Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Система для автоматического описания способа получения файлов в проекте по действиям пользователя в консоли

Серока А.В.
Научный руководитель: Сергушичев А. А.

Санкт-Петербург
2015

ОГЛАВЛЕНИЕ

Стр.

ВВЕДЕНИЕ

В наше время очень часто приходится заниматься обработкой информации. Требуется найти правильные пути превращения из одного состояния в другое. Для преобразования полученной информации на компьютере часто используются не графические приложения, а наборы небольших утилит, запускаемых из консоли, каждая из которых делает свою отдельную задачу.

Сейчас не так часто обычные пользователи используют для работы командную строку. Но существует группа людей, которая это делает очень часто или всегда. Например программисты и некоторые ученые. Очень быстро развивается биоинформатический анализ. Появляется множество программ для обработки результатов, полученных из изучения цепочек ДНК. Данные программы запускаются друг за другом в консоли, получаются результаты, обрабатываются. Если человек получил удовлетворительный результат и хочет его повторить на другом наборе данных, то ему будет сложно воспроизводить предыдущие шаги. А если в какой-то момент какой-то шаг был не нужен, а человек уже не помнит об этом?

Рассмотрим пример обычной работы для обработки данных. Человек, получив данные, дает их на вход первой программе и получает результат. Затем этот результат идет другой программе и так далее. Заканчивается этот процесс по достижению необходимого результата. Но ведь никто не застрахован от ошибок. Если на каком-то этапе человек сделает ошибку, то ему будет необходимо заново воссоздавать информацию (при условии, что она не была отдельно сохранена). Чаще всего о некорректности результата узнается на последнем этапе, когда получается результат. Но, предположив, что финальное состояние удовлетворительно, и хочется сделать тоже самое с другими входными, ему надо будет отделить все предыдущие **правильные** шаги от побочных, которые были получены в результате отработки анализа. С этим возникают трудности.

Таким образом, было бы удобно иметь инструмент, который позволяет по действиям человека в консоли воспроизводить необходимую цепочку действий, которые привели к финальному результату. Также, хотелось бы иметь возможность отмены действий, просмотра зависимостей одной команды от

другой, чтобы убрать ненужные действия. А также было бы очень практично иметь возможность воспроизводить данную цепочку событий для получения нужной информации и на другом компьютере.

В работе были проанализированы существующие варианты решения данной задачи и рассмотрены несколько случаев использования. Для решения задачи потребовалось использовать сторонние библиотеки и программы для реализации, уделено внимание их возможностям, положительным сторонам и отрицательным.

Потребовалось исследовать способы получения информации о действии пользователя в консоли, возможные варианты дополнения их своими.

Результатом проделанной работы стал программный комплекс **gshell**, обеспечивающий человеку однозначное представление последовательности действий, для получения результата его работы в консоли. Также были разработаны вспомогательные инструменты, упрощающие и расширяющие возможности работы человека в командной строке.

В первой главе рассматриваются основные понятия, используемые в работе, основные технологии, а так же предметная область, в которой будет применяться разработанный программный комплекс.

Во второй главе рассматриваются цели, которые требовалось достичь, современные альтернативные способы решения данной задачи.

В третьей главе рассматривается реализованное программное обеспечение, его особенности, реализация, а также трудности, возникшие во время разработки.

ГЛАВА 1

ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В наше время существует несколько наиболее популярных вариантов операционных систем:

- *Microsoft Windows*
- Операционные системы на базе ядра *Linux*
- *Mac OS X*

Все эти три системы предоставляют человеку возможность работы с компьютером посредством графического интерфейса. Но так было не всегда, изначально надо было работать через интерактивный текстовый режим — консоль.

Но и в наше время многие люди работают в этом режиме. Он позволяет более гибко управлять системой, минимизировать действия для получения результата, получать необходимую информацию в “сухом” виде.

Работа в консоли наиболее популярна на ОС семейства Linux. Существуют различные оболочки для работы с системой в этом режиме. Наиболее популярные это:

- *Bash*
- *zsh*
- *fish*
- *tcsh*

Bash — наиболее популярная командная оболочка. *Fish* и *zsh* являются более продвинутыми оболочками и все чаще используются вместо *bash*.

Для унификации работы с данными непосредственно на устройствах для их хранения, используются файловые системы.

В системах семейства Linux монтирование файловых систем происходит в поддиректорию дерева всей файловой системы. Таким образом для человека точка монтирования будет являться просто папкой, в которой можно смотреть, редактировать, удалять содержимое.

Файловые системы, которые используются непосредственно на разделах устройств, могут быть примонтированы, как это сказано выше в виде папок. Таким образом программы могут не задумываться о том, что есть суть директории, им важен только интерфейс работы с ней. То есть это позволяет даже не знать о том, что папка — это точка монтирования. Так же существу-

ют файловые системы, которые можно использовать уже поверх других ФС. Этой возможностью пользуется файловая система *unionfs*, которая позволяется объединять несколько директорий в одну и предоставлять возможность ее монтирования и дальнейшей работы.

1.1 Операционные системы на базе ядра Linux

Операционные системы, построенные на базе ядра *Linux*[1], используют консоль и консольные приложения больше других ОС (напр. Windows). Данное семейство операционных систем предоставляет разработчику огромное количество возможностей для работы с файловыми структурами, системами. Так же они позволяют в пользовательском окружении создавать собственные точки монтирования через библиотеку *FUSE*. Эти возможности являются ключевыми в реализации данного проекта.

Если человек примонтировал другую файловую систему в некий каталог в Linux, то он будет видеть его содержимое так же, как и содержимое других папок. Таким образом будет поддерживаться древовидная структура каталога системы.

1.2 UnionFS-fuse

Программа *unionfs-fuse*[2] — является альтернативной реализацией *unionfs*[3] из ядра Linux, с возможностью использования в пользовательском окружении.

Данная программа позволяет объединять множество файлов и каталогов прозрачно от (***от?***) пользователя, предоставляя единую точку монтирования, в которой будет видно объединение вышеуказанных директорий. Так же имеются функции монтирования каталогов только для чтения или для чтения-записи. Это позволяет объединять папки для чтения и делать одну папку для записи, куда будут помещены все новые изменения без потери старых данных. Вспомогательной для этого функцией является технология копирования при записи (*copy on write*), из-за чего существенно уменьшается объем записываемой информации и нагрузки на устройства хранения информации.

Скорость работы данной файловой системы проигрывает реализации из ядра *Linux*, но все равно является приемлемой. В случае использования с HDD скорость будет зависеть только от возможностей данного устройства, поэтому говорить о снижении производительности в случае использования данной файловой системы нет необходимости.

<<<<<<< HEAD *Unionfs* позволяет объединять любые каталоги, что позволяет нам использовать различные другие файловые системы. Это возможно, если мы примонтируем ======= *Unionfs* позволяет хранить и объединять любые каталоги, что позволяет нам объединять различные другие файловые системы. Это возможно, если мы примонтируем >>>>>>> origin/comments файловую систему *A* в какой-либо каталог, и в дальнейшем передадим этот каталог в качестве аргумента для *unionfs*. (***к чему это?***)

1.3 ZSH

Командная оболочка *UNIX* под названием *ZSH*[4] является очень популярной. Она является аналогом командной оболочки *BASH*, но не является ее продолжением или ответвлением. *ZSH* предоставляет пользователю множество улучшений и дополнительных возможностей при сравнении с *BASH*.

Для реализации данного проекта была использована возможность данной командной оболочки, позволяющая добавлять собственные дополнительные действия при <<<<<<< HEAD определенных событиях при работе в консоли в *ZSH*. Так называемые *hooks*. ======= определенных события при работе в консоли в *ZSH*, так называемые *hooks*.

(***смешано, что использовал, и что просто существует***)

(***С точки зрения сохранения действий пользователя* ZSH *позволяет добавлять… В частности: …***) >>>>>>> origin/comments

Были использованы:

- *precmd* hook — позволяет добавлять действия перед появлением поля ввода командной строки (фактически, после заврешения команды).
- *preexec* hook — позволяет добавлять действия перед непосредственным выполнением команды.

(***Это должно быть во второй части***) Использование этих двух функций позволило наладить автоматизацию действий. То есть человек не должен бу-

дет вызывать дополнительные команды для работы, все будет выполняться без его участия при наступлении необходимых событий. Возможность ручного вызова функций не была отключена из соображений, что пользователь может придумать свой сценарий использования.

ГЛАВА 2

ПОСТАНОВКА ЗАДАЧИ

В данной главе описывается задача, которую требуется решить, и определяется конечная цель. Рассматриваются те способы решения, которые уже можно применить для получения результата, приведено их сравнение.

2.1 Проблема

В наше время пользователи, использующие командную строку в качестве инструмента для работы с данными, часто не имеют возможности однозначно воссоздать историю того, как они получили тот или иной результат (***простую/минимальную историю, без всего лишнего, просто история — берешь весь .zhistory и все***). Нет способа описания получения файлов по действиям пользователя в консоли.

Существуют различные варианты того, как можно, объединив несколько различных утилит, получить необходимый результат. Но все равно не получится добиться того, что бы полностью были выполнены необходимые условия. А именно:

- Показать (***минимальную***) историю того, как получился результат.
- Увидеть все зависимости между командами и файлами.
- Из истории убрать те шаги, что привели к ложным результатам или вовсе не дали никакого результата.

Таким образом в настоящий момент существует проблема отсутствия необходимого программного обеспечения для предоставления человеку необходимого удобства и комфорта в решении задачи автоматического описания способа получения файлов по его действиям в консоли (***больше одного и того же разными словами! (это сарказм)***).

2.2 Цель работы

Makefile - файл для утилиты автоматического создания файлов, описывающий то, как получить финальный результат. Данные способы генерации и

описывания процесса компиляции и линковки программ широко распространены.

В настоящий момент на основе идеи makefile, технологии от 1977 года, создано большое количество утилит подобного рода. Например:

- CMake
- Cabal
- Cargo
- Nix

Все это утилиты совершенно разные, предназначены для разных языков программирования, но объединяет их одна возможность, которой все они обладают - это описание способа автоматического получения результата (скомпилированной программы, документа в PDF, состояния файловой системы и тп.) на основе исходных данных.

Но все они создаются таким образом, что человек сам изначально должен задать каждое действие и поддействие. И, естественно, от каждого из них зависит следующее. Возникает проблема - при ошибке одного из пунктов алгоритма получения результата, все дальнейшие пункты будут уже работать на основе этих ошибочных данных.

Стоит отметить, что некоторые системы сборки умеют по исходным данным и минимальной дополнительный информации получать makefile'ы для воссоздания проекта. Например *cabal*. Данная программа умеет по исходному коду проекта на языке *haskell* и информации о том, что же является ключевым файлом с главной функцией (*main*), генерировать файл проекта с расширением `.cabal`. Этот файл позволяет скомпилировать проект и получить необходимый результат. Но, к сожалению, данная программа умеет работать только с исходными кодами будущих (***каких будущих?***) программ или библиотек.

Что же можно предпринять для общего случая? Наиболее простым способом генерации таких файлов автоматической сборки является самостоятельное выполнение всех пунктов, а потом, убедившись в том, что результат является искомым, запись данных действий в определенном формате в makefile.

Но в таком подходе существует сложность в построении правильной последовательности действий для каждой компоненты (напр. (***не сокращ.***) файла) результата. Надо учитывать каждое действие, которое затрагивало файл,

будь то действие которое его изменяло или просто открывало.

К сожалению, на данный момент нет инструмента, который бы позволял наиболее простым образом выполнять данную задачу.

2.3 Постановка задачи

Необходимо создать инструмент, который позволит однозначно описывать историю получения файла, а так же создание необходимых инструментов окружения, позволяющих человеку удобно модифицировать, а так же находиться в разных этапах построения данного компонента.

Требования к программному обеспечению, позволяющее выполнить задачу:

а) Вывод графа IO операции для конкретного файла.
б) Возможность отменять действия над файлом.
в) Восстановление любой версии файла или проекта.
г) Параллельная работа над проектом.
д) Возможность объединения изменений из разных временных версий проекта.

Таким образом (***не понятно следствие***), программное обеспечение для данной проблемы можно разделить на 2 части реализации:

а) Демонический режим.
б) Интерактивный режим.

Демонический режим подразумевает выполнение действий, необходимых для поддержания выполнения требований к решению задачи, без участия человека. То есть автоматическое выполнение и активация необходимых функций и процессов.

Интерактивный режим подразумевает выполнение действий при участии человека. То есть человек должен самостоятельно вызывать команды необходимые ему.

Демонический режим требуется для автоматизации и упрощения процесса создания истории для проекта, а интерактивный режим будет необходим для использования утилит и других инструментов для модификации, просмотра или работы с историей, либо другой информацией, связанной, непосредственно, с задачей.

2.4 Существующие методы решения

В обзор!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

2.4.1 Системы управления версиями

Для сохранения в явной истории ключевых состояний проекта можно использовать систему контроля версий.

В наше время широко распространены:

- SVN[5]
- Git[6]
- Mercurial[7]

SVN (Subversion) — централизованная система управления версиями. В наше время ее активно вытесняет распределенная система управления версиями — *Git*. Отличие *svn* от *git* заключается в том, что вторая, как было сказано выше, распределенная. Другими словами, если есть несколько разработчиков работающих с репозиторием у каждого на локальной машине будет полная копия этого репозитория. <<<<<<< HEAD Разумеется есть где-то и центральная машина, с которой можно клонировать репозиторий. В *git* это похоже на SVN. Основной плюс в том, что если вдруг у ======= Разумеется, есть где-то и центральная машина, с которой можно клонировать репозиторий (***совершенно необязательно***). В *git* это похоже на SVN. Основной плюс в том, что если вдруг у >>>>>>> origin/comments вас нет доступа к интернету, сохраняется возможность работать с репозиторием. Потом только один раз сделать синхронизацию и все остальные разработчики получат полную историю (**хаха, один раз**).

В целом суть работы системы контроля версий отдаленно похожа на то, что мы хотим решить. Но цель этих инструментов совсем в ином.

Поскольку самая популярная система (***вероятно неправда: http://programmers.stackexchange.com/a/136207***) контроля версий на данный момент это *git*, то рассмотрим ее.

С помощью *git* мы можем:

- Фиксировать состояние интересующих нас файлов.
- Смотреть историю по каждому из них.

- Работать параллельно.

Но данное решение нельзя назвать удовлетворительным в рамках нашей задачи. Ведь придется фиксировать каждое изменение, вручную указывать команду, использованную для шага. А так же мы не сможем отследить все зависимости во время IO-действий с файлом, что бы в дальнейшем сказать о том, какие файлы в проекте являлись необходимыми для достижения результата, а какие были “мусором”. Так же эта система — не самый лучший выбор для использования с большими не текстовыми файлами.

2.4.2 Galaxy

Одним из существующих на данный момент инструментов, что может решить нашу задачу — является Galaxy[8]. Это всесторонний инструмент для воспроизводимых, доступных и прозрачных вычислений в биоинформатическом анализе. Но в нашей задаче мы не можем применить данный инструмент, так все действия в galaxy происходят в веб интерфейсе. И команды, которые может использовать человек, они связаны только с биоинформатическим анализом. Таким образом, данное программное обеспечение ограничивает сферу своего применения.

2.5 Инструменты для реализации

Для реализации данной идеи необходимо было использовать инструмент, который позволяет:

- Работать со структурой файловой системы.
- Уметь вызывать другие программы.
- Обрабатывать результат выполнения других программ.

Таким образом, в реализации данного проекта не важен язык программирования, так как задача, которую надо реализовать, в принципе, выполнима при помощи большинства современных прикладных ЯП.

Изначально для разработки был выбран язык *Rust*. Он превосходно подходит для такой задачи, так как изначально позиционировался как инструмент для разработки системных приложений. Так же он имеет строгую систему

типизации, что позволяет избежать многих ошибок. *Rust* компилируется в нативный код, что сказывается на его быстродействии. Но на момент начала работы над задаче, он находился в стадии активной разработки, не имея стабильной версии. Разработчики часто вносили необратимые изменения в язык, из-за которых необходимо было проделывать много работы заново. По этой причине, от этого инструменты отказались.

В итоге, из-за личных предпочтений, решено было использовать язык *Haskell*.

Данный язык программирования является высокоуровневым функциональным языком.

Его преимущества:

- Строго типизирован.
- Компилируется в нативный код.
- Позволяет работать в операционной системе Linux.
- Широко распространен.
- Требует меньшего написания кода для выполнения тех же задач (если сравнивать с *C++* или *Java*, например).

Недостатки:

- Требует высокую квалификацию программиста для реализации программы.

ГЛАВА 3

РЕАЛИЗАЦИЯ

В итоге, при рассмотрении всего, что было описано во второй главе, было решено разработать свое программное обеспечение, которое бы полностью выполняло поставленную задачу.

Программа была названа *gshell* от слов git и shell, так как работает по схожему принципу действий с данной системой контроля версий, а также работает в командной строке — в shell.

3.1 Описание gshell

Данная программа выполняет поставленную задачу создания истории файла по тем действиям, что пользователь сделал в консоли.

Проект под управлением gshell представляет из себя рабочую папку, индивидуальную для каждой открытой консоли. Каждая такая папка — это сессия. Каждое состояние проекта — это ревизия. Таким образом получается древовидная структура, в которой узлы это и есть ревизии. Каждая сессия имеет свою собственную историю, начало которой берется из того узла, что был записано в качестве последней ревизии в мастер ветке. Мастер ветка — это главная история, куда можно добавлять новые ревизии, или же брать оттуда все ревизии в свою текущую сессию.

На рисунке 3.1 черные круги — это мастер ветка, а круги разных цветов — это различные сессии.

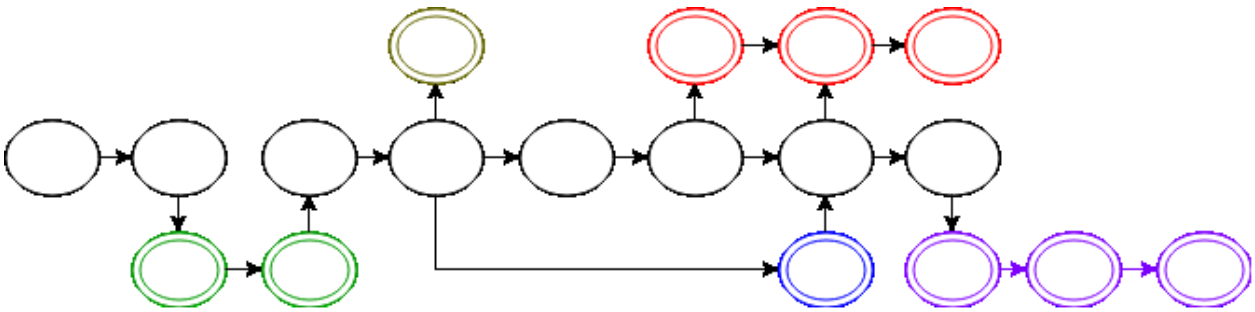


Рис. 3.1 — Граф истории

Работа gshell по созданию ревизий происходит в демоническом режиме. Обновление мастер ветки, создание сессии, создание проекта – все это происходит в интерактивном режиме.

Ниже приведен список команд, которые пользователь может использовать:

- gshell init — начальное создание проекта.
- gshell enter — создание новой сессии для работы с проектом.
- gshell log — просмотр истории для всех файлов в сессии.
- gshell log — просмотр истории для заданного файла в рамках данной сессии.
- gshell push — добавление текущей сессии в мастер ветку.
- gshell pull — добавление ревизий из мастер ветки в текущую сессии.
- gshell exit — выход из сессии.
- gshell off — выключение функции автоматического фиксирования изменения файлов в сессии.
- gshell on — включение функции автоматического фиксирования изменения файлов в сессии.
- gshell clear — очистка проекта.
- gshell makefile — создание makefile для воссоздания заданного файла.
- gshell rollback — отмена предыдущей команды.
- gshell checkout — переход на конкретную ревизию.
- gshell graph — вывод в формат png графа файлов и зависимостей между ревизий.

3.2 Пример работы в консоли с помощью gshell

Пример процедуры работы пользователя с проектом *project* при помощи *gshell*:

```
$ gshell init project
$ gshell enter project
work-id8172 $ touch x
work-id8172 $ gshell exit
```

Изначально пользователь вызывает *gshell* с аргументами *init* и названием проекта. Это создает необходимые начальные файлы и папки. Создается папка проекта, `.gshell`, нулевая ревизия (в которой записываются пустые предки, пустое сообщение commit и время создания), в мастер ветку она же

и записывается.

Далее пользователь вызывает *gshell* с аргументами *enter* и названием проекта.

Что происходит “внутри” во время исполнения этой команды:

- Создается подпапка в папке проекта для новой сессии.
- Берется последняя ревизия из мастер ветки.
- Строится дерево из всех ее предшественников (основываясь на файле parents).
- Создается новая ревизия, в которую будет писать новая сессия.
- Монтируется при помощи *unionfs* получившаяся ветка в новосозданную папку для сессии.
- Получив путь к рабочей директории, папка делается активной для пользователя.
- Пользователь перемешается в новую сессию.

Затем пользователь может выполнять любые действия в данной сессии. За исключением действий по выходу из папки на уровни выше, так как это уже не будет управляться *gshell*.

Каждое действие пользователя сохраняется.

Когда введена команда и нажата клавиша для ее исполнения:

- Фиксируется состояние проекта.
- Выполняется команда.
- Текст команды записывается как сообщение данной ревизии, говорящее о том, как
- это состояние было получено.
- Фиксируется время исполнения в ревизии.

Все это нужно для того, чтобы множество ревизий отображало проект на фазе каждой команды. Имея множество состояний мы можем перемещаться в любой момент работы. Если была сделана ошибка, мы можем “откатиться” назад, исправить ошибку и получить новое состояние системы без прошлой оплошности. Это и позволяет нам в дальнейшем, получать информацию о том, что мы делали с тем или иным файлом. Поскольку в каждой ревизии фиксируется какие файлы были изменены, а так же какие файлы из каких предыдущих ревизий были затронуты. На запрос об истории файла мы можем пройтись по всем ревизиям, которые используются в текущей сессии и

по информации о доступе однозначно сказать, какие команды были использованы по отношению к файлу.

Когда пользователь получит необходимый результат он может выйти из программы, оставив весь проект как есть. Но так же присутствует возможность удаления проекта, но не результирующих файлов.

Человек может работать в проекте одновременно из нескольких терминалов. Это было реализовано при помощи создания мастер ветки ревизий. Изначально, каждая сессия не зависит от другой (только от тех ревизий, что были уже в мастер ветке, тк новая сессия всегда начинается от последнего состояния в главной ветке). Она имеет собственную историю и список используемых ревизий. Но если пользователю нужно сделать данную ревизию и ее родителей основной, то есть что бы другие результаты были основаны на ней, он может добавить ее в мастер ветку. При существовании конфликтов файлов происходит процесс объединения. Он прост — тот файл, что старше, будет заменен файлом который моложе. Это реализовано за счет того, что ревизии монтируются по времени создания (завершения команды, что записана в качестве сообщения).

Стоит отметить, что во время сессии команды *gshell* не фиксируются в истории.

3.3 Основной компонент программы

Наиболее частой операцией в программе является взаимодействие с файлами и директориями. Поэтому было решено сделать самую гравную структуру в виде дерева файлов.

В качестве реализации использована библиотека directory-tree [9].

Сигнатура этой структуры выглядит следующим образом:

```
data DirTree a = Dir { name :: FileName,
                       contents :: [DirTree a] }
               | File { name :: FileName,
                        file :: a }
               | Failed { name :: FileName,
                          err :: Exception }
```

Таким образом мы можем загрузить образ нашей файловой системы в память программы и свободно ей манипулировать.

3.4 Внутреннее устройство gshell

3.4.1 Базовая структура

Для обработки всей информации было решено состоянием программы сделать так же директории и их содержимое. С помощью выше описанной структуры мы считываем следующее:

```
project-root/
├─ .gshell/
│  ├─ commits/
│  │  ├─ master
│  │  ├─ rev1-rev1Hash/
│  │  │  ├─ to-mount/
│  │  │  ├─ commit
│  │  │  ├─ parents
│  │  │  ├─ time-stamp
│  │  │  └─ vars
│  │  ├─ rev2-rev2Hash/
│  │  │  ├─ to-mount/
│  │  │  ├─ commit
│  │  │  ├─ parents
│  │  │  ├─ time-stamp
│  │  │  └─ vars
│  │  └─ …
│  ├─ work-id1/
│  │  └─ work-information
│  └─ work-id2/
│     └─ work-information
├─ work-id1/
└─ work-id2/
```

gshell — программа, которая инициализируется отдельно для каждого проекта. Мы можем видеть нашу структуру, которая будет одинаковой по своему строению для каждого из проекта. Но, естественно, содержать раз-

ную информацию.

- *project-root* — изначальная папка, которая дается gshell в качестве начальной для инициализации.
- *.gshell* — папка, в которой находятся ревизии, а так же дополнительная информация о каждой запущенной сессии.
- *commits* — папка, в которой находятся ревизии.
- *master* — файл, который содержит название главной ревизии.
- *rev-revhash* — папка для ревизии. *revhash* — случайная строка, которая создается для каждой ревизии.
- *to-mount* — папка, которая содержит файлы пользователя, относящиеся для конкретной ревизии.
- *commit* — файл, который содержит сообщение, поясняющие, как была создана данная ревизия.
- *parents* — файл, содержащий в себе информацию о ревизиях, которые были предками данной. То есть те ревизии, из которых она получилась.
- *time-stamp* — файл, содержайщий запись о том, когда ревизия получилась. Стоит отметить, что в файле записано не время создания, а момент, когда действие, необходимое для получения результата было завершено.
- *vars* — файл, содержайщий запись о том, какие были переменные среды во время сессии.
- *.gshell/work-id** — папка, которая содержит в себе информацию, необходимую для работы сессии.
- *work-information* — файл, который содержит в себе список ревизий, которые загружены для данной сессии.
- *project-root/work-id** — папка, в которой “находится” пользователь и в которой в данный момент загружено множество ревизий. Пользователь в ней видит только свои файлы.

3.5 Граф зависимости и истории

Одной из возможностей *gshell* является создание графа зависимостей и истории. На рисунке 3.2 виден пример типичной сессии.

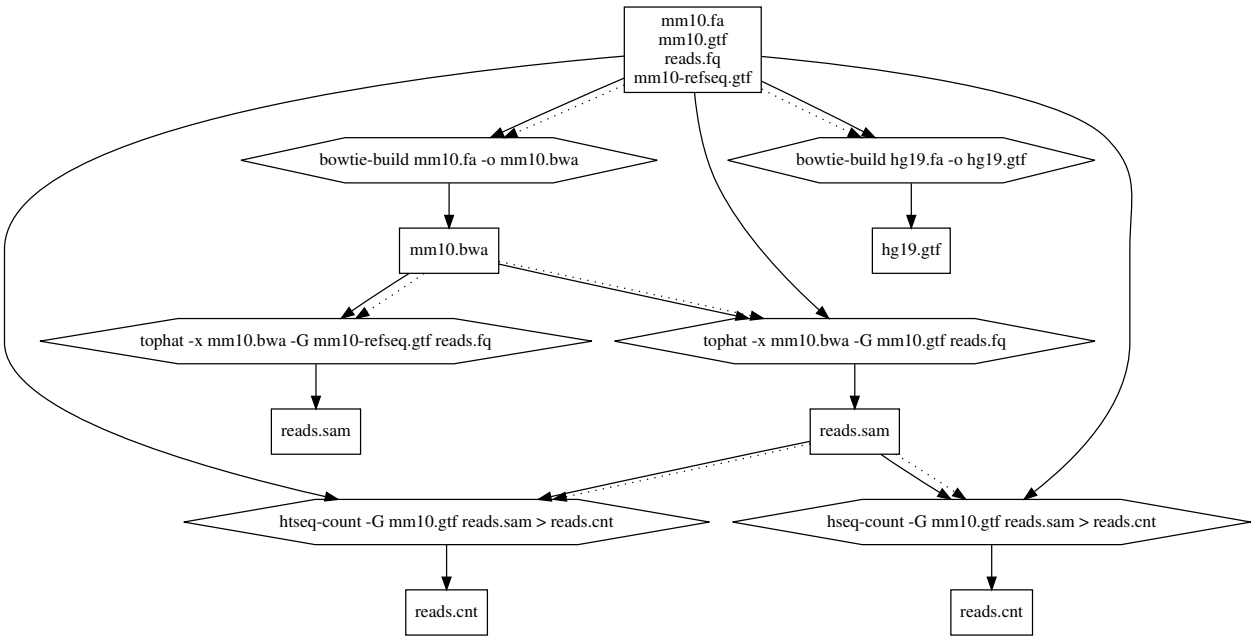


Рис. 3.2 — Граф истории и зависимостей

- Пунктирные стрелки — показывают отношение родитель-ребенок, переход между состояниями с точки зрения истории. То есть та ревизия, откуда идет стрелка, является родителем, но не обязательно зависимостью ревизии, куда стрелка указывает.
- Непрерывные стрелки — показывают отношение зависимости одной ревизии от другой.
- Шестиугольники — команда в консоли.
- Квадрат — результат. Получившиеся новые или измененные старые файлы.

Когда пользователь получит необходимый ему результат (файл *reads.sam*), то он может вызвать команду *gshell graph reads.sam* и получить результат, который показан на рисунке 3.3.

Таким образом пользователь получит удобное представление истории файла.

Если же будет использована команада *gshell makefile reads.sam*, то будет выведен лог данного файла в виде списка команд, что его “породили”.

```
$ bowtie-build mm10.fa -o mm10.bwa
$ tophat -x mm10.bwa -G mm10.gtf reads.fq
$ htseq-count -G mm10.gtf reads.sam > reads.cnt
```

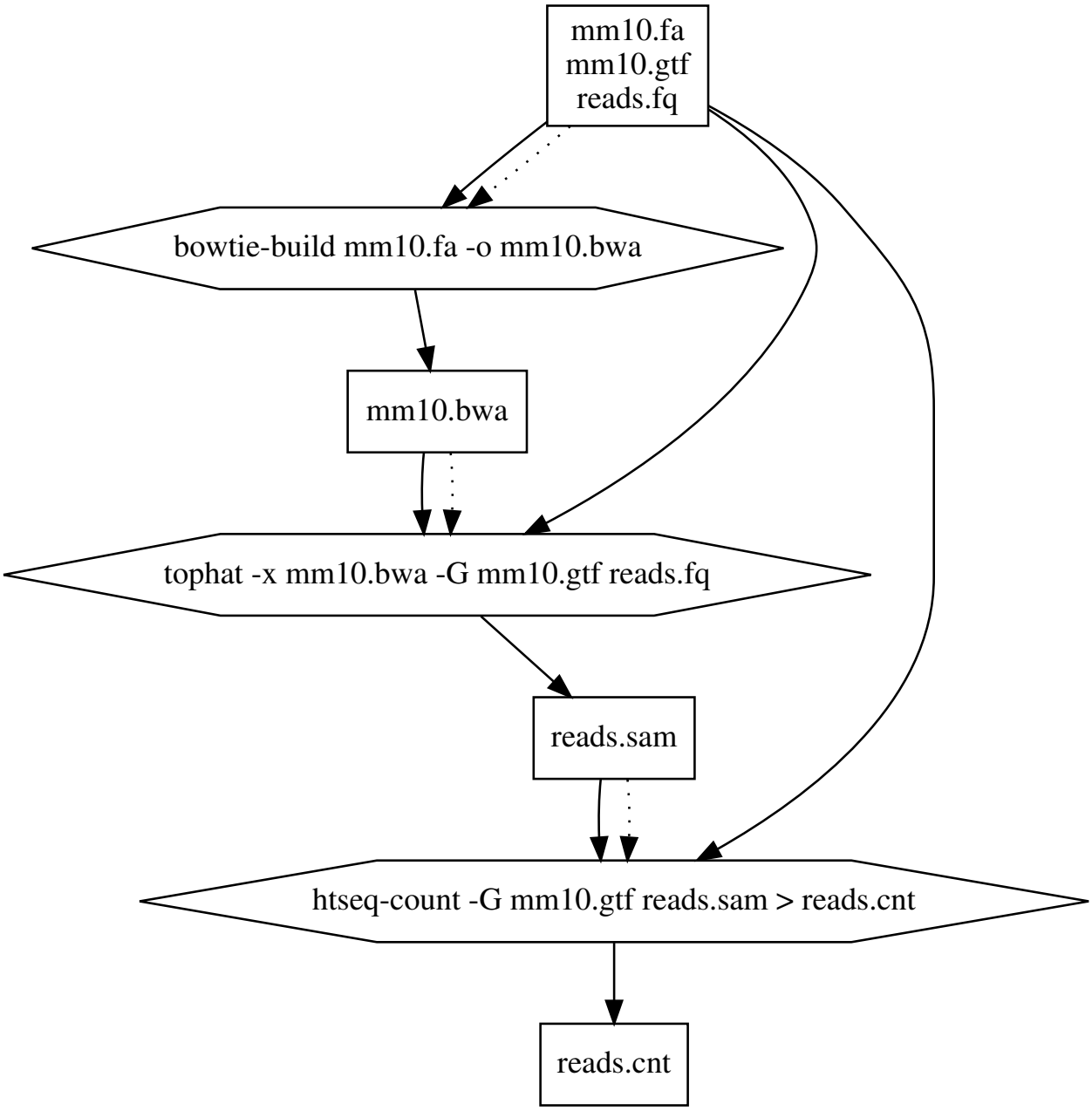


Рис. 3.3 — Граф истории и зависимостей

3.6 Механизм работы gshell

Когда на вход программе подаются аргументы, перед тем как понять, что хочет пользователь, *gshell* всегда проверяет наличие папки проекта, и генерирует внутренее состояние на основе ее содержимого.

Далее в зависимости от необходимого действия происходит вызов соответствующей функции. Ни одна из команд не будет работать, если для ее осуществления отсутствует, например, директория проекта. *gshell* выдаст ошибку.

3.6.1 Монтирование ревизий

Для каждой сессии существует свой список ревизий, которые будут в ней использоваться. Ревизии отсортированы по времени, и, таким образом, те из них, что старше, будут примонтированы только для чтения. Самая последняя же, то есть самая новая (пустая) — будет для записи. Это и обеспечивает возможность создания ревизии для каждого действия. Если у нас в ревизии, что замонтирована для чтения, есть файл, который пользователь изменит, то измененная версия файла будет уже записана в новую ревизию, что смонтирована для записи. Это реализовано с помощью программы *unionfs*. Она обеспечивает copy-on-write технологию. То есть файл будет записан только в том случае, если его начнут изменять. Если файл был просто открыт и с ним ничего не сделали, то он так и останется в той ревизии, где и был. Если же мы открыли файл и изменили его, то он скопируется и изменится. Если мы удалили файл в нашей ревизии, то *unionfs* пометит данный файл как удаленный, и при дальнейшей работе, при дальнейшем использовании данной ревизии, мы этот файл уже наблюдать не будем. Но если он нам понадобится, то мы можем просто взять то состояние проекта, когда он существовал!

3.6.2 Обновление сессии и фиксирование изменений

Когда пользователь используется проект под управлением *gshell*, он выполняет команды в консоли. После каждой команды фиксируются изменения и обновляется текущая сессия. Обновление — это есть создание новой ревизии и монтирование предыдущих только для чтения. Когда монтирование завершено, для пользователя обновляется автоматически сама сессия. Это реализовано при помощи командной оболочки UNIX ZSH. В нее добавляются необходимые операции, которые выполняются после любых команд, при условии инициализированного и включенного *gshell*.

3.7 Реализация и используемые возможности языка

3.7.1 Линзы

Для работы со внутренней структурой была использована библиотека lens[10].

Данная библиотека позволяет очень просто модифицировать объекты. Например у нас есть проект со своей структурой и эта структура в памяти программы. Чтобы получить время создания второй ревизии без использования библиотеки lens, нам было бы необходимо писать очень длинную рекурсивную функцию, которая бы распаковывала структуру, проверяла наличие необходимых подпапок и так далее. Код получился бы очень громоздким. Но с помощью данной библиотеки мы можем написать:

```
timeStamp name = revisionRoot name.traverse.filteredByName
    timeStampFileName._file
```

На вход функции мы просто даем имя интересующей нас ревизии и саму структуру.

Линзы - это функции первого класса для доступа к данным в структурах. Отдаленным аналогом в императивных язык программирования являются getters и setters. В русском языке правильней было бы их назвать лупой, так как они позволяют сфокусироваться на каком-то значении. А фокусироваться нам нужно как раз для того, чтобы изменять (set) или получать (get).

Ряд функций, для упрощения работы с состоянием программы приведен ниже:

3.7.2 Монада State

Монада в языке Haskell — это класс типов, для вхождения в который, на типе m должны быть определены функции:

- `>>= :: m a -> (a -> m b) -> m b`
- `return a -> m a`

```
projectRoot = _dirTree._contents

gshellRoot = projectRoot.traverse.filteredByName gshellDirName._contents

commitsRoot = gshellRoot.traverse.filteredByName commitsDirName._contents

revisionRoot name = commitsRoot.traverse.filteredByName name._contents

workDirs = projectRoot.traverse.filteredByName workDirName

workingState name = gshellRoot.traverse.filteredByName
    name._contents.traverse.filteredByName workHelperFileName._file

masterState = commitsRoot.traverse.filteredByName masterFileName._file

parents name = revisionRoot name.traverse.filteredByName
    parentsFileName._file

timeStamp name = revisionRoot name.traverse.filteredByName
    timeStampFileName._file

revCommit revision = commitsRoot.traverse.filteredByName
    revision._contents.traverse.filteredByName commitFileName._file
```

Вопреки расхожему мнению, >>= может иметь самые различные семантические нагрузки — от изощрённого составления цепочки вызовов (“пустая” монада), до поддержки и передачи состояния (монада State), до ввода-вывода (монада IO). Поэтому, по некоторой информации, императивным программистам проще думать о монадах в Haskell как о программируемом операторе “точка с запятой”.

Данная монада State позволяет манипулировать состоянием программы, передавая результат и новое состояние из одной функции в другую.

3.7.3 Трансформеры

Монадный трансформер — это некая “матрешка”. Он объедиет две монады в одну, с целью получения свойст двух монад в одной.

В моей программе используется монадный трансформер *StateT* вместе с монадой *IO*. Это позволяло менять состояние программы, а так же выполнять IO действия, при этом сохраняя строгую типизированность программы и быть в безопасности от побочных эффектов.

Совместное использование трех технологий: линзы, состояние и монадный трансформер — позволяет очень легко писать код программы, быстро

его изменять и просто читать.

3.7.4 Результат использования технологий

Поскольку было решено использовать монаду state с линзами, то код получился очень легко читаем. Но, в тоже время, он сохраняет все свойства и плюсы функционального подхода. То есть — мы получили состояние (считали с диска), затем необходимым образом его изменили, и только после этого, включая необходимые проверки, записали его обратно на диск, а также, по необходимости, выполнили и другие IO действия.

На листинге 3.1 мы можем видеть пример функции реализованной при помощи данных библиотек и техник. Как можно заметить, код получился в императивном стиле, легко понять, что делает любая строка. Хотя, за каждой из них стоит много другого, вспомогательного, кода.

```
commitGshell :: String -> FilePath -> StateT GState IO Result
commitGshell message currentWork = do
   lift $ unmountWorkspace currentWork
   workState <- getWorkState currentWork
   let parent = last $ workState ^. revisions
   setTimeStamp parent
   writeCommitMessage message parent
   revName <- createCommitDir $ Parents [parent]
   workState' <- gets $ WorkingState . generateBranch [revName]
   workingState (takeFileName currentWork) .= show workState'
   writeStateToDisk
   get >>= lift . createWorkspace currentWork (workState' ^. revisions)
```

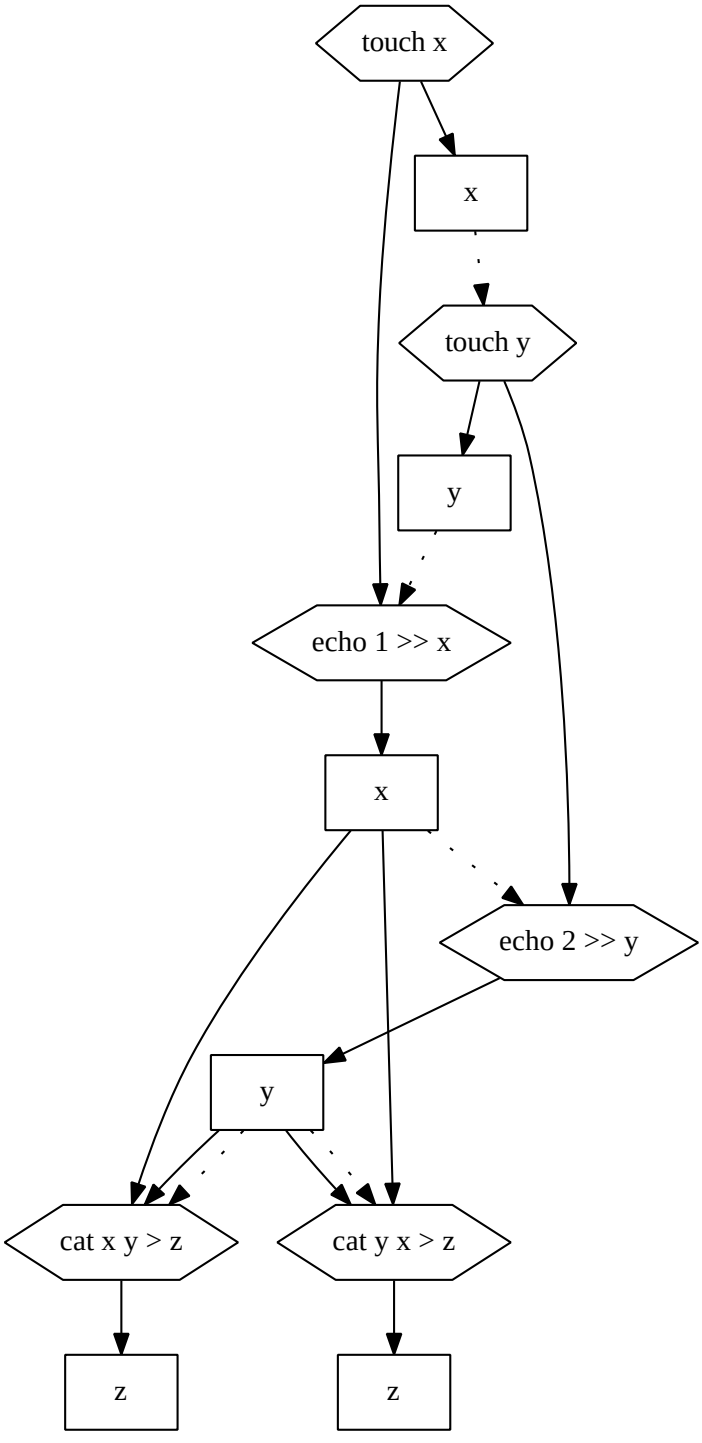
Листинг 3.1 — Функция для создания коммита

3.8 Дополнительные примеры работы

3.8.1 Пример №1

В данном примере рассмотрено использование примитивных команд, которые часто используются в консоли. Так же используется функция отмены предыдущего действия так как были переданы аргументы в не правильном порядке, что дало не верный результат.

```
$ gshell init project
$ gshell enter project
work-id6174 $ touch x
work-id6174 $ touch y
work-id6174 $ echo 1 >> x
work-id6174 $ echo 2 >> y
work-id6174 $ cat x y > z
work-id6174 $ gshell rollback
work-id6174 $ cat y x > z
```



На рисунке виден пример граф данной сессии пользователя. Видно, какие файлы были задействованы для каких команд.

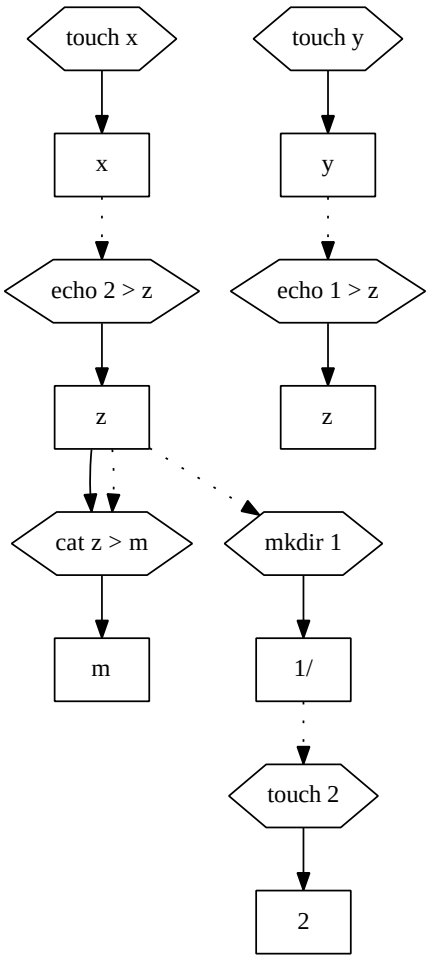
3.8.2 Пример №2

В данном примере рассмотрено использование gshell для параллельной работы в нескольких ветвях.

```
$ gshell init project
$ gshell enter project
work—id6174 $ touch x
work—id6174 $ echo 1 > z

$ gshell enter project
work—id4416 $ touch y
work—id4416 $ echo 2 > z
work—id4416 $ cat z > m

$ gshell enter project
work—id1510 $ mkdir 1
work—id1510 $ touch 2
```



На рисунке виден пример граф данной сессии пользователя. Показано, что в графе есть несколько сессий, которые работали параллельно.

ЗАКЛЮЧЕНИЕ

В работе была реализована программа на языке Haskell для семейства операционных систем на базе ядра Linux, выполняющая задачу автоматического описания способа получения файлов в проекте по действиям пользователя в консоли. Особенностью данного программного обеспечения является возможность привычной работы в консоли, без использования дополнительных навыков или знаний. Данный комплекс полноценно выполняет поставленную перед ним задачу, а так же предоставляет дополнительные возможности для работы.

Полный список возможностей программы:

- Просмотр истории файла, с включением всех необходимых шагов для его воссоздания.
- Отмена изменения(-ий) для исправления неверных шагов, при начальном интерактивном построении истории файлов.
- Параллельная работа в проекте, позволяющая запускать длительные действия в консоли, и одновременно с ними продолжать работу в нескольких сессиях.
- Генерация истории для файла, с возможностью воспроизведения на другом компьютере, при условии установленных там средств программного обеспечения, использованного в сессии создания файла(-ов).

В проекте активно используется программа *unionfs*, командная оболочка *zsh*, средства POSIX совместимых файловых систем для ОС на базе Linux.

Отличительной чертой данного проекта является скорость работы, совершенно не заметная для пользователя, а так же полная интеграция в существующие решения, что позволяет человеку не задумываться о том, чтобы “изучать” новый инструмент.

Реализация была выполнена на языке haskell, с активным применением линз и монады состояния, что положительно сказалось на простоте написания кода и читаемости кода.

СПИСОК ИСТОЧНИКОВ

1. Community. Linux. [Электронный ресурс]. URL: http://linux.org.
2. Podgorny Radek. Unionfs-fuse. [Электронный ресурс]. URL: https://github.com/rpodgorny/unionfs-fuse.
3. Unionfs: User-and community-oriented development of a unification filesystem / David Quigley, Josef Sipek, Charles P Wright [и др.] // Proceedings of the 2006 Linux Symposium. Т. 2. 2006. С. 349–362.
4. Paul Falstad Peter Stephenson. Z Shell. [Электронный ресурс]. URL: http://www.zsh.org/.
5. CollabNet. SVN. [Электронный ресурс]. 2000. URL: subversion.apache.org.
6. Torvalds Linus, Hamano Junio. Git: Fast version control system. [Электронный ресурс]. 2005. URL: http://git-scm.com.
7. Mackall Matt. Mercurial. [Электронный ресурс]. 2005. URL: https://mercurial.selenic.com/.
8. Team The Galaxy. galaxy. [Электронный ресурс]. URL: https://galaxyproject.org/.
9. Simmons Brandon. A simple directory-like tree datatype. [Электронный ресурс]. URL: https://hackage.haskell.org/package/directory-tree.
10. Kmett Edward. The—lens—package. [Электронный ресурс]. URL: http://hackage.haskell.org/package/lens.