МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра дискретной математики и алгоритмики

РЫЖЕВИЧ Александра Анатольевна

ПРОБЛЕМА ПРЕДСТАВЛЕНИЯ ГРАФОВ В ФУНКЦИОНАЛЬНОМ ПРОГРАММИРОВАНИИ

Магистерская диссертация

специальность 1-31 81 09 «Алгоритмы и системы обработки больших объемов информации»

Научный руководитель:
Котов Владимир Михайлович
доктор физико-математических наук, профессор

Допущена к защите

«____» _____ 2020 г.

Зав. кафедрой дискретной математики
и алгоритмики
доктор физико-математических наук,
профессор В. М. Котов

Минск, 2020

ОГЛАВЛЕНИЕ

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Магистерская диссертация, 57 с., 8 иллюстраций, 7 листингов, 13 источников, 1 приложение.

Ключевые слова: ГРАФЫ; ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ; HASKELL; БЕЗОПАСНОСТЬ ТИПОВ; БИБЛИОТЕКА ГРАФОВ; МАКСИМАЛЬНЫЙ ПОТОК.

Объектом исследования являются представление графов в чистом функциональном окружении, а также проблемы, затрудняющие написание идиоматического кода без потери эффективности.

Цель работы включает в себя анализ существующих подходов, определение проблем, которые встречаются при работе с ними, а также проектирование библиотеки в качестве решения данных проблем, которая предлагает безопасный интерфейс для конструирования и эффективной обработки графов в чистом функциональном стиле.

Методы исследования, использованные в данной работе, включают анализ, проектирование, экспериментирование и тестирование.

Результатом работы является библиотека для обработки графов, реализованная на языке программирования Haskell. Библиотека предлагает средства для безопасного с точки зрения типов построения графов и оперирования ими. Она предоставляет низкоуровневый интерфейс для написания эффективного кода в императивном стиле, а также высокоуровневые абстракции для разработки идиоматических чистых функциональных алгоритмов на их основе. Библиотека включает обширный набор тестов и встраивается в существующие системы тестирования производительности различных библиотек для работы с графами на языке Haskell.

Область применения результатов работы покрывает различные области исследования, для которых необходима библиотека обработки графов общего назначения, например компьютерные сети, организация данных, анализ социальных сетей и др.

АГУЛЬНАЯ ХАРАКТАРЫСТЫКА РАБОТЫ

Магістарская дысертацыя, 57 с., 8 малюнкаў, 7 лістынгаў, 13 крыніц, 1 дадатак.

Ключавыя словы: ГРАФ; ФУНКЦЫЯНАЛЬНАЕ ПРАГРАМАВАННЕ; БЯСПЕЧНАСЦЬ ТЫПАЎ; HASKELL; БІБЛІЯТЭКА ГРАФАЎ; МАКСІМАЛЬНЫ ПАТОК.

Аб’ектам даследавання з’яўляюцца прадстаўленне графаў у чыстым функцыянальным асяроддзі, а таксама праблемы, якія ўскладняюць напісанне ідыяматычнага кода без страты эфектыўнасці.

Мэта работы ўключае ў сябе аналіз існуючых падыходаў, вызначэнне праблем, якія сустракаюцца пры працы з імі, а таксама праектаванне бібліятэкі ў якасці рашэння дадзеных праблем, якая прапануе бяспечны інтэрфейс для канструявання і эфектыўнай апрацоўкі графаў у чыстым функцыянальным стылі.

Метады даследавання, выкарыстаныя ў дадзенай працы, уключаюць аналіз, праектаванне, эксперыментаванне і тэставанне.

Вынікам працы з’яўляецца бібліятэка для апрацоўкі графаў, рэалізаваная на мове праграмавання Haskell. Бібліятэка прапануе сродкі для бяспечнай па тыпах пабудовы графаў і апэравання імі. Яна дае нізкаўзроўневы інтэрфейс для напісання эфектыўнага кода ў імператыўным стылі, а таксама высокаўзроўневыя абстракцыі для распрацоўкі ідыяматычных чыстых функцыянальных алгарытмаў на іх аснове. Бібліятэка ўключае шырокі набор тэстаў і ўбудоўваецца ў існуючыя сістэмы тэставання прадукцыйнасці розных бібліятэк для працы з графамі на мове Haskell.

Вобласць ужывання вынікаў працы пакрывае розныя галіны даследавання, для якіх патрэбна бібліятэка апрацоўкі графаў агульнага прызначэння, напрыклад кампутарныя сеткі, арганізацыя дадзеных, аналіз сацыяльных сетак і інш.

ABSTRACT

Master thesis, 57 p., 8 figures, 7 code snippets, 13 sources, 1 appendix.

Keywords: GRAPHS; FUNCTIONAL PROGRAMMING; HASKELL; TYPE SAFETY; GRAPH LIBRARY; MAXIMUM FLOW.

The object of the study is graph representation in a purely functional setting and problems that make it difficult to write idiomatic code without sacrificing performance.

The objective of the thesis is to analyze existing approaches, determine the problems that can be met while working with them and design a library as a solution to these problems that provides a safe interface to construct and process graphs in a purely functional and efficient way.

Research methods used in this work include analysis, design, experimentation, testing, and benchmarking.

The result of the thesis is a graph library implemented in the Haskell programming language. The library provides tools for constructing and manipulating graphs in a type-safe way. It exposes a lower-level interface for writing efficient imperative-style code and higher-level abstractions for building idiomatic purely functional algorithms based on them. The library includes a comprehensive test suite and plugs into existing state-of-the-art benchmark of various Haskell graph libraries.

The field of application covers all kinds of research areas where a general-purpose graph library is needed, e.g. computer networks, data organization, social network analysis, etc.

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

1. $\mathbb{N}$ — тип неотрицательных целых чисел, соответствующий типу «`Int`» в Haskell.
2. $\mathbb{B}$ — тип логических значений, соответствующий типу «`Bool`» в Haskell.
3. $(\,)$ — тип пустого кортежа, который имеет единственное значение «`()`».
4. $[\alpha]$ — тип списка значений типа α.
5. $\alpha \to \beta$ — тип функции, ставящей в соответствие значениям типа α значения типа β.
6. $\alpha \times \beta$ — тип пары, где первый элемент имеет тип α, а второй — тип β, что соответствует записи «`(a, b)`» в Haskell.
7. $\forall\alpha.\ \phi$ — тип более высокого ранга, где ϕ — любое выражение типов, которое также может использовать α. Соответствует записи «`forall a. f`» в Haskell.
8. $\mathrm{value} : \alpha$ — глобальный объект с именем value и типом α, что соответствует записи «`value :: a`» в Haskell.
9. $x : \alpha$ — локальный объект с именем x и типом α, что соответствует записи «`x :: a`» в Haskell.
10. $\bullet$ — обозначение аргумента, который не используется в функции и чьё имя не играет роли, что соответствует записи «_» в Haskell.
11. $\lambda x.\ f\ x$ — лямбда-функция, которая принимает аргумент x и возвращает результат применения к нему функции f, что соответствует записи «`\x -> f x`» в Haskell.
12. $f_1 \circ f_2$ — композиция функций f_1 и f_2, что соответствует записи «`f1 . f2`» в Haskell.
13. $x :: xs$ — добавление элемента в начало списка, что соответствует записи «`x:xs`» в Haskell.
14. $xs_1 +\!\!+\, xs_2$ — конкатенация списков, что соответствует записи «`xs1 ++ xs2`» в Haskell.

ВВЕДЕНИЕ

Представление графов в функциональном программировании не является тривиальной задачей. Основной причиной возникновения проблем в эффективной по времени реализации является отсутствие произвольного доступа к памяти.

Простейшей моделью алгоритмов для функциональных языков программирования служит машина указателей[13] (pointer machine). Модель памяти этой машины представляет собой граф указателей, а единственной допустимой операцией является создание записи. Такая запись представляет собой пару, элементами которой могут быть либо значение примитивного типа, либо указатель на другую запись. При этом нельзя обратиться к какой-либо записи напрямую: для этого придётся пройти весь путь по указателям от корневой записи до искомой.

Подобная организация памяти позволяет достичь чистоты операций, то есть получить гарантию того, что структуры данных, переданные какой-либо функции, не будут изменены, а результат будет получен в виде новой структуры, что предоставляет огромные преимущества по сравнению с императивными языками. Однако такая модель несёт также и недостатки, например, невозможность создания такой простой структуры данных, как массив.

Важным следствием описанных ограничений является невозможность обновления какой-либо записи. Чтобы изменить данные, хранящиеся в какой-либо записи, создается новая запись, однако к ней невозможно добраться, так как не существует указателей, которые бы на неё указывали. По этой причине необходимо пересоздать не только целевую запись, но и все записи на пути до корня, изменяя только указатели.

Однако что происходит в случае наличия контура из указателей? Понятно, что изменение записи в таком случае повлечёт за собой изменение всех записей, принадлежащих компоненте сильной связности, образованной контуром, в противном случае структура памяти будет нарушена. Таким образом, обновление какой-либо записи требует пересоздания не только всех записей за пути от неё до корня, но также всех компонент сильной связности, которым эти записи принадлежат. Очевидно, что графы относятся к циклическим структурам, поэтому работа с ними представляет собой некоторые трудности.

Создание циклической структуры также не является простой задачей. Как можно заметить, если создаваемая запись содержит указатель, то этот указатель уже должен быть известен. Таким образом, потомки должны быть созданы раньше своих предков, но как же тогда создать указатель на предка, чтобы получить контур, если предок еще не создан? Это было бы невозможной задачей при отсутствии ленивых вычислений, однако в Haskell можно вызвать функцию, используя в качестве аргумента результат выполнения этой функции, ко-

торый еще неизвестен. Такая техника называется «tying the knot» и позволяет создавать циклические структуры в Haskell[9].

Но что если стоит задача изменить циклический граф? Проблема состоит в том, что при обработке структуры данных Haskell не делает различий между циклической и бесконечной структурами. Таким образом, нет возможности отследить начало и конец контура, используя только чистые вычисления. По этой причине техника «tying the knot» не является сильно полезной в задаче представления графов.

Описанные выше причины делают написание многофункциональных библиотек для работы с графами нетривиальной задачей. В данной работе рассматривается новый подход к созданию такой библиотеки, основанный на абстрактном представлении графа и выделении минимума операций над ним, необходимых для написания полезных алгоритмов над графами.

ГЛАВА 1
ИССЛЕДОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Преимущества функционального программирования

Термин «функциональное программирование» относится как к совокупности идиом программирования, которые можно использовать практически в любом языке программирования, так и к семейству языков программирования, предназначенных для них, включая Haskell, OCaml и Coq.

Функциональное программирование разрабатывалось на протяжении многих десятилетий — его корни уходят к введению лямбда-исчисления Чёрча в 1930-х годах, ещё задолго до появления первых компьютеров. Но с начала 1990-х он вызвал всплеск интереса у промышленных инженеров и создателей языков программирования, играя ключевую роль в таких компаниях, как Microsoft, Facebook и Ericsson.

Основной принцип функционального программирования — это сделать вычисления чистыми, насколько это возможно, в том смысле, что единственным результатом выполнения должно быть получение результата: оно не должно иметь каких-либо побочных эффектов, таких как операции ввода-вывода, изменение значений переменных, перенаправление указателей[11]. Например, если императивная функция сортировки может упорядочить переданный список, модифицировав его, то функция чистой сортировки возьмет исходный список и вернет новый отсортированный список.

Чистота функций в функциональном программировании также обеспечивает их ссылочную прозрачность, что позволяет относиться к ним, как к математическим функциям, применяя математические теоремы, законы и правила, например подстановку функций и удаление общих подвыражений[10].

Существенным преимуществом такого стиля программирования является то, что он облегчает понимание и анализ программ. Если каждая операция в структуре данных даёт новую структуру данных, оставляя старую нетронутой, нет необходимости беспокоиться о том, как эта структура используется и может ли её изменение одной частью программы нарушить какой-либо инвариант в другой её части, на которую она опирается. Эти соображения особенно важны в параллельных системах, где любые изменяемые данные, которые разделяются между потоками, являются потенциальным источником ошибок. Действительно, большая часть недавнего интереса к функциональному программированию в промышленности обусловлена его более простым поведением при наличии параллелизма.

Еще одно преимущество функционального программирования связано с предыдущим: функциональные программы часто намного легче распаралле-

лить, чем их императивные аналоги. Если выполнение вычисления не оказывает никакого влияния, кроме получения результата, то не имеет значения, где оно выполняется. Действительно, идиома Map-Reduce, лежащая в основе Hadoop и используемая Google для индексации сети, является классическим примером функционального программирования.

1.2 Анализ альтернативных решений

1.2.1 Стандартная библиотека контейнеров Haskell

Стандартная библиотека `containers` предлагает модуль `Data.Graph` для работы с графами, который содержит функции для построения графа, а также предлагает некоторые готовые алгоритмы.

Внутри библиотеки граф хранится в виде списка смежности, который представляет собой отображение из ключей вершин в списки смежных вершин. Рассматриваемые графы являются ориентированными, поэтому для моделирования неориентированных графов необходимо использовать симметрические ориентированные графы. Стоит отметить, что библиотека не предусматривает работу со взвешенными графами и в принципе не предлагает механизмов разметки дуг.

Для хранения списка смежности используется стандартный неизменяемый массив, который позволяет обращаться к элементам по ключу за константное время, но не позволяет изменять их. Тип ключей вершин является открытым, а проверка существования ключа не производится, что позволяет обращаться к несуществующей вершине. Такое поведение может привести к ошибке времени выполнения.

В основе представленных алгоритмов лежит единственная идея — возможность построить остовный лес графа, соответствующий обходу этого графа в глубину[12]. При этом построенный лес обладает одним важным свойством: для любой вершины леса и любой пары поддеревьев, образованных сыновьями этой вершины, отсутствуют дуги из левого в правое поддерево с учётом существования строгого порядка сыновей для каждой вершины.

Описанный выше инвариант позволяет выполнять алгоритмы, основанные на алгоритме поиска в глубину, с помощью обхода построенного дерева. Одним из самых важных представленных алгоритмов является топологическая сортировка графа. Благодаря отсутствию дуг «слева направо» порядок вершин топологической сортировки полностью совпадает с порядком вершин во время обратного левого обхода остовного дерева.

Помимо топологической сортировки библиотека предлагает и другие алгоритмы, основанные на поиске в глубину. Среди них поиск компонент графа, сильно связных компонент графа, двусвязных компонент графа, а также алгоритм поиска достижимых вершин. Библиотекой также предоставляется интерфейс для работы с остовным лесом, который позволит построить любой алго-

ритм, основанный на обходе дерева, например, динамику по дереву, с помощью красивого функционального кода. К сожалению, для реализации алгоритмов другого типа необходимо проектировать весь интерфейс с нуля самостоятельно.

Ключевая идея библиотеки состоит в построении остовного леса с инвариантом отсутствия дуг «слева направо». Для этого первым шагом строится циклическая структура, представляющая собой бесконечное дерево в случае наличия вершин, достижимых из корня несколькими способами. Вторым этапом производится прямой левый обход дерева, при котором обрубаются ветви, ведущие к уже посещённым вершинам. Для возможности выставления меток посещённых вершин за константное время используется ST-монада. Однако при таком подходе не гарантируется отсутствие обратных дуг, что может привести к некорректному выполнению алгоритма топологической сортировки, которая невозможна при наличии контура.

Таким образом, модуль `Data.Graph` несомненно обладает некоторыми достоинствами:

- возможность написания красивого идиоматического кода при решении задач на основе обхода в глубину;
- линейное время работы представленных алгоритмов, что сравнимо с императивными аналогами;
- интересный подход к решению задачи, отличающийся от классического императивного обхода в глубину.

Однако библиотека также имеет несколько важных недостатков:

- недопустимая ограниченность интерфейса для стандартной библиотеки, не позволяющая работать даже со взвешенными графами, не говоря о более общих классах графов;
- наличие эффективных алгоритмов только на основе обхода в глубину;
- наличие небезопасных указателей на вершины, которое может повлечь ошибку времени выполнения, что не является идиоматическим решением;
- невозможность отследить некорректное завершение алгоритма при нарушении инварианта отсутствия обратных дуг в графе.

1.2.2 Библиотека индуктивных графов

Последствием доминирования императивных языков программирования и разработки алгоритмов под их особенности является то, что граф в большинстве случаев представляется как монолитная структура в виде пары множества и вершин и множества дуг. По этой причине большинство алгоритмов на графах используют такой механизм, как пометка посещённых вершин, что делает невозможным их реализацию в чистом функциональном стиле. Автор библиотеки FGL (functional graph library) предложил абсолютно другой подход к представлению графов, который не нуждается указанном выше механизме.

Автор предложил рассматривать каждый граф как комбинацию графа меньшего на единицу порядка и одной вершины с контекстом[4]. Контекст включает

в себя список входящих дуг, метку вершины и список исходящих дуг. Если граф является тривиальным, то он представляет собой комбинацию вершины с пустым контекстом и пустого графа. Таким образом, последовательно добавляя по одной вершине с контекстом, можно построить любой граф, который будет представлять собой последовательность применённых операций комбинации. Такое представление даёт не что иное как индуктивное определение графа.

Операция комбинации и проверки на пустоту даёт возможность использовать сопоставление по образцу и реализовать простейшие функции, не требующие определённого порядка посещения вершин, работающие за линейное время. К таким функциям относится отображение и свёртка графа. Однако по той причине, что вершина определяется ключом, по которому происходит сопоставление по образцу, такой интерфейс также не является безопасным для использования.

Однако, так как большинство алгоритмов требуют посещать вершины именно в определённом порядке, автор ввёл ещё одну операцию над графом — получение контекста конкретной вершины. Благодаря этой операции становится возможным реализовать алгоритмы обхода в глубину и ширину, а также основанных на них, алгоритм Дейкстры, и другие, которые, при условии работы операций получения контекста за константу, имеют трудоёмкость того же порядка, что и императивные реализации. При этом сохраняется идиоматический подход к написанию кода и его простота.

К сожалению, ещё не было представлено такой реализации описанных операций, которая бы обеспечивала константное время их выполнения. Реализация библиотеки для языка Haskell использует поисковые деревья и сжатые префиксные деревья для внутреннего представления графа, что, очевидно, нелинейно увеличивает асимптотики операций над графом.

Таким образом, библиотека FGL предлагает весьма интересный подход к проблеме представления графов и обладает важными достоинствами:

- возможность использования сопоставления по образцу при обработке графа благодаря индуктивному определению графа;
- идиоматический подход к написанию кода и его простота;
- простой интерфейс, выраженный лишь одной операцией, но позволяющий реализовать достаточно сложные алгоритмы.

Однако данная библиотека имеет также и несколько существенных недостатков:

- множество реализаций представленного интерфейса, ни одна из которых не позволяет достигнуть теоретической оценки;
- наличие небезопасных указателей на вершины, сопоставление по образцу на которые может привести к ошибке времени выполнения, если вершина не существует.

1.2.3 Алгебраические графы

Ещё один интересный подход лежит в основе библиотеки Alga[1]. Проблемой описанных выше библиотек является то, что неаккуратное использование указателей на вершины может привести к ошибке времени выполнения, то есть их интерфейс не является безопасным. Библиотека Alga лишена данного недостатка.

Интерфейс библиотеки состоит из всего четырёх конструкторов графа: пустой граф, вершина, наложение и соединение. Пустой граф представляет собой нейтральный элемент для операций наложения и соединения. Конструктор вершины создаёт тривиальный граф с вершиной с указанной меткой. Операция наложения принимает два графа и возвращает граф, множество вершин и дуг которого являются объединением множества вершин и дуг операндов. Операция соединения также принимает два графа и возвращает их соединение с добавленными дугами между всеми парами вершин из разных операндов. При этом очевидно, что с помощью представленных конструкторов можно построить абсолютно любой граф, и при этом нельзя построить несуществующий граф, случайно использовав указатель на несуществующую вершину.

Множество конструируемых графов вместе с нейтральным элементом и операциями наложения и соединения образуют нечто близкое к полукольцу, что позволяет строить различные аксиомы поверх такого определения графа. Простота определения графа и множество аксиом позволяют использовать такое представление для доказательства корректности алгоритмов и различных теорем.

Особенностью библиотеки является предположение, что все вершины имеют уникальные метки. Это означает, что если в операции над двумя графами присутствуют две вершины с равной меткой, то они будут трактоваться как одна и та же вершина, например, соединение двух вершин с равной меткой всё ещё вернет тривиальный граф. Также из-за особенностей подхода в библиотеке невозможно использование размеченных дуг. В экспериментальной версии библиотеки разработчики пытаются внедрить возможность размеченных дуг, однако на метки наложены настолько серьёзные ограничения, что их использование не несёт практической пользы.

Прочие функции библиотеки, включая алгоритмы, реализованы на основе этих конструкторов, при этом не зависят от реализации самих конструкторов. Реализация интерфейса не является строго прописанной, однако в предлагаемой авторами версии для внутреннего представления графа используются множества. Разработчики постарались выполнить алгоритмы над графами также в функциональном стиле, однако не смогли избежать проблемы с метками посещённых вершин. Для реализации данного механизма они также использовали множества, что вкупе со внутренним представлением графа стало причиной более высокого порядка трудоёмкости реализованных алгоритмов.

Итак, библиотека Alga обладает несколькими важными достоинствами:

- простота интерфейса и реализации внутреннего представления графа;
- ценность с точки зрения математического подхода.

Однако она обладает также и недостатками, которые делают нецелесообразным выбор данной библиотеки для практического использования:

- сильное отставание в трудоёмкости алгоритмов по сравнению с их императивными аналогами;
- ограничение уникальности меток вершин графа;
- отсутствие возможности работы с помеченными дугами.

1.3 Актуальность поставленной задачи

Как следует из предыдущих разделов, функциональные языки программирования, в том числе Haskell, предлагают ряд преимуществ над императивными языками, которые делают их весьма востребованными в настоящее время. В то же время алгоритмы на графах применяют в большой доле задач программирования, что делает необходимым предоставить пользователям хорошие инструменты для работы с графами, которые должны быть сравнимы по эффективности с их реализациями в императивном программировании. Однако язык Haskell не может предоставить большого выбора библиотек для работы с графами, а существующие библиотеки обладают рядом недостатков, которые делают невозможным написание безопасных алгоритмов на графах в виде чистого функционального и эффективного по времени кода. Всё вышеперечисленное даёт широкий простор в разработке новой библиотеки, учитывающей все недостатки существующих решений.

ГЛАВА 2
АБСТРАКТНОЕ ПРЕДСТАВЛЕНИЕ ГРАФА

Одной из основных идей данной работы служит написание библиотеки с интерфейсом, позволяющим писать чистый функциональный код при работе с графами, лишённый недостатков решений, описанных в предыдущей главе.

Одним из общих недостатков было наличие указателей на вершины, которые позволяли неправильно использовать библиотеку и в результате получать ошибку времени выполнения. Соответственно, главным требованием к новой библиотеке является создание такого интерфейса, который сделал бы невозможным обращение к несуществующей вершине уже по дизайну.

Ещё одним требованием является наличие интерфейса, позволяющего писать идиоматический код на основе разрабатываемой библиотеки. Механизмом выполнения такого требования является нахождение необходимого набора общих операций над графом, которые позволят реализовать любой алгоритм на их основе.

Также библиотека должна подходить для большинства практических задач и решать ещё один общий недостаток других библиотек — неразмеченность вершин и дуг графа. И дуги, и вершины графа могут иметь любой тип метки, при этом на них не накладывается ограничение уникальности, то есть метка вершины не является её идентификатором.

2.1 Внутреннее представление графа

Будем считать, что все графы относятся к типу $\mathrm{Graph}\ e\ v$, где e — тип меток дуг, v — тип меток вершин. Если пользователю нужны непомеченные вершины или дуги, то в качестве типа меток можно использовать тип $(\,)$. Для наиболее частого частного случая — невзвешенного графа — заведем псевдоним $\mathrm{Graph'}\ v$. Таким образом, вершины и дуги могут иметь любой тип меток, так как на них не накладывается никакого ограничения, в том числе ограничения уникальности.

Чтобы библиотека следовала функциональному стилю, тип графа должен быть неизменяемым, то есть все операции над графом должны возвращать новый объект, оставляя предыдущий объект нетронутым. Это позволяет скрыть от пользователя внутреннее представление графа, позволяя ему работать с типом $\mathrm{Graph}\ e\ v$, как с некоторым абстрактным типом данных, не делая никаких предположений о его внутреннем устройстве и пользуясь лишь предоставленным набором операций.

Взгляд на $\mathrm{Graph}\ e\ v$ как на абстрактный тип помогает выполнить одно из важнейших требований к библиотеке: отсутствие доступных пользователю небезопасных указателей на вершины. Если тип абстрактен, то пользователь

работает со всем объектом целиком, но никак ни с его элементами, поэтому предоставим ему такой набор функций над графом, который лишит необходимости обращаться к отдельным вершинам и позволит писать чистый функциональный код на его основе.

Сокрытие внутреннего представления позволяет выбрать любое из возможных представлений графа. Тогда будем использовать список смежности, так как он является наиболее удобным и экономным по памяти при реализации многих алгоритмов.

Для хранения списка смежности можно использовать неизменяемый вектор, предлагаемый стандартной библиотекой. Вектор позволяет обращаться к элементам за константное время, так как он реализован с использованием знаний об архитектуре памяти компьютера в обход функциональной модели. Однако его использование допустимо по причине сокрытия внутренней реализации от пользователя. Представим граф внутри библиотеки следующим образом:

$$\textbf{data}\ \mathrm{Adj}\ e = \mathrm{Adj}\ \{\mathrm{target} : \mathbb{N},\ \mathrm{label} : e\}$$
$$\textbf{data}\ \mathrm{Graph}\ e\ v = \mathrm{Graph}\ \{\mathrm{values} : \mathrm{Vector}\ v,\ \mathrm{adjs} : \mathrm{Vector}\ (\mathrm{Vector}\ (\mathrm{Adj}\ e))\}$$

Граф в представлении выше задан вектором меток вершин values и списком смежности adjs. Во внутреннем представлении вершины определяются числовыми указателями, однако они безопасны, так как недоступны снаружи.

На основании данного представления уже можно реализовать простейшие функции. Так как пользователь не имеет доступа к внутреннему представлению и не может обратиться к вершине напрямую, ему необходимо предоставить интерфейс для получения всех данных, хранящихся в графе, в противном случае пользователь не сможет обработать результат операций над графом. Роль этого интерфейса будут выполнять функции $\mathrm{vertices}$ и edges, которые возвращают список вершин и дуг соответственно:

$$\mathrm{vertices} : \mathrm{Graph}\ e\ v \to [v]$$
$$\mathrm{edge} : \mathrm{Graph}\ e\ v \to [v \times e \times v]$$

Заметим, что возвращаемые функциями значения никак не связаны с внутренними указателями на вершины, и пользователь получает только метки вершин и дуг.

После того, как мы определили функции извлечения данных из графа, можно определить некоторые операции над ним. Согласно концепции функционального программирования и по условию неизменяемости типа графа, функции не могут модифицировать переданный граф и могут лишь вернуть новый граф в качестве результата. Часто результатом различных вычислений над графами становятся новые метки вершин или дуг. Например, результатом работы алгоритма Дейкстры становится массив меток вершин, соответствующий расстояниям от начальной вершины до всех остальных вершин. Так как представление графа абстрактно, возвращать список меток не имеет смысла, если

пользователь библиотеки пожелает использовать их для дальнейших вычислений, так как у него не будет больше возможности соотнести метки с самими вершинами (порядок вершин никак не ограничен). По этой причине результатом выполнения операций будет новый граф структуры исходного графа, метки вершин или дуг которого содержат результат вычислений. Дальше рассмотрим несколько примеров таких операций.

К самым элементарным функциям можно отнести функции отображения множества меток вершин vmap и отображения множества меток дуг emap. Они достаточно просты и всего лишь меняют тип меток, не учитывая структуры графа. Пользователю необходимо лишь предоставить функцию отображения старого типа в новый тип:

$$\mathrm{vmap} : (v_1 \to v_2) \to \mathrm{Graph}\ e\ v_1 \to \mathrm{Graph}\ e\ v_2$$
$$\mathrm{emap} : (e_1 \to e_2) \to \mathrm{Graph}\ e_1\ v \to \mathrm{Graph}\ e_2\ v$$

Выполняя различные операции над графами, которые не меняют его структуру, можно возвращать результат, специфичный для каждой вершины, в качестве новой метки этой вершины. Так как структура остаётся неизменной, то сопоставить результат с оригинальной меткой не составит труда. В качестве примера определим функцию получения потомков вершин succs, которая возвращает граф, метками вершин которого являются списки пар метки дуги и метки вершины-потомка в исходном графе:

$$\mathrm{succs} : \mathrm{Graph}\ e\ v \to \mathrm{Graph}\ e\ [e \times v]$$

Через функцию succs можно выразить функцию вычисления степеней вершин degree как длин списков потомков каждой вершины:

$$\mathrm{degree} : \mathrm{Graph}\ e\ v \to \mathrm{Graph}\ e\ \mathbb{N}$$
$$\mathrm{degree} = \mathrm{vmap}\ \mathrm{length} \circ \mathrm{succs}$$

Одной из важных операций над графом является транспонирование графа $\mathrm{transpose}$, то есть реверсирование всех дуг графа. Такая операция меняет структуру графа, поэтому необходимо построить список смежности графа заново и создать новый граф. На основе транспонирования графа можно реализовать функцию получения списка предков каждой вершины preds, которая возвращает граф, метками вершин которого являются списки пар метки вершины-предка и метки дуги в исходном графе:

$$\mathrm{transpose} : \mathrm{Graph}\ e\ v \to \mathrm{Graph}\ e\ v$$
$$\mathrm{preds} : \mathrm{Graph}\ e\ v \to \mathrm{Graph}\ e\ [v \times e]$$

Стоит заметить, что все описанные функции работают за линейное время, что не уступает трудоёмкости аналогичных операций в императивных реализациях. Реализация может быть найдена в модулях `Data.Graph.Abstract` и `Data.Graph.Abstract.Internal`.

2.2 Базовые операции над графом

Ещё одной важной задачей, поставленной перед разрабатываемой библиотекой является выделение набора базовых операций над графом, позволяющих реализовать большую часть алгоритмов над графами за сравнимое с императивными реализациями время.

Первой функцией, которую можно выделить как базовую, является функция $\mathrm{transpose}$, выполняющая транспонирование графа. Эта функция уже была описана в первом разделе. Данная функция очень важна, так как разворот дуг применяется во многих алгоритмах на графах, например в поиске компонент сильной связности.

Следующей функцией скорее прикладного характера, но не менее важной является функция склеивания графов zip. Она принимает два графа с разными типами меток дуг и вершин, но с одинаковой структурой, и возвращает новый граф той же структуры, но со склеенными метками вершин и дуг:

$$\mathrm{zip} : \mathrm{Graph}\ e_1\ v_1 \to \mathrm{Graph}\ e_2\ v_2 \to \mathrm{Graph}\ (e_1 \times e_2)\ (v_1 \times v_2)$$

Важность данной функции заключается в том, что согласно подходу, использующемуся в данной библиотеке, результат операции над вершиной возвращается в качестве новой метки этой вершины, поэтому может возникнуть необходимость склеить исходный и полученные графы, чтобы соотнести результаты вычислений с оригинальными вершинами. Например, $\mathrm{zip}\ g\ (\mathrm{degree}\ g)$ вернёт граф с метками вершин в виде пар исходных меток и степеней вершин.

Из типа функции этот факт неочевиден, но функция zip обладает важным свойством сохранения структуры исходных графов, которое может быть выражено двумя равенствами:

$$\mathrm{vmap}\ \mathrm{fst}\ (\mathrm{zip}\ g_1\ g_2) = g_1$$
$$\mathrm{vmap}\ \mathrm{snd}\ (\mathrm{zip}\ g_1\ g_2) = g_2$$

В первом разделе была рассмотрена функция emap, которая отображает метки дуг из одного типа в другой. Однако часто для правильного выставления новой метки дуги необходимо знать её контекст — метки инцидентных ей вершин. Для этого введём более общую функцию emapc, которая принимает функцию, отображающую три типа в новый тип метки вместо одного:

$$\mathrm{emapc} : (v \to e_1 \to v \to e_2) \to \mathrm{Graph}\ e_1\ v \to \mathrm{Graph}\ e_2\ v$$

Стоит заметить, что теперь функцию emap можно выразить через функцию emapc, так как вторая является обобщением первой:

$$\mathrm{emap}\ f = \mathrm{emapc}\ (\lambda \bullet\ e\ \bullet .\ f\ e)$$

Для многих алгоритмов важен обход вершин в определённом порядке, которые основываются в своих вычислениях на полученных ранее результатах для

других вершин. Однако предоставить пользователям самим определять этот порядок и обращаться к произвольным вершинам графа невозможно, ведь внутреннее представление графа скрыто. Более того, для этого им потребуется использование ST-монады, что крайне нежелательно, поэтому необходимо самим предоставить такую функцию, которая не раскроет внутреннего представления, но позволит обойти граф в нужном порядке и предоставит необходимый контекст для вычислений. Для этой цели и предназначены функции $\mathrm{transformu}$ и $\mathrm{transformd}$:

$$\begin{aligned}\mathrm{transformu} : (v_1 \to \mathbb{B}) \to (v_1 \to [e \times v_2] \to v_2) \to (v_1 \to v_2) \\ \to \mathrm{Graph}\ e\ v_1 \to \mathrm{Graph}\ e\ v_2\end{aligned}$$

$$\begin{aligned}\mathrm{transformd} : (v_1 \to \mathbb{B}) \to ([v_2 \times e] \to v_1 \to v_2) \to (v_1 \to v_2) \\ \to \mathrm{Graph}\ e\ v_1 \to \mathrm{Graph}\ e\ v_2\end{aligned}$$

В качестве порядка обхода был выбран порядок обхода в ширину. Обход выполняется прозрачно для пользователя, а сами функции $\mathrm{transformu}$ и $\mathrm{transformd}$ оперируют уже с построенным деревом обхода, имея знания лишь о родителе и потомках каждой вершины.

Контроль пользователя над скрытым обходом ограничивается указанием начальных вершин обхода с помощью аргумента типа $v_1 \to \mathbb{B}$. Подобным образом библиотека избегает использования указателей на вершины, то есть пользователь предоставляет предикат, который по метке вершины определяет, является ли она начальной или нет. При этом на предикат не накладывается никаких ограничений, то есть начальных вершин может быть несколько либо вообще ни одной.

Следующий параметр различает функции $\mathrm{transformu}$ и $\mathrm{transformd}$ между собой. Иногда при обходе графа значение новой метки вычисляется на основании меток потомков, а в некоторых случаях — на основании меток предка.

Первый случай назовём обходом вверх (up), который соответствует функции $\mathrm{transformu}$. В этом случае вычисление новых меток вершин начинается с листьев дерева обхода, причём порядок обхода вершин ограничен лишь инвариантом нахождения родителя дальше любого его потомка. Для расчёта новой метки вершины пользователю необходимо предоставить функцию типа $v_1 \to [e \times v_2] \to v_2$, которая вычисляет новую метку вершины на основании старой метки этой вершины и списка пар меток дуг и новых меток детей в дереве обхода.

Второй случай назовём обходом вниз (down), который соответствует функции $\mathrm{transformd}$. В этом случае вычисление новых меток вершин начинается с корня (корней, если деревьев несколько) дерева обхода, при этом порядок обхода вершин гарантирует посещение родителя прежде посещения любого из его сыновей. Для вычисления новой метки вершины пользователю необходимо предоставить функцию типа $[v_2 \times e] \to v_1 \to v_2$, которая вычисляет новую

метку вершины на основании новой метки родителя, метки дуги от него в дереве обхода (список нужен, так как у некоторых вершин может не быть предка) и старой метки этой вершины.

Однако функция трансформирует граф полностью, то есть меняет метки всех вершин. Что случится, если какие-либо вершины недостижимы из начальных вершин? Мы не можем оставить их неизменными или выбросить из графа, ведь тогда изменится его структура. Решением служит третий аргумент типа $v_1 \to v_2$, который ставит в соответствие старым меткам недостижимых вершин новые метки со значениями по умолчанию. Таким образом, не останется вершин с неизменёнными метками.

Иногда при решении задач на графах не все дуги можно использовать при переходе. Для поддержки такой возможности добавим ещё два трансформера transformd' и transformu' с фильтрацией дуг. Фильтрацию дуг будет обеспечивать предикат типа $e \to \mathbb{B}$, передаваемый в качестве дополнительного аргумента:

$$\begin{aligned}\text{transformu'} : (v_1 \to \mathbb{B}) \to (e \to \mathbb{B}) \to (v_1 \to [e \times v_2] \to v_2) \to (v_1 \to v_2) \\ \to \text{Graph } e\ v_1 \to \text{Graph } e\ v_2\end{aligned}$$

$$\begin{aligned}\text{transformd'} : (v_1 \to \mathbb{B}) \to (e \to \mathbb{B}) \to ([v_2 \times e] \to v_1 \to v_2) \to (v_1 \to v_2) \\ \to \text{Graph } e\ v_1 \to \text{Graph } e\ v_2\end{aligned}$$

В качестве примера рассмотрим задачу нахождения расстояния между парой вершин в невзвешенном графе. Решим её с помощью функции `distance`, которую теперь возможно реализовать в чистом функциональном стиле благодаря предоставленному библиотекой интерфейсу:

```haskell
distance :: (v -> Bool) -> (v -> Bool) -> Graph' v -> Int
distance isStart isTarget g =
    snd . head . filter (isStart . fst) . vertices $ zip g g'
  where
    g' = transformu isStart choosePath (const -1) g
    choosePath v succs
        | isTarget v = 0
        | otherwise =
            let ds = filter (>= 0) . map snd $ succs
            in if null ds then -1 else min ds + 1
```

В данном примере мы используем обход «снизу вверх» по дереву, корнем которого является начальная вершина. В качестве новых меток вершин будем использовать расстояние до конечной вершины в дереве обхода, поэтому всем недостижимым вершинам из начальной поставим метку -1. При посещении какой-либо достижимой вершины будем действовать следующим образом: если вершина является конечной, то расстояние от неё до самой себя равно 0,

в противном случае расстояние до конечной вершины на единицу больше минимального расстояния от сыновей, из которых достижима конечная вершина. Если конечная вершина не является достижимой ни из одного из сыновей, то расстояние также полагаем равным -1.

Заметим, что все представленные функции работают за линейное от числа вершин и дуг время, что сравнимо с трудоёмкостью алгоритмов над графами в функциональном программировании. Реализация может быть найдена в модуле `Data.Graph.Abstract.Transform`.

2.3 Безопасное построение графа

В предыдущих разделе описывалось, как представлен граф изнутри и как организована работа с объектами этого типа, чтобы оставлять внутреннее представление скрытым. Закономерно возникает вопрос, как построить такой граф.

Изначально предполагался вариант построения графа с помощью функции, которая принимает набор вершин и дуг в каком-то доступном пользователю представлении. Ниже представлены рассмотренные варианты:

$$\begin{aligned}
&\text{build} : [v] \to [\text{Edge } e] \to \text{Graph } e\ v \\
&\text{build} : (\text{Vertex} \to v) \to [\text{Edge } e] \to \text{Graph } e\ v \\
&\text{build} : [v \times [e \times \text{Offset}]] \to \text{Graph } e\ v
\end{aligned}$$

Первая функция принимает список меток вершин и список дуг. Главным преимуществом этой функции является её естественность, так как такой вид конструктора наиболее ожидаем для тех, кто привык работать с графами в императивном программировании. Однако такой подход не является безопасным, так как список дуг никак не ограничен значениями в списке вершин и вполне может содержать указатель на несуществующую вершину. Использование метки вершины вместо указателя на неё при описании дуги невозможно из-за свойства неуникальности меток вершин.

Вторая функция принимает лишь список дуг, а метки вершин вычисляются уже исходя из встреченных «идентификаторов» вершин типа Vertex. Такой подход не позволяет построить изолированные вершины, но это легко исправить, добавив третий аргумент типа $[\text{Vertex}]$. Специальный тип Vertex может показаться гарантией безопасности, однако ничто не защищает пользователя от возможной ошибки во время вызова функции отображения вершины в её метку.

Последняя функция принимает список пар вершины и списка исходящих из неё дуг, где каждая дуга также представлена парой метки дуги и смещением до концевой вершины дуги в списке вершин. Если ограничить тип смещения Offset только целочисленностью, то возможны выходы за пределы границ. В функциональных языках программирования с более строгой динамической типизацией возможно было бы индуктивно задать специальный тип для подобного списка вершин, в который входило бы также число свободных (ещё не объявленных, с

положительным смещением) и занятых (с отрицательным смещением) вершин, которые бы позволили ограничить тип Offset допустимым интервалом смещений. Однако в Haskell такой механизм возможно реализовать лишь в теории и невозможно использовать на практике.

Итак, более-менее знакомые подходы из императивного программирования не гарантируют достаточной безопасности. Как оказалось, самой главной проблемой является возможность создать указатель на несуществующую вершину. С одной стороны, совсем отказываться от указателей не хочется, так как они очень удобны в использовании, но с другой стороны необходимо найти способ сделать их безопасными. Учитывая особенности функционального программирования и языка Haskell в частности, хотелось достичь безопасности ещё на уровне типов, то есть каким-то образом связать тип указателя с графом, которому вершина принадлежит.

Идея связать типы указателя и графа натолкнула на мысль об ST-монаде, определённой в стандартной библиотеке Haskell и в своей основе несущей похожую идею связывания типов. ST-монада зависит от дополнительного типа s, ещё называемого фантомным типом, который не используется и не ограничивается ни одной из функций, работающих с ST-монадой. Единственная функция, которая позволяет извлечь хранимое значение из ST-монады, с помощью обобщения типа s посредством квантора всеобщности $\forall s$ не позволяет возвращаемому значению иметь тип, зависящий от s. Таким образом, все типы, зависящие от s не могут выбраться за пределы ST-монады[5].

Данный подход, согласно известным мне данным, не используется в других модулях стандартной библиотеки и в других библиотеках, но ничто не мешает адаптировать подобную нетривиальную идею под свои нужды в библиотеке для работы с графами. Создадим монаду $\mathrm{Builder}\ s\ e\ v\ \alpha$, хранящую значение типа α и зависящую от типа меток вершин v и дуг e, а также от свободного типа s. Этим же типом s ограничим тип указателя $\mathrm{Vertex}\ s$, тем самым связав тип указателя с типом $\mathrm{Builder}$:

$$\textbf{newtype}\ \mathrm{Vertex}\ s = \mathrm{Vertex}\ \mathbb{N}$$

Скрыв конструктор указателя от пользователя, мы можем быть уверены, что пользователь не сможет создать самостоятельно какой-либо указатель на вершину, включая указатели на несуществующие вершины, а созданные внутри библиотеки указатели не смогут быть использованы при построении других графов, что гарантируется проверкой корректности типов. Неверное использование указателей попросту приведет к ошибке компиляции. Таким образом, библиотека обеспечивает безопасность указателей на вершины, что и являлось целью создания конструктора графа.

В итоге, пользователь сможет создавать вершины для одного графа только в одном окружении, в результате создания вершин получать безопасные указатели на них и уже после этого использовать их для создания дуг. Следовательно, необходимо предоставить соответствующие функции добавления вершины и

дуги. Однако необходимо также обеспечить сохранность уже созданных вершин и дуг. Для этого воспользуемся монадой State, обернув в неё состояние BState, зависящее от типа s и хранящую список вершин, дуг, а также количество созданных вершин, позволяющее за константу вычислить указатель на новую вершину. Саму монаду State обернём в уже представленную монаду Builder, которая позволит изменять состояние при добавлении вершины или дуги:

$$\textbf{data}\ \mathrm{BState}\ s\ e\ v = \mathrm{BState}\ \{\mathrm{count} : \mathbb{N},\ \mathrm{verts} : [v],\ \mathrm{edges} : [\mathrm{Edge}\ e]\}$$

$$\textbf{newtype}\ \mathrm{Builder}\ s\ e\ v\ \alpha = \mathrm{Builder}\ (\mathrm{State}\ (\mathrm{BState}\ s\ e\ v)\ \alpha)$$

Состояние конструктора хранит список дуг, поэтому определим тип дуги Edge e. Дуга определяется начальной и концевой вершинами, выраженными в виде указателей, а также меткой дуги:

$$\textbf{data}\ \mathrm{Edge}\ e = \mathrm{Edge}\ \{\mathrm{source} : \mathbb{N},\ \mathrm{target} : \mathbb{N},\ \mathrm{label} : e\}$$

Добавление вершины будет возможно только с помощью функции vertex, умеющей извлекать состояние из конструктора и менять его, а также возвращающей указатель на созданную вершину. Заметим, что тип Vertex s не используется в определении типа состояния BState, а содержится лишь в его объявлении, так как в этом нет необходимости. Более того, мы должны иметь возможность извлечь значения из состояния BState, что было бы невозможно, если бы они зависели от типа s. Функция добавления вершины, в свою очередь, возвращает объект типа Vertex s, чтобы ограничить пользователя в использовании указателя, сделав его безопасным.

Рассмотрим, как меняется состояние при добавлении вершины. Так как внутренние указатели на вершины являются всего лишь их порядковым номером в списке вершин, то указатель созданной вершины равняется предыдущему количеству вершин. Метка новой вершины добавляется в начало списка вершин, так как эта операция выполняется за константу. Самой функции достаточно принять лишь метку новой вершины в качестве аргумента:

```
vertex : v → Builder s e v (Vertex s)
vertex v = Builder (do
    s ← State.get
    State.put (s {count = (count s) + 1, verts = v :: (verts s)})
    pure (Vertex (count s)))
```

Добавление дуги также выполняется с помощью функции edge, меняющей состояние конструктора, а именно, добавлением нового объекта типа Edge e в начало списка дуг. Дуга содержит указатели на начальную и концевую вершины, поэтому функция должна принимать два объекта типа Vertex s, то есть защищённые указатели на ранее созданные вершины, а также метку дуги. Так как

в построении графа добавленная дуга больше участвовать не будет, то функция ничего не возвращает:

$$\begin{aligned}&\text{edge} : e \to \text{Vertex } s \to \text{Vertex } s \to \text{Builder } s\ e\ v\ ()\\&\text{edge } \ell\ (\text{Vertex } v_s)\ (\text{Vertex } v_t) = \text{Builder } (\textbf{do}\\&\quad \text{State.modify } (\lambda s.\ s\ \{\text{edges} = e :: (\text{edges } s)\}))\\&\ \ \textbf{where}\\&\quad e = \text{Edge } \{\text{source} = v_s,\ \text{target} = v_t,\ \text{label} = \ell\}\end{aligned}$$

После построения графа в состоянии конструктора хранятся список вершин и дуг, соответственно, необходимо реализовать функцию build', которая построит граф согласно внутреннему представлению графа:

$$\text{build'} : [v] \to [\text{Edge } s] \to \text{Graph } e\ v$$

Можно заметить, что тип функции build' совпадает с типом первого рассмотренного варианта конструктора, однако, во-первых, данная функция скрыта внутри библиотеки и недоступна пользователю, а во-вторых, при её выполнении невозможна ошибка из-за обращения по указателю на несуществующую вершину, так как списки вершин и дуг строятся внутри безопасного конструктора.

Теперь, когда у нас имеется состояние, хранящее созданные вершины и дуги, нам необходимо построить граф и избавиться от типа s с помощью квантора всеобщности $\forall s$. Так как тип $\text{Graph } e\ v$ не зависит от s, то он может являться возвращаемым типом функции, которая не позволяет защищённым типам покинуть монаду Builder. Роль такой функции будет выполнять функция build. В ней также необходимо задать начальное состояние конструктора с пустыми списками вершин и дуг. Заметим, что вершины добавлялись каждый раз в начало списка, поэтому относительно указателей на вершины список перевёрнут, и его необходимо снова развернуть, прежде чем передать в функцию build':

$$\begin{aligned}&\text{build} : (\forall s.\ \text{Builder } s\ e\ v\ \alpha) \to \text{Graph } e\ v\\&\text{build } (\text{Builder } b) = \text{build'}\ (\text{reverse } (\text{verts } s_f))\ (\text{edges } s_f)\\&\ \ \textbf{where}\\&\quad s_0 = \text{BState } \{\text{count} = 0,\ \text{verts} = [\,],\ \text{edges} = [\,]\}\\&\quad s_f = \text{State.execState } b\ s_0\end{aligned}$$

В качестве примера рассмотрим построение графа K_3:

```
build $ do
    v <- vertex "v"
    u <- vertex "u"
    w <- vertex "w"
    edge "vw" v w
    edge "uw" u w
    edge "uv" u v
```

Таким образом, разработанный конструктор достаточно гибок для построения графа, но в то же время не может быть ошибочно использован, то есть предлагает чистый с точки зрения функционального программирования интерфейс, что и было целью данной библиотеки. Реализация может быть найдена в модуле `Data.Graph.Abstract.Builder`.

2.4 Определения распространённых видов графов

Часто пользователю не нужно слишком общее определение графа, и ему необходимо работать с более узкими классами. По этой причине необходимо предоставить пользователю упрощённый интерфейс для работы с наиболее частыми видами графов.

В большой доле задач на графах нет необходимости помечать дуги, то есть граф может быть и невзвешенным. Для этого введём новый тип графа $\mathrm{Graph'}$, указав тип меток дуг как тип $(\,)$, при этом пользователь всё ещё должен указать тип меток вершин:

$$\textbf{type}\ \mathrm{Graph'} = \mathrm{Graph}\ ()$$

Аналогично введём функцию добавления непомеченной дуги $\mathrm{edge'}$, которая указывает тип метки дуги как тип $(\,)$:

$$\begin{aligned}&\mathrm{edge'} : \mathrm{Vertex}\ s \to \mathrm{Vertex}\ s \to \mathrm{Builder}\ s\ ()\ v\ ()\\&\mathrm{edge'} = \mathrm{edge}\ ()\end{aligned}$$

Некоторые виды невзвешенных графов имеют структуры, позволяющие предоставить безопасный тип функций-конструкторов этих графов, то есть без необходимости работать с указателями на вершины и с конструктором графов напрямую, передавая лишь метки вершин в каком-либо формате.

Примером такого класса графов является класс пустых графов O_n. Для однозначного определения пустого графа достаточно указать лишь список меток всех вершин. При создании графа с помощью конструктора просто вызовем функцию vertex для каждой метки вершины:

$$\begin{aligned}&\mathrm{isolated} : [v] \to \mathrm{Graph}\ e\ v\\&\mathrm{isolated}\ \ell s = \mathrm{build}\ (\mathrm{mapM\_}\ \mathrm{vertex}\ \ell s)\end{aligned}$$

Для создания графа порядка 0 достаточно вызвать функцию $\mathrm{isolated}$ с пустым списком меток, но так как такой граф довольно часто используется, например, в тестировании различных алгоритмов в качестве крайнего случая, то целесообразным будет выделить для него собственную функцию-конструктор empty, которая не добавляет ни вершин, ни дуг:

$$\begin{aligned}&\mathrm{empty} : \mathrm{Graph}\ e\ v\\&\mathrm{empty} = \mathrm{build}\ (\mathrm{pure}\ ())\end{aligned}$$

В тех же целях можно выделить в отдельную функцию singleton создание тривиального графа. Тривиальный граф однозначно определяется меткой единственной вершины. Так как такой граф является частным случаем пустого графа, а именно пустым графом порядка 1, то функция singleton может сводиться просто к вызову функции isolated с передачей в неё метки единственной вершины в виде списка из одного элемента:

$$\begin{aligned}&\text{singleton} : v \to \text{Graph}\ e\ v\\&\text{singleton}\ \ell = \text{isolated}\ [\ell]\end{aligned}$$

К распространённым видам графов также относятся графы-пути P_n. Если принять порядок переданных меток вершин за порядок вершин в пути, то такой граф также однозначно определяется лишь списком меток вершин. Будем считать, что дуги ориентированы в сторону вершины с большим индексом в списке. Для создания графа достаточно лишь вызвать функцию edge' для всех пар соседних вершин:

$$\begin{aligned}&\text{path} : [v] \to \text{Graph'}\ v\\&\text{path}\ [\,] = \text{empty}\\&\text{path}\ \ell s = \text{build}\ (\textbf{do}\\&\qquad vs \leftarrow \text{mapM vertex}\ \ell s\\&\qquad \text{mapM}\ (\text{uncurry edge'})\ (\text{zip}\ vs\ (\text{tail}\ vs)))\end{aligned}$$

Ещё одним из часто используемых классов графов являются графы-контуры C_n. Казалось бы, что такой контур можно задать последовательностью вершин, где первая и последняя вершины совпадают, но это приведет к построению пути с двумя вершинами с совпадающими метками, поэтому для контура нужен отдельный конструктор, который, впрочем, имеет такой же тип, как и конструктор для пути. По причине того, что граф ориентирован, будем считать, что дуги направлены в сторону большего индекса, за исключением дуги, соединяющей последнюю вершину с первой. Для создания дуг сместим список вершин на одну позицию и вызовем функцию edge' для всех пар вершин, лежащих на одной позиции. Также необходимо отдельно обработать случаи графов порядка 0 и 1, когда дуги создавать не нужно:

$$\begin{aligned}&\text{cycle} : [v] \to \text{Graph'}\ v\\&\text{cycle}\ [\,] = \text{empty}\\&\text{cycle}\ [\ell] = \text{singleton}\ \ell\\&\text{cycle}\ (\ell :: \ell s) = \text{build}\ (\textbf{do}\\&\qquad v \leftarrow \text{vertex}\ \ell\\&\qquad vs \leftarrow \text{mapM vertex}\ \ell s\\&\qquad \text{mapM}\ (\text{uncurry edge'})\ (\text{zip}\ ([v] +\!\!+ vs)\ (vs +\!\!+ [v])))\end{aligned}$$

Также упрощённый конструктор можно создать и для графа-турнира K_n. Если считать, что между каждой упорядоченной парой вершин есть дуга, то список меток вершин однозначно определяет такой граф. Для создания дуг графа достаточно вызвать функцию edge' для всех элементов декартова квадрата множества вершин, за исключением пар, лежащих на диагонали:

```
clique : [v] → Graph' v
clique ℓs = build (do
    vs ← mapM vertex ℓs
    let ivs = zip [1 ...] vs
    sequence_ [edge' v₁ v₂ | (i₁, v₁) ← ivs, (i₂, v₂) ← ivs, i₁ ≠ i₂])
```

Подобная похожая на математическую запись стала доступна благодаря разработанному конструктору графов и встроенным особенностям языка Haskell.

С помощью меток вершин однозначно можно определить и полные двудольные графы $K_{n,m}$. Для этого достаточно задать два списка меток вершин, соответствующие двум долям графа. Будем считать, что дуги направлены от первой доли ко второй, тогда для создания дуг необходимо вызвать функцию edge' для всех элементов декартова произведения множеств вершин долей:

```
biclique : [v] → [v] → Graph' v
biclique ℓs₁ ℓs₂ = build (do
    vs₁ ← mapM vertex ℓs₁
    vs₂ ← mapM vertex ℓs₂
    sequence_ [edge' v₁ v₂ | v₁ ← vs₁, v₂ ← vs₂])
```

Можно заметить, что если один из списков пуст, то функция biclique работает аналогично конструктору пустого графа.

Определим также функцию-конструктор star для графа-звезды. Будем считать, что все дуги направлены от центра к другим вершинам. В таком случае, звезда однозначно определяется меткой центральной вершины и списком меток остальных вершин и её можно рассматривать как частный случай полного двудольного графа $K_{1,n}$. Тогда функция star может быть выражена через функцию biclique:

```
star : v → [v] → Graph' v
star ℓ = biclique [ℓ]
```

Представленные конструкторы отдельных видов графов позволяют пользователям облегчать построение графов, избегая работы с общим конструктором, но и не жертвуя безопасностью, при этом построение всех представленных графов производится на линейное от числа вершин и дуг время.

Реализация представленных конструкторов использует внутри Builder и является хорошим примером того, насколько лаконичным и функциональным получается код построения графа с использованием общего конструктора. Функции расположены в модуле `Data.Graph.Abstract.Common`.

ГЛАВА 3

МОНАДА ГРАФА

3.1 Реализация экземпляров классов Functor, Applicative, Monad

Классы типов Functor, Applicative и Monad являются одним из важнейших механизмов в Haskell и, как известно, определены следующим образом[6]:

class Functor f **where**

$$(\langle\$\rangle) : (\alpha \to \beta) \to f\ \alpha \to f\ \beta$$
$$f\ \langle\$\rangle\ m_x = \text{pure}\ f\ \langle*\rangle\ m_x$$

class Functor $f \Rightarrow$ Applicative f **where**

$$\text{pure} : \alpha \to f\ \alpha$$

$$(\langle*\rangle) : f\ (\alpha \to \beta) \to f\ \alpha \to f\ \beta$$
$$m_f\ \langle*\rangle\ m_x = m_f \gg\!= (\lambda f.\ m_x \gg\!= \text{pure} \circ f)$$

class Applicative $f \Rightarrow$ Monad f **where**

$$\text{join} : f\ (f\ \alpha) \to f\ \alpha$$
$$\text{join}\ m_{m_x} = m_{m_x} \gg\!= \text{id}$$

$$(\gg\!=) : f\ \alpha \to (\alpha \to f\ \beta) \to f\ \beta$$
$$m_x \gg\!= f = \text{join}\ (f\ \langle\$\rangle\ m_x)$$

Как видно из циклических зависимостей функций друг от друга, чтобы какой-либо тип стал монадой, необходимо определить для него функцию pure, а также минимальный набор некоторых функций из fmap ($\langle\$\rangle$), ap ($\langle*\rangle$), bind ($\gg\!=$) и join, через которые можно было бы выразить оставшиеся функции.

Монада зависит лишь от одного типа, причём последнего указанного в определении, поэтому если тип графа Graph $e\ v$ является монадой, то это будет монада над типов меток вершин v. Попробуем определить недостающие функции для монады Graph e.

Посмотрим на тип функции pure для монады графа и заметим, что она ставит в соответствие метке одной вершины граф с указанным типом меток:

$$\mathrm{pure} : v \to \mathrm{Graph}\ e\ v$$

Если внимательно посмотреть на рассмотренные в предыдущей главе конструкторы классов графов, то можно увидеть функцию с совпадающим типом — конструктор тривиального графа $\mathrm{singleton}$. Эта реализация подходит и по типу и по смыслу: мы получаем метку вершины и возвращаем граф с единственной вершиной, обладающей указанной меткой.

Типы четырёх других функций для монады графа выглядят следующим образом:

$$(\langle\$\rangle) : (v_1 \to v_2) \to \mathrm{Graph}\ e\ v_1 \to \mathrm{Graph}\ e\ v_2$$
$$(\langle*\rangle) : \mathrm{Graph}\ e\ (v_1 \to v_2) \to \mathrm{Graph}\ e\ v_1 \to \mathrm{Graph}\ e\ v_2$$
$$\mathrm{join} : \mathrm{Graph}\ e\ (\mathrm{Graph}\ e\ v) \to \mathrm{Graph}\ e\ v$$
$$(\gg\!=) : \mathrm{Graph}\ e\ v_1 \to (v_1 \to \mathrm{Graph}\ e\ v_2) \to \mathrm{Graph}\ e\ v_2$$

Обычно тип функции практически полностью отражает суть того, что она делает. Однако если посмотреть на типы функций ap и bind, то они не являются такими простыми для понимания. Сложно сразу понять, как граф функций с одной структурой может воздействовать на граф с другой структурой. Поэтому обратим пока внимание на функции fmap и join.

Из типа fmap видно, что функция ставит в соответствие графу с метками вершин одного типа граф с метками вершин другого типа. Так как монада реализована над типом вершин, то логично предположить, что структура графа измениться не должна, а в этом случае по типу и по смыслу вполне подходит функция vmap, описанная в предыдущей главе.

Теперь посмотрим на функцию join. Получается, что она ставит в соответствие графу, каждая вершина которого является графом, обычный граф. При этом между графами также могут существовать дуги. Таким образом, задача функции join состоит в том, чтобы преобразовать «вложенный» граф в обычный. Можно было бы просто вынести вершины из нижнего уровня вложенности на верхний и получить граф с несколькими компонентами связности, однако в таком случае потеряются дуги между графами. Тогда можно рассматривать дугу между графами как множество дуг между всеми парами вершин из декартова произведения множеств вершин этих графов. В таком случае не составит труда добавить эти дуги между парами вершин из разных графов в соответствии с дугами между графами.

Таким образом, мы определили для типа $\mathrm{Graph}\ e\ v$ функции pure, fmap и join, через которые можно выразить bind и ap, значит граф является монадой над типом вершин.

Реализация может быть найдена в модуле `Graph.Data.Abstarct.Instance`. Там же находится реализация функций для классов типов $\mathrm{Foldable}$ и $\mathrm{Traversable}$

для $\mathrm{Graph}\ e\ v$, позволяющие строить различные свёртки относительно меток вершин графа.

3.2 Расширение монады списка

Самой распространённой монадой, которая может хранить несколько значений, является список. Суть этой монады представляется не в упорядоченности элементов, а в вариативности исходов, то есть список представляет собой множество возможных исходов, которые между собой равнозначны[3][7]. Можно ли тогда добиться такого же поведения от монады графа?

Граф отличается от списка тем, что вершины в нём не упорядочены, а также наличием дуг. Если первое для монады списка не играет никакой роли, и неупорядоченные вершины графа вполне соответствуют требованиям, то дуги можно просто удалить и рассматривать пустой граф.

Теперь последовательно рассмотрим функции класса монады. Начнём с функции pure (см. утверждение (1)). В левой части функция pure монады списка преобразовывает элемент x в список $[x]$, после чего к нему применяется функция $\mathrm{isolated}$, которая преобразовывает его в тривиальный граф. В правой части функция pure монады графа сразу преобразовывает элемент x в тривиальный граф с меткой вершины x. Таким образом, равенство выполняется.

$$\forall x : \alpha.\ \mathrm{isolated}\ (\mathrm{pure}\ x) = \mathrm{pure}\ x \tag{1}$$

Следующей рассмотрим функцию fmap (см. утверждение (2)). Нам даются функция отображения f из одного типа в другой и список значений xs. Мы можем вынести функцию fmap за конструктор $\mathrm{isolated}$, поскольку в первом случае сначала функция f применится ко всем элементам списка xs и затем из них будет построен пустой граф, а во втором случае сначала будет построен пустой граф с метками вершин xs и затем ко всем меткам вершин будет применена функция f. Значит, равенство выполняется.

$$\forall f : \alpha \to \beta,\ xs : [\alpha].\ \mathrm{isolated}\ (f\ \langle\$\rangle\ xs) = f\ \langle\$\rangle\ \mathrm{isolated}\ xs \tag{2}$$

Теперь рассмотрим функцию ap (см. утверждение (3)). Нам даются список функций fs, отображающих один тип во второй, и список значений xs. Функция ap монады списка в участке $f\ \langle*\rangle\ xs$ левой части равенства строит список результатов применения функции $f(x)$ для всевозможных пар (f, x) из декартова произведения fs и xs, а уже применение конструктора $\mathrm{isolated}$ строит пустой граф с соответствующими метками вершин. В правой части сначала строится пустой граф функций и пустой граф значений, а затем к ним применяется функция ap монады графа, которая строит пустой граф с множеством вершин, равным декартову произведению множеств функций и значений, в котором каждая пара сводится к результату применения функции к значению. Таким образом, в

обоих случаях строится один и тот же граф, что позволяет нам вынести функцию ap за конструктор. Значит, равенство выполняется.

$$\forall f : [\alpha \to \beta],\ xs : [\alpha].\ \mathrm{isolated}\ (f\ \langle * \rangle\ xs) = \mathrm{isolated}\ f\ \langle * \rangle\ \mathrm{isolated}\ xs \qquad (3)$$

И, наконец, рассмотрим функцию bind (см. утверждение (4)). Нам даются функция f, ставящая в соответствие значение первого типа в список значений второго типа, и список значений xs. В левой части равенства функция bind монады списка применяет функцию f к каждому элементу x из xs, а затем разворачивает вложенный список в одномерный список значений второго типа, после чего вызывает конструктор $\mathrm{isolated}$, который строит пустой граф по переданным значениям. В правой части равенства сначала строится пустой граф из значений xs, после чего функция bind монады графа применяет функцию f к каждой метке вершин графа. В результате получается граф, метками вершин которого являются списки значений второго типа. После функция bind применяет конструктор пустого графа к меткам вершин, получая пустой граф в качестве метки для каждой вершины, а затем применяет функцию join, которая убирает лишний уровень вложенности. Таким образом, равенство выполняется.

$$\forall f : \alpha \to [\beta],\ xs : [\alpha].\ \mathrm{isolated}\ (xs \gg\!= f) = \mathrm{isolated}\ xs \gg\!= (\mathrm{isolated} \circ f) \quad (4)$$

Так как функция join выражается через другие функции, то для неё аналогичное утверждение вытекает из утверждений для остальных функций. Таким образом, мы смогли выразить все функции монады списка с помощью функций монады графа над пустым графом, а это значит, что монада графа может восприниматься, как расширение монады списка, что является весьма интересным наблюдением.

ГЛАВА 4
МОНАДА ACCESSOR

Проектирование алгоритмов на графах в функциональном программировании вне сомнений является нетривиальной задачей, поскольку большинство алгоритмов было разработано с оглядкой на особенности императивных языков, и перенести их в функциональные языки не всегда возможно. Временами необходимо искать совершенно новый подход к давно известной и решённой задаче, чтобы добиться схожей асимптотики времени работы, однако и в таких случаях константа может быть слишком велика. К сожалению, к настоящему моменту существующие библиотеки для работы с графами в Haskell не предоставляют возможности в исходном виде перенести эффективные императивные алгоритмы в код на Haskell. В данной главе описывается предложенное решение проблемы.

4.1 Основные принципы

При разработке алгоритмов на графах в императивном программировании часто вершины и дуги рассматриваются как отдельные объекты, которые обрабатываются независимо, что достигается за счет использования указателей на вершины и дуги. Так как целью данной части работы является обеспечить возможность реализации императивных алгоритмов на графах в Haskell, то необходимо также предоставить возможность обращаться к вершинам по указателю.

Основная проблема наличия указателей на вершины и дуги разбиралась ещё в прошлых главах. Она заключается в том, что указатели несут потенциальную угрозу и при ошибочном их использовании могут вызвать ошибку времени исполнения, в то время как одним из принципов Haskell является гарантия безопасности кода, если его компиляция была завершена успешно. Это значит, что возникновение ошибочной ситуации в библиотеке должно быть невозможно.

Из вышеописанного следует, что разрабатываемый механизм должен поддерживать указатели на вершины и дуги, но при этом обязан обеспечивать их безопасность. Такая проблема уже встречалась ранее при разработке данной библиотеки, а именно при разработке механизма безопасного построения графа. Можно попробовать применить схожий подход и в данном случае.

Идея заключается в том, чтобы разрешить писать код в императивном стиле только внутри специально созданной для этого монады. При этом мы защитим потенциально опасные данные от их попадания за пределы монады с помощью фантомного типа s, от которого будет зависеть сама монада и все защищаемые типы данных.

Так как монада служит для доступа к информации о вершинах и дугах графа, то назовём её $\mathrm{Accessor}\ s\ e\ v\ \alpha$. Как и $\mathrm{Builder}$, она зависит не только от свободного типа s, но также и от типа меток вершин v и дуг e. Тип α является типом значения, хранимого в монаде.

Теперь мы можем определить типы указателей на вершины $\mathrm{Vertex}\ s$ и на дуги $\mathrm{Edge}\ s$. Чтобы обеспечить безопасность указателей, сделаем их также зависимыми от типа s и скроем от пользователя их конструкторы. Таким образом, указатели могут создаваться только внутри библиотеки, что обеспечивает их корректность, и не могут покинуть монаду $\mathrm{Accessor}$, что обеспечивает их безопасность. Так как внутренне граф представлен списком смежности, то указатель на вершину является обёрткой над целым числом, а указатель на дугу — обёрткой над вершиной и сдвигом в списке исходящих из неё дуг:

$$\textbf{newtype}\ \mathrm{Vertex}\ s = \mathrm{Vertex}\ \mathbb{N}$$
$$\textbf{data}\ \mathrm{Edge}\ s = \mathrm{Edge}\ \{\mathrm{source} : \mathrm{Vertex}\ s,\ \mathrm{offset} : \mathbb{N}\}$$

Несмотря на то, что внутри монады код будет выглядеть достаточно императивным, в конечном итоге он будет обернут в функцию в чистом функциональном стиле, на которую накладываются такие же ограничения, как и на любой другой функциональный код. Это означает, что такая функция не может изменять переданные ей в качестве аргументов объекты, а может только возвращать новые. В связи с этим, монада $\mathrm{Accessor}$ будет служить лишь для доступа к графу и построении на его основе новых объектов (включая новые графы), но никак не для модификации графа, что является большим преимуществом, так как позволит сохранить чистый функциональный интерфейс библиотеки.

Выбранный подход даёт ещё одно преимущество. Зависимость монады $\mathrm{Accessor}$ от типов меток вершин и дуг позволяет скрыть сам граф, для которого запущен алгоритм, то есть его присутствие в монаде будет неявным, что позволит писать достаточно обобщённый код и избавит от необходимости передавать граф в качестве аргумента, что сильно упростит интерфейс.

Теперь необходимо определить набор функций, которые обеспечат интерфейс, достаточный для полноценного переноса императивных алгоритмов в код на Haskell практически без потерь. Все функции из этого набора будут монадическими, то есть возвращаемое ими значение будет обёрнуто в монаду $\mathrm{Accessor}$ с целью защиты от использования вне разрешённого окружения.

Начнем с того, что мы пока запретили создавать пользователям свои указатели на вершины, но при этом не предоставили им возможность получить указатели, созданные внутри библиотеки. Функция $\mathrm{vertices}$ решает данную проблему и возвращает список указателей на все вершины графа:

$$\mathrm{vertices} : \mathrm{Accessor}\ s\ e\ v\ [\mathrm{Vertex}\ s]$$

Указатель на вершину сам по себе не несёт никакой ценности, если его нельзя нигде использовать. В императивных алгоритмах указатели на функции, в ос-

новном, используются для получения метки вершины, а также списка исходящих дуг. В таком случае, определим функции value и $\mathrm{outgoing}$, причем список исходящих дуг представим в виде списка указателей на дуги, что даст больше свободы действий при их обработке:

$$\begin{aligned}&\mathrm{value} : \mathrm{Vertex}\ s \to \mathrm{Accessor}\ s\ e\ v\ v\\&\mathrm{outgoing} : \mathrm{Vertex}\ s \to \mathrm{Accessor}\ s\ e\ v\ [\mathrm{Edge}\ s]\end{aligned}$$

Указатели на дуги пока можно получить лишь через вершины, которым они инцидентны, однако некоторые алгоритмы основаны на обработке именно дуг, а не вершин, поэтому естественным будет добавить функцию edges, которая вернула бы список указателей на все дуги графа. Эту функцию можно выразить через конкатенацию списков исходящих дуг всех вершин, используя только что определённые функции:

$$\begin{aligned}&\mathrm{edges} : \mathrm{Accessor}\ s\ e\ v\ [\mathrm{Edge}\ s]\\&\mathrm{edges} = \mathrm{pure} \circ \mathrm{concat} =\!\ll \mathrm{traverse}\ \mathrm{outgoing} =\!\ll \mathrm{vertices}\end{aligned}$$

Так же, как указатели на вершины, указатели на дуги являются невостребованными, если мы не можем с их помощью получить информацию о дугах, на которые они указывают. Дуга определяется начальной и концевой вершинами, а также меткой дуги. Начальную вершину мы можем получить с помощью уже определённой функции source в типе $\mathrm{Edge}\ s$. Определим функции label и target для доступа к метке дуги и к её концевой вершине:

$$\begin{aligned}&\mathrm{label} : \mathrm{Edge}\ s \to \mathrm{Accessor}\ s\ e\ v\ e\\&\mathrm{target} : \mathrm{Edge}\ s \to \mathrm{Accessor}\ s\ e\ v\ (\mathrm{Vertex}\ s)\end{aligned}$$

Так как теперь мы можем получить концевую вершину дуги по её указателю, мы можем определить функцию $\mathrm{successors}$, которая возвращает список прямых потомков вершины. Эта функция особенно полезна при работе с невзвешенными графами, когда получение дуги вместо вершины не даёт дополнительной информации, но вызывает неудобства в работе. Список прямых потомков легко выражается через список исходящих дуг вершины:

$$\begin{aligned}&\mathrm{successors} : \mathrm{Vertex}\ s \to \mathrm{Accessor}\ s\ e\ v\ [\mathrm{Vertex}\ s]\\&\mathrm{successors}\ v = \mathrm{traverse}\ \mathrm{target} =\!\ll \mathrm{outgoing}\ v\end{aligned}$$

Достаточно часто для алгоритма необходима определённая вершина, с которой нужно начинать обработку графа, или множество таких вершин. Все они обладают каким-то свойством, согласно которому принадлежат этому множеству. Для упрощения адаптации таких случаев под Haskell добавим функцию vfind, которая найдёт вершины, удовлетворяющие условию некоторого предиката. Эту функцию можно выразить через фильтрацию списка указателей на

вершины графа:

$$\begin{aligned}&\text{vfind} : (v \to \mathbb{B}) \to \text{Accessor}\ s\ e\ v\ [\text{Vertex}\ s]\\&\text{vfind}\ f = \text{filterM}\ (\text{liftM}\ f \circ \text{value}) =\!\ll \text{vertices}\end{aligned}$$

Аналогичная ситуация возникает и с поиском дуг, подходящих под некоторый критерий. Благодаря возможности получить список указателей на дуги, минуя работу со внутренним представлением графа, функцию efind можно тоже определить через фильтрацию простого списка:

$$\begin{aligned}&\text{efind} : (e \to \mathbb{B}) \to \text{Accessor}\ s\ e\ v\ [\text{Edge}\ s]\\&\text{efind}\ f = \text{filterM}\ (\text{liftM}\ f \circ \text{label}) =\!\ll \text{edges}\end{aligned}$$

Таким образом, монада Accessor позволяет реализовывать алгоритмы в императивном стиле, работающие на одном неявно заданном графе, при этом предоставляет весь необходимый интерфейс для работы с указателями на вершины и дуги, что делает возможным переносить существующие императивные алгоритмы на графах в практически неизменном стиле.

До этого было описано, какие возможности предоставляет монада Accessor, также было объяснено, каким образом она обеспечивает безопасность, однако ещё не было раскрыто, за счёт чего она позволяет скрыть граф и обращаться к нему неявно. Эта возможность реализована с помощью конструктора монады, который принимает в качестве аргумента функцию, переводящую граф в ST-действие:

$$\textbf{newtype}\ \text{Accessor}\ s\ e\ v\ \alpha = \text{Accessor}\ (\text{Graph}\ e\ v \to \text{ST}\ s\ \alpha)$$

Таким образом, все основные функции, описанные выше, возвращают обёрнутые в монаду Accessor функции, принимающие граф и возвращающие нужное значение, например, метку вершины. Откуда же тогда берётся сам граф, который передаётся во все обёрнутые в монаду функции? Напомним, что в конечном итоге для монады Accessor будет вызвана функция, извлекающая независящее от типа s значение. Именно эта функция будет принимать исходный граф, который будет передан во все обёрнутые в монаду функции:

$$\text{execute} : \text{Graph}\ e\ v \to (\forall s.\ \text{Accessor}\ s\ e\ v\ \alpha) \to \alpha$$

Также, если внимательно посмотреть на определение типа монады Accessor, то можно заметить, что данный тип может быть выражен как монада поверх композиции монад Reader и ST:

$$\textbf{newtype}\ \text{Accessor}\ s\ e\ v\ \alpha = \text{Accessor}\ (\text{ReaderT}\ (\text{Graph}\ e\ v)\ (\text{ST}\ s)\ \alpha)$$

Реализацию рассмотренных в данном разделе функций можно найти в модуле `Data.Graph.Abstract.Accessor`.

4.2 Разметка вершин и дуг

Во время разработки алгоритмов в императивном стиле нам часто приходится ассоциировать с вершинами и дугами некоторое изменяемое состояние, которое модифицируется в ходе алгоритма. Механизм $\mathrm{Accessor}$ также должен обеспечивать такую возможность.

Так как пользователь оперирует с указателями на вершины $\mathrm{Vertex}\ s$ и дуги $\mathrm{Edge}\ s$, которые не являются целочисленными типами, он не может использовать готовые стандартные контейнеры. По этой причине библиотеке необходимо предоставить свои контейнеры для хранения состояний, которые, к тому же, будут обеспечивать безопасность их использования за счёт механизма $\mathrm{Accessor}$. Введём тип $\mathrm{VArray}\ s\ \alpha$ для хранения состояния вершин и тип $\mathrm{EArray}\ s\ \alpha$ для хранения состояния дуг графа.

В целях безопасности существование контейнера с незаполненными начальными состояниями недопустимо, поэтому при создании контейнера необходимо его сразу заполнять. Одним из возможных решений является заполнение контейнера значением по умолчанию, которое задаёт сам пользователь согласно типу хранимого в контейнере состояния. Для этого определим функции varray и earray, которые будут создавать заполненные переданным значением контейнеры состояний вершин и дуг соответственно:

$$\mathrm{varray} : \alpha \to \mathrm{Accessor}\ s\ e\ v\ (\mathrm{VArray}\ s\ \alpha)$$
$$\mathrm{earray} : \alpha \to \mathrm{Accessor}\ s\ e\ v\ (\mathrm{EArray}\ s\ \alpha)$$

Однако не всегда существует значение по умолчанию, одинаковое для всех объектов, иногда оно может быть вычислено на основании знаний о вершине или дуге. Функции vbuild и ebuild позволят создавать контейнеры состояний, вычисленных с помощью переданной пользователем функции:

$$\mathrm{vbuild} : (\mathrm{Vertex}\ s \to \mathrm{Accessor}\ s\ e\ v\ \alpha) \to \mathrm{Accessor}\ s\ e\ v\ (\mathrm{VArray}\ s\ \alpha)$$
$$\mathrm{ebuild} : (\mathrm{Edge}\ s \to \mathrm{Accessor}\ s\ e\ v\ \alpha) \to \mathrm{Accessor}\ s\ e\ v\ (\mathrm{EArray}\ s\ \alpha)$$

Теперь мы можем выразить varray и earray через эти функции, преобразовав значение по умолчанию в константную функцию:

$$\mathrm{varray} = \mathrm{vbuild} \circ \mathrm{const} \circ \mathrm{pure}$$
$$\mathrm{earray} = \mathrm{ebuild} \circ \mathrm{const} \circ \mathrm{pure}$$

Данные контейнеры должны обеспечивать доступ к хранимым состояниям по указателю на вершину или дугу. Для этого определим функции vget и eget, возвращающие состояния вершины и дуги соответственно:

$$\mathrm{vget} : \mathrm{VArray}\ s\ \alpha \to \mathrm{Vertex}\ s \to \mathrm{Accessor}\ s\ e\ v\ \alpha$$
$$\mathrm{eget} : \mathrm{EArray}\ s\ \alpha \to \mathrm{Edge}\ s \to \mathrm{Accessor}\ s\ e\ v\ \alpha$$

Так как состояния могут меняться в ходе выполнения алгоритма, необходимо также определить функции для модификации состояний в контейнерах. Функции vset и eset будут заменять хранимое состояние вершины или дуги соответственно на новое переданное состояние:

$$\text{vset} : \text{VArray}\ s\ \alpha \to \text{Vertex}\ s \to \alpha \to \text{Accessor}\ s\ e\ v\ ()$$
$$\text{eset} : \text{EArray}\ s\ \alpha \to \text{Edge}\ s \to \alpha \to \text{Accessor}\ s\ e\ v\ ()$$

Часто состояния вершин или дуг, полученные в ходе выполнения алгоритма, и являются результатом его работы, который нужно передать пользователю за пределы монады Accessor. Как уже упоминалось, результатом выполнения чистой функции является новый объект, а в случае с графом этим новым объектом может быть и новый граф. Функции vgraph и egraph служат для создания нового графа, повторяющего структуру исходного графа, но метки вершин или дуг которого соответственно равны хранящимся в массиве состояниям:

$$\text{vgraph} : \text{VArray}\ s\ \alpha \to \text{Accessor}\ s\ e\ v\ (\text{Graph}\ e\ \alpha)$$
$$\text{egraph} : \text{EArray}\ s\ \alpha \to \text{Accessor}\ s\ e\ v\ (\text{Graph}\ \alpha\ v)$$

Результатом выполнения функции над графом может быть не только новый граф, но и любое другое значение, которое может быть вычислено на основании состояний вершин или дуг, например сумма или максимум значений по всем вершинам или дугам. Для упрощения реализации подобных вычислений служат функции vfold и efold, позволяющие выполнять свёртки контейнеров состояний:

$$\text{vfold} : (\alpha \to \beta \to \beta) \to \beta \to \text{VArray}\ s\ \alpha \to \text{Accessor}\ s\ e\ v\ \beta$$
$$\text{efold} : (\alpha \to \beta \to \beta) \to \beta \to \text{EArray}\ s\ \alpha \to \text{Accessor}\ s\ e\ v\ \beta$$

Описанные в данном разделе типы и функции над ними можно найти в модуле `Data.Graph.Abstract.Accessor`.

4.3 Встраивание ST-действий

Использование ST-монады в конструкторе монады Accessor позволяет оборачивать в монаду Accessor другие ST-действия. Так как извне конструктор монады Accessor недоступен, а также для удобства пользователей определим в самой библиотеке функцию liftST, позволяющую это сделать:

$$\text{liftST} : \text{ST}\ s\ \alpha \to \text{Accessor}\ s\ e\ v\ \alpha$$

Теперь для того, чтобы встроить какое-либо ST-действие в блок внутри монады Accessor, достаточно просто применить данную функцию к результату ST-действия.

Возможность встраивать ST-действия в блоки $\mathrm{Accessor}$ позволит пользователям использовать множество других библиотек, работающих с ST, включая различные изменяемые структуры данных и контейнеры, например $\mathrm{STVector}$, что значительно расширяет диапазон возможностей $\mathrm{Accessor}$.

Некоторые структуры данных очень часто используются в алгоритмах на графах, и потому можно заранее упростить работу с ними, реализовав простые обёртки поверх них. Ниже перечислены такие структуры.

• Ссылка. Тип $\mathrm{STRef}\ s\ \alpha$ обёрнут в тип $\mathrm{Ref}\ s\ \alpha$, который позволяет записывать, читать и инкрементировать хранимое значение типа α. Данная структура довольно часто используется при присваивании инкрементных меток вершинам, например в порядке обхода.

• Очередь. Тип $\mathrm{STQueue}\ s\ \alpha$ обёрнут в тип $\mathrm{Queue}\ s\ \alpha$, который реализует саморасширяющуюся очередь значений типа α на векторах. Эта структура данных важна, в первую очередь, для алгоритмов, основанных на поиске в ширину.

• Приоритетная очередь. Тип $\mathrm{STPriorityQueue}\ s\ \alpha$ обёрнут в тип $\mathrm{PQueue}\ s\ \alpha$, который реализует саморасширяющуюся приоритетную очередь значений типа α на куче с определённой пользователем функцией сравнения. Эта структура данных важна при обработке объектов в порядке их приоритетов, например, в алгоритме Дейкстры.

• Система непересекающихся множеств. Данная структура задана типом $\mathrm{Dsu}\ s$ и не является обёрткой поверх существующего ST-контейнера, а реализована поверх VArray с использованием эвристик сжатия путей и объединения по размеру.

Реализации перечисленных структур данных можно найти в соответствующих подмодулях модуля `Data.Graph.Abstract.Accessor`.

4.4 Реализация алгоритмов

В предыдущих разделах описывался интерфейс, предоставляемый пользователю при использовании монады $\mathrm{Accessor}$, а также его преимущества. Теперь давайте посмотрим, как выглядит реализация алгоритмов с помощью $\mathrm{Accessor}$ на практике.

В качестве примера возьмём вычисление расстояния от заданной начальной вершины до остальных вершин невзвешенного графа, реализованного с помощью $\mathrm{Accessor}$, и сравним с реализацией аналогичной функции на языке Python (см. листинг 1). Данный алгоритм реализовывается с помощью поиска в ширину, где каждой следующей обрабатываемой вершине присваивается метка, на единицу большая метки родителя этой вершины в дереве обхода.

Можно заметить, что код с использованием $\mathrm{Accessor}$ использует знакомые императивные паттерны и выглядит достаточно похожим на альтернативную реализацию на языке Python. Однако он также обладает преимуществами функционального подхода.

Основным из этих преимуществ является безопасность, обеспеченная си-

```haskell
distances s = do
    ds <- varray Nothing
    q <- Queue.new

    vset ds s (Just 0)
    Queue.push q (s, 0)

    whileM_ (not <$> Queue.empty q) $ do
        (v, dv) <- Queue.pop q
        succs <- successors v
        forM_ succs $ \u -> do
            visited <- isJust <$> vget ds u
            unless visited $ do
                let du = dv + 1
                vset ds u (Just du)
                Queue.push q (u, du)

    pure ds
```

```python
def distances(s, g):
    ds = [None] * len(g)
    q = Queue()

    ds[s] = 0
    q.push((s, 0))

    while not q.empty():
        (v, dv) = q.pop()

        for u in g[v]:
            visited = ds[u] is not None
            if not visited:
                du = dv + 1
                ds[u] = du
                q.push((u, du))

    return ds
```

Листинг 1 — Реализация функции `distances` на языке Haskell с использованием Accessor (слева) и на языке Python (справа)

стемой проверки типов Haskell. Например, в реализации на Python при создании списка `ds` нужно явно задавать длину, тогда как в Haskell граф передан неявно и массив нужной длины будет создан автоматически. Также реализация на Python сильно зависит от представления графа, полагаясь в том числе и на тот факт, что вершина имеет целочисленный тип `int`. Выходит, что можно, например, случайно перепутать порядок вершины и расстояния до неё при добавлении в очередь, и получить неочевидную ошибку, которую потом будет не так просто отыскать. В реализации же на Haskell такая ситуация попросту невозможна, так как указатель на вершину имеет свой собственный тип, и при подобной описке будет получена ошибка компиляции. Более того, реализация на Haskell никак не зависит от внутреннего представления графа и продолжит корректно работать даже после её изменения.

Ещё одним преимуществом является то, что пользователь может всё так же использовать чистый функциональный код внутри блоков Accessor, что позволяет, например, красиво и лаконично работать со списками, для чего в императивных языках придётся добавлять лишние циклы.

Разрабатываемая библиотека предлагает реализацию следующих алгоритмов с использованием монады Accessor:

- Алгоритмы, основанные на поиске в ширину:

– Обобщённый поиск в ширину. Алгоритм реализован в двух вариантах: с заданием начальных вершин и без. Пользователь может указать монадическую функцию вычисления новой метки вершины на основе информации о родителе. Среди существующих библиотек для работы с графами в Haskell поиск в глубину реализован лишь библиотекой Alga, однако время его работы значительно превышает линейное.

– Алгоритм вычисления расстояний невзвешенном графе с указанием на-

чальных вершин. В результате выполнения алгоритма каждой вершине ставится в соответствие расстояние до неё от ближайшей из начальных вершин, если вершина из них достижима.

– Алгоритм нахождения кратчайших путей в невзвешенном графе с указанием начальных вершин. В результате алгоритма каждой вершине ставится в соответствие вершина-родитель, если она существует.

• Алгоритмы, основанные на поиске в глубину:

– Обощённый поиск в глубину прямым обходом. Алгоритм реализован в двух вариантах: с заданием начальных вершин и без. Пользователь может указать монадическую функцию вычисления новой метки вершины на основе информации о родителе.

– Обощённый поиск в глубину обратным обходом. Алгоритм реализован в двух вариантах: с заданием начальных вершин и без. Пользователь может указать монадическую функцию вычисления новой метки вершины на основе информации о посещённых и непосещённых прямых потомках.

– Алгоритм построения леса обхода в глубину. В результате алгоритма каждой вершине ставится в соответствие вершина-родитель, если она существует.

– Алгоритм проверки графа на ацикличность. Данный алгоритм невозможно реализовать с помощью графов из стандартной библиотеки контейнеров, так как все обратные дуги удаляются в процессе построения дерева обхода.

– Алгоритм топологической сортировки. В результате выполнения алгоритма каждой вершине ставится в соответствие её номер в порядке топологической сортировки, если такой порядок существует. Данный алгоритм исправляет неверное поведение аналогичного алгоритма, реализованного в стандартной библиотеке, вследствие игнорирования наличия контуров в графе.

– Алгоритм определения компонент связности графа. В результате выполнения алгоритма вершинам ставится в соответствие номер компоненты связности. Данный алгоритм и три последующих разработаны для неориентированного графа, в котором рёбра представлены парой дуг.

– Алгоритм проверки графа на связность.

– Алгоритм раскраски графа в два цвета. В результате выполнения алгоритма каждой вершине ставится в соответствие её цвет. Гарантируется, что при двудольности графа раскраска будет верна.

– Алгоритм проверки графа на двудольность. Данный алгоритм сначала раскрашивает граф, а затем проверяет корректность раскраски.

• Алгоритмы вычисления расстояний во взвешенных графах:

– Алгоритм Дейкстры. В результате алгоритма вершинам ставится в соответствие расстояние от ближайшей из заданных начальных вершин.

– Алгоритм Беллмана-Форда. Данный алгоритм возвращает высчитанные расстояния только в случае отсутствия контуров отрицательного веса.

• Алгоритмы построения минимального остовного дерева:

– Алгоритм Краскала. Данный алгоритм также работает на неориентированном графе и возвращает список указателей на дуги, соответствующие рёб-

рам, которые входят в построенное остовное дерево или лес, если компонент связности несколько.

– Алгоритм Прима. В отличие от стандартной интерпретации данный алгоритм запускается на всех компонентах связности неориентированного графа с целью обеспечить то же поведение на несвязных графах, что и алгоритм Краскала.

• Алгоритмы нахождения максимального потока в сети:

– Алгоритм Эдмондса-Карпа. Так как изменение графа с целью добавления обратных дуг невозможно, остаточная сеть строится дополнительно, при этом для каждой дуги исходного графа добавляются две псевдо-дуги в остаточную сеть, которые связаны с исходной дугой. В результате выполнения алгоритма каждой дуге ставится в соответствие величина пропущенного по этой дуге потока.

– Алгоритм Диница. В отличие от императивной реализации, в данном алгоритме каждый раз при нахождении блокирующего потока остаточная сеть строится заново, чтобы обеспечить физическое удаление тупиковой дуги из сети.

Возможность реализации перечисленных выше алгоритмов с использованием монады Accessor доказывает его мощность и способность к адаптации любого императивного алгоритма к работе с данной библиотекой без потерь в читаемости кода и оценке трудоёмкости, что и являлось целью данной части работы.

Реализации алгоритмов могут быть найдены в соответствующих подмодулях модуля `Data.Graph.Abstract.Accessor.Algorithm`.

4.5 Accessor как низкоуровневая абстракция

Для каждого из алгоритмов, перечисленных в предыдущем разделе, также определена немонадическая обёртка, принимающая исходный граф и возвращающая новый граф, хранящий результат вычислений в виде меток вершин или дуг. Помимо этого, разобранные в предыдущей главе функции-трансформеры transformd и transformu также реализованы с помощью монады Accessor. Таким образом, можно считать, что Accessor представляет собой более низкоуровневую абстракцию, которая используется для реализации более высокоуровневых абстракций, которые предлагают чистый функциональный интерфейс. В свою очередь, пользователям крайне рекомендуется использовать именно высокоуровневые абстракции, чтобы получать в результате красивый идиоматический для языка Haskell код.

ГЛАВА 5

РЕШЕНИЕ ЗАДАЧ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИОНАЛЬНЫХ АБСТРАКЦИЙ

Рассмотрим пример решения задачи поиска максимального потока в сети с использованием лишь высокоуровневых чистых функциональных абстракций. Напомним, что такие функции преобразуют исходный граф в новый граф той же структуры. Соответственно, нашей целью является разбить задачу поиска максимального потока на несколько этапов преобразования графа, в результате которых мы получим граф с пропущенным по нему максимальным потоком.

По причине того, что функциональные преобразования графа не меняют его структуру, будем считать, что мы изначально работаем с остаточной сетью типа $\mathrm{Graph\ Edge}\ v$, где Edge — тип метки дуг, имеющий поля $\mathrm{capacity}$ (пропускная способность) и flow (пропущенный по дуге поток), а v — тип метки вершины. Назовём остаточную сеть G_f, где G — исходный граф. Также обозначим через $V(G)$ множество вершин G, а через $A(G)$ — множество его дуг. Тогда будем считать, что перед началом алгоритма выполнены следующие условия:

1. Множества вершин исходного графа и остаточной сети совпадают, то есть $V(G) = V(G_f)$.

2. Метки вершин остаточной сети уникальны, то есть $\forall v, u \in V(G_f) : \ v \neq u \Rightarrow \ell(v) \neq \ell(u)$, где ℓ — функция разметки вершин.

3. Пропускная способность дуг в исходном графе неотрицательна, то есть $\forall a \in A(G) : \ c(a) \geqslant 0$, где c — функция пропускной способности в исходном графе.

4. Каждой дуге исходного графа соответствует прямая и обратная дуги в остаточной сети, то есть $\forall a = (v, u) \in A(G),\ \exists a' = (v, u), a'' = (u, v) \in A(G_f) : c_f(a') = c(a),\ c_f(a'') = 0$, где c_f — функция пропускной способности в остаточной сети.

5. Для любой пары начальной и концевой вершин остаточной сети существует не более одной дуги, то есть $\forall a = (v, u), a' = (v', u') \in A(G_f) : \ a \neq a' \wedge v = v' \Rightarrow u \neq u'$.

6. Поток в остаточной сети не пропущен ни по одной дуге, то есть $\forall a \in A(G_f) : f(a) = 0$, где f — функция потока в сети.

Заметим, что данные условия не ограничивают использование алгоритма на графах с несколькими дугами с совпадающими начальной и концевой вершиной и с неуникальными метками вершин, так как такой граф может быть приведён к требуемому виду следующими преобразованиями:

1. Перенумеруем вершины ведением новой функции разметки ℓ', то есть $\forall v \in V(G) : \ \ell'(v) = (i(v), \ell(v))$, где $i(v) \in \{1, 2, \ldots, |V(G)|\},\ \forall v, u \in V(G) : v \neq u \Rightarrow i(v) \neq i(u)$.

2. Склеим дуги с совпадающими начальной и концевой вершинами, то есть построим такую новую функцию пропускной способности c' и такое новое множество дуг $A'(G) = \{a_{vu} \mid a_{vu} = (v,u), c'(a_{vu}) = \sum\limits_{a \in A_{vu}(G)} c(a), v,u \in V(G), \exists a = (v,u) \in A(G)\}$, где $A_{vu}(G) = \{a \mid a = (v,u) \in A(G)\}$.

3. Склеим пары дуг с совпадающими неупорядоченными парами начальной и концевой вершин, то есть построим такую новую функцию пропускной способности c' и такое новое множество дуг $A'(G) = A_s(G) \cup A_u(G)$, где $A_s(G) = \{a \mid a = (v,u) \in A(G),\ c'(a) = c(a),\ \nexists a' = (u,v) \in A(G)\}$ и $A_u(G) = \{a \mid a = (v,u),\ c'(a) = c(a') - c(a''),\ \exists a' = (v,u), a'' = (u,v) \in A(G),\ c(a') > c(a'')\}$.

Итак, будем считать, что у нас есть остаточная сеть, построенная по исходному графу, для которых выполняются все описанные выше условия, а также две выбранные вершины: источник и сток. При реализации алгоритма будем опираться на общий алгоритм Форда-Фалкерсона поиска максимального потока в сети[8], который состоит из следующих шагов:

1. Находим путь между источником и стоком, следуя только по дугам, чья остаточная пропускная способность положительна.

2. Если путь не найден, переходим к последнему шагу.

3. Среди всех дуг, входящих в найденный путь, находим дугу с наименьшей остаточной пропускной способностью, которая вычисляется как разность пропускной способности и уже пропущенной величины потока. Значение наименьшей пропускной способности этой дуги и будет величиной потока, который пропустим по найденному пути.

4. Для каждой дуги пути увеличиваем поток, пропущенный по данной дуге, на найденную величину, уменьшая поток по обратной дуге на ту же величину.

5. Переходим к первому шагу.

6. Завершаем алгоритм.

Для начала оговорим, что источник и сток заданы двумя предикатами типа $v \to \mathbb{B}$. Уникальность меток вершин гарантирует существование предиката, который будет выполнен лишь для одной вершины.

С целью лучшего понимания происходящих преобразований графа рассмотрим их также на примере одной итерации алгоритма, запущенного на графе, изображённом на рисунке 5.1. В данном графе вершины S и T являются источником и стоком соответственно, а метки на дугах записаны в формате «поток/пропускная способность». Будем считать, что в данный момент мы начинаем четвёртую итерацию, тогда как на предыдущих итерациях были найдены пути $S \to A \to D \to T$ с потоком 2, $S \to A \to C \to T$ с потоком 1 и $S \to B \to C \to T$ с потоком 1.

Первое преобразование графа должно помочь нам построить путь от источника к стоку. В данном случае нам потребуется использовать знания о вершинах и дугах, поэтому воспользуемся одним из трансформеров, запустив его из источника. Если мы будем идти по графу «снизу вверх», то сможем передавать

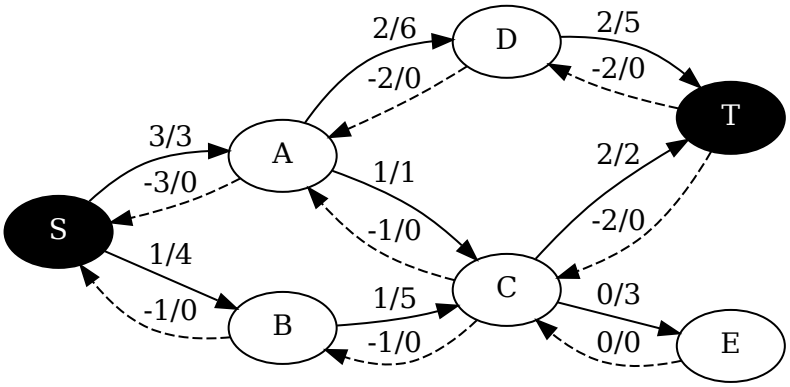

Рисунок 5.1 — Представление графа перед началом преобразований

родительской вершине информацию о том, как добраться до стока, и которая, в конечном счёте, будет доставлена источнику. При этом необходимо использовать трансформер с фильтрацией дуг по неотрицательной остаточной пропускной способности, так как переходить по заблокированным дугам нельзя. Выделим информацию, которую необходимо хранить в вершине, чтобы суметь восстановить путь из неё к стоку. Предполагая, что сыновья тоже хранят необходимую для восстановления пути информацию, в самой вершине достаточно хранить метку сына, от которого можно добраться до стока. Помня о том, что трансформер позволяет обойти не сам граф, а дерево, в котором между двумя достижимыми вершинами существует единственный путь, можно сделать вывод, что информация для восстановления пути будет доступна только вершинам, лежащим на этом пути. По этой причине новая метка должна быть обёрнута в монаду Maybe, чтобы суметь обозначить через значение $\mathrm{Nothing}$, что из какой-либо вершины путь к стоку не найден или же она вовсе недостижима из источника.

Вместе с построением пути можно заодно посчитать наибольшую величину потока, который можно пропустить по данному пути. Для этого воспользуемся утверждением, что эта величина для какой-либо вершины, лежащей на пути и не являющейся стоком, будет равна минимуму из остаточной пропускной способности дуги, ведущей к сыну, лежащему на пути, и наибольшей величины потока, который можно пропустить по пути от сына до стока.

Таким образом, в результате описанного преобразования источник в качестве метки будет хранить наибольшую величину потока и метку сына, лежащего на пути к стоку, если этот путь существует. Реализуем данное преобразование в функции `findPath` (см. листинг 2).

Результат выполнения преобразования `findPath` показан на рисунке 5.2. Чёрным выделены дуги, которые участвовали в преобразовании. Можно заметить, что заблокированные (с остаточной пропускной способностью 0), а также многие обратные дуги не принадлежат построенному дереву обхода, зато по дуге $C \to A$ можно пройти за счёт заблокированной исходной дуги $A \to C$. Также можно увидеть, что по пути от вершины A до стока T ещё можно пропустить

```haskell
findPath :: (v -> Bool) -> (v -> Bool)
         -> Graph Edge v -> Graph Edge (Maybe (Int, v), v)
findPath isSource isSink =
    transformu' isSource ((> 0) . available) backtrack (Nothing,)
  where
    backtrack v [] = (Nothing, v)
    backtrack v ((e, (mc, u)):chs) = case mc of
        Nothing | isSink u -> (Just (available e, u), v)
        Nothing | otherwise -> backtrack v chs
        Just (c, _) -> (Just (min c (available e), u), v)
```

Листинг 2 — Реализация функции `findPath`

поток величины 3, в то время как по пути от источника к стоку — уже только 1. Помимо этого, можно обратить внимание на вершину E. Она является достижимой из источника, потому участвует в преобразовании, однако сток из неё недостижим, и потому эта вершина хранит метку Nothing.

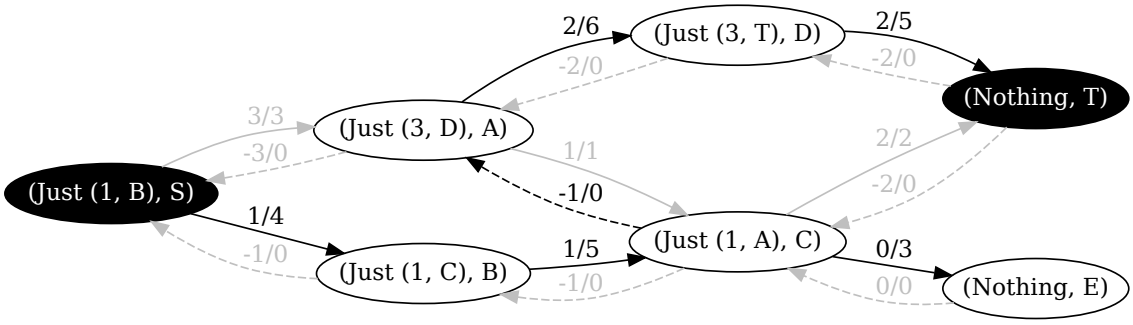

Рисунок 5.2 — Представление графа после преобразования `findPath`

Однако после преобразования `findPath` величина потока для найденного пути известна только источнику, поэтому необходимо передать эту информацию всем вершинам на пути к стоку. Для этого снова воспользуемся трансформером, но теперь будем обходить граф «сверху вниз», передавая данные от родителей к сыновьям. Таким образом, если от вершины добраться до стока невозможно, то метка измениться не должна. В противном случае вершина наследует метку своего родителя. Описанное преобразование реализуем в функции `propagatePath` (см. листинг 3).

Результат выполнения преобразования `propagatePath` показан на рисунке 5.3. Можно заметить, что теперь все вершины на пути к стоку хранят величину потока 1.

Теперь мы построили путь при условии его существования. Однако мы всё ещё не знаем, существует он или нет, эта информация скрывается среди меток вершин. Представим, что путь между источником и стоком не найден. Это означает, что сток недостижим из источника. Функции-трансформеры гарантируют, что в случае недостижимости вершины ей будет присвоена метка по умолчанию, которая в данном случае равна Nothing. Все недостижимые вер-

```haskell
propagatePath :: (v -> Bool) -> Graph Edge (Maybe (Int, v), v)
              -> Graph Edge (Maybe (Int, v), v)
propagatePath isSource =
    transformd' (isSource . snd) ((> 0) . available) descend id
  where
    descend [] pv = pv
    descend _ (Nothing, v) = (Nothing, v)
    descend (((Nothing, _), _):preds) pv = descend preds pv
    descend (((Just (c, _), _), _):preds) (Just (_, u), v) = (Just (c, u), v)
```

Листинг 3 — Реализация функции `propagatePath`

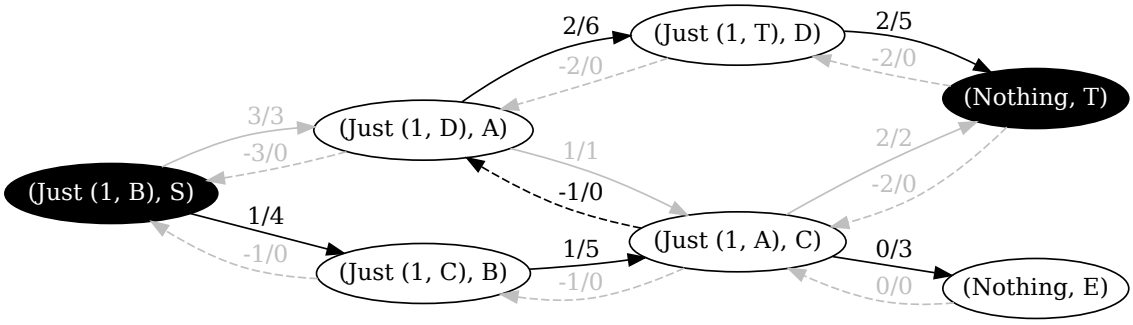

Рисунок 5.3 — Представление графа после преобразования `propagatePath`

шины обрабатываются независимо, без рассмотрения их окружения. Из этого следует, что при отсутствии пути между источником и стоком все вершины будут иметь метку Nothing, так как ни одна из них не будет родителем стока в дереве обхода. Обратное утверждение также справедливо: если метки всех вершин равны Nothing, то не существует вершины, из которой сток достижим, значит, он недостижим и из источника, что говорит об отсутствии пути между ними. Значит, путь между источником и стоком существует, если существует хотя бы одна вершина, метка которой не равна Nothing.

Для вычисления ответа воспользуемся свёрткой or, которую допустимо применять к графу, поскольку он реализует Foldable. Данная свёртка возвращает True, если хотя бы один обработанный объект равен True. Для отображения меток вершин в булево множество воспользуемся функцией отображения vmap и предикатом, возвращающим False, если значение равно Nothing, и True в противном случае. Реализуем данную проверку в функции `reachable` (см. листинг 4).

Теперь, когда мы знаем, что путь от источника к стоку действительно найден, и каждая вершина, лежащая на пути, знает величину потока, которую нуж-

```haskell
reachable :: Graph Edge (Maybe (Int, v), v) -> Bool
reachable = or . vmap (isJust . fst)
```

Листинг 4 — Реализация функции `reachable`

```haskell
pushPath :: (Eq v) => Graph Edge (Maybe (Int, v), v)
         -> Graph Edge (Maybe (Int, v), v)
pushPath = emapc reflow
  where
    reflow (Nothing, _) e (Nothing, _) = e
    reflow (Just (c, u), v) e (Nothing, v')
        | u == v' = push c e
        | otherwise = e
    reflow (Nothing, v) e (Just (c', u'), v')
        | u' == v = push (-c') e
        | otherwise = e
    reflow (Just (c, u), v) e (Just (c', u'), v')
        | u == v' = push c e
        | u' == v = push (-c') e
        | otherwise = e
```

Листинг 5 — Реализация функции `pushPath`

но пропустить по дуге к сыну, тоже лежащему на пути, мы можем пропустить поток по найденному пути. Так как при этом придётся изменять метки дуг сети, основываясь на метках инцидентных дугам вершин, нам потребуется отображение с контекстом emapc.

Тогда осталось понять, в каких случаях нужно пропустить поток по дуге. Во-первых, когда обе вершины лежат на пути и метка сына начальной вершины совпадает с меткой концевой вершины дуги. Во-вторых, когда обе вершины лежат на пути и метка начальной вершины совпадает с меткой сына концевой вершины. Во втором случае мы попали на дугу, обратную дуге, входящей в найденный путь, и это значит, что по ней нужно пропустить обратный поток. Реализуем описанное преобразование в функции `pushPath` (см. листинг 5). В реализации рассмотрено на два случая больше, так как вершина-сток не имеет сына, и этот случай необходимо обрабатывать отдельно.

Результат выполнения преобразования `pushPath` показан на рисунке 5.4. Чёрным выделены дуги, которые изменили свои метки в ходе данного преобразования. Можно увидеть, что к таким дугам относятся только дуги, лежащие на найденном пути от источника к стоку, а также обратные им дуги.

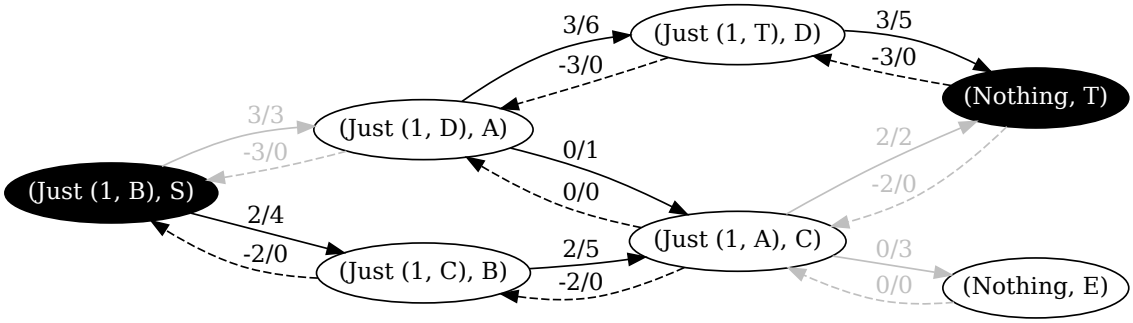

Рисунок 5.4 — Представление графа после преобразования `pushPath`

```haskell
unmark :: Graph Edge (Maybe (Int, v), v) -> Graph Edge v
unmark = vmap snd
```

Листинг 6 — Реализация функции `unmark`

```haskell
maxFlow :: (Eq v) => (v -> Bool) -> (v -> Bool) -> Graph Edge v -> Graph Edge v
maxFlow isSource isSink g
    | reachable g' = maxFlow isSource isSink . unmark . pushPath $ g'
    | otherwise = unmark g'
  where
    g' = propagatePath isSource . findPath isSource isSink $ g
```

Листинг 7 — Реализация функции `maxFlow`

После того, как мы пропустили поток по найденному пути, нам нужно вернуть исходные метки вершин. Для этого воспользуемся простым отображением меток вершин vmap, оставив только вторую часть метки, которая и является исходной меткой вершины. Реализуем это преобразование в функции `unmark` (см. листинг 6).

Результат выполнения преобразования `unmark` показан на рисунке 5.5. Можно заметить, что в итоге мы получили граф в том же виде, как и до преобразований, но с изменёнными метками дуг согласно пропущенному по ним потоку.

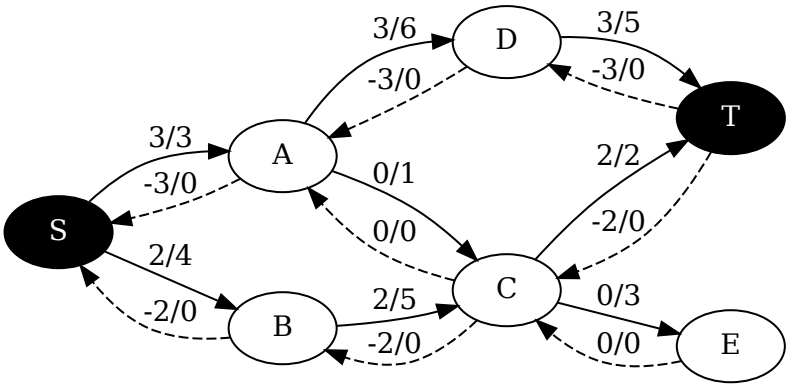

Рисунок 5.5 — Представление графа после преобразования `unmark`

Объединим все преобразования в функцию `maxFlow` (см. листинг 7). В полученной сети пытаемся построить путь от источника к стоку с помощью функций `findPath` и `propagatePath`, затем с помощью функции `reachable` проверяем существование пути. Если путь существует, то пропускаем по нему поток функцией `pushPath`, очищаем метки вершин функцией `unmark` и снова вызываем для полученного графа функцию `maxFlow`. Если путь не был найден, то максимальный поток уже построен, поэтому очищаем метки вершин функцией `unmark` и возвращаем полученный граф.

Так как функции-трансформеры гарантируют обход вершин согласно порядку поиска в ширину, можно заметить, что каждый раз мы находим кратчай-

ший путь от источника к стоку. Это означает, что данное решение реализует алгоритм Эдмондса-Карпа, и для него справедливы все оценки времени работы, что и для императивных реализаций.

Таким образом, используя лишь определённые в библиотеке высокоуровневые чистые функциональные операции над графом можно реализовать достаточно сложные алгоритмы, такие как поиск максимального потока в сети, которые не отстают по трудоёмкости от их императивных аналогов, но при этом код выглядит красивым и идиоматическим с точки зрения языка Haskell, а также использует лишь функциональные паттерны, такие как сопоставление по образцу, стражи, свёртки и различные комбинаторы, позволяя взглянуть на знакомую задачу с другой стороны.

ГЛАВА 6
ТЕСТИРОВАНИЕ

6.1 Модульное тестирование

В целях проверки корректности работы, библиотека была подвержена модульному тестированию. Всего разработано 265 тестов, покрывающих все экспортируемые функции библиотеки.

6.2 Тестирование производительности

Разработанную библиотеку можно подключить к системе тестирования производительности[2], которая сравнивает среднее время работы библиотек языка Haskell на различных операциях над графами и уже поддерживает библиотеки Alga, FGL и контейнеры. Для встраивания собственной библиотеки достаточно реализовать создание графа по заданному образцу, а также определить адаптеры к тестируемым операциям. К таким операциям относятся:

- создание графа (`creation`);
- получение количества вершин графа (`vertexCount`);
- получение списка вершин графа (`vertexList`);
- получение списка рёбер графа (`edgeList`);
- алгоритм поиска в ширину (`bfs`);
- построение леса обхода в глубину (`dff`);
- топологическая сортировка графа (`topSort`);
- определение достижимости вершин в графе из заданной (`reachable`);
- транспонирование графа (`transpose`).

Разработанная библиотека была протестирована в сравнении с другими на различных графах порядка 1000:

- на графах-сетках (см. рисунок 6.1);
- на графах-турнирах (см. рисунок 6.2);
- на реалистичных графах (см. рисунок 6.3).

Результаты тестирования показали, что данная библиотека не уступает другим при выполнении перечисленных операций, однако следует понимать, что проведённые тесты не полностью отражают ситуацию. Система тестирования предлагает лишь ограниченный набор операций, который не включает сложных нелинейных алгоритмов, на которых разработанная библиотека значительно превосходит конкурентов, однако включает операции (такие как `dff` и `topSort`), под которые была заточена лидирующая на них библиотека контейнеров, не позволяющая реализовывать более никакие другие алгоритмы. Если же сравнивать лишь с библиотеками общего назначения (Alga и FGL), то разработанная библиотека показывает лучшие результаты практически на всех тестах.

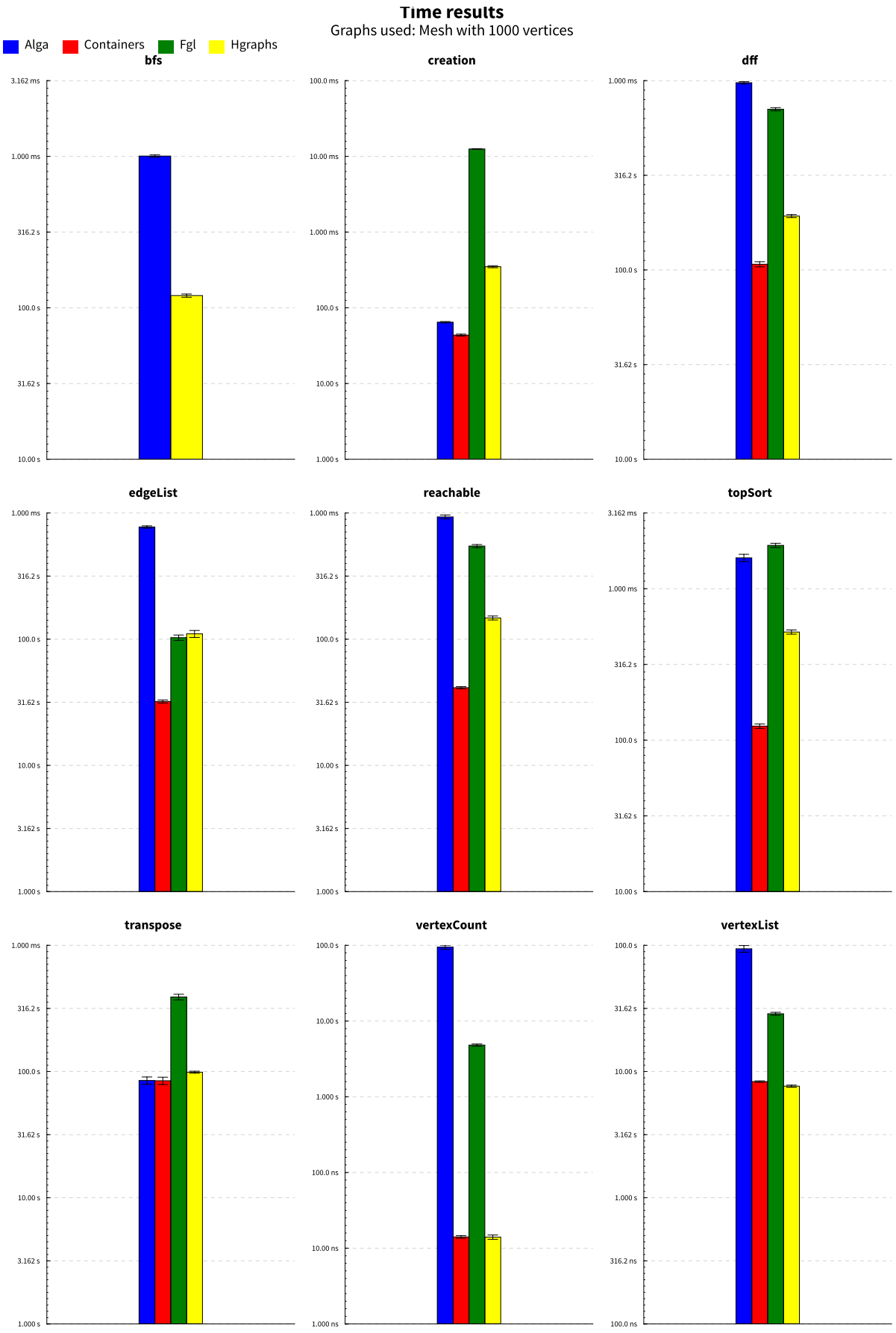

Рисунок 6.1 — Результаты тестирования на графе-сетке (разработанная библиотека показана жёлтым)

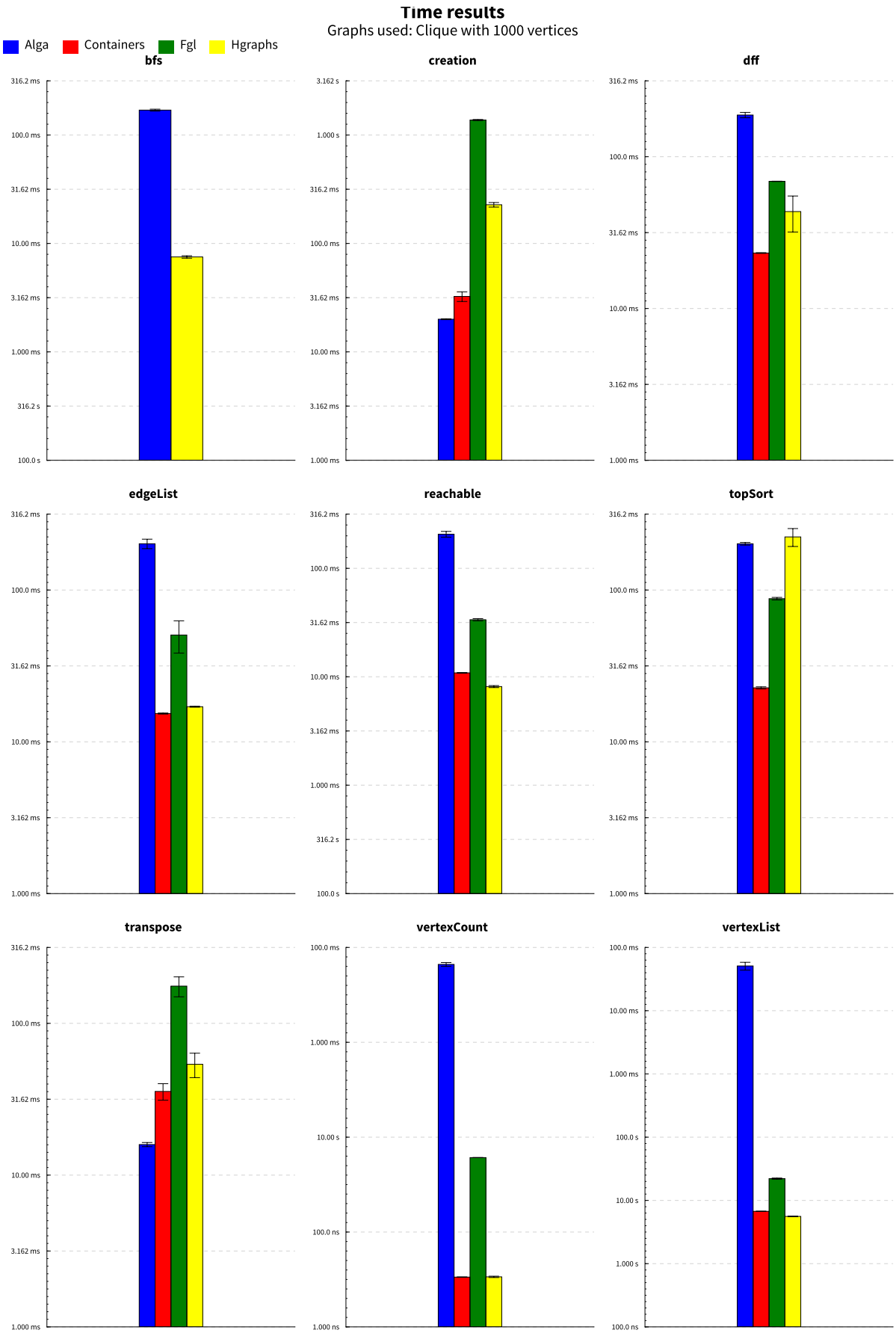

Рисунок 6.2 — Результаты тестирования на графе-турнире (разработанная библиотека показана жёлтым)

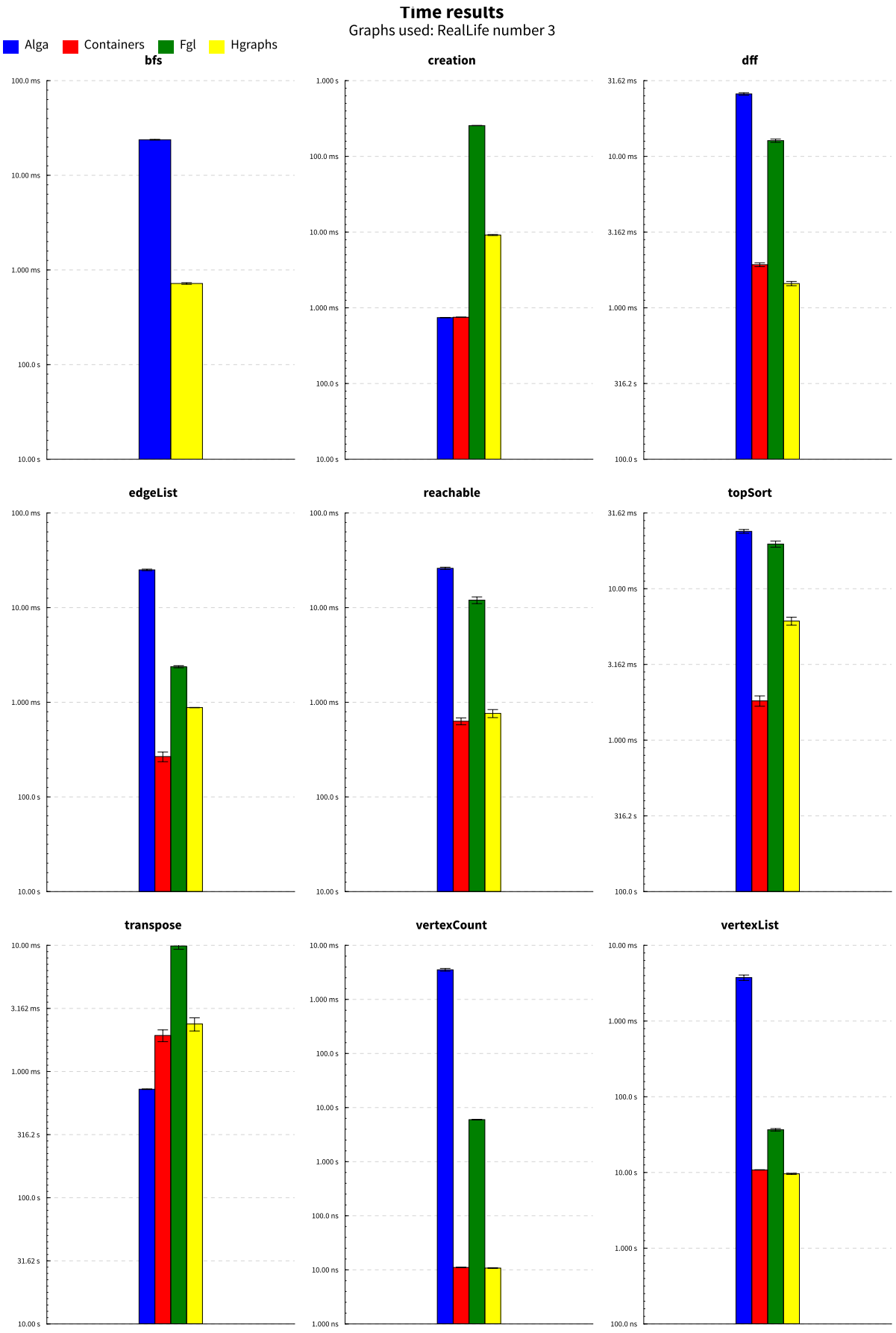

Рисунок 6.3 — Результаты тестирования на реалистичном графе (разработанная библиотека показана жёлтым)

ЗАКЛЮЧЕНИЕ

В рамках работы были исследованы недостатки функционального программирования в контексте проблемы представления графов, а также были изучены предлагаемые варианты библиотек для работы с графами в Haskell, были проанализированы их преимущества и недостатки и был разработан список требований к новой библиотеке, решающих эти недостатки.

После проведения исследовательской работы была начата разработка новой библиотеки для работы с графами, в ходе которой было предложено внутреннее представление графа, был разработан механизм безопасного построения графов. Помимо этого, был выделен набор базовых операций, позволяющий реализовывать алгоритмы на графах в чистом функциональном стиле и предложены упрощённые конструкторы для распространённых видов графов.

Также в процессе разработки были обнаружены интересные с точки зрения функционального программирования свойства типа графа. В частности, был представлен экземпляр монады для типа графов и проведено аналитическое сравнение с монадой списка.

Помимо этого, был спроектирован интерфейс, позволяющий разрабатывать безопасные с точки зрения типов, но эффективные алгоритмы на графах в императивном стиле, и который был использован в качестве низкоуровневой реализации высокоуровневых функциональных абстракций. В целях демонстрации работоспособности данного интерфейса, на его основе было реализовано множество известных алгоритмов на графах с достижением оценки времени работы, сравнимой с результатами императивных реализаций.

В качестве обобщения было предложен нетривиальный идиоматический подход к решению задачи о максимальном потоке с помощью разработанной библиотеки, демонстрирующий возможность использования чистых функциональных паттернов при описании алгоритмов.

В рамках проверки корректности работы разработанная библиотека была покрыта модульными тестами, а также подвержена тестированию производительности в целях сравнения с существующими библиотеками языка Haskell, где показала свою конкурентоспособность.

Таким образом, поставленные в начале диссертации цели были достигнуты.

ПРИЛОЖЕНИЕ А. РЕПОЗИТОРИЙ С РЕАЛИЗАЦИЕЙ БИБЛИОТЕКИ

Репозиторий в системе GitHub с реализацией разработанной библиотеки доступен по ссылке: `https://github.com/larandaA/hgraphs`.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Mokhov, A. Algebraic Graphs with Class (Functional Pearl) / A. Mokhov // Haskell 2017: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. — 2017. — P. 2–13.

2. Moine, A. Benchmark suite for graph libraries // A. Moine // GitHub [Electronic resource]. — 2018. Mode of access : https://github.com/haskell-perf/graphs. — Date of access : 24.05.2020.

3. Wadler, P. Comprehending Monads / P. Wadler // LFP ’90: Proceedings of the 1990 ACM conference on LISP and functional programming. — 1990. – P. 61–78.

4. Erwig, M. Inductive Graphs and Functional Graph Algorithms / M. Erwig // Journal of Functional Programming. — Vol. 11, No. 5. — 2001.

5. Launchbury, J. Lazy Functional State Threads / J. Launchbury, S. Peyton Jones // PLDI ’94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation. — 1994. – P. 24–35.

6. Lipovaca, M. Learn You a Haskell fo Great Good / M. Lipovaca. — San Francisco : No Starch Press, 2011. — 400 p.

7. Wadler, P. Monads for functional programming / P. Wadler // Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text. — 1995. – P. 24–52.

8. Williamson, David P. Network Flow Algorithms / David P. Williamson. — New York : Cambridge University Press, 2019. — 326 p.

9. Bird, R. On building cyclic and shared structures in Haskell / R. Bird // Form Asp Comp. — No. 24. — 2012. – P. 609–621.

10. Bird, R. Pearls of Functional Algorithm Design / R. Bird. — New York : Cambridge University Press, 2010. — 277 p.

11. Okasaki, C. Purely Functional Data Structures / C. Okasaki. — New York : Cambridge University Press, 1998. — 220 p.

12. King, David J. Structuring Depth-First Search Algorithms in Haskell / David J. King, J. Launchbury // POPL ’95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1995. – P. 344–354.

13. Ben-Amram, A. What is a ”Pointer Machine”? / A. Ben-Amram // ACM SIGACT News. — Vol. 26, No. 2. — 1995.