaSep lexer
whiteSpace  = T.whiteSpace lexer

-- | Error Either stuff

infixl <!+>

(<!+>) :: Either String a -> String -> Either String a
(<!+>) (Left l) s = Left $ s ++ "\n" ++ l
(<!+>) (Right r) _     = Right r

either_s :: (Show e) => Either e a -> Either String a
either_s (Left l) = Left $ show l
either_s (Right r)     = Right r

-- | Parser stuff

infixl <?|>
(<?|>) a b = (try a) <|> b

oper0 :: Parser Operator
oper0 =
      (stringL ">="     >> return GreaterE)     <?|>
      (charL      '>'         >> return Greater)      <?|>
      (stringL "<="     >> return LessE)  <?|>
```

```
      (charL      '<'         >> return Less)         <?|>
      (stringL "!="     >> return NotEqual)     <?|>
      (stringL "=="     >> return Equal)

oper1 :: Parser Operator
oper1 =
      (charL '+' >> return Add) <|>
      (charL '-' >> return Sub)

oper2 :: Parser Operator
oper2 =
      (charL      '*'   >> return Mul) <|>
      (charL      '/'   >> return Div) <|>
      (charL      '%'   >> return Rem)

property :: Parser Expr
property = do
      ise <- index_se
      charL '.'
      s <- identifier
      return $ ExprProp $ Prop ise s

constant :: Parser Expr
constant = do
      ise <- index_se
      return $ ExprConst ise

function :: Parser Expr
function = do
      s <- identifier
      l <- parens $ commaSep expr
      return $ ExprFunc s l

int :: Parser Expr
int = do
      n <- integer
      return $ ExprAtom (fromIntegral n)

mult :: Parser Expr
mult =
      int <?|>
      property <?|>
      function <?|>
```

```
      constant <?|>
      parens expr

term :: Parser Expr
term = do
            a     <- mult
            op    <- oper2
            b     <- term
            return $ ExprOp op a b
      <?|> mult

bterm :: Parser Expr
bterm = do
            a     <- term
            op    <- oper1
            b     <- bterm
            return $ ExprOp op a b
      <?|> term

sub_expr :: Parser Expr
sub_expr = do
            a     <- bterm
            op    <- oper0
            b     <- sub_expr
            return $ ExprOp op a b
      <?|> bterm

if_then_else :: Parser Expr
if_then_else = do
      cond  <- sub_expr
      charL '?'
      th          <- sub_expr
      charL ':'
      el          <- sub_expr
      return $ ExprIfThenElse cond th el

expr :: Parser Expr
expr = if_then_else <?|> sub_expr

expr_list :: Parser [Expr]
expr_list = nothing_or $ brackets $ commaSep expr

index_se :: Parser IndexSE
```

```
index_se = do
      s     <- identifier
      e     <- expr_list
      return $ IndexS s e

io_type :: Parser IOType
io_type =
      (kw "in"    >> return In) <?|>
      (kw "inout" >> return InOut) <?|>
      (kw "out"   >> return Out)

eps :: Parser ()
eps = return ()

df_section :: Parser [(IOType, IndexSE)]
df_section = do
      io    <- io_type
      colon
      ise   <- commaSep index_se
      return $ zip (repeat io) ise

cf_rule :: Parser CfRule
cf_rule = do
      kw "cf"
      cf_def            <- index_se
      stringL ":="
      code_name   <- identifier
      vars        <- nothing_or $ braces $ commaSep index_se
      params            <- nothing_or $ angles $ commaSep expr
      dfs               <- nothing_or $ parens $ semiSep df_section
      return $ CfRule { cf          = cf_def,
                              code  = code_name,
                              var   = vars,
                              param       = params,
                              df          = concat dfs }

order_rule :: Parser OrderRule
order_rule = do
      a <- index_se
      charL '<'
      b <- index_se
      return $ OrderRule (a,b)
```

```
const_type :: Parser ConstType
const_type =
      (kw "dynamic"     >> return Dynamic) <?|>
      (kw "static"      >> return Static) <?|>
      return Static

const_rule :: Parser ConstRule
const_rule = do
      tp    <- const_type
      kw "const"
      ise   <- index_se
      charL '='
      e     <- expr
      return $ ConstRule tp ise e

df_def :: Parser PropRule
df_def = do
      kw "df"
      ise <- index_se
      stringL ":="
      kw "block"
      e <- parens expr
      return $ PropRule Df (Prop ise "df_size") e

dot_prop :: Parser PropRule
dot_prop = do
      ise   <- index_se
      dot
      s     <- identifier
      charL '='
      e     <- expr
      return $ PropRule Some (Prop ise s) e

prop_rule :: Parser PropRule
prop_rule = df_def <?|> dot_prop

diap :: Parser (Expr,Expr)
diap =
      do
            stringL ".."
            b     <- expr
            return (ExprAtom 0, ExprOp Sub b (ExprAtom 1))
      <?|>
```

```
      do
            a     <- expr
            stringL ".."
            b     <- expr
            return (a,b)
      <?|>
      do
            a     <- expr
            return (a,a)

range :: Parser Range
range = do
      s <- identifier
      charL '='
      r <- diap
      return $ Range s r

nothing_or p = p <?|> return []

rule :: Parser a -> Parser (RangeRule a)
rule p = do
      rl    <- p
      rng   <- nothing_or $ (charL '|' >> commaSep range)
      return $ RangeRule rl rng 0

program_item :: Parser FProgram
program_item =
            do
                  kw "program"
                  s <- identifier
                  return $ mempty { fpname = s }
            <?|>
            do
                  r <- rule cf_rule
                  return $ mempty { cfrule = [r] }
            <?|>
            do
                  r <- rule prop_rule
                  return $ mempty { proprule = [r] }
            <?|>
            do
                  r <- rule const_rule
                  return $ mempty { construle = [r] }
```

```
            <?|>
            do
                  r <- rule order_rule
                  return $ mempty { orderrule = [r] }

fprogram :: Parser FProgram
fprogram = do
      whiteSpace
      l <- endBy program_item semi
      eof
      return $ enumRules $ mconcat l

enumRules :: FProgram -> FProgram
enumRules fp =
      fp {
            construle   = enum $ construle fp,
            proprule    = enum $ proprule fp,
            cfrule            = enum $ cfrule fp,
            orderrule   = enum $ orderrule fp
            }
      where
            enum :: forall a . Rule a => [RangeRule a] -> [RangeRule a]
            enum l = [RangeRule r rng i | (RangeRule r rng _, i) <- zip l [0..]]

-- | Settings parser

setting_item :: Parser Setting
setting_item = do
      let notSpace = many1 (satisfy (not . isSpace))
      char '@'
      key <- notSpace
      spaces_not_nl
      value <- notSpace
      return $ Map.singleton key [value]

skip_to_eol :: Parser String
skip_to_eol = many $ satisfy (/= '\n')

spaces_not_nl :: Parser String
spaces_not_nl = many $ satisfy (\x -> isSpace x && (x /= '\n'))

comment :: Parser Setting
comment =
```

```
      do
            spaces_not_nl
            l <- sepBy setting_item spaces_not_nl
            skip_to_eol
            return $ Map.unionsWith (++) l
      <?|>
      do
            skip_to_eol
            return Map.empty

src_line :: Parser Setting
src_line =
      do
            manyTill (noneOf "\n") (try $ string "//")
            st <- comment
            return st
      <?|>
      do
            skip_to_eol
            return Map.empty

setting :: Parser Setting
setting = do
      st <- sepBy src_line newline
      eof
      return $ Map.unionsWith (++) st

-- | Compose parsers

get_ast :: String -> Either String (FProgram, Setting)
get_ast src = do
      settings    <- (either_s $ parse setting "setting" src)          <!+>
"Setting & comments parsing error."
      fp                <- (either_s $ parse fprogram "fprogram" src)  <!+>
"FProgram parsing error."
      return (fp, settings)
```

Файл FProgram.hs, в котором описаны функции транслирующие абстрактное синтаксическое дерево в код на языке C++.

```
{-# LANGUAGE RankNTypes #-}
```

```
module FProgram where

import qualified Data.Map as Map
import Data.List
import Data.Char
import Data.Monoid
import Data.Either
import Control.Monad

import Common

-- | Operator datatype

data Operator = Add | Sub |
                        Mul | Div | Rem |
                        NotEqual | Equal |
                        Greater | Less |
                        GreaterE | LessE
                        deriving (Eq, Ord)

instance Show Operator where
      show Add          = "+"
      show Sub          = "-"
      show Mul          = "*"
      show Div          = "/"
      show Rem          = "%"
      show NotEqual     = "!="
      show Equal        = "=="
      show Greater      = ">"
      show Less         = "<"
      show GreaterE     = ">="
      show LessE        = "<="

-- | Expression datatype

data Expr = ExprOp Operator Expr Expr |
                  ExprIfThenElse Expr Expr Expr |
                  ExprAtom Int |
                  ExprFunc String [Expr] |
                  IndexVar String Int |
                  ExprConst IndexSE |
                  ExprProp Prop
                  deriving (Eq, Ord)
```

```
getInt (ExprAtom i) = i
getInt x = error $ "getInt from " ++ show x

isInt (ExprAtom i) = True
isInt _ = False

lookForInt :: Expr -> Maybe Int
lookForInt (ExprAtom i) = Just i
lookForInt _ = Nothing

-- it's kind of Functor functionality
map_expr :: (Expr -> Expr) -> Expr -> Expr
map_expr f (ExprOp op a b)               = f $ ExprOp op (f a) (f b)
map_expr f (ExprIfThenElse a b c)  = f $ ExprIfThenElse (f a) (f b) (f c)
map_expr f (ExprFunc s l)                = f $ ExprFunc s (map f l)
map_expr f e@(IndexVar _ _)              = f e
map_expr f e@(ExprAtom _)                = f e
map_expr f e@(ExprConst _)               = f e
map_expr f e@(ExprProp _)                = f e

mapM_expr :: Monad m => (Expr -> m Expr) -> Expr -> m Expr
mapM_expr f (ExprOp op a b) = do
      a' <- f a
      b' <- f b
      f (ExprOp op a' b')
mapM_expr f (ExprIfThenElse a b c) = do
      a' <- f a
      b' <- f b
      c' <- f c
      f (ExprIfThenElse a' b' c')
mapM_expr f (ExprFunc s l) = do
      l' <- mapM f l
      f (ExprFunc s l')
mapM_expr f e@(IndexVar _ _) = f e
mapM_expr f e@(ExprAtom _) = f e
mapM_expr f e@(ExprConst (IndexS s le)) = do
      le' <- mapM f le
      f (ExprConst (IndexS s le'))
mapM_expr f e@(ExprProp (Prop (IndexS s le) s2)) = do
      le' <- mapM f le
      f (ExprProp (Prop (IndexS s le') s2))
```

```
-- it's kind of Foldable functionality
fold_expr :: Monoid m => (Expr -> m) -> Expr -> m
fold_expr f e@(ExprOp op a b)            = mconcat [f e, fold_expr f a,
fold_expr f b]
fold_expr f e@(ExprIfThenElse a b c)     = mconcat [f e, fold_expr f a,
fold_expr f b, fold_expr f c]
fold_expr f e@(IndexVar _ _)             = f e
fold_expr f e@(ExprAtom _)                     = f e
fold_expr f e@(ExprFunc _ l)                         = (f e) `mappend` mconcat
(map (fold_expr f) l)
fold_expr f e@(ExprConst (IndexS _ l))               = (f e) `mappend` mconcat
(map (fold_expr f) l)
fold_expr f e@(ExprProp (Prop (IndexS _ l) _)) = (f e) `mappend` mconcat (map
(fold_expr f) l)

--instance Show Expr where
--     show = transECpp

instance Show Expr where
      show (ExprOp op a b)          = "(" ++ show a ++ show op ++ show b ++ ")"
      show (ExprIfThenElse i t e) = "(" ++ show i ++ " ? " ++ show t ++ " : " ++
show e ++ ")"
      show (ExprAtom a)
            | a < 0           = "(" ++ s ++ ")"
            | otherwise = s
            where s = show a
      show (ExprFunc s l)                = s ++ "(" ++ intercalate ", " (map
show l) ++ ")"
      show (IndexVar s i)                = s ++ "{" ++ show i ++ "}"
      show (ExprConst ise)          = show ise
      show (ExprProp pr)                 = show pr

transOpCpp :: Operator -> String
transOpCpp Add          = "Add"
transOpCpp Sub          = "Sub"
transOpCpp Div          = "Div"
transOpCpp Mul          = "Mul"
transOpCpp Rem          = "Rem"
transOpCpp NotEqual     = "Neq"
transOpCpp Equal  = "Eq"
transOpCpp Greater      = "Gt"
transOpCpp Less         = "Lt"
transOpCpp GreaterE     = "Geq"
```

```
transOpCpp LessE  = "Leq"

transECpp e = case e of
      (ExprOp op a b)               -> exprS $ triple (tr a) (transOpCpp op) (tr
b)
      (ExprIfThenElse i t e)  -> exprS $ triple (tr i) "Question" (exprS $
triple (tr t) "Colon" (tr e))
      (ExprAtom i)                  -> exprS $ show i
      (IndexVar s i)                -> exprS $ show s ++ ", " ++ show i
      (ExprConst ise)               -> exprS $ "Variable(" ++ transISECppBk ise
++ ")"
      (ExprProp (Prop ise p)) -> exprS $ "Variable(" ++ transISECppBk ise ++ ",
" ++ show p ++ ")"
      (ExprFunc s [x])        -> exprS $ triple (exprS $ show s) "Function1" (tr
x)
      e@(ExprFunc _ _)        -> error "Number of arguments must by equal 1 in
expression: " ++ show e
      where
            triple a b c            = intercalate ", " [a,b,c]
            exprS s                       = "Expression" ++ "(fp, " ++ s ++ ")"
            expr e                        = exprS $ transECpp e
            tr                            = transECpp

-- | Indexed names datatype

data IndexS a     = IndexS String [a] deriving (Eq, Ord)
type IndexSE      = IndexS Expr
type IndexSI      = IndexS Int

instance Functor IndexS where
      fmap f (IndexS s l) = IndexS s (fmap f l)

fmapM_indexS :: Monad m => (Expr -> m Expr) -> IndexSE -> m IndexSE
fmapM_indexS f (IndexS s l) = do
      l' <- mapM f l
      return $ IndexS s l'

instance (Show b) => Show (IndexS b) where
      show (IndexS h [])      = h
      show (IndexS h l) = h ++ "[" ++ intercalate ", " (map show l) ++ "]"

transLECpp :: [Expr] -> String
transLECpp l = "vExpr()" ++ (concat $ map ((" << "++) . transECpp) l)
```

```
transISECppBk ise@(IndexS s l) =
      if null l
            then show s
            else transISECpp ise
transISECpp (IndexS s l) = show s ++ ", " ++ transLECpp l

-- | Rule

data RuleType = TCf | TProp | TConst | TOrder
      deriving Show

-- Node – unique identifier of each language elements like Cf,Const,Prop ...
-- Eq, Ord for Map, Ord also for Monoid (Map k v)
data FPNode =
      ConstNode IndexSI |
      PropNode IndexSI String |
      CfNode IndexSI |
      OrderRel IndexSI IndexSI
      deriving (Eq, Ord)

instance Show FPNode where
      show (ConstNode isi) = "const " ++ showISI isi
      show (CfNode isi) = "cf " ++ showISI isi
      show (PropNode isi s) = "propery " ++ showISI isi ++ "." ++ s
      show (OrderRel a b) = showISI a ++ " < " ++ showISI b

showISI :: IndexSI -> String
showISI (IndexS s l) = s ++ "[" ++ (intercalate "," (map show l)) ++ "]"

-- Int for index of the rule in FProgram
-- Maybe not just adj list, but Set.Set
data FPNodeAttribute =
      ConstAttr {
            -- ConstType = Static | Dynamic
            constType::ConstType,
            constValue::Int
      } |
      PropAttr {
            propValue::Int
      } |
      CfAttr {
            cfAttrParam::[Int],
```

```
            cfAttrCode::String,
            cfAttrVar::[IndexSI],
            cfAttrDf::[(IOType, IndexSI)] -- they all will be in adj list as
well
      } |
      OrderAttr {
      }
      deriving Show

integers :: IndexSE -> Maybe IndexSI
integers (IndexS s le)
      | all isInt le = Just $ IndexS s (map (\(ExprAtom i) -> i) le)
      | otherwise = Nothing

class Rule a where
      map_rule :: (Expr -> Expr) -> a -> a
      mapM_rule :: Monad m => (Expr -> m Expr) -> a -> m a

      noExprForm :: a -> Maybe (FPNode, RuleType, FPNodeAttribute)
      ruleType ::a -> RuleType

      fold_rule :: Monoid m => (Expr -> m) -> a -> m
      transCpp :: a -> [String]
      -- construction of FProgram with only one rule
      construct_fp :: RangeRule a -> FProgram
      -- replace one rule to another in fprogram
      change_rule :: RangeRule a -> RangeRule a -> FProgram -> FProgram

-- | Range

data Range = Range String (Expr, Expr) deriving Eq

map_range :: (Expr -> Expr) -> Range -> Range
map_range f (Range s (a,b)) = Range s (f a, f b)
mapM_range f (Range s (a,b)) = do
      a' <- f a
      b' <- f b
      return $ Range s (a', b')

instance Show Range where
      show (Range s (a,b)) = s ++ "' = " ++ show a ++ ".." ++ show b

intercalateE _ [] = ""
```

```
intercalateE sep l      = intercalate sep l ++ sep

transRangeCpp :: (Int, Range) -> String
transRangeCpp (i,(Range s (a,b))) =
      let tr = transECpp in
      "entity.AddRange(" ++ intercalate ", " [show s, show i, "make_pair(" ++ tr
a ++ ", " ++ tr b ++ ")"] ++ ")"

-- | Rule with Range

data RangeRule a = RangeRule a [Range] Int
      deriving Eq

data RuleID = RuleID Int RuleType
      deriving Show

ruleID :: Rule a => RangeRule a -> RuleID
ruleID (RangeRule a r i) = RuleID i (ruleType a)

transRuleCpp :: (Rule a) => RangeRule a -> String
transRuleCpp (RangeRule r l _) =
      let m = map transRangeCpp $ zip [0..] l in
      intercalateE ";" $ transCpp r ++ m

instance Show a => Show (RangeRule a) where
      show (RangeRule rl rng _) = show rl ++ showRng rng
            where
                  showRng [] = ""
                  showRng l = " | " ++ intercalate ", " (map show l)

get_range_vars :: (Rule a) => RangeRule a -> [String]
get_range_vars (RangeRule _ l _) = [s | (Range s _) <- l]

map_range_rule f (RangeRule r l i) = RangeRule (map_rule f r) (map (map_range f)
l) i
mapM_range_rule f (RangeRule r l i) = do
      r' <- mapM_rule f r
      l' <- mapM (mapM_range f) l
      return $ RangeRule r' l' i

-- | Order rule

data OrderRule = OrderRule (IndexSE, IndexSE) deriving Eq
```

```
instance Rule OrderRule where
      map_rule f (OrderRule (a,b)) = OrderRule (fmap f a, fmap f b)
      mapM_rule f (OrderRule (a,b)) = do
            a' <- fmapM_indexS f a
            b' <- fmapM_indexS f b
            return $ OrderRule (a', b')

      noExprForm (OrderRule (a,b)) = do
            a' <- integers a
            b' <- integers b
            return $ (OrderRel a' b', TOrder, OrderAttr)

      ruleType _ = TOrder

      construct_fp r = mempty { orderrule = [r] }
      change_rule s r fp = fp { orderrule = replace_elem s r (orderrule fp) }

      fold_rule f o@(OrderRule (IndexS _ l1,IndexS _ l2)) =
            fn l1 `mappend` fn l2
                  where
                        fn l = mconcat $ map (fold_expr f) l

      transCpp (OrderRule (a,b)) =
            ["Relation& entity = fp.AddRuleOrder()",
            "entity.SetFirst(" ++ transISECpp a ++ ")",
            "entity.SetSecond("++ transISECpp b ++ ")"]

instance Show OrderRule where
      show (OrderRule (a,b)) = show a ++ " < " ++ show b

-- | Property himself – df_size, location e.t.c.
-- | Df rule = Property rule with "df_size" property name

data ObjType = Cf | Df | Const | Some deriving (Show, Eq)
data Prop = Prop IndexSE String deriving (Eq, Ord)

instance Show Prop where
      show (Prop ise s) = show ise ++ "." ++ s

-- | Property rule

data PropRule = PropRule ObjType Prop Expr deriving Eq
```

```
instance Show PropRule where
      show (PropRule _ pr@(Prop ise s) e)
            | s == "df_size"  = "df " ++ show ise ++ " := block(" ++ show e ++
")"
            | otherwise             = show pr ++ " = " ++ show e

instance Rule PropRule where
      map_rule f (PropRule tp (Prop ise s) e) = (PropRule tp (Prop ise' s) e')
            where
                  ise' = fmap f ise
                  e' = f e

      mapM_rule f (PropRule tp (Prop ise s) e) = do
            ise' <- fmapM_indexS f ise
            e' <- f e
            return $ PropRule tp (Prop ise' s) e'

      noExprForm (PropRule objtp (Prop ise s) e) = do
            ise' <- integers ise
            e' <- lookForInt e
            return $ (PropNode ise' s, TProp, PropAttr e')

      ruleType _ = TProp

      construct_fp r = mempty { proprule = [r] }
      change_rule s r fp = fp { proprule = replace_elem s r (proprule fp) }

      fold_rule f (PropRule _ (Prop (IndexS _ l) _) e) =
            fold_expr f e `mappend` mconcat (map (fold_expr f) l)

      transCpp (PropRule _ (Prop ise s) e) =
            ["Property& entity = fp.AddProperty(" ++
                        show s ++ ", " ++
                        transISECpp ise ++ ", " ++
                        "vExpr() << " ++ transECpp e ++ ")"]

-- | Constant declaration rule /ConstType, ConstRule/

data ConstType = Static | Dynamic deriving Eq

instance Show ConstType where
      show Static       = "static"
```

```
      show Dynamic      = "dynamic"

data ConstRule = ConstRule ConstType IndexSE Expr deriving Eq

instance Show ConstRule where
      show (ConstRule tp ise e) = show tp ++ " const " ++ show ise ++ " = " ++
show e

instance Rule ConstRule where
      map_rule f (ConstRule tp ise e) = ConstRule tp (fmap f ise) (f e)
      mapM_rule f (ConstRule tp ise e) = do
            ise' <- fmapM_indexS f ise
            e' <- f e
            return $ ConstRule tp ise' e'

      noExprForm (ConstRule tp ise e) = do
            ise' <- integers ise
            e' <- lookForInt e
            return $ (ConstNode ise', TConst, ConstAttr { constType = tp,
constValue = e' } )

      ruleType _ = TConst

      construct_fp r = mempty { construle = [r] }
      change_rule s r fp = fp { construle = replace_elem s r (construle fp) }

      fold_rule f (ConstRule _ (IndexS _ l) e) =
            fold_expr f e `mappend` mconcat (map (fold_expr f) l)

      transCpp (ConstRule tp ise e) =
            ["Constant& entity = fp.AddConstant(" ++
                        transISECpp ise ++ ", " ++
                        "vExpr() << " ++ transECpp e ++ ", " ++ str tp ++ ")"]
            where
                  str Static  = "STATIC"
                  str Dynamic = "DYNAMIC"

-- | Cf rule /CfRule, IOType/

data IOType = In | Out | InOut deriving Eq

instance Show IOType where
      show In = "in"
```

```
        show Out = "out"
        show InOut = "inout"

data CfRule = CfRule {
        cf          :: IndexSE,
        code  :: String,
        var   :: [IndexSE], --list of variables that can be changed in runtime
        param       :: [Expr],
        df          :: [(IOType, IndexSE)]
} deriving Eq

instance Show CfRule where
        show ru =
                "cf " ++ show (cf ru) ++
                " := " ++ code ru ++ vars ++ params ++ dfs
                where
                        vars  = (fslist show "{" (var ru) "}" ", ")
                        params      = (fslist show "<" (param ru) ">" ", ")
                        dfs         = (fslist (\(tp,d) -> show tp ++ ": " ++ show d)
"(" (df ru) ")" "; ")

                        fslist _ _ [] _ _ = ""
                        fslist f l body r sep = l ++ intercalate sep (map f body) ++ r

instance Rule CfRule where
        map_rule f rl = rl {
                cf          = fmap f $ cf rl,
                var         = map (fmap f) $ var rl,
                param       = fmap f $ param rl,
                df          = map (\(tp, ise) -> (tp, fmap f ise)) $ df rl
                }

        mapM_rule f rl = do
                cf' <- fmapM_indexS f $ cf rl
                var' <- mapM (fmapM_indexS f) $ var rl
                param' <- mapM f $ param rl
                df' <- mapM fun $ df rl
                return $ rl { cf = cf', var = var', param = param', df = df'}
                where
                        fun (tp, ise) = do
                                ise' <- fmapM_indexS f ise
                                return $ (tp, ise')
```

```
	noExprForm rl = do
		cf' <- integers (cf rl)
		let code' = code rl
		var' <- mapM integers (var rl)
		param' <- mapM lookForInt (param rl)
		df' <- mapM (\(tp,ise) -> do { ise' <- integers ise; return
(tp,ise') }) (df rl)
		return (CfNode cf', TCf, CfAttr { cfAttrParam = param', cfAttrCode =
code', cfAttrVar = var', cfAttrDf = df' })

	ruleType _ = TCf

	construct_fp r = mempty { cfrule = [r] }
	change_rule s r fp = fp { cfrule = replace_elem s r (cfrule fp) }

	fold_rule f rl =
		mconcat $ map (fold_expr f) la
			where
				IndexS _ l = cf rl
				l' = concat [ls | IndexS _ ls <- var rl]
				l'' = param rl
				l''' = concat [ls | (_,IndexS _ ls) <- df rl]
				la = concat [l,l',l'',l''']

	transCpp rl = concat
		[[title],
		[cff		$ cf rl],
		[codef		$ code rl],
		map dff		$ df rl,
		map varf	$ var rl,
		[paramf		$ param rl]]
		where
			title		= "RuleCf& entity = fp.AddRuleCf()"
			cff ise			= "entity.SetCf(" ++ transISECpp ise ++
")"
			codef s			= "entity.SetCodeName(" ++ show s ++
")"
			dff (tp, ise)	= "entity.AddDf(" ++ str tp ++ ", " ++
transISECpp ise ++ ")"
			varf ise	= "entity.AddDynamicConst(" ++ transISECpp
ise ++ ")"
			paramf l	= "entity.SetParameter(" ++ transLECpp l ++
")"
```

```
                  str In                  = "IN"
                  str Out                 = "OUT"
                  str InOut         = error "Support of kw 'inout' not
implemented yet."

-- | Fragmented program

data FProgram = FProgram {
      fpname            :: String,
      construle   :: [RangeRule ConstRule],
      proprule    :: [RangeRule PropRule],
      cfrule            :: [RangeRule CfRule],
      orderrule   :: [RangeRule OrderRule]
} deriving Eq

-- [TODO]: ugly, ugly
{--
modify_fprogram :: Rule a => RangeRule a -> RangeRule a -> FProgram -> FProgram
modify_fprogram s@(RangeRule (ConstRule _ _ _) _) r src = FProgram {
      fpname            = fpname src,
      construle   = replace_elem s r (construle src),
      proprule    = proprule src,
      cfrule            = cfrule src,
      orderrule   = orderrule src}

modify_fprogram s r src = src
--}

map_fprogram :: (forall a. Rule a => RangeRule a -> RangeRule a) -> FProgram ->
FProgram
map_fprogram f fp =
      mempty { construle = map f $ construle fp } `mappend`
      mempty { proprule = map f $ proprule fp } `mappend`
      mempty { cfrule = map f $ cfrule fp } `mappend`
      mempty { orderrule = map f $ orderrule fp }

fold_fprogram :: (Monoid m) => (forall a. Rule a => RangeRule a -> m) ->
FProgram -> m
fold_fprogram f fp =
      mempty `mappend`
      (mconcat $ map f (construle fp)) `mappend`
      (mconcat $ map f (proprule fp)) `mappend`
      (mconcat $ map f (cfrule fp)) `mappend`
```

```
      (mconcat $ map f (orderrule fp))

-- This signature is blowing my mind!
foldM_fprogram :: (Monad m) => (forall a. (Rule a, Show a) => b -> RangeRule a
-> m b) -> b -> FProgram -> m b
foldM_fprogram f s fp = do
      x <- foldM f s $ construle fp
      y <- foldM f x $ proprule fp
      z <- foldM f y $ cfrule fp
      u <- foldM f z $ orderrule fp
      return u

instance Monoid FProgram where
      mempty = FProgram {
      fpname            = "a_noname_fp",
      cfrule            = [],
      proprule    = [],
      orderrule   = [],
      construle   = []
      }

      mappend a b = FProgram {
            fpname            = max (fpname a) (fpname b),
            cfrule            = cfrule a ++ cfrule b,
            proprule    = proprule a ++ proprule b,
            orderrule   = orderrule a ++ orderrule b,
            construle   = construle a ++ construle b
      }

-- take care of tabs around {, } and ;
indentCpp :: Int -> String -> String
indentCpp n s = unlines . filter (not . all isSpace) . lines . ind n $ tabs n ++
s
      where
            ind :: Int -> String -> String
            ind _ "" = ""
            ind n (x:xs)
                  | x == ';' = [x] ++ tabs n ++ ind n xs
                  | x == '{' = tabs n ++ [x] ++ tabs (n+1) ++ ind (n+1) xs
                  | x == '}' = tabs (n-1) ++ [x] ++ tabs (n-1) ++ ind (n-1) xs
                  | otherwise= x : ind n xs
            tabs n = '\n' : (take n $ repeat '\t')
```

```
transFPCpp :: FProgram -> String
transFPCpp fp =
      "fp.SetFPName(" ++ (show $ fpname fp) ++ ");" ++
      (unlines . map f . concat)
      [map transRuleCpp $ construle fp,
      map transRuleCpp $ proprule fp,
      map transRuleCpp $ cfrule fp,
      map transRuleCpp $ orderrule fp]
      where
            f s = "{" ++ s ++ "}"

instance Show FProgram where
      show fp = (endByConcat "\n" . map (endByConcat ";\n")) [a,b,c,d,e]
            where
                  a = ["program " ++ fpname fp]
                  b = tol construle
                  c = tol proprule
                  d = tol cfrule
                  e = tol orderrule
                  tol f = map show $ f fp
                  endByConcat sep l = intercalate sep l ++ (if null l then ""
else sep)

-- | Settings like file with code procedures e.t.c

type Setting = Map.Map String [String]
```

Файл Main.hs, в используются функции из предыдущих двух файлов и осуществляется непосредственно парсинг текста программы и его трансляция в C++ код.

```
module Main where

import FProgram
import Parser
import FPSemantic
import Common

import Data.List
import System.IO
import System.Environment
```

```
import System.Console.ANSI    --for Red, Blue constructors of data Color

-- find out all occurunce of index variables in expression around fprogram and
add to them
-- index of corresponding index variable
index_var_replace :: FProgram -> FProgram
index_var_replace fp =
      fp {
            construle   = mp_rule construle,
            cfrule            = mp_rule cfrule,
            proprule    = mp_rule proprule,
            orderrule   = mp_rule orderrule
      }
      where
            mp_rule getter = map mf $ getter fp
            mf r = map_range_rule (map_expr (f (get_range_vars r))) r

            f :: [String] -> Expr -> Expr
            f v c@(ExprConst (IndexS s [])) =
                  case s `elemIndex` v of
                        Just i      -> IndexVar s $ fromIntegral i
                        Nothing     -> c
            f v e = e

process_ast :: FProgram -> FProgram
process_ast = index_var_replace

translateFpCpp :: FilePath -> FProgram -> IO ()
translateFpCpp fname fp = do
      let cppSrc = indentCpp 1 . transFPCpp $ fp
      h <- readFile "translate/header.cpp"
      f <- readFile "translate/footer.cpp"
      writeFile fname $ h ++ cppSrc ++ f

main = do
      pname <- getProgName
      args <- getArgs
      if length args /= 2
            then error $ "Usage: " ++ pname ++ " <source .fp file>
<destination .cpp file>"
            else return ()
      let [srcFile, destFile] = args
      s <- readFile srcFile
```

```
let pr = get_ast s
case pr of
      Left err          -> do
            clr_print Red err
            error "Parsing failed"
      Right (fp,st)     -> do
            let pfp = process_ast fp
            putStrLn "Successful parsing."
            clr_print Blue "Parsed settings:"
            print st
            clr_print Blue "Parsed program:"
            print pfp
            clr_print Red "Semantic analisys..."
            let pfp' = semantic_check pfp
            putStrLn $ show pfp'
            if not $ isDiagOk pfp'
                  then error "Compile process failed because of errors."
                  else return ()
            let npfp = getDiagCore pfp'
            clr_print Red "Updated fprogram:"
            putStrLn $ show npfp
            translateFpCpp destFile npfp
            clr_print Blue $ "Cpp code was written to " ++ destFile
```