САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Климов Иван Олегович

Трансляция функциональных языков программирования в императивные

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А.Н.

Научный руководитель:
Магистр информационных технологий, ст. преп. Григорьев С. В.

Рецензент:
Ведущий архитектор, ООО "Belkasoft" Тимофеев Н. М.

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Klimov Ivan

Translation of functional programming languages into imperative ones

Graduation Thesis

Admitted for defence.
Head of the chair:
DSc, Professor A.N. Terekhov

Scientific supervisor:
Master of Information Technology, sen. lector Semyon Grigoriew S.V. Grigoriev

Reviewer:
N. M. Timofeev

Saint-Petersburg
2014

Оглавление

Введение

На данный момент компьютерные языки развиваются и совершенствуются. Наибольшей интерес представляют два молодых и активно прогрессирующих класса языков высокого уровня (ЯВУ). Многие алгоритмические решения задач удобно записываются математическими терминами, и в свою очередь синтаксис функциональных языков близок к данной терминологии, что позволяет упростить реализацию алгоритма на вычислительном устройстве. Но не смотря на удобство реализации, функциональные языки известны не самой высокой производительность, в отличии от языков, которые позволяют производить вычисления на графических процессорах общего назначения(General-purpose graphics processing units, GPGPU) и распараллеливать вычисления там, где это возможно. Но минусом подобных языков, что для их использования необходимо учитывать их особую специфику для взаимодействия с GPGPU, что может значительно замедлить процесс разработки. Откуда и появляется необходимость трансляции функциональных языков в специфические для GPGPU языки.

С каждым годом всё больше и больше появляется сложных программных задач достаточно трудоёмких и, занимающих существенное время вычислений, поэтому распараллеливание программы в тех частях кода, где это возможно, является важной задачей. Перспективным направлением в этой области выглядит использование возможностей графических процессоров, активно развивающихся в последнее время. Выигрыш при использовании GPGPU можно получить за счёт массового параллелизма.

Одними из важных факторов для разработки больших программных продуктов, является удобство языка и скорость разработки на нём, не зависимо от того, на каком устройстве ведётся разработка. Реализация решение задач на машинном коде или хотя бы ассемблере привело бы к большому временному периоду разработки и скорее всего не удалось сразу избежать ошибок в огромном и трудном для чтения коде. Высокоуровневые языки прекрасно подходят для разборки сложных программных продуктов. В них обычно присутствует поддержка множества полезных структур данных и различных операций для работы с ними. Важным плюсом является их кроссплатформенность, а производительность одного и того же алгоритма на том же языке зависит только от его транслятора.

В рамках данной работы в качестве ЯВУ был выбран функциональный язык F# [6], на котором можно распараллелить. В F# присутствует строгая типизация, неявные преобразования типов полностью отсутствуют, что полностью исключает ошибки, связанные с приведением типов. В данной работе будет описана трансляция важных для функционального языка конструкций в язык способный взаимодействовать с графическим процессором.

В качестве целевого языка способного взаимодействовать с GPGPU был выбран

OpenCL [5], из-за своего широкого круга поддержки графических процессоров общего назначения.

Данная работа описывает реалзиацию расширения транслятора основных функциональных конструкций F# в язык, позволяющий взаимодействовать с GPGPU. А именно доработка функционала библиотеки Brahma.FSharp [2] для трансляции языка F# в OpenCL. В Brahma.FSharp уже реализована трансляция простейших императивных конструкций, но всё ещё не хватает поддержки функциональных конструкций. В частности трансляции вложенных функций, которых тривиально не удастся транслировать в императивный язык без преобразований над абстрактным синтаксическим деревом (Abstract syntax tree ,AST).

1. Постановка задачи

Целью данной работы является интеграция ЯВУ с вычислениями на GPGPU, что позволит увеличить производительность вычислений критически медленных задач за счёт массового параллелизма. Для достижения данной цели была выбрана библиотека Brahma.FSharp и поставлены следующие задачи.

- Изучить особенности языка F#.
- Разработать шаблоны трансляции в язык OpenCL.
- Реализовать поддержку трансляции вложенных функций.
- Апробация.

  Применение расширенной библиотеки Brahma.FSharp для EigenCFA[3]

2. Обзор существующих решений

В обзоре существующих решений рассмотрены участвующие в трансляции языки, описаны используемого в ходе работы инструмента, а так же их аналоги.

2.1. Рассматриваемые ЯВУ

2.1.1. F# как исходный язык

Благодаря растущей популярности и его мощности в качестве исходного ЯВУ был выбран функциональный язык программирования F#. На самом деле F# является мультипарадигмальный языком, включающий в себя и функциональный стиль, и императивный, и объектно-ориентированный. Синтаксис F# построен на математической нотации, а программирование чем-то похоже на алгебру, что делает F# похожим на Haskell.

Пример кода на F# (в алгебре эквивалентен $f(x) = x - 3$):

let f x = x - 3

2.1.2. F# Quotations

Quotations[1] — это выражение языка F# в коде, выделенное таким образом, что оно не компилируется как часть программы, а компилируется в объект, представляющий выражение языка F#. Цитируемое выражение можно отметить двумя способами: со сведениями о типе либо без сведений о типе.

Используя F# Quotations можно получить дерево представление цитируемого выражения. Это позволяет писать код, который принимает код, написанный на F # в качестве данных и выполняет некоторые анализ кода или компилирует этот код на другом языке.

2.1.3. OpenCL как целевой язык

В качестве целевого язык способного взаимодействовать с GPGPU был выбран OpenCL. OpenCL — открытый стандарт для GPGPU, который разрабатывается консорциумом Khronos, и по замыслам, позволит использовать мощности GPU на различных программных и аппаратных платформах. Во фреймворк OpenCL входят язык программирования, который базируется на стандарте C99, и интерфейс программирования приложений. OpenCL обеспечивает параллелизм на уровне кода и на уровне данных.

Отличие от привычного языка C.

[1] F# Quotations: http://msdn.microsoft.com/en-us/library/dd233212.aspx

- Отсутствие поддержки указателей на функции, рекурсии, битовых полей, массивов переменной длины, стандартных заголовочных файлов.
- Расширения языка для параллелизма: векторные типы, синхронизация, функции для Work-items/Work-Groups.
- Квалификаторы типов памяти: __global, __local, __constant, __private.
- Иной набор встроенных функций.

2.2. Существующие инструменты трансляции

Идея трансляции F# в код для взаимодействия с GPGPU не новая. И существует уже хорошие инструменты для трансляции такие, как Alea.cuBase [1] и FSCL [4].

2.2.1. Alea.cuBase

Особых сложностей в освоении библиотеки Alea.cuBase нет, документация достаточно удобная для изучения. В основном необходимо освоить как правильно реализовывать и запускать ядро, которое в дальнейшем будет исполняться на GPGPU. Установка библиотеки подробно описано в документации и не занимает много времени. Имеется возможность установки с помощью NuGet [2] в Visual Studio [3].

Огромным плюсом является возможность контроля количество запускаемых потоков, как в CUDA [4]. Всю логику исполнения можно так же реализовывать в самом ядре, что позволяет экономить на копировании данных с GPGPU, что значительно замедляет процесс вычислений. Есть транслирование кода на этапе компиляции, что позволяет не затрачивать время на неё во время исполнения кода.

В Alea.cuBase произведена поддержка вложенных функций.

Из-за сильной привязки к CUDA, библиотека не имеет возможности работать на ATI GPGPU. Alea.cuBase является коммерческим продуктом с платной лицензией.

2.2.2. FSCL

FSCL — открытый компилятор из F# в OpenCL , направленной на повышение абстракции над OpenCL программирования, позволяющий программистам выражать OpenCL ядро внутри F#. F# на виртуальной машине, позволяет автоматизировать большую часть сил потраченных для запуска кода на OpenCL.

Для установки данной библиотеки потребуется потратить значительно больше времени, но не критично. Пользоваться библиотекой можно на примерах, представленных в проекте библиотеки.

[2]Пакетный менеджер NuGet для платформы разработки Microsoft: http://docs.nuget.org/
[3]Среда разработки Microsoft Visual Studio: http://www.visualstudio.com/
[4]Программно-аппаратная архитектура CUDA от компании NVIDIA http://www.nvidia.com

2.2.3. Brahma.FSharp

Brahma.FSharp молодая и активно развивающаяся библиотека. Как Alea.cuBase и FSCL основана на трансляции F# Quotations в OpenCL. Что позволяет использовать эту библиотеку не только на nVidia GPGPU, но на других процессорах поддерживающих OpenCL. Проста в установке и освоении.

Так же имеется возможность контроля запускаемых потоков. Огромным плюсом Brahma является поддержка атомарных операций. Данная библиотека является бесплатной. Из-за использованию ранее на практике Brahma.FSharp, то хотелось бы заняться доработкой именной этой библиотеки.

3. Особенности архитектуры Brahma.FSharp

Чтобы рассказать более подробно о технической части работы, нужно описать устройство библиотеки Brahma.FSharp, внутри которой и была реализована данная работа. Глубокого вдаваться в её устройство производить не будем, так как не имеет смысла погружать в детали, которые не использовались в рамках данной работы.

3.1. Код ядра

Для получения методов способных выполняться на графическом процессоре, используются F# Qountations, которые позволяют получать промежуточное представление кода и в дальнейшем производить трансляцию относительно просто на основе стандартного модуля. Данный модуль содержит коллекцию частично параметризованные активные шаблоны для подмножества языковых конструкций F#. Сама трансляция будет производится в модуле Brahma.FSharp.OpenCL.Translator, в котором и будет производится основная часть данной работы.

Необходимо отметить, что на трансляцию накладываются ограничения связанные с OpenCL, в частности отсутствие поддержки рекурсивных функций. Важны ограничением является то, что в коде ядра возможны только массивы и числовые типы данных. Так же содержаться специфичные для OpenCL служебные классы, содержащие, например количество одновременно запускаемых потоков и размер рабочей группы.

В основном в ядре в проекте Brahma.FSharp.OpenCL.Core и задаются параметры с которыми код будет исполнятся на устройстве. В частности, использование нативного кода, уменьшение точности дробных чисел для ускорения арифметический операций, а так же упомянутые ранее конфигурации запускаемых потоков, групп и памяти используемой ими.

3.2. Трансляция

Одним из ключевых моментов при преобразовании F# кода в OpenCL, является этап трансляции кода. Основным классом является FSQountationToOpenCLTranslator, включающий в себя метод Translate разбитый на два модуля Body и QuotationsTransformer, в которых и происходят необходимые преобразования.

В QuotationsTransformer происходит преобразование дерева, полученного при обработке F# Qountations. Дерево трансформируется таким образом, чтобы в дальнейшем можно было из функциональных конструкций получить корректный императивный код. Данная работа посвящена в основном именно этому разделу и более подробно в следующей главе.

После необходимых преобразований, полученное дерево обрабатывается в модуле

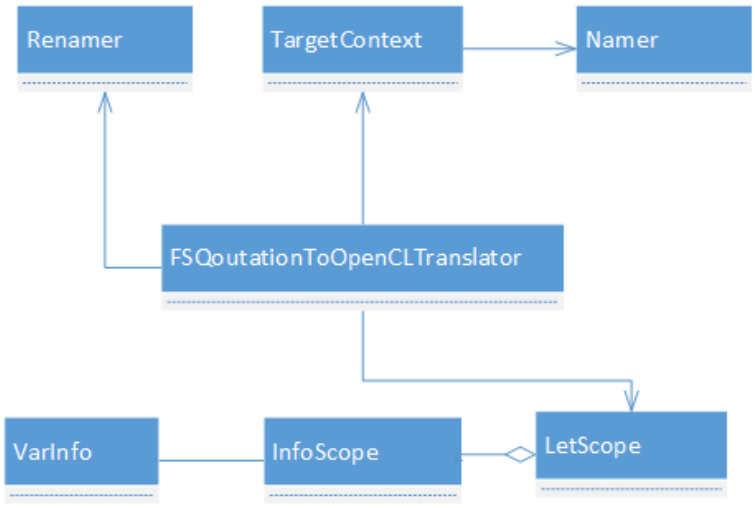

Рис. 1: Схема проекта

Body. На данном этапе происходит построение специфичного для OpenCL абстрактного синтаксического дерева с учётом поддерживаемых конструкций трансляции.

3.3. Абстрактное представление

После преобразований, произведённых в трансляторе, получается абстрактное синтаксическое дерево, обычно элементами верхнего уровня являются несколько #pragma-выражений, специфичных для OpenCL, а так же функции ядра и объявления типов.

В модуле Brahma.FSharp.OpenCL.AST присутствует класс Statement, который является родителем для классов поддерживаемых конструкций, котором объявление типов содержит список аргументов, тип возвращаемого значения и тело функции.

3.4. Генерация кода

Окончательный код печатается с помощью проекта Brahma.FSharp.OpenCL.Printer используя рекурсивный обход AST. В результате получается код OpenCL C. В качестве целевого языка он был выбран, так как OpenCL является открытым стандартом и его используют не только GPGPU.

В результате такого обхода получается экземпляр класса Layout, входящего в коллекцию инструментов и библиотек F# PowerPack [5]. Отформатированный код в дальнейшем печатается используя готовую функцию из той же коллекции.

[5]Коллекция инструментов и библиотек для F: http://fsharppowerpack.codeplex.com/

4. Особенности реализации

Так как трансляция функциональных конструкций в императивный код вызывают большие сложности, то необходимо произвести преобразование над ними таким образом, чтобы это стало возможным. Все необходимые преобразования производятся на этапе трансляции кода.

4.1. Трансляция вложенных функций

Чтобы разобрать в то, что такое вложенные функции, рассмотрим на примерах изображённых ниже. Откуда станет понятно для чего нужно производить необходимые преобразования программы.

Рассмотрим небольшой F# код и примерно равносильный ему Си код.

```
let f(x:int) =
    x + 1

f 3
```

```
int f(int x) {
return x + 1; }
void main() {
... f(3) ...}
```

Выше слева код на F#, а справа код на языке Си (без преобразований над деревом). Ниже представлен аналогичные соответствия, но уже с вложенными F# функциями.

```
let f (x:int) =
    let g (y:int) =
        y - 1
    g x
f3
```

```
int f(int x) {
    int g(int y) {
        return y - 1;}
    return g(x); }
void main() {... f(3);...}
```

Очевидно, что без каких либо преобразований, полученный код упадёт с ошибкой на этапе компиляции. Чтобы этого избежать, были реализованы классы Renamer, VarInfo, InfoScope, LetScope и расширен модуль QuotationsTransformer.

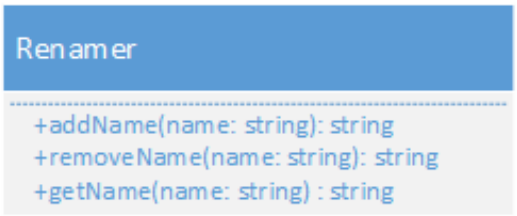

Рис. 2: Класс Renamer

Класс **Renamer** позволяет переименовывать названия уникальным образом.

Метод *addName* принимает на вход строку, переименовывает её (если это необходимо) и кладёт её в словарь. Ключом в словаре является исходное имя, а значением стек уникальных имён. Возвращает новое уникальное имя.

Метод *removeName* удаляет верхнее уникальное имя со стека по ключу, который подаётся на вход. Возвращает удаляемое уникальное имя.

Метод *getName* по ключу возвращает его последнее уникальное имя.

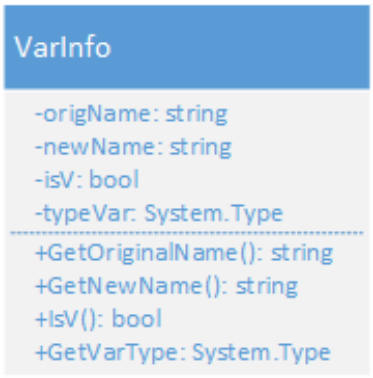

Рис. 3: Класс VarInfo

Класс **VarInfo** предназначен для хранения информации о переменной (функции).

Экземпляр класса хранит оригинальное имя, новое уникальное имя, тип, а так же информацию о том, являлся это объект именем переменной или именем функции в isV.

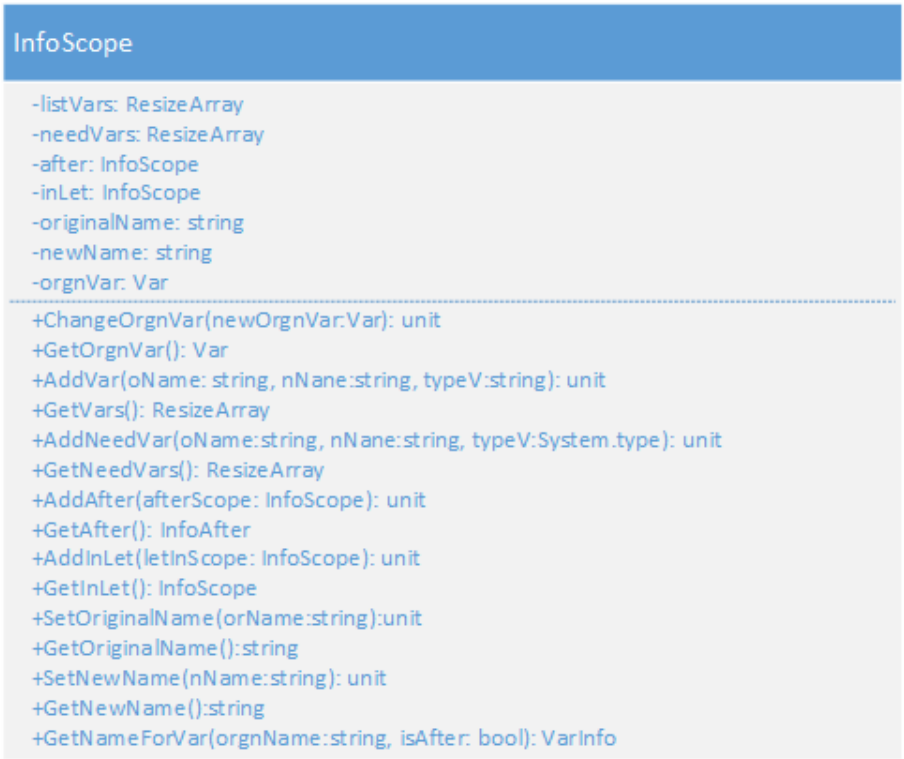

Рис. 4: Класс InfoScpe

Чтобы произвести необходимые преобразования над кодом, необходимо хранить информацию о каждом let. Экземпляр класса **InfoScope** хранит следующую информацию:

- listVars – список входных параметров. Элемент списка имеет тип VarInfo;
- needVars – список входных параметров, которые необходимо добавить для дальнейшей корректной трансляции;
- after – ссылка на объект типа InfoScope, в теле которого находился текущий let в исходном коде;
- inLet – ссылка на объект типа InfoScope, в выражении которого находился текущий let в исходном коде;
- originalName – оригинальное имя let;
- newName – новое уникальное имя let;
- orngVar – переменная в первую очередь предназначенная для хранения типа возвращаемого значения.

Для каждого из этих полей есть свои get и set/add методы. Метод *GetNameForVar* используется для поиска нового имени для переменной вызываемой в выражении или теле let.

Рис. 5: Класс InfoScpe

Объект класса **LetScope** содержит в себе все список словарь и стек InfoScope и методы для работы. Вспомогательным является поле isInLastLet, которое говорит нам о том находимся мы в выражении последнего let или в его теле.

Метод *AddLetInfo* принимает новый объект типа InfoScope и добавляет его в словарь allLet.

Метод *LetIn* вызывается, когда заходим в очередной let. Если данный метод вызван, внутри выражения внешнего let, то добавляется текущий let в словарь и добавляется информацию о внешнем. Если вызван в теле let, то добавляется информацию к текущему о let в теле, которого находимся и информацию let о внешнем let. После чего кладём новое имя текущего let в lastInLet.

Метод *LetOut* удаляет из стека имя последнего let, в который заходили.

Метод *GetNameForVarInLet* определяет новое имя для переменной вызываемой в выражении или теле let.

4.1.1. Подготовительный этап

На данном этапе происходит переименовывание необходимых вложенных конструкций, чтобы при дальнейшей трансляции избежать коллизии в именах функций и аргументов. А также запоминаем исходную зависимость конструкций, чтобы не растерять её на этапе преобразования. Для всех этих операций были реализованы следующие классы.

- Renamer — класс, который позволяет переименовывать названия уникальным образом.
- VarInfo — класс, который хранит в себе информацию нового и старого имени переменной.
- nfoScope — класс, хранящий информацию для конкретного let и методы для работы с этой информацией.
- Leinfo - класс, содержащий в себе InfoScope и методы для работы с ними.

Let в F# в терминах F# Qountations имеет тип (Var * Expr * Expr). Если более точно то он имеет переменную (var:Var), в которой хранится название let, выражение (ExprIn:Expr) входящее в него и выражение (Body:Expr), следующее за ним. Далее опишем алгоритм подготовительного этапа.

Необходимо рекурсивно спускаться по дереву и переименовывать (если это нужно) функций let, а так же имена вложенных в них Lambda-выражений.

Когда заходим в ExprIn и встречаем let, то переименовываем его, запоминаем информацию, о том в какой из let он был вложен.

Если мы заходим в ExprIn и встречаем Lambda-выражение, то переименовываем переменную выражения и добавляем в последний let, в который мы зашли, информацию нового и старого имени переменной, входящий в let.

Когда заходим в Body и встречаем let, то переименовываем его, запоминаем информацию, о том в какой из let он был вложен и за каким let он идёт.

Следующая часть алгоритма касается переменных и вызовов методов, которые находятся в ExprIn и Body. Для ясности let, которые в исходном коде обладали аргументами далее называются let-функциями, а без аргументов let-переменными.

Если находимся в ExprIn, то когда встречаем переменную, то рекурсивно поднимаемся вверх по информационном дереву, которое мы строили, когда спускались по исходному. Для начала проверяем наличие этой переменной (поиск производим по старому имени) в том let, в котором находимся, если есть, то получаем новое имя. В том случае если не нашли, то смотрим по названиям(по старыми именам) в предшествующих let, если это так, то переименовываем, и если мы вышли за пределы let-функции, то рекурсивно добавляем ко всем let-функциям имена переменных в список neededVars, предшествующим текущему let. Если и предшествующих не нашли, то продолжаем подниматься вверх, но теперь при встрече нужной lambda-переменной или let-переменной, мы добавляем все необходимые переменные в neededVars , во все предшествующие let-функции. Благодаря строгой типизации в F#, нам удастся найти нужный нам метод или lambda-переменную.

Если находимся в выражении Body, то поступаем аналогично предыдущему шагу. Только не будем искать в переменных текущего let. А так же при встречи нужной компоненты, не добавляются необходимые переменные в список neededVars текущего let.

4.1.2. Этап перестроения.

Переименованное дерево на данном этапе честно рекурсивно перестраиваем таким образом, что каждый вложенную let-функцию с его ExprIn выталкиваем на верх. Аналогично для let-переменных выталкиваем наверх, но не выходя за пределы let-функции, в которую были вложены. При этом сохраняется очерёдность, так как производим это рекурсивно для каждого let. В результате получается вытянутое дерево. Которое в дальнейшем будет легко разбить на и список функций и продолжить этап трансляции в модуле Body. Ниже представлен пример преобразований.

```
let f x y =
    let y = y
    let y = y
    let g x m = m + x
    g x y
f 1 7
```

```
let g x0 m = m + x0
let f x y =
    let y0 = y
    let y1 = y0
    g x y1
f 1 7
```

4.2. Шаблоны трансляции

Для внесение ясности в расширенный функционал Brahma.FSharp, далее рассматриваются основные шаблоны трансляции.

Шаблон выноса вложенных функций.

```
let f x =
    let g y = 5 + y
    let m k = k - 1
    g x
f 5
```

```
let g y = 5 + y
let m k = k - 1
let f x =
    g x
f 5
```

Шаблон выноса вложенных функций позволяет выносить любое количество вложенных функций на внешний уровень.

Шаблон рекурсивного выноса функций.

```
let f x =
    let g y =
        let m k = k - 1
        5 + (m y)
    g x
f 5
```

```
let m k = k - 1
let g y = 5 + (m y)
let f x =
    g x
f 5
```

Шаблон рекурсивного выноса функций позволяет выносить вложенные функции рекурсивно на самый верхний уровень.

Шаблон переименования вложенных функций.

```
let f x =
    let f y =
        5 + y
    f x
f 5
```

```
let f0 y = 5 + y
let f x =
    f0 x
f 5
```

Шаблон переименования вложенных функций позволяет избегать коллизий при трансляции и переименовывает вложенные функции там, где это необходимо.

Шаблон рекурсивного переименования вложенных функций.

```
let f x =
    let f y =
        let f k = k - 1
        5 + (f y)
    f x
f 5
```

```
let f1 k = k - 1
let f0 y = 5 + (f1 y)
let f x =
    f0 x
f 5
```

Шаблон рекурсивного переименования вложенных функций позволяющий рекурсивно переименовывать вложенные функции для избегания коллизий при дальнейшей трансляции.

Шаблон сохранения аргументов вложенной функции.

```
let f x =
    let m =
        let g k = k - 1
        5 + (g y)
    m + x
f 5
```

```
let g k = k - 1
let f x =
    let m =
        5 + (g y)
    m + x
f 5
```

Шаблон сохранения аргументов вложенной функции позволяет не терять лямбда-аргументы вложенной функции при преобразовани исходного кода и для дальнейшей трансляции.

Шаблон переименования аргументов вложенных функций.

```
let f x =
    let g f x =
        x + f
    g x 5
f 5
```

```
let g f0 x0 =
        x0 + f0
let f x =
    g x 5
f 5
```

Шаблон переименования аргументов вложенных функций позволяет переименовывать переменные вложенной функции для избегания коллизий при дальнейшей трансляции. Так же присутствует схожий с ним шаблон для рекурсивного переименования аргументов вложенных функций.

Шаблон добавления аргументов во вложенные функции.

```
let f x =
    let k = 7
    let g y =
        k + y
    g x
f 5
```

```
let g k y = k + y
let f x =
    let k = 7
    g k x
f 5
```

Данный шаблон позволяет добавлять необходимые аргументы во вложенные функции при выносе их на верхний уровень.

Шаблон подъёма вложенных функций.

```
let f =
    let r = 7
    let k =
        let g y =
            5 + y
        g r
    let m t = t - 1
    m k
f
```

```
let g y = 5 + y
let m t = t - 1
let f =
    let r = 7
    let k =
        g r
    m k
f
```

Шаблоны подъёма позволяет вложенным let-функциям подниматься на верхний уровень, учитывает порядок их объявления и не нарушает корректность кода.

5. Апробация

5.1. Алгоритм EigenCFA

EigenCFA – это алгоритм, позволяющий производить статический анализ кода с помощью многопоточного устройства. Для получения большой производительности на GPGPU алгоритм статического анализа кода Shivers' 0CFA[6] для continuationpassing style (CPS)[7] специализировали в каноническую форму binary continuation-passing style (binary CPS). Граматика для binary CPS содержит вызовы и выражения, а выражения являются *lambda*-термом или переменной:

$call \in Call ::= (f\ e_1\ e_2)$

$f, e \in Exp = Var + Lam$

$v \in Var\ is\ a\ set\ of\ identiers$

$lam \in Lam ::= (\lambda(v_1\ v_2)\ call).$

В дальнейшем все вызовы ($call$), функции (f), аргументы (e_1, e_2), и переменные (v) последовательно пронумеровываются. Основываясь на этой нумерации строятся матрицы зависимостей.

while $\boldsymbol{\sigma}$ *changes* **do**
  foreach *call* **do**
    // Lookup function and arguments in *call*
    $\vec{L} := (\langle\langle call \rangle\rangle \times \mathbf{Fun}) \times \boldsymbol{\sigma}$
    $\vec{L}_1 := (\langle\langle call \rangle\rangle \times \mathbf{Arg}_1) \times \boldsymbol{\sigma}$
    $\vec{L}_2 := (\langle\langle call \rangle\rangle \times \mathbf{Arg}_2) \times \boldsymbol{\sigma}$

    // Formal arguments of function, $\vec{L}$
    $\vec{v}_1 := (\vec{L} \times \mathbf{Var}_1) \times \boldsymbol{\sigma}$
    $\vec{v}_2 := (\vec{L} \times \mathbf{Var}_2) \times \boldsymbol{\sigma}$

    // Update store
    $\boldsymbol{\sigma} := \boldsymbol{\sigma} + \underbrace{\vec{v}_1^{\top} \times \vec{L}_1}_{Bind\ L_1\ to\ v} + \underbrace{\vec{v}_2^{\top} \times \vec{L}_2}_{Bind\ L_2\ to\ v'}$
  end
end

Рис. 6: Алгоритм EigenCFA.

[6]Control Flow Analysis in Scheme / Olin Shivers:http://www.iro.umontreal.ca/ feeley/cours/ift6232/doc/cfa-in-scheme.pdf

[7]http://en.wikipedia.org/wiki/Continuation-passing_style

Более детальное описание алгоритма в статье "EigenCFA: Accelerating Flow Analysis with GPUs"[3]

Для финального тестирования расширеной версии Brahma.FSharp на алгоритме EigenCFA использовалась следуюя конфигурация оборудования.

- Двухъядерный процессор Intel Core i5-2430 с тактовой частотой 2.40 ГГц.
- Видеокарта GeForce GT 540M 2 ГБ GDDR3.
- Оперативная память 3x2 ГБ DDR3 с частотой 663 МГц.
- Жёсткий диск 7200 оборотов в минуту.

5.2. Сравнение производительности

Тестирование производилось на случайно сгенерированных данных. По оси ординат представлено время в секундах. По оси абсцисс данные для EigenCFA (количество обрабатываемых lambda-выражений на количество их вызовов).

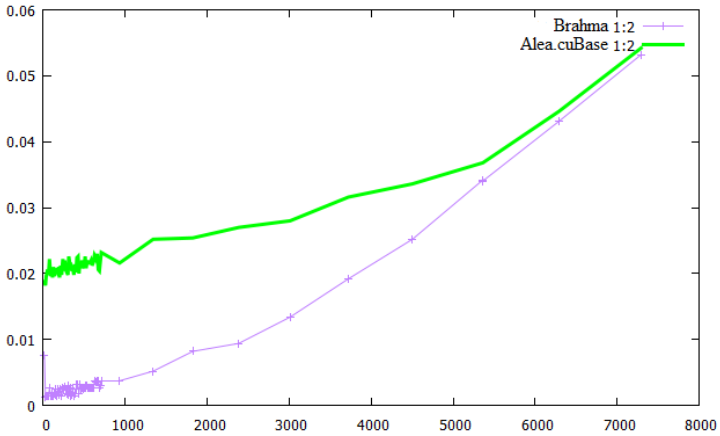

Рис. 7: Алгоритм EigenCFA.

По результатом тестирования на алгоритме EigenCFA выявлено, что расширенная Brahma.FSharp не уступает по производительности аналогичной библиотеки Alea.cuBase. Важным замечанием являтеся то, что Brahma.FSharp позволяет производить вычисления не только на графических процессорах NVIDIA.

Заключение

Результаты

- Изучить особенности языка F#, откуда было выявлено, что удобным способом получения абстрактного синтаксического дерева исходного пода можно получить с помощью F# Qountations и в дальнейшем производить необходимые преобразования над ним.

- Разработаны и успешно протестированы шаблоны трансляции F# кода в OpenCL для вложенных функций.

- Разработан алгоритм позволяющий делать преобразования над деревом, с помощью которой появилась возможность трансляции вложенных функций. Это значительно упрощает процесс разработки алгоритмов на основе библиотеки Brahma.FSharp.

- Апробация.

  Применена расширенная библиотеки Brahma.FSharp для алгоритма статического анализа кода EigenCFA. По полученным данным выявлено, что расширенная версия Brahma.FSharp не уступает по производительности Alea.cuBase на алгоритме EigenCFA.

Список литературы

[1] Alea.cuBase Documentation. — https://www.quantalea.net/products/resources/.

[2] Brahma.FSharp. — https://sites.google.com/site/semathsrprojects/home/brahma-fsharp.

[3] Prabhu Tarun, Ramalingam Shreyas, Might Matthew, Hall Mary. EigenCFA: Accelerating Flow Analysis with GPUs. — 2011. — http://matt.might.net/papers/prabhu2011eigencfa.pdf.

[4] FSCL Documentation. — https://github.com/GabrieleCocco/FSCL.Compiler.

[5] The OpenCL Specification. — https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[6] Syme Don, Granicz Adam, Cisternino Antonio. Expert F# 3.0. — Apress, 2012. — ISBN: 978-1-4302-4650-3. — URL: `http://www.apress.com/9781430246503`.