Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Реализация языка для обработки графов

Дипломная работа студента 544 группы
Крутихина Егора Александровича

Научный руководитель	.... /подпись/	к. ф.-м. н. Д.Ю. Булычев
Рецензент	.... /подпись/	Д.С. Ломов
"Допустить к защите" зав. кафедры	.... /подпись/	д. ф.-м. н., проф. А.Н. Терехов

Санкт-Петербург
2007

Содержание

1 Введение **3**

2 Существующие подходы **4**
2.1 Библиотеки и синтаксические расширения традиционных языков . . . 4
2.2 PROGRES . . . 6
2.3 GXL . . . 7

3 Постановка задачи **8**

4 Описание языка GRAIL **9**
4.1 Основные языковые конструкции . . . 10
4.1.1 Значения и образцы . . . 10
4.1.2 Абстракции и применения . . . 12
4.1.3 Множества, включения множеств и задание порядка 13
4.1.4 Управляющие конструкции . . . 15
4.1.5 Операции . . . 15
4.1.6 Локальное связывание . . . 16
4.1.7 Другие конструкции . . . 16
4.2 Конструкции, специфичные для графов . . . 17
4.2.1 Конструкторы графов . . . 17
4.2.2 Let-связывание для графов . . . 19
4.2.3 Операции над графами . . . 19
4.2.4 Графовые образцы и запросы . . . 19
4.3 Примеры алгоритмов на языке GRAIL . . . 23
4.3.1 Простейшие примеры . . . 23
4.3.2 Нумерации вершин . . . 25

5 Описание реализации **28**
5.1 Система типов . . . 28
5.2 Проекции конструкций языка GRAIL в язык Haskell . . . 29
5.2.1 Простой конструктор графа . . . 30
5.2.2 Включение графов . . . 31
5.2.3 Графовые запросы . . . 32
5.3 Внутреннее представление графов . . . 34
5.4 Взаимодействие с другими системами и библиотеками . . . 37

6 Применение GRAIL: задача о поиске всех вхождений **37**

7 Заключение **42**

1 Введение

Понятие графа является одним из фундаментальных в математике и информатике. Огромное количество теоретических и практических задач сводится к алгоритмам на графах. Сейчас существует достаточно много специализированных библиотек для практически всех популярных языков программирования, в которых многие фундаментальные алгоритмы уже реализованы. Однако далеко не все задачи можно решить комбинацией типовых алгоритмов. Кроме того, многие из этих алгоритмов (например, проверка изоморфности графов, поиск всех вхождений) имеют большую вычислительную сложность, а потому в применении к реальным задачам требуют внесения каких-либо эвристик, ускоряющих работу алгоритма для некоторого класса задач.

Скорость работы алгоритма чаще всего определяется именно подобными эвристиками с учётом знания предметной области решаемой задачи, а не эффективностью представления графа, быстродействием используемого языка программирования и аналогичными низкоуровневыми показателями. Однако описание таких эвристик порой оказывается намного сложнее основного алгоритма и требует более высокого уровня абстракции, которого не добиться в терминах универсальных языков программирования: программы получаются громоздкими, сложными для понимания и модификации. Возникает потребность в создании выразительного специализированного языка, позволяющего описывать сложные операции над графами в лаконичной и понятной форме.

Неформальным показателем выразительности подобного языка можно считать малое количество стандартных функций: его синтаксис должен быть настолько гибким, чтобы выражать большинство типовых операций с помощью небольшого количества языковых конструкций. Подобные синтаксические конструкции должны вбирать в себя функциональность наиболее часто используемых операций для работы с графами. Можно выделить две основные группы подобных операций: операции, предназначенные для создания графов, а также операции по извлечению некоторой информации из имеющегося графа и представлению её в удобном для дальнейших модификаций виде. Дополнительным плюсом для такого языка можно считать как можно более декларативный характер.

2 Существующие подходы

Данный раздел содержит обзор основных подходов в создании платформ, позволяющих описывать различные алгоритмы и операции на графах. Все эти решения можно классифицировать следующим образом:

- библиотеки для традиционных языков программирования;
- синтаксические расширения существующих языков программирования;
- самостоятельные специализированные языки;
- полноценные среды разработки со своими специализированными языками.

2.1 Библиотеки и синтаксические расширения традиционных языков

Как уже было сказано выше, на сегодняшний день существует достаточно большое количество библиотек для всех основных традиционных языков программирования, реализующих основную функциональность для работы с графами (различные эффективные представления графов, основные алгоритмы, интерфейсы для реализации собственных алгоритмов и композиции уже имеющихся). Однако выразительная сила таких средств не очень велика, так как при создании традиционных языков программирования не предполагалось использование операций над графами в качестве базовых синтаксических единиц.

Предпочтение здесь можно отдать языкам, предполагающим в той или иной мере возможность расширения собственного синтаксиса (либо достаточно гибкие средства для переопределения уже существующих синтаксических конструкций) и обладающим богатыми средствами для написания обобщенных алгоритмов (не привязанных к конкретному представлению графов).

Языки **GTPL** (расширение FORTRAN II), **GEA** (Graph Extended ALGOL, расширение ALGOL 60), **GASP** (Graph Algorithm and Software Package, расширение PL/I), **GRAAL** (GRAph Algorithmic Language, расширение ALGOL 60 и FORTRAN) и **GRASPE** (расширение SNOBOL4, SLIP-FORTRAN и LISP 1.5) [1], появившиеся в конце 60-х годов, были первыми попытками расширить синтаксические возможности традиционных языков программирования для возможности реализации алго-

ритмов на графах. Перечисленные базовые языки не имели средств для расширения собственного синтаксиса, а потому все упомянутые подходы требовали предварительного препроцессирования при компиляции для преобразования текста программы в базовый язык. Появление подобных расширений было обусловлено не стремлением к выразительности и высокому уровню абстракции, а просто отсутствием в языках того времени механизмов для достаточно общего представления графов. Стандартные алгоритмические языки (такие как FORTRAN и ALGOL 60) с их ограниченными возможностями в представлении данных оказались непригодны для реализации алгоритмов на графах, в то время как языки для обработки списков (производные от LISP) предоставляли более удобные структуры данных, однако скрывали теоретико-графовый характер алгоритмов, а также приводили к медленному выполнению программ и большим затратам оперативной памяти.

Пакет **GRAAP** [1] (GRaph Algorithmic Applications Package), выполненный в виде расширения ALGOL 68, появился в 1980 году и использовал возможности базового языка по расширению собственного синтаксиса, не требуя препроцессирования при компиляции. Данная библиотека решала многие проблемы ее предшественниц, предоставляла необходимые общие структуры для представления графа и, ко всему прочему, обладала обширным набором уже реализованных алгоритмов.

Язык **Graphscript** [2] (как часть инструментального средства Graphlet для редактирования графов) является синтаксическим расширением языка Tcl/Tk. Включает в себя базовые обобщённые структуры и средства для реализации алгоритмов. К значительным недостаткам можно отнести интерпретируемость, отсутствие статической типизации (язык является скриптовым) и проверку корректности синтаксиса во время выполнения, а не на стадии компиляции (наследие Tcl/Tk). Кроме того, сказывается узкая направленность языка (причиной его появления было стремление расширить редактор Graphlet некоторым скриптовым языком): наличие в языке таких низкоуровневых характеристик, как координатная привязка вершин, а также отсутствие каких-либо стандартных алгоритмов (хотя есть все возможности для их реализации).

Библиотека **BGL** [3] (The Boost Graph Library) для языка C++ включает фундаментальные обобщённые алгоритмы на графах и богатые средства для реализации собственных обобщённых алгоритмов, не привязанных к конкретному представлению графов. Она не использует син-

таксические расширения, однако возможность перегрузки базовых операторов и богатые возможности шаблонов C++ позволяют включить её в описываемую группу библиотек с некоторой долей условности.

Библиотека **FGL** [4] (A Functional Graph Library) для языков ML и Haskell включает основные алгоритмы и богатые средства для создания новых. Вариант для Haskell использует возможности языка по определению новых операторов и переопределению семантики существующих синтаксических конструкций. Кроме того, в настоящее время ведется работа по расширению синтаксических возможностей Haskell для поддержки «активных образцов» (active patterns), которые позволят значительно увеличить выразительную силу библиотеки.

Библиотека **OcamlGraph** [5] для языка Objective Caml является самой общей библиотекой из перечисленных, не привязанной к конкретному представлению графов. Содержит большое количество стандартных алгоритмов и богатые возможности по созданию новых. Не использует синтаксических расширений.

2.2 PROGRES

Язык PROGRES [6] (PROgrammed Graph REwriting Systems), входящий в состав одноимённого средства для создания приложений, требующих различных видов взаимодействия с графами (например, конструирование графов, анализ, трансформация, использование графоподобных структур), является очень интересной разработкой, широко используемой на практике. Отличительной чертой этого языка является возможность использования в тексте программы графических конструкций, позволяющих наиболее выразительно описывать необходимые действия (это оказывается удобным при конструировании графов и сопоставлении с образцом, см. рис. 1).

PROGRES является строго типизированным и даже почти полностью статически типизированным языком, основанным на концепции программируемых систем переписывания графов. Он предусматривает объектно-ориентированное описание атрибутных графовых структур, поддерживая множественное наследование и параметрический полиморфизм. Также имеется возможность декларативного описания выводимости атрибутов и бинарных отношений, визуального описания правил переписывания графов со сложными условиями применения, а также недетерминированного и императивного описания композитных графо-

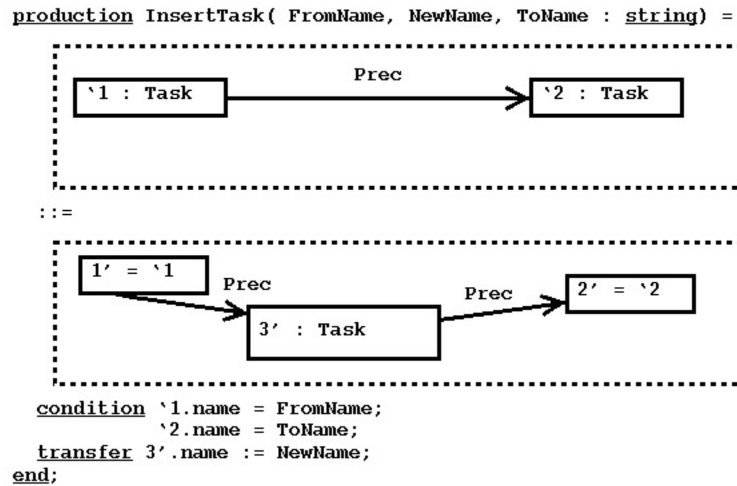

Рис. 1: Пример описания трансформации на языке PROGRES

вых трансформаций (с использованием встроенного механизма перебора с возвратами).

В качестве недостатков языка можно отметить сложность синтаксиса и громоздкость результирующего кода (в частности, из-за использования графических конструкций, которые оказываются удобными для описания простых «плоских» преобразований, однако сильно усложняют описание подграфов, чья структура сложна или не определена статически).

2.3 GXL

Язык GXL [7] является попыткой расширить популярный язык переписывания деревьев TXL для возможности работы с графами, используя общие принципы переписывания графов. От TXL он унаследовал синтаксис, языковые особенности и стиль, адаптировав и расширив их для работы с обобщёнными направленными графами. Среди важных черт TXL, перенесенных в GXL, стоит отметить строгую типизацию, принцип локальности действия правил (возможность ограничения действия трансформации на некоторый подграф или поддерево, а не на весь граф) и параметризацию (возможность работы с несколькими независимыми копиями подграфа или поддерева и передачи их в качестве параметров). Это выгодно отличает его от языка PROGRES, в котором все правила переписывания носят глобальный характер.

Ниже представлен простейший пример описания правила переписывания на языке GXL, которое для всех рёбер, имеющих метку `"calls"`,

добавляет обратное ребро с меткой "called by":

```
rule addCalledByEdges
   replace $ [subgraph]
      P [node] -- calls --> Q [node]
   by
      P -- calls --> <-- called by -- Q
end rule
```

Языку GXL свойственен общий недостаток всех систем переписывания — необходимость заботиться о завершаемости процесса переписывания (процесс может не иметь неподвижной точки и привести к зацикливанию). Кроме того, ощущается недостаточная гибкость языка: описание достаточно простых трансформаций может потребовать значительных усилий для реализации этого в рамках данной семантической модели.

3 Постановка задачи

Проведённое исследование позволяет сделать вывод о том, что на сегодняшний день не существует языка для работы с графами, который бы в полной мере удовлетворял перечисленным выше требованиям. Приведённые в обзоре языки и синтаксические расширения либо предоставляют лишь низкоуровневые средства для осуществления манипуляций с графами, либо имеют нетрадиционную модель вычислений (модель переписывания графов) и тем самым сложны для понимания и создания программ. Это и послужило причиной создания языка, который смог бы решить поставленные задачи.

Д. Ю. Булычевым был создан прототип подобного специализированного языка, получивший название GRAIL и предназначенный для быстрого прототипирования алгоритмов на графах. В рамках данной дипломной работы требовалось создать компилятор для данного языка. Особое внимание должно было уделяться производительности: очевидно, что программы написанные на подобном языке высокого уровня будут уступать в скорости аналогичным программам на традиционных языках программирования, однако этот проигрыш должен быть приемлем. Кроме того, требовалось оценить выразительность данного языка, реализовав на нём некоторые алгоритмы и сопоставив с аналогами на других языках.

4 Описание языка GRAIL

Основные отличительные черты алгоритмов на графах и программ, требующих работы с графовыми структурами, позволяют выделить ряд требований к разрабатываемому языку:

- Многие алгоритмы на графах основаны на переборе с возвратами и в некоторых случаях требуют отката на предыдущий шаг выполнения с возобновлением всей структурной информации. Наиболее удобный способ для реализации такого процесса — это использование неизменяемых данных (immutable data). Поэтому в качестве основы для языка была выбрана функциональная модель вычислений.

- Важным требованием в алгоритмах на графах, требующих значительного объема вычислений, является недетерминизм. В связи с этим была выбрана «ленивая» (lazy) вычислительная стратегия, реализующая последовательное вычисление структур данных для предотвращения массовой генерации значений, которые в конечном счете не будут использоваться в дальнейших вычислениях.

- Граф должен являться базовой сущностью языка, которая может передаваться в качестве параметра, участвовать в основных операциях, а также использоваться при построении более сложных структур данных: метаграфов (графов, метками вершин и рёбер которых являются другие вершины, рёбра и даже графы), графовых множеств (множеств, элементами которых являются графы) и т.д.

- Создание графов и запросы к графам (выделение в конкретном графе вершин и рёбер, удовлетворяющих некоторым условиям) с возможностью гибкого использования результатов этого запроса должны представляться в как можно более лаконичной и простой форме, так как эти конструкции являются наиболее часто используемыми.

С учётом всего вышеперечисленного был создан язык GRAIL — язык программирования, нацеленный на описание алгоритмов обработки графов. В качестве модели вычислений была выбрана чистая функциональная модель языка Haskell [8] с присущим этому языку ленивым характером вычислений.

4.1 Основные языковые конструкции

В данном разделе описываются основы языка GRAIL, чей синтаксис во многом схож с синтаксисом языка Haskell.

Язык имеет особую семантическую модель. Все вершины и рёбра создаются в едином **пространстве** (universe), причём ребро не может быть создано, если пространство не содержит начальной или конечной вершины. Графы по сути являются лишь упорядоченными множествами вершин и рёбер (подмножествами этого пространства). Порядок в графе имеет значение лишь при осуществлении запросов (см. раздел 4.2.4); графы, содержащие одни и те же вершины и рёбра, но в различном порядке, считаются равными. Под «графом» здесь и далее понимается упорядоченный помеченный мультиграф (допускающий несколько рёбер между одними и теми же вершинами; вершины и рёбра могут быть помечены значениями различных типов, в том числе и графами).

Все операции над графами носят локальный характер, т. е. применяются только к конкретным графам, а не к пространству в целом. Исключением являются лишь функции `src`, `dst` и `label`, которые могут применяться к вершине или ребру вне зависимости от того, в рамках какого графа они рассматриваются.

Программа на GRAIL состоит из набора **определений функций**. Каждое определение объявляет имя функции, её аргументы и тело. Определения могут быть взаимно рекурсивными.

Помимо определений, в качестве основных синтаксических категорий стоит выделить **выражения** и **образцы** (patterns). Выражения определяют способ вычисления значений, в то время как образцы используются для сопоставления вычисленных значений какому-либо шаблону и для выбора пути вычислений на основе этого сопоставления.

4.1.1 Значения и образцы

Значения — это объекты структур данных; **образцы** отражают их некоторые структурные свойства и используются для их сопоставления. Значения могут быть нескольких видов:

- целые числа;
- строки;
- кортежи;
- составные значения, полученные применением пользовательского конструктора к некоторым аргументам;

- пустое значение (unit).

Например, `1024` — целочисленное значение, `"Monday"` — строка, `()` — пустое значение (unit), `(12, "Jan", 2007)` — кортеж, `City ("St.Petersburg", 1703)` — значение, полученное применением пользовательского конструктора `City` к аргументам `"St.Petersburg"` и `1703`.

Использование образцов двояко: с одной стороны, исследуется структура значения, а с другой — происходит связывание некоторых частей составного значения с определёнными идентификаторами. В основном, синтаксис образцов соответствует синтаксису значений, которым они сопоставляются. Например, `"Monday"` — строковый образец, (p_1, ..., p_k) (где p_i также являются образцами) — образец-кортеж, `Day` p (где p является образцом) — образец-конструктор и т.д. Идентификаторы и символ подчеркивания тоже являются образцами. Каждый образец может быть снабжён **ограничением** (constraint), отделенным ключевым словом `when`. Любой образец (как самостоятельный, так и входящий в состав другого) может быть связан с некоторым идентификатором, который отделяется ключевым словом `as`. Ниже приведены некоторые примеры образцов:

```
 x
(x, y) as t
(x, (y, z) as t) when x<y+z
(C x, (a, b) when a+b>0)
(C x, C y, C z)
```

Сопоставление значения v с образцом p помимо результата сопоставления (успешно/неуспешно) порождает также множество **связываний** (bindings) согласно следующим правилам:

- если p является символом подчеркивания (`_`), тогда сопоставление считается успешным и никаких связываний не происходит;

- если p — некоторый идентификатор *name*, тогда сопоставление считается успешным и идентификатор *name* связывается с сопоставляемым значением v;

- если p — это (p_1, p_2, ..., p_k), а v — кортеж с компонентами v_1, v_2, ..., v_k, и все значения v_i успешно сопоставляются с образцами p_i с множествами связываний b_i, тогда сопоставление v с p считается успешным, а множество связываний является объединением всех b_i;

- если p — это C `(`p_1`,` p_2`, ... ,` p_k`)`, а v — применение конструктора C `(`v_1`,` v_2`, ... ,` v_k`)`, и все значения v_i успешно сопоставляются с образцами p_i с множествами связываний b_i, тогда сопоставление v с p считается успешным, а множество связываний является объединением всех b_i;

- если p — это `()`, а v — пустое значение `()`, тогда сопоставление считается успешным и никаких связываний не происходит;

- если p имеет вид p_1 `as` *name* и значение v успешно сопоставляется с p_1 с множеством связываний b, тогда сопоставление v с p также считается успешным, а множество связываний состоит из всех элементов b, а также идентификатора *name*, связанного с v;

- если p имеет вид p_1 `when` *expr* и значение v успешно сопоставляется с p_1 с множеством связываний b, а *expr* имеет истинное значение, тогда сопоставление v с p также считается успешным с множеством связываний b;

- если p является целым числом или строкой, а v равняется этому целому числу или строке соответственно, тогда сопоставление считается успешным и никаких связываний не происходит;

- во всех остальных случаях сопоставление считается неуспешным.

Например, сопоставление `()` с `k` успешно и `k` оказывается связанным с `()`, сопоставление `City (Moscow, Capital)` с `City (name, Capital)` успешно и `name` оказывается связанным с `Moscow`, а сопоставление `((), ())` с `()` — неуспешно.

4.1.2 Абстракции и применения

Абстракция позволяет описывать и переиспользовать некий процесс вычислений. Другими словами, абстракция позволяет создавать функцию из выражения (выбирая в качестве аргументов функции свободные переменные в выражении). Например, лямбда-выражение

```
\ x y -> x + y
```

является абстракцией сложения, или просто функцией, складывающей два своих аргумента.

Другой конструкцией абстракции является **определение**. Например,

```
addTwo x y = x + y
```

аналогично первому примеру с тем лишь различием, что в данном случае абстрагируемое значение связывается с идентификатором `addTwo`, в то время как в предыдущем примере оно остаётся анонимным.

Синтаксически абстракция имеет следующую форму:

\\p_1 ... p_k -> $expr$

где p_i являются образцами, а $expr$ — выражением. Определение имеет аналогичную форму:

$ident$ p_1 ... p_k = $expr$

Семантически противоположной к абстракции конструкцией является **применение**, синтаксически выражаемое простым приписыванием значений аргументов к имени функции. Например, `addTwo 1 2` соответствует применению функции `addTwo` к аргументам `1` и `2`. Применение рассматривается в **каррированной форме** (curryfied form), т.е., к примеру, `addTwo 1` является функцией, получающей один аргумент и возвращающей значение на единицу больше.

4.1.3 Множества, включения множеств и задание порядка

Множества (sets) — это наборы каких-либо значений, не содержащие повторяющихся элементов. Множества могут быть заданы перечислением своих элементов с помощью следующей конструкции:

{v_1, v_2, ..., v_k}

где v_i — некоторые выражения, являющиеся элементами множества. Кроме того, множества могут быть получены из уже имеющихся с помощью операций `+` (объединение), `*` (пересечение) и `-` (разность), соответствующих стандартным операциям теории множеств, а также с помощью конструкции **включения множеств** (set comprehension; аналог конструкции list comprehension языка Haskell, перенесенный на множества). Включение множеств позволяет осуществлять проходы по множествам и конструировать новые множества из уже имеющихся, и синтаксически оформляется следующим образом:

{ e | $name_1$<-s_1, $name_2$<-s_2, ..., $name_k$<-s_k }

Здесь создается множество, элементы которого описываются выражением e, зависящим от $name_1$, $name_2$, ..., $name_k$, которые в свою очередь пробегают по всем элементам множеств s_1, s_2, ..., s_k соответственно (s_i могут быть любыми выражениями, значениями которых являются мно-

жества). Конструкция *name*`<-`*expr* называется **генератором**. Все связывания, порождаемые внешними генераторами, могут быть использованы в правых частях внутренних генераторов. Кроме того, каждый генератор может быть снабжён дополнительным ограничением. Для примера ниже приведены выражения, создающие множества, и их значения:

`{1,2,3,4,5}`	$=$	$\{1,2,3,4,5\}$
`{(x,y) | y<-{1,2}, x<-{3,4}}`	$=$	$\{(3,1),(3,2),(4,1),(4,2)\}$
`{(y,x) | y<-{1,4} when y>x, x<-{2,3}}`	$=$	$\{(4,2),(4,3)\}$

Множества — неупорядоченные наборы: элементы хранятся и выбираются в том порядке, в котором они были добавлены. Однако иногда возникает потребность в изменении этого порядка. Этой цели служит конструкция

[*set* | *ord*]

которая переупорядочивает элементы множества, являющегося значением выражения *set*, с помощью функции порядка *ord*. Функция порядка должна быть булевой функцией двух аргументов, удовлетворяющей аксиомам отношения строгого полного порядка (*ord* возвращает `True`, если первый аргумент «меньше» второго, и `False` в противном случае):

1. $\forall x \forall y \forall z(ord\ x\ y$ `&&` $ord\ y\ z$ `==` $ord\ x\ z)$;

2. $\forall x \forall y(ord\ x\ y$ `&&` $ord\ y\ x$ `== False`).

3. $\forall x \forall y(ord\ x\ y$ `||` $ord\ y\ x$ `== True`).

Например, выражение

```
[{3,4,2,1,5} | \ x y -> x < y]
```

будет иметь значение $\{1,2,3,4,5\}$.

Стандартные операции теории множеств приобретают следующую семантику:

- Объединением двух упорядоченных множеств является множество, полученное в результате добавления к первому множеству (в котором порядок элементов остаётся тем же) всех элементов из второго, которые не содержатся в первом (с сохранением порядка следования).

- Пересечением двух упорядоченных множеств является множество, полученное в результате удаления из первого множества всех элементов, которые не содержатся во втором (порядок оставшихся элементов сохраняется таким же, как и в первом множестве).

- Разностью двух упорядоченных множеств является множество, полученное в результате удаления из первого множества всех элементов, содержащихся во втором (с сохранением порядка).

4.1.4 Управляющие конструкции

Управляющие конструкции используются для условного разветвления процесса вычислений. В языке GRAIL имеется две такие конструкции: `if` и `match`.

Выражение

`if` *cond* `then` *true-value* `else` *false-value* `fi`

равно значению *true-value*, если значение выражения *cond* равняется `True`, и *false-value* в противном случае.

Конструкция `match` используется для разветвления процесса вычисления в зависимости от результата сопоставления с образцом. Выражение

```
match expr with
    p1 -> expr1
    ...
    pk -> exprk
end
```

равно значению $expr_i$, если *expr* успешно сопоставляется с p_i, а все сопоставления с p_j для $j < i$ оказались неуспешными. В выражении $expr_i$ возможно использование связываний, порождённых образцом p_i. В случае, когда ни одно из сопоставлений не прошло успешно, выполнение всей программы прерывается.

4.1.5 Операции

Набор стандартных унарных и бинарных операций унаследован от языка Haskell, и все они синтаксически и семантически полностью соответствуют своим прообразам в Haskell.

4.1.6 Локальное связывание

Локальное связывание (сопоставление некоторого выражения какому-либо идентификатору в рамках текущего контекста) может быть выполнено с помощью конструкций `let` и `where`.

Конструкция `let` играет роль выражения и может иметь одну из следующих синтаксических форм:

`let` $p_1, \ldots, p_k$ `=` $expr_1, \ldots, expr_k$ `in` $expr$

или

`let` $ident\ p_1\ \ldots\ p_k$ `=` def `in` $expr$

`let`-связывание осуществляет сопоставление выражений $expr_i$ с образцами p_i и вычисляет выражение $expr$ с учетом всех связываний, порождённых этими сопоставлениями. Вторая форма записи является сокращением выражения

`let` $ident$ `= \`$p_1\ \ldots\ p_k$ `->` def `in` $expr$

Конструкция `where` является постфиксным аналогом `let`-конструкции и может использоваться в выражениях, участвующих в объявлении функции на верхнем уровне. Синтаксис этой конструкции имеет следующий вид:

`where` $def_1; \ldots; def_k;$

Все объявления `where`-конструкции доступны внутри выражения, к которому оно примыкает. Например:

```
sumsqr x y = sqr x + sqr y where
   sqr x = x * x;
;
```

4.1.7 Другие конструкции

В языке GRAIL существует возможность вставлять код на языке Haskell на месте объявления функций верхнего уровня, а также в любом месте, где предполагается наличие выражения (если не возникает семантического и типового противоречия). Оформляется это в виде текста, заключённого в обратные апострофы (‘. . . ‘) и предназначается для осуществления действий, не поддерживаемых напрямую в GRAIL (например, ввод-вывод).

4.2 Конструкции, специфичные для графов

Граф в языке GRAIL является базовой сущностью, с которой можно работать так же, как и с любыми другими значениями: передавать в качестве параметра функции, создавать множества графов, графы графов и т.д.

В GRAIL существует 4 вида конструкций для манипуляций с графами:

- конструкторы графов;
- let-связывание для графов;
- базовые операции над графами;
- графовые запросы.

Все эти конструкции фактически являются специальными видами выражений.

4.2.1 Конструкторы графов

Конструктор графа является единственным способом построения графа «с нуля». Существует две формы конструкторов: **простой конструктор графа** и **включение графов** (graph comprehension, аналог set comprehension).

Простой конструктор записывается в следующем виде:

`#`$item_1$`, ..., `$item_k$`#`

где $item_i$ определяет вершину или ребро создаваемого графа.

Вершина определяется с помощью выражения, заключенного в круглые скобки и соответствующего метке узла. Вершина может быть снабжена некоторым идентификатором, связывающим создаваемую вершину с этим именем. Например, `(2)` определяет вершину с меткой `2` без каких-либо связываний, а `n:((17, "August", 1970))` определяет вершину `n`, помеченную кортежем `(17, "August", 1970)`.

Определение ребра в общем случае имеет следующий вид:

src_node`-`*edge*`->`*dst_node*

где *src_node* и *dst_node* являются объявлениями начальной (source) и конечной (destination) вершин, *edge* — объявлением метки ребра (аналогично объявлению метки вершины) с необязательным указанием имени создаваемого ребра. Например,

```
n:(3)-e:("edge")->k:(4)
```

объявляет ребро `e` с меткой `"edge"`, идущее из вершины `n` с меткой `3` в вершину `k` с меткой `4`. Конструктор, содержащий подобное объявление ребра, создаст ребро `e` вместе с инцидентными вершинами `n` и `k`.

Существует другая форма объявления вершины или ребра — с помощью использования идентификаторов, связанных с уже созданными вершинами и рёбрами. Для этого нужно использовать символ `!` перед именем идентификатора. В таком случае никакого ребра или вершины не создаётся. Например,

```
!x-("label")->!y
```

соединяет *существующие* вершины `x` и `y` *новым* ребром с меткой `"label"`.

Несколько других примеров:

- `#n:("node")-("loop")->!n#` создаёт циклический граф из одной вершины, помеченной строкой `"node"`, и ребра, помеченного строкой `"loop"`;
- `#x:(1)-(1)->y:(2), !y-(2)->z:(3), !z-(3)->!x#` создаёт треугольный граф с вершинами и рёбрами, занумерованными числами `1`, `2` и `3`;
- `#-!e->, -!f->#` задаёт граф, состоящий из двух рёбер другого графа (идентификаторы `e` и `f` должны быть связаны с некоторыми рёбрами). Отметим, что получаемый граф не содержит вершин (оба ребра являются висячими).

Включение графов синтаксически состоит из простого конструктора, снабжённого последовательностью генераторов. Синтаксис и семантика генераторов такова же, как и для включения множеств. Конструируемый граф является объединением всех графов, получаемых в результате выполнения конструктора графа для всех комбинаций пробегаемых значений. Например,

```
#(n) | n<-{1,2,3,4}#
```

соответствует графу из четырёх вершин, помеченных числами `1`, `2`, `3` и `4`.

4.2.2 Let-связывание для графов

Иногда в вычислениях требуется использовать конкретные рёбра или вершины графа. Для осуществления выборки таких сущностей из имеющегося графа в язык введены конструкции **запросов** (queries). Однако часто возникает потребность осуществлять выборку из только что созданного графа. Для этой цели в язык введена особая форма конструкции `let`, которая позволяет использовать во внешнем контексте внутренние связывания конструктора графа. Данная форма конструкции `let` имеет следующий синтаксис:

`let` *simple_graph_constructor* [`as` *ident*] `in` *expr*

Здесь все связывания, порождённые конструктором графа, могут быть использованы в выражении *expr*. Кроме того, весь сконструированный граф может быть связан с некоторым идентификатором *ident*, который также может использоваться в *expr*.

4.2.3 Операции над графами

Имеется три встроенные операции над графами: `or`, `and` и `but`, соответствующие объединению, пересечению и разности графов в теоретико-графовом смысле. Порядок рёбер и вершин в результирующем графе определяется описанными выше правилами выполнения аналогичных операций на упорядоченных множествах.

4.2.4 Графовые образцы и запросы

Запросы (graph queries) предназначены для извлечения информации о внутренней структуре графов. Существует 4 типа запросов:

- all-запрос — выборка всех сущностей, удовлетворяющих некоторым условиям, и сопоставление идентификаторам, участвующим в запросе, *множеств* сущностей, входящих в результат выборки;
- some-запрос — нахождение *хотя бы одной конфигурации* сущностей, удовлетворяющих некоторому условию;
- set-запрос — нахождение *всего множества конфигураций*, удовлетворяющих условию запроса;
- subgraph-запрос — выделение *подграфа*, являющегося объединением всех конфигураций, удовлетворяющих условиям запроса.

Несмотря на то, что все эти виды запросов различаются семантически, синтаксически они имеют много общего и все опираются на понятие **графовых образцов** (graph patterns). Подобно обычным образцам, синтаксическая форма которых определяется записью соответствующих значений, графовые образцы имеют много схожего с конструкторами графов. Неформальная семантика графовых образцов достаточно очевидна: **вершинный образец** (node pattern) выбирает все вершины, чьи метки удовлетворяют образцу; **рёберный образец** (edge pattern) выбирает рёбра, у которых метка, начальная и конечная вершина сопоставляются с соответствующими образцами. Точная семантика образцов зависит от типа запроса, в котором они используются.

Например:

- вершинному образцу `n:Start` сопоставляются все вершины, имеющие метку `Start`, и идентификатор `n` связывается с такими вершинами;
- рёберному образцу `_-e->!n` (или, в более короткой записи, `-e->!n`) сопоставляются все рёбра с конечной вершиной `n` (идентификатор `n` должен быть известен в текущем контексте), и идентификатор `e` связывается с такими рёбрами;
- рёберному образцу `n-!e->k` сопоставляется конкретное ребро `e` (которое должно быть известно в текущем контексте), связывая начальную и конечную вершины с идентификаторами `n` и `k` соответственно.

All-запрос имеет следующую синтаксическую форму:

$graph$`{` p_1 `, ... ,` p_k `} :` $expr$

где $graph$ — выражение, являющееся графом, p_i — графовые образцы. Семантика этого запроса такова:

- граф $graph$ сопоставляется с каждым образцом p_i в порядке их следования;
- во время сопоставления с образцом p_i доступны все связывания, возникающие в результате сопоставления с образцами p_j при $j < i$;
- каждое сопоставление с образцом выбирает *все* сущности (вершины и рёбра) графа, удовлетворяющие образцу;

- каждое связывание, порожденное каждым образцом запроса, доступно в выражении *expr* в качестве *множества* рёбер и вершин.

Примеры:

- `g{n}:n` — выбрать все вершины из графа `g`;
- `g{-e->}:e` — выбрать все рёбра из графа `g`;
- `g{!n-e->}:e` — выбрать все рёбра, выходящие из вершины `n`;
- `g{!n-e->!k}:e` — выбрать все рёбра, ведущие из вершины `n` в вершину `k`;
- `g{!n-_->k}:k` или просто `g{!n-->k}:k` — выбрать все вершины, достижимые из `n` «за один шаг».

Some-запрос записывается следующим образом:

graph`<`p_1`, ... , `p_k`> ? `$expr_1$` : `$expr_2$

где *graph* — выражение, являющееся графом, p_i — графовые образцы. Семантика сопоставлений с образцами такова же, как и в all-запросе, но семантика связываний в образцах отличается. Произвольным образом выбирается набор связываний, порождаемый *каким-либо* вариантом сопоставления с образцами p_i, и этот набор связываний оказывается доступен в выражении $expr_1$, которое и является результатом всего запроса. В случае, когда ни одного такого набора не существует, возвращается значение выражения $expr_2$.

Примеры:

- `g<n> ? n : ‘error "..."‘` — выбрать одну произвольную вершину из графа `g` (или выдать ошибку, если граф не содержит вершин);
- `g<-e->> ? e : ‘error "..."‘` — выбрать одно произвольное ребро из графа `g` (аналогично предыдущему примеру);
- `g<!n-e->k> ? f e k : ‘error "..."‘` — выбрать одну (произвольную) конфигурацию, состоящую из ребра `e`, выходящего из вершины `n`, и вершины `k`, в которую это ребро входит, и вызвать функцию `f` от этого ребра и вершины; выдать ошибку, если подобная конфигурация не может быть найдена в графе `g`.

Set-запрос действует так же, как some-запрос, но возвращает *все* конфигурации, удовлетворяющие всем образцам. Данный запрос имеет следующую синтаксическую форму:

graph{< p_1, ..., p_k > : *expr*}

Здесь *graph* — выражение, являющееся графом, p_i — графовые образцы, *expr* — выражение, вычисляющееся каждый раз, когда все сопоставления проходят успешно. Результирующее множество состоит из всех таких *expr*. Например,

```
g{<n-o->, -i->!n> : (i,n,o)}
```

возвращает множество всех кортежей `(i,n,o)`, в которых `n` — вершина, а `i` и `o` — её входящее и выходящее рёбра соответственно.

Subgraph-запрос имеет форму

graph[p_1, ..., p_k]

и возвращает граф, содержащий в себе все конфигурации вершин и рёбер, успешно сопоставляющиеся образцам p_i запроса. Например, запрос

```
g[x-->y, !y-->z, !z-->!x]
```

возвращает подграф, состоящий из всех треугольников, содержащихся в графе `g`.

Для демонстрации различий в семантике описанных четырёх видов запросов приведём еще один пример. Пусть `g` — это граф, изображённый на рис. 2 слева. Тогда:

- `g{-e1->n:"X", !n-e2->} : (e1,n,e2)` возвращает кортеж из трёх множеств: `({"e","c","d"}, {"X","X"}, {"b","a"})`[1];
- `g<-e1->n:"X", !n-e2->> ? (e1,n,e2) : ‘error "..."‘` возвращает кортеж `("c","X","a")`.
- `g{<-e1->n:"X", !n-e2->> : (e1,n,e2)}` возвращает множество кортежей: `{("c","X","a"), ("d","X","a"), ("e","X","b")}`
- `g[-e1->n:"X", !n-e2->]` возвращает подграф, изображённый на рис. 2 справа.

[1]Элементами множества являются вершины и рёбра, которые для наглядности здесь представлены своими метками.

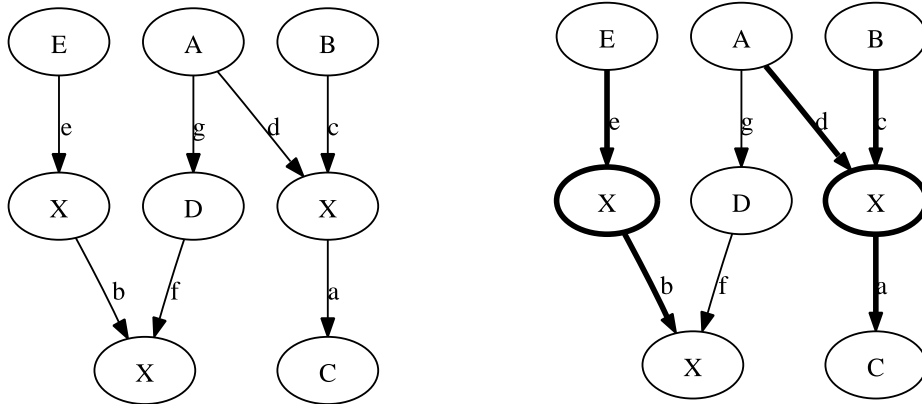

Рис. 2: Входной граф и результат subgraph-запроса

4.3 Примеры алгоритмов на языке GRAIL

В данном разделе приводится несколько примеров программ, демонстрирующих основные возможности языка GRAIL. Примеры сопровождаются описанием решаемой задачи (с определениями всех необходимых понятий, требуемых для её формулировки) и разъяснением всех выполняемых в программе действий.

4.3.1 Простейшие примеры

Для начала приведём простейшие примеры, иллюстрирующие использование языка GRAIL.

Функция, которая для каждого ребра графа создаёт новое ребро с теми же входной и выходной вершинами и меткой `"new"`:

```
dup g = g or #!(src e)-("new")->!(dst e) | e<-g{-e->}:e#;
```

Функция, «корректно» удаляющая все вершины с меткой `"X"` (всякая такая вершина `o`, лежащая между какими-либо вершинами `x` и `y`, удаляется вместе с рёбрами, ведущими из `x` в `o` и из `o` в `y`, но с добавлением нового ребра из `x` в `y`):

```
buildClosure g =
  let set = g{<x-e->o:"X",!o-f->y> : (x,y,o,e,f)} in
    (g or #!x-()->!y | (x,y,_,_,_)<-set#) but
    #-!e->, -!f->, !o | (_,_,o,e,f)<-set#;
```

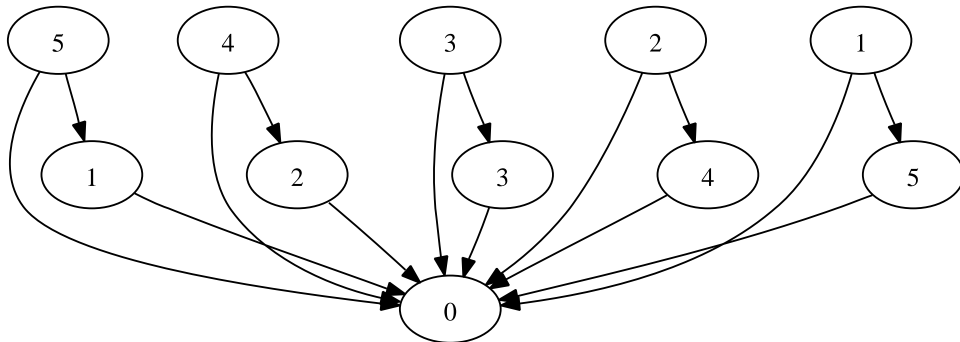

Рис. 3: Граф, построенный функцией constr

«Пополнение» графа (соединение попарно всех вершин, между которыми нет ребра):

```
full g =
  g or #!j-()->!k | j<-g{n}:n,
                    k<-g{n}:n when (g{!j-e->!k}:e)=={}#;
```

Построение графа, изображенного на рис. 3:

```
constr _ =
  let #o:(0)# as g in
    g or #x:(k)-()->y:(n), !x-()->!o, !y-()->!o |
         k<-{1, 2, 3, 4, 5}, n<-{1, 2, 3, 4, 5} when (k+n)==6#;
```

Нахождение всех вершин в графе, из которых достижима данная:

```
anc g n = inner {n} {n} where
  inner s acc =
    let diff _ = {m | m<-g{m-->!k}:m, k<-s} - acc in
      if diff () == {} then
        acc
      else
        inner (diff ()) (acc + (diff ()))
      fi
  ;
;
```

4.3.2 Нумерации вершин

Перед формулировкой задачи дадим необходимые определения [9].

Определение 1 *Пусть $G = (V, E)$ — граф (V — множество вершин, E — множество рёбер), $|V| = n$.* ***Нумерацией*** *(ordering) вершин графа G называется биекция $F : V \to [1 : n]$.*

Определение 2 *Пусть $G = (V, E)$ — граф, v — некоторая его вершина.* ***M-нумерацией*** *(preorder, прямой нумерацией) называется нумерация вершин графа G в порядке их обхода при поиске в глубину со стартовой вершиной v (соответствует порядку включения вершин в путь).*

Определение 3 *Пусть $G = (V, E)$ — граф, v — некоторая его вершина.* ***N-нумерация*** *(postorder, обратная нумерация) соответствует порядку, обратному порядку исключения вершин графа G из пути при обходе в глубину со стартовой вершиной v.*

Определение 4 *Пусть $G = (V, E)$ — граф ($|V| = n$), F — некоторая его нумерация.* ***F-областью*** *(F-region) называется множество вершин, из которых достижима вершина p в подграфе, порождённом вершинами с номерами $[i : n]$, и обозначается $F\langle i\rangle$.*

Определение 5 ***K-нумерация*** *графа $G = (V, E)$ ($|V| = n$) со стартовой вершиной v определяется как последний член K_n последовательности нумераций K_1, K_2, ..., K_n, в которой K_1 — обратная нумерация для графа G и стартовой вершины v, и для любых $i \in [1 : n)$ и $p, q \in V$ справедливы следующие свойства:*

- *если $K_i(p) < i$, то $K_{i+1}(p) = K_i(p)$;*
- *в противном случае:*
  - *если вершины p и q обе либо принадлежат, либо не принадлежат $K_i\langle i\rangle$, то $K_{i+1}(p) < K_{i+1}(q) \Leftrightarrow K_i(p) < K_i(q)$;*
  - *в противном случае $K_{i+1}(p) < K_{i+1}(q)$.*

Описанные нумерации имеют большую сферу применения в задачах анализа потока управления. На рис. 4 приведён пример построения нумераций для некоторого графа.

Задача *Написать программу для построения M-, N- и K-нумераций.*

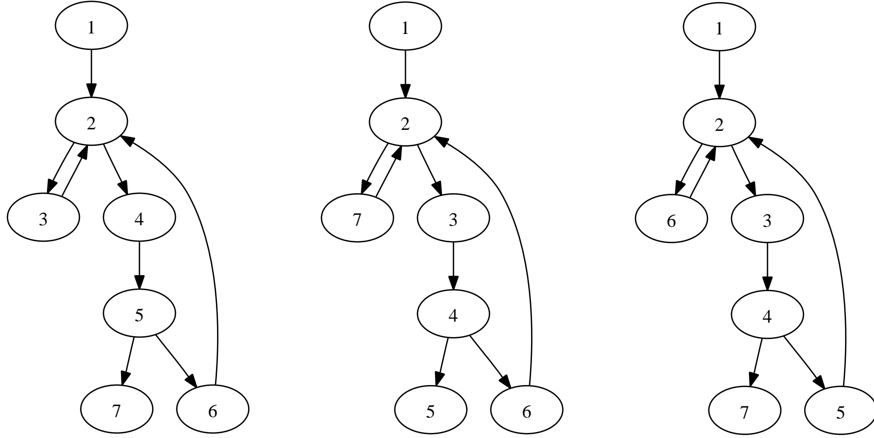

Рис. 4: M-, N- и K-нумерация

Программа:

```
m_enum g start = (inner start ((g{n}:n)-{start})) or g[-e->] where
 inner n ns = #!n# or fromSet {inner k (ns-{k}) | k<-ns*(g{!n-->k}:k)};
;

n_enum g start = rev (inner start ((g{n}:n)-{start})) or g[-e->] where
 inner n ns = fromSet {inner k (ns-{k}) | k<-ns*(g{!n-->k}:k)} or #!n#;
;

k_enum g start = (inner ## ((n_enum g start)[n])) or g[-e->] where
  inner newg oldg =
    if oldg == ## then
      newg
    else
      let (n,_) = choose (oldg{k}:k) in
      let region _ = oldg and #!k | k<-anc g n# in
        inner (newg or #!n#) (((region ()) or oldg) but #!n#)
    fi;
;
```

В приведённой программе нумерации задаются не отображениями, а порядком вершин в графе.

Алгоритмы построения M- и N-нумераций достаточно просты: осуществляется обход графа в глубину, и в результат добавляются обрабатываемые вершины в той последовательности, которая задаётся в определении этих нумераций. Остаётся лишь пояснить, что стандартная функция `fromSet`, получая множество графов, возвращает граф, явля-

ющийся их объединением, а `rev` меняет порядок вершин и рёбер в графе на противоположный.

Алгоритм построения K-нумерации состоит в следующем. Строится N-нумерация для графа и осуществляется проход по всем вершинам согласно устанавливаемому ей порядку. На каждой итерации граф перенумеровывается особым образом (в зависимости от вершины, обрабатываемой на данной итерации, и нумерации графа, полученной на предыдущем шаге алгоритма). Нумерация, полученная после обработки последней вершины, является искомой. Опишем конкретнее действия, производимые на каждом шаге. Пусть обрабатывается i-ая вершина (т.е. выполняется i-ая итерация алгоритма). Для данной вершины строится F-область, и вершины во всём графе перенумеровываются следующим образом: сначала идут все вершины, номера которых меньше i, затем все вершины из найденной F-области (в том же порядке), а затем все остальные (в том же порядке).

Переходя к программе на языке GRAIL для построения K-нумерации, отметим следующий факт, используемый в реализации: на i-ом шаге алгоритма первые $i - 1$ вершин уже имеют нужный порядок и будут входить в целевую нумерацию именно в таком виде. Поэтому внутренняя функция `inner` принимает два аргумента: граф `newg`, состоящий из первых $i-1$ вершин, которые будут входить в результат в таком же порядке, и граф `oldg`, содержащий оставшиеся вершины в порядке, имеющемся на текущей итерации. На каждом шаге строится F-область `region`, зависящая только от графа `oldg`, и осуществляется переход на следующий шаг алгоритма с добавлением в `newg` текущей вершины и переупорядочиванием графа `oldg` согласно правилам, описанным выше. После обработки n-ой вершины граф `oldg` оказывается пуст, а `newg` содержит необходимый порядок вершин.

5 Описание реализации

Данный раздел содержит описание особенностей реализации компилятора для языка GRAIL. Приводятся идеи по осуществлению проекций основных синтаксических конструкций и операций в язык Haskell, а также обосновывается выбор внутреннего представления для графов.

5.1 Система типов

GRAIL — нетипизированный язык, поэтому все выражения имеют значения одного вариантного типа `L`:

```
data L = C String L -- значение алгебраического типа,
                    -- порождённое применением контруктора
       | I Integer  -- целочисленное значение
       | S String   -- строка
       | B Bool     -- булево значение
       | M (Set L)  -- множество
       | L [L]      -- упорядоченное множество
       | T [L]      -- кортеж
       | U          -- пустое значение (unit)
       | V S        -- граф
       | N N        -- вершина
       | E E        -- ребро
```

На этот тип естественным образом индуцируются все стандартные операции (`+`, `-`, `*`, `div`, `mod`, `abs`, `show`, `&&`, `||`, `not`) с динамической проверкой совместимости типов (в случае, если операция не применима к некоторым аргументам, возбуждается исключение). Особо стоит отметить операции, связанные с отношением порядка (`<`, `<=`, `>`, `>=`, `==`, `/=`), так как для их реализации пришлось немного отступить от общего подхода в реализации подобных стандартных операций. В случае, когда аргументы не могут сравниваться, возвращается `False` вместо того, чтобы возбуждать исключение. Необходимо это, например, в запросах вида `g{n:l when l<=4}`, который должен выбрать лишь те вершины, метки которых имеют целочисленное значение и не превосходят 4, а не возбуждать исключение при обнаружении первой вершины с меткой другого типа. Подобное переопределение семантики операций корректно и позволяет, в частности, создавать множества из элементов типа `L`.

5.2 Проекции конструкций языка GRAIL в язык Haskell

Как уже объяснялось выше, во время работы программы в GRAIL существует некоторое глобальное состояние (universe), которое может меняться в ходе выполнения любой функции. Поэтому все объявления функций языка GRAIL проецируются в аналогичные конструкции языка Haskell с тем лишь отличием, что в сигнатуру функции добавляется дополнительный параметр `u`, соответствующий глобальному состоянию на момент начала выполнения функции, а в качестве результата возвращается пара `(x,u)` из истинного значения `x` и изменённого глобального состояния. Например, функция

```
add x y = x + y;
```

спроецируется в Haskell в следующем виде:

```
add x y u = (x + y, u);
```

Соответственно, и любое выражение GRAIL в действительности будет представлять из себя пару из значения выражения и глобального состояния. Поэтому необходимо вручную осуществлять корректную передачу этого глобального состояния от одного выражения к другому. Например, выражение

```
add (add a b) (add c d)
```

будет иметь примерно следующую проекцию:

```
let (t1, u1) = add a b u in
let (t2, u2) = add c d u1 in
  add t1 t2 u2
```

Прежде чем описывать проекции синтаксических конструкций языка GRAIL в Haskell, выделим ряд низкоуровневых функций для работы с графами, необходимых для осуществления таких проекций. Отметим особо, что эти функции должны предоставлять полноценный интерфейс для взаимодействия с графами, чтобы позволять генерировать код программ на языке Haskell, не привязываясь к конкретному представлению графов. Различные варианты таких представлений будут описаны в следующем разделе. Ниже приведён список функций, которых оказалось достаточно для осуществления всех проекций:

- `createView u` создаёт новый пустой граф;
- `createNode u l g` создаёт новую вершину с меткой `l` и добавляет её в граф `g`;
- `createEdge u l g s d` создаёт новое ребро с меткой `l`, начальной вершиной `s` и конечной вершиной `d`, и добавляет его в граф `g`;
- `attachNode (g,u) n` добавляет существующую вершину `n` в граф `g`;
- `attachEdge (g,u) e` добавляет существующее ребро `e` в граф `g`;
- `filterSrc u g es ns` получает список рёбер `es` и вершин `ns`, и возвращает список пар `(n,e)` таких, что `n` является начальной вершиной `e` (`n` и `e` содержатся в `ns` и `es` соответственно);
- `filterDst u g es ns` возвращает список пар `(n,e)` таких, что `n` является конечной вершиной `e` (аналогично предыдущей функции);
- `filterSrcDst u g es ss ds` получает список рёбер `es`, начальных вершин `ss` и конечных вершин `ds`, и возвращает список троек `(s,d,e)` таких, что `s` является начальной вершиной `e`, а `d` — конечной (`s`, `d` и `e` содержатся в соответствующих входных списках);
- `findNodesByLabel u g p` возвращает все вершины графа `g`, метки которых удовлетворяют предикату `p`;
- `findEdgesByLabel u g p` возвращает все рёбра графа `g`, метки которых удовлетворяют предикату `p`;
- функции для осуществления базовых операций над графами (`or`, `and`, `but`).

5.2.1 Простой конструктор графа

Проекция простого конструктора достаточно тривиальна. Она осуществляется путём последовательного выполнения функций `createView`, `createNode` и `createEdge` с осуществлением необходимых связываний там, где это необходимо. Например, конструктор

```
#x:("A")-()->y:("B"), !y-()->z:("C"), !x-()->!z#
```

проецируется в примерно следующее выражение на языке Haskell:

```
let (v1, u1) = createView u in
let (y, v2, u2) = createNode u1 (S "B") v1 in
let (x, v3, u3) = createNode u2 (S "A") v2 in
let (_, v4, u4) = createEdge u3 U v3 x y in
let (z, v5, u5) = createNode u4 (S "C") v4 in
let (_, v6, u6) = createEdge u5 U v5 y z in
let (_, v7, u7) = createEdge u6 U v6 x z in
  (v7, u7)
```

5.2.2 Включение графов

Конструкция включения графов проецируется в выражение, строящееся аналогично случаю с простым конструктором, которое заключается во вложенные вызовы функции `fold` (каждому генератору соответствует один такой вызов) с порождением всех необходимых связываний. Например, выражение

```
#(x)-()->(y) | x<-{1, 2, 3}, y<-{1,2} when x>=y#
```

имеет следующую проекцию:

```
let (v1,u1) = createView u in
let (u2,v2) =
  fold
    (\ x (u,v) ->
       fold
         (\ y (u,v) ->
            if x>=y then
              let (dst,v1,u1) = createNode u y v in
              let (src,v2,u2) = createNode u1 x v1 in
              let (_,v3,u3) = createEdge u2 U v2 src dst in
                (u3,v3)
            else
              (u,v))
         (u,v)
         (M (Set.fromList [I 1, I 2])))
    (u1,v1)
    (M (Set.fromList [I 1, I 2, I 3])) in
(v2,u2)
```

5.2.3 Графовые запросы

Прежде чем приступать к описанию проекций для всех видов запросов, рассмотрим, каким образом генерируются графовые образцы, находящиеся внутри них. Пусть `g` — граф, к которому осуществляется запрос. Суть проекций понятна из примеров:

- `n` переходит в `findNodesByLabel u g (_->B True)`;
- `n:"A"` переходит в

  ```
  findNodesByLabel u g (\ arg -> case arg of {
                          S "A" -> B True;
                          _     -> B False})
  ```

- `-e->` переходит в `findEdgesByLabel u g (_->B True)`;
- `-e:"A"->` переходит в

  ```
  findEdgesByLabel u g (\ arg -> case arg of {
                          S "A" -> B True;
                          _     -> B False})
  ```

- `-e:l when l<=4->` переходит в

  ```
  findEdgesByLabel u g (\ arg -> case arg of {
                          l | l <= I 4 -> B True;
                          _            -> B False})
  ```

- `n-e->` переходит в

  ```
  filterSrc u g
    (findEdgesByLabel u g (\_->B True))
    (findNodesByLabel u g (\_->B True))
  ```

- `n-e->` переходит в

  ```
  filterDst u g
    (findEdgesByLabel u g (\_->B True))
    (findNodesByLabel u g (\_->B True))
  ```

- `n-e->k` переходит в

```
filterSrcDst u g
  (findEdgesByLabel u g (\_->B True))
  (findNodesByLabel u g (\_->B True))
  (findNodesByLabel u g (\_->B True))
```

- `!n-e->` переходит в

  ```
  filterSrc u g (findEdgesByLabel u g (\_->B True)) [n]
  ```

- `-e->!n` переходит в

  ```
  filterDst u g (findEdgesByLabel u g (\_->B True)) [n]
  ```

- `!n-e->!k` переходит в

  ```
  filterSrcDst u g
    (findEdgesByLabel u g (\_->B True)) [n] [k]
  ```

Как видно, результатом сопоставления любого образца являются списки значений, связанные с идентификаторами, определенными в теле запроса. Проекцией для всех 4 типов запросов являются вложенные вызовы функции `fold` (по одному вызову на каждое используемое связанное значение) по этим спискам, различающиеся лишь способом использования элементов списков.

- all-запрос просто возвращает все необходимые списки, преобразуя их в упорядоченные множества (убирая повторяющиеся элементы);
- graph-запрос выполняет операцию `attachNode` или `attachEdge` для каждого элемента списка, в зависимости от того, является ли он вершиной или ребром;
- set-запрос строит декартово произведение полученных списков, преобразованное в упорядоченное множество;
- some-запрос осуществляет те же действия, что и set-запрос, и, если полученное множество не пусто, извлекает его первый элемент; в противном случае возвращается выражение, заданное в else-ветке запроса.

5.3 Внутреннее представление графов

Как видно из предыдущего подраздела, генерация кода в язык Haskell не привязана к внутреннему представлению графов и оперирует фиксированным набором функций, осуществляющих всю необходимую работу с ними. Под «внутренним представлением графов» понимается выбор конкретного представления для глобального состояния (universe), а также конкретизация понятия «граф» в рамках выбранного представления. Это даёт простор для проведения исследований по выбору наилучшего такого представления с учётом специфики языка, а также для осуществления оптимизаций для описанных базовых функций.

В ходе работы было рассмотрено два основных представления. В обоих представлениях вершины и рёбра отождествляются с некоторыми целыми числами (уникальными идентификаторами). Каждому ребру и вершине сопоставляется некоторая метка типа `L`. Кроме того, каждое ребро имеет ровно по одной начальной и конечной вершине. Отсюда возникает следующий минимальный структурный состав глобального состояния (состояние представляется в виде кортежа, а нижеперечисленные атрибуты являются компонентами этого кортежа):

- `nodes :: Int` — наибольший идентификатор из всех созданных на данный момент вершин;
- `edges :: Int` — наибольший идентификатор из всех созданных на данный момент рёбер;
- `nodeLabel :: IntMap L` — сопоставление каждой вершине некоторой метки;
- `edgeLabel :: IntMap L` — сопоставление каждому ребру некоторой метки;
- `edgeSrc :: IntMap Int` — сопоставление каждому ребру идентификатора начальной вершины;
- `edgeDst :: IntMap Int` — сопоставление каждому ребру идентификатора конечной вершины.

Основа первого подхода заключается в выделении двух понятий: view и selector. View — это граф, построенный с помощью какого-либо конструктора графов. Selector — граф, получаемый в результате выполнения операций `or`, `and` и `but` над view. Каждый view задаётся идентификатором, а принадлежность вершины или ребра какому-либо view — с

помощью соответствующих структур. Следовательно, глобальное состояние в этом случае дополняется следующими компонентами:

- `views :: Int` — наибольший идентификатор из всех созданных на данный момент view;
- `nodeViews :: IntMap IntSet` — каждой вершине сопоставляется множество view, в которых она содержится;
- `edgeViews :: IntMap IntSet` — каждому ребру сопоставляется множество view, в которых оно содержится.

Второй подход рассматривает графы как некоторые множества вершин и рёбер. Такое представление не требует хранения дополнительной информации в глобальном состоянии.

Реализация базовых функций по конструированию графов (`createView`, `createNode`, `createEdge`, `attachNode` и `attachEdge`) достаточно тривиальна в обоих представлениях. Временная сложность таких операций составляет $O(\log(|V_G|+|E_G|))$ (где V_G и E_G — множества вершин и рёбер графа, в который происходит добавление). Однако выбор наилучшей реализации для функций осуществления запросов к графам является наиболее важной задачей, потому что, как показывает практика, именно они выполняются чаще всего в реальных программах: работа этих функций может отнимать до 80% всего времени выполнения.

Функции `filterSrc`, `filterDst` и `filterSrcDst` реализуются независимо от выбора конкретного способа представления графов, так как опираются на использование информации, общей для обоих подходов (`edgeSrc` и `edgeDst`). Рассмотрим результаты, которых удалось достичь в попытках оптимизации этих функций, на примере `filterSrcDst`. Эта функция получает три списка и теоретически должна построить их декартово произведение, а затем выбрать из него только те элементы, которые соответствуют реальным тройкам «начальная вершина–ребро–конечная вершина». Данное решение имеет временную сложность $O(|E| \cdot |V|^2)$. Однако возможен другой вариант: достаточно преобразовать получаемые списки вершин в множества, а затем, пройдя по всем элементам списка рёбер, отобрать лишь те, у которых начальная и конечная вершина принадлежат этим множествам. Временная сложность такого решения — $O(|E| \cdot (\log|E| + \log|V|))$, что существенно повышает производительность даже с учетом накладных расходов по созданию временных множеств.

Что касается функций `findNodesByLabel` и `findEdgesByLabel`, то первый вариант оказывается крайне неэффективным для них: необходимо осуществить проход по *всем* вершинам или рёбрам, отобрав из них

лишь те, которые удовлетворяют некоторому предикату и принадлежат конкретному view (для этого необходимо для каждой сущности найти множество view, которым оно принадлежит, и проверить, не содержит ли оно данное). В случае, когда граф является некоторым selector, а не view, всё оказывается ещё сложнее. Selector представляется в виде двух множеств: множество view, которые входят в selector как объединение, и множество view, которые должны исключаться из него. Соответственно, реализация перечисленных функций требует дополнительного прохода по этим множествам.

Во втором варианте графы являются множествами вершин и рёбер, а потому функции `findNodesByLabel` и `findEdgesByLabel` реализуются тривиальным образом (проходом по всем элементам с проверкой на соответствие предикату) и имеют временную сложность $O(|V_G|)$ и $O(|E_G|)$. Базовые операции над графами сводятся к базовым операциям над множествами и имеют временную сложность $O(|V_{G_1}| + |V_{G_2}| + |E_{G_1}| + |E_{G_2}|)$.

Приведённый обзор реализаций базовых функций позволяет сделать вывод о том, что второй вариант представления графов является намного более эффективным. Кроме того, работа с тестовыми примерами на языке GRAIL показала необходимость внедрения в язык понятия порядка для множеств и графов. Данное нововведение не вписывалось в первый вариант представления, однако во втором вылилось в изящную и эффективную реализацию. Суть её в следующем: упорядоченное множество является парой из обычного множества и списка (список содержит те же элементы, что и множество, но с учётом порядка). Элементы в списке содержатся в порядке, обратном порядку добавления элементов в упорядоченное множество (для эффективности операции добавления элемента). Базовые операции над множествами в таком случае реализуются посредством удаления или добавления в список первого аргумента элементов списка из второго. Наличие не только списка, но и множества, позволяет сохранить временную сложность всех операций на том же уровне, что и для обычных множеств. За счёт того, что все вычисления носят ленивый характер, список будет конструироваться только в случае, когда он действительно понадобится в программе. В обычном же случае упорядоченные множества ведут себя абсолютно так же, как и обычные, и имеют такие же показатели производительности.

5.4 Взаимодействие с другими системами и библиотеками

Немаловажной составляющей в реализации языка GRAIL была необходимость наглядного представления графов для пользователя и взаимодействие с другими системами, языками и библиотеками. Было решено осуществить это наиболее простым и универсальным способом — реализовать экспорт графовых структур в DOT-формат, а также импорт из него. DOT — это общий язык описания графов в системе визуализации графов Graphviz [10].

Экспорт в этот формат позволяет, во-первых, предоставлять пользователю выходные данные программ на GRAIL в наглядной и понятной форме (существует возможность отображать несколько графов на одной схеме с использованием различных цветов и атрибутов, что позволяет показывать пересечение и вложенность графов), а во-вторых, передавать эти данные в качестве входных в другие системы (формат DOT является стандартом де-факто во взаимодействии между графовыми библиотеками и системами). Импорт из формата DOT делает возможным работу в GRAIL с данными, полученными из других систем.

Кроме того, стоит отметить, что описанную функциональность по работе с форматом DOT можно также использовать в качестве средства для сериализации промежуточных данных на диск (в целях отладки, а также для декомпозиции сложных процессов на несколько независимых этапов).

6 Применение GRAIL: задача о поиске всех вхождений

В данном разделе рассматриваются вопросы практической применимости языка GRAIL: формулируется некоторая практическая задача, строится её решение на языке GRAIL, приводятся различные идеи по реализации эвристик для ускорения работы, строится аналогичное решение на языке Haskell и проводится сравнительный анализ их времени работы.

Определение 6 *Пусть* $P = (V_P, E_P)$ *и* $S = (V_S, E_S)$ *— некоторые графы,* $M_V \subseteq V_P \times V_S$ *и* $M_E \subseteq E_P \times E_S$ *— отношения, заданные на вершинах и рёбрах этих графов.* ***Вхождением*** *графа* P *в граф* S *назовём пару инъективных отображений* $F_V : V_P \to V_S$ *и* $F_E : E_P \to E_S$ *таких, что:*

- *для любой вершины* $v \in V_P \quad (v, F_V(v)) \in M_V$*;*

- *для любого ребра $e \in E_P$ выполняются следующие соотношения:*

$$\begin{cases} (e, F_E(e)) \in M_E \\ src(F_E(e)) = F_V(src(e)) \\ dst(F_E(e)) = F_V(dst(e)) \end{cases}$$

Задача *Написать программу для нахождения всех возможных вхождений графа P в граф S с отношениями M_V и M_E.*

Программа:

```
matchAll p s me mn = matchEdge ## ## (p{-e->}:e) where
  matchEdge corr i c =
    if c == {} then
      {(corr, i)}
    else
      let (e, c') = choose c in
      flatten {
       matchEdge
         (corr or #(src e)-()->(src f), (dst e)-()->(dst f)#)
         (i or #-!f->#)
         c'
       |
       f<-((s but i){-f->, _ when me e f && matchEnds e f}:f)
      }
    fi where
    matchEnds e f = matchEnd src && matchEnd dst where
      matchEnd r   = mn (r e) (r f) && matched (r e) (r f);
      matched  u v = (corr<w-->:!v> ? label w == u : True) &&
                     (corr<:!u-->w> ? label w == v : True);
    ;
  ;
;
```

Алгоритм основан на переборе с возвратами всех рёбер графа `p` с сохранением сопоставлений в метаграфе `corr`, представляющем отображение F_E (вершины графа `corr` помечены вершинами графов `p` и `s`; две вершины соединены ребром только в том случае, если метка начальной вершины сопоставляется с меткой конечной). Стандартная функция `choose` получает в качестве аргумента множество и выделяет из него первый элемент (`let (e, c') = choose c` означает, что `e` связывается с первым элементом `c`, а `c'` — с множеством из оставшихся элементов). Стандартная функция `flatten` получает множество, элементами которого являются множества, и возвращает их объединение.

Пример использования (вершины сопоставляются, если их метки равны; метки на рёбрах игнорируются):

```
runMatching p s = matchAll p s me mn where
  me _ _ = True;
  mn x y = label x == label y;
;
```

Приведённый алгоритм поиска всех вхождений является достаточно общим и может использоваться во многих задачах. Однако время его выполнения на больших тестах оказывается слишком длительным. Можно увеличить скорость работы этого алгоритма, написав некоторые эвристики, ускоряющие процесс для какого-либо класса задач или входных данных.

В качестве одной из эвристик может служить следующая идея. Для случая, когда M_E и M_V проверяют метки на равенство, можно преобразовать графы P и S таким образом, чтобы метки рёбер хранили некоторую информацию из окрестности этого ребра (например, помимо собственной метки, еще и метки начальной и конечной вершин). Возможны различные модификации этой идеи: хранение информации из более широкой окрестности, сопоставление некоторого хеш-кода такой информации для быстроты сравнения и т. д. Это позволит отсечь многие заведомо «неудачные» пути перебора на ранней стадии, тем самым значительно сократив размер дерева перебора.

Приведём пример ещё одной эвристики, применимой к алгоритму в общем случае (не зависящей от вида графов P и S и отношений M_E и M_V), предварительно дав некоторые определения.

Определение 7 ***Степенью*** *(degree) вершины называется число входящих и выходящих из неё рёбер.*

Определение 8 *Пусть P и S — графы из определения вхождения, M_E и M_V — соответствующие отношения для рёбер и вершин.* ***Степенью уникальности вершины*** *$v \in V_P$ назовём число вершин $u \in V_S$, которые находятся с v в отношении M_V. Аналогично,* ***степенью уникальности ребра*** *$e \in E_P$ назовём число рёбер $f \in E_S$, которые находятся с e в отношении M_E.*

Основная задача эвристики — изменить порядок обхода рёбер графа P так, чтобы большинство «неудачных» путей отсекалось как можно

раньше. Для этого сопоставим каждому ребру некоторый вес и будем перебирать рёбра в порядке роста этого веса. Будем искать весовую функцию в следующем виде:

$$cost(e) = \big(orig_E(e)\big)^{\alpha}\Big(\big(orig_V(u)deg(u)\big)^{\beta} + \big(orig_V(v)deg(v)\big)^{\beta}\Big),$$

где $deg(n)$ — степень вершины n, u и v — начальная и конечная вершины ребра e, $orig_V(n)$ и $orig_E(e)$ — степени уникальности вершины и ребра соответственно. Тесты показывают, что удачным является следующий набор параметров: $\alpha = 1$, $\beta = 2$.

Ниже приведены реализация описанной эвристики и новый вариант использования алгоритма:

```
convert g me mn =
  let degree n = size (g{!n-e->}:e) + size (g{-e->!n}:e) in
  let nodeOrig n = size (g{m, _ when mn n m} : m) in
  let edgeOrig e = size (g{-f->, _ when me e f} : f) in
  let weight n = (nodeOrig n) * (degree n) in
  let sqr x = x * x in
  let cost e =
    (edgeOrig e) * (sqr (weight (src e)) + sqr (weight (dst e))) in
      g[n] or #-!e-> | e<-[g{-e->}:e | \x y -> cost x < cost y]#;

runMatching p s = matchAll (convert p me mn) s me mn where
  me _ _ = True;
  mn x y = label x == label y;
;
```

Приведём код аналогичной программы на языке Haskell с использованием библиотеки FGL [4] (программа содержит реализацию как общего алгоритма, так и описанной эвристики):

```
import Data.Map (Map)
import Data.IntMap (IntMap)
import Data.Graph.Inductive.Tree

import qualified Data.List as List
import qualified Data.Map as Map
import qualified Data.IntMap as IntMap
import qualified Data.Graph.Inductive.Graph as G

{- Base types -}
type L = String
type Graph = Gr L L
type Node = G.Node
type LNode = G.LNode L
type Edge = G.Edge
type LEdge = G.LEdge L

label g n = let Just l = G.lab g n in l
```

```
orderedEdges g me mn =
  let edges = G.labEdges g
      degree = G.deg g
      nodeOrig n = length $ filter (\(_,l) -> mn (label g n) l) (G.labNodes g)
      edgeOrig (_,_,l) = length $ filter (\(_,_,l') -> me l l') edges
      sqr x = x * x
      cost e@(src, dst, _) = (e, (edgeOrig e) * (sqr (nodeOrig src * degree src) +
                                                 sqr (nodeOrig dst * degree dst)))
  in map fst (List.sortBy (\ (_,d1) (_,d2) -> compare d2 d1) (map cost edges))

insertNode n to from | fst (G.match n to) == Nothing = G.insNode (n, label from n) to
insertNode _ to _ = to

insertEdge e@(src, dst, lab) to from =
  G.insEdge e (insertNode dst (insertNode src to from) from)

matchAll p s me mn es =
  matchEdges (Map.empty, IntMap.empty, G.empty, s, es, []) where
    matchEdges (eCorr, nCorr, newg, oldg, [], results) = (eCorr, nCorr, newg):results
    matchEdges (eCorr, nCorr, newg, oldg, pe@(pSrc, pDst, pLab):pes, results) =
      foldl f results (filter matchEdge (G.labEdges oldg)) where
        f results se@(sSrc, sDst, _) =
          matchEdges (eCorr', nCorr', newg', oldg', pes, results) where
            eCorr' = Map.insert pe se eCorr
            nCorr' = insNCorr pDst sDst $ insNCorr pSrc sSrc nCorr
            newg'  = insertEdge se newg oldg
            oldg'  = G.delLEdge se oldg
        insNCorr pn sn m = IntMap.insert pn sn $ IntMap.insert sn pn m
        matchNodes n1 n2 = case IntMap.lookup n1 nCorr of
                             Nothing  -> True
                             Just n2' -> n2 == n2'
        matchEdge (sSrc, sDst, sLab) =
          me pLab sLab &&
          mn (label p pSrc) (label s sSrc) &&
          matchNodes sSrc pSrc && matchNodes pSrc sSrc &&
          mn (label p pDst) (label s sDst) &&
          matchNodes sDst pDst && matchNodes pDst sDst

matchAll' p s me mn = matchAll p s me mn (orderedEdges p me mn)
```

Программа на Haskell сопоставима по размеру с программой на GRAIL, однако менее выразительна. Кроме того, Haskell-вариант не работает с мультиграфами ввиду специфики библиотеки FGL. Однако это не мешает сравнивать быстродействие программ.

При тестировании производительности в качестве входных данных использовались следующие: в роли графа P выступал граф из 30 вершин и 34 рёбер, в роли S — граф из 38 вершин и 74 рёбер; общее число вхождений графа P в граф S равно 1280.

Haskell-версия, использующая эвристику, отрабатывала за 30 секунд и использовала около 30 мегабайт памяти. GRAIL-версия с эвристикой работала 5 минут 20 секунд, используя порядка 40 мегабайт.

Если сравнивать быстродействие алгоритма с его первоначальным вариантом (без эвристик и с изначальным представлением графов и базовых функций), то можно отметить увеличение скорости работы более

чем в 2000 раз. Причём немаловажную роль здесь сыграла реализованная высокоуровневая эвристика, которая была придумана только благодаря гибкости и удобству языка GRAIL для прототипирования подобных алгоритмов.

7 Заключение

В рамках данного дипломного проекта был реализован компилятор языка для быстрого прототипирования алгоритмов на графах GRAIL. Для оценки выразительности и демонстрации основных возможностей языка был разработан набор тестовых примеров. В ходе работы язык доказал свою выразительную силу, позволив реализовать эвристику, увеличившую скорость работы общего алгоритма поиска всех вхождений в 20 раз.

Несмотря на то, что компилятор в текущем состоянии не является оптимизирующим, он оказывается вполне пригоден для написания реальных алгоритмов и эвристик. Однако его производительность (проигрыш аналогичным программам на Haskell составляет 10-12 раз) даёт простор для дальнейшей работы и реализации специализированных оптимизаций, направленных на доведение быстродействия до приемлемого уровня.

Список литературы

[1] The History of Programming Languages,
http://hopl.murdoch.edu.au

[2] The Graphscript Language,
http://www.infosun.fim.uni-passau.de/Graphlet/graphscript

[3] The Boost Graph Library,
http://www.boost.org/libs/graph/doc

[4] A Functional Graph Library,
http://web.engr.oregonstate.edu/~erwig/fgl

[5] OcamlGraph Library,
http://www.lri.fr/~filliatr/ocamlgraph

[6] Andy Schürr. PROGRES for Beginners

[7] Medha Shukla Sarkar, Dorothea Blostein, James R. Cordy. GXL — A Graph Transformation Language with Scoping and Graph Parameters

[8] Haskell Language,
http://www.haskell.org

[9] В. Н. Касьянов, В. А. Евстигнеев. Графы в программировании: обработка и визуализация // «БХВ-Петербург», 2003.

[10] Graphviz,
http://www.graphviz.org