Оглавление

Предисловие

Как один из разработчиков компилятора языка Haskell GHC (Glasgow Haskell Compiler) на протяжении почти 15 лет я наблюдал его развитие от нишевого, чисто исследовательского языка до богатейшей процветающей экосистемы. Большая часть этого времени была потрачена на реализацию поддержки параллелизма и конкурентности. Первое, что я сделал в GHC в 1997 году, было переписывание системы времени исполнения, в то время мы приняли ключевое решение встроить конкурентность непосредственно в ядро системы, отказавшись от её реализации в виде дополнительной библиотеки. Мне нравится думать, что это решение основывалось на гениальном предвидении, хотя в действительности на него сильно повлияло обнаружение нами способа снизить накладные расходы на конкурентность почти до нуля (прежде они составляли порядка 2%; мы всегда были одержимы производительностью). Сделать конкурентность обязательной уже означало, что она автоматически станет важнейшей частью реализации, и я убеждён, что принятое решение поспособствовало появлению в GHC надёжной и быстрой её поддержки.

Haskell уже давно ассоциируется с параллелизмом. Достаточно вспомнить такие проекты, как pH, вариант языка Haskell, построенный на основе специально спроектированного с поддержкой параллелизма языка Id, система GUM, обеспечивавшая запуск параллельных программ на языке Haskell на нескольких машинах вычислительного кластера, система GRip, полная компьютерная архитектура, предназначенная для запуска параллельных функциональных программ. Все эти проекты имели место задолго до распространения многоядерных процессоров и связанной с ними революции в программировании, проблема заключалась в том, что в те времена закон Мура ещё позволял получать всё более быстрые компьютеры. Достижение параллелизма было трудной задачей, в которой не было большого смысла в условиях, когда компьютеры становились экспоненциально быстрее.

Около 2004 года мы решили сделать параллельную реализацию си-

стемы времени исполнения GHC для запуска на мультипроцессорах с разделяемой памятью, чего раньше не было. Многоядерная революция пока не произошла, процессоры с несколькими ядрами ещё были впереди, тогда как машины с несколькими процессорами уже встречались довольно часто. И снова я предпочёл бы думать, что решение заняться в этот момент параллелизмом было продиктовано даром предвидения, но, честно говоря, поработать с реализацией параллелизма для систем с разделяемой памятью было просто-напросто очень интересно, это была увлекательная исследовательская задача. Существенным здесь была чистота Haskell — это означало, что можно избежать некоторых накладных расходов на блокировки в системе времени исполнения и при сборке мусора, что, в свою очередь, означало, что мы можем снизить накладные расходы на использование параллелизма до очень небольших значений. До того как новой реализацией стало возможно пользоваться и ускорять выполнение широкого спектра программ, потребовалось провести довольно много исследований, переписать планировщик и реализовать новую параллельную сборку мусора. Работа, которую я представил на международной конференции по функциональному программированию (ICFP) в 2009 году, ознаменовала переход от интересного прототипа к реальному инструменту.

Исследовать и реализовывать всё это было крайне интересно, однако качественных ресурсов для обучения программистов использованию параллелизма и конкурентности в Haskell практически не было. В течение последних двух лет я имел счастье читать на двух летних школах курсы по параллельному и конкурентному программированию на языке Haskell: первой была центрально-европейская летняя школа по функциональному программированию (CEFP) 2011 года в Будапеште, а второй — летняя школа CEA/EDF/INRIA 2012 года в Кадараше на юге Франции. Готовя материалы к этим курсам, я впервые написал расширенный туториал и начал собирать хорошие демонстрационные примеры. После школы 2012 года в моём туториале было около 100 страниц, и благодаря побуждению от одного или двух человек (см. Благодарности) я решил превратить его в книгу. Тогда мне казалось, что готово 50%, хотя на самом деле там было ближе к 25%. Мне было, о чём писать! Надеюсь, вам понравится результат.

Для кого предназначена эта книга

Вам потребуется уверенное владение языком Haskell, соответствующего материала в этой книге нет. Возможно, лучше предварительно прочитать какую-либо вводную книгу по языку Haskell, например «Изучай Haskell во имя добра!» (ДМК Пресс, 2012).

Как читать эту книгу

Основная цель книги — в том, чтобы научить вас программировать с использованием таких инструментов, как Parallel Haskell и Concurrent Haskell. Однако, как вам должно быть известно, обучение программированию не может состоять только из чтения книги. Поэтому эта книга имеет явный практический уклон: имеется множество примеров, которые вы можете запустить, модифицировать и расширить. В некоторых главах содержатся предложения относительно упражнений, которые вы могли бы попробовать решить, чтобы освоиться в представленных в этих главах темах. Я настоятельно рекомендую либо поработать с этими упражнениями, либо запрограммировать что-нибудь своё.

По мере прохождения различных разделов книги я не буду стесняться указывать на недостатки и иные несовершенства тех или иных фрагментов системы. Хотя Haskell развивается уже более 20 лет, сегодня темпы его роста выше, чем когда бы то ни было в прошлом. Поэтому мы неминуемо будем натыкаться на несогласованности и места, менее тщательно отполированные, нежели что-то другое. Некоторые обсуждающиеся в книге вопросы представляют собой результаты современных исследований: библиотеки из глав 4, 5, 6 и 14 возникли за последние несколько лет.

Книга в целом состоит из двух преимущественно независимых частей: первой и второй. Вы можете начать с любой из них или даже постоянно между ними переключаться (то есть читать конкурентно!). Между двумя частями есть только одна зависимость: читать главу 13 имеет смысл после первой части, в особенности перед чтением раздела «Монада ParIO» на с. 292 следует прочитать главу 4.

Главы в пределах одной части лучше читать последовательно. Эта книга не является справочником, в ней есть примеры и темы, которые развиваются на протяжении нескольких глав.

Используемые соглашения

Всюду в книге мы используем следующие типографские соглашения:

Курсив
: используется для выделений, новых терминов, названий команд и утилит Unix, имён файлов и каталогов.

`Моноширинный шрифт`
: обозначает переменные, функции, типы, параметры, объекты и другие конструкции языка программирования.

Так оформляются советы, предложения, уточняющие или обобщающие замечания.

А так мы указываем на известные ловушки, за которыми стоит следить, обычно это что-то, не сразу очевидное.

Примеры программного кода выглядят следующим образом:

timetable1.hs

```
search :: ( partial -> Maybe solution )    -- (1)
       -> ( partial -> [ partial ] )
       -> partial
       -> [solution]
```

В заголовке указано имя исходного файла, в котором этот фрагмент кода содержится; в разделе «Примеры программ» на с. 20 сказано, как можно получить все необходимые файлы. Если приводится несколько фрагментов одного файла, то имя обычно указывается перед первым таким фрагментом.

(1) Таким образом часто будут комментироваться отдельные строки фрагментов кода.

Команды, которые нужно будет вводить в командной строке, будут выглядеть так:

```
$ ./logger
hello
bye
logger: stop
```

Символ `$` обозначает приглашение к вводу, за ним следует имя команды, а всё остальное — вывод этой команды.

Сеанс интерпретатора GHCi выглядит так:

```
> extent arr
(Z :. 3) :. 5
> rank (extent arr)
2
> size (extent arr)
15
```

Приглашение к вводу в GHCi у меня обычно установлено в символ `>`, за которым следует пробел, поскольку принятое по умолчанию в GHCi

приглашение с подключением нескольких модулей становится чрезмерно длинным. Вы можете сделать то же самое, выполнив в GHCi следующую команду:

```
Prelude> :set prompt "> "
>
```

Примеры исходного кода

Сопровождающие эту книгу примеры с исходным кодом доступны в Интернете; подробности относительно их загрузки и сборки приведены в разделе «Примеры программ» на с. 20. Информация о ваших правах на использование, модификацию и распространение исходного кода имеется в файле *LICENSE* в архиве с примерами.

Благодарности

На протяжении нескольких месяцев я занимался исключительной этой книгой, не оставляя времени ни на что другое, поэтому в первую очередь и в наибольшей степени я хотел бы поблагодарить свою жену за её содействие, терпение и особенно за её пирожные в течение всего это времени.

Во-вторых, я в долгу у Саймона Пейтона Джонса, который возглавляет проект GHC с самого его возникновения, он всегда был для меня богатейшим источником вдохновения. Неустанный энтузиазм и техническая проницательность Саймона являются постоянной движущей силой GHC.

Благодарю Мэри Ширан (Mary Sheeran) и Андреса Лоха (Andres Löh), которые в числе других убеждали меня превратить свои заметки в эту книгу, и благодарю также организаторов летних школ CEFP и CEA/EDF/INRIA за приглашение прочитать курсы, что подвигло меня на эту работу, и студентов, слушавших эти курсы и сыгравших роль подопытных свинок.

Я очень благодарен своему редактору Энди Ораму (Andy Oram) и другим сотрудникам O'Reilly, которые помогли превратить эту книгу в реальность.

Перечислю также людей, которые так или иначе помогали мне с этой книгой, рецензируя черновики, посылая мне свои предложения, комментируя опубликованные в сети главы, реализуя код, который я (надеюсь, с должным указанием на авторство) использовал, публикуя записи в блогах, из которых я заимствовал идеи, или делая что-либо ещё (прошу прощения, если я вас забыл): Joey Adams, Lennart Augustsson, Tuncer Ayaz, Jost

Berthold, Manuel Chakravarty, Duncan Coutts, Andrew Cowie, Iavor Diatchki, Chris Dornan, Sigbjorn Finne, Kevin Hammond, Tim Harris, John Hughes, Mikolaj Konarski, Erik Kow, Chris Kuklewicz, John Launchbury, Roman Leshchinskiy, Ben Lippmeier, Andres Löh, Hans-Wolfgang Loidl, Ian Lynagh, Trevor L. McDonell, Takayuki Muranushi, Ryan Newton, Mary Sheeran, Wren ng Thornton, Bryan O'Sullivan, Ross Paterson, Thomas Schilling, Michael Snoyman, Simon Thomson, Johan Tibell, Phil Trinder, Bas Van Dijk, Phil Wadler, Daniel Winograd-Cort, Nicolas Wu и Edward Yang.

Наконец, благодарю всё сообщество Haskell за его дружелюбность, включённость, желание помочь и стимулирование к работе, других таких сетевых сообществ у проектов с открытым кодом я не знаю. Ребята, нам есть чем гордиться, давайте это сохранять.

1. Введение

В программистском сообществе уже довольно давно бытует мнение о том, что программировать с применением потоков и блокировок сложно. Высокий уровень мастерства необходим даже в простых случаях, программы зачастую содержат трудно диагностируемые ошибки. Вместе с тем эти инструменты позволяют выразить весь спектр требующих решения задач: от параллельной обработки изображений до конкурентных веб-серверов, а единый простой интерфейс имеет неоспоримые выгоды. Однако при желании упростить написание параллельного и конкурентного программного обеспечения следует принять идею о том, что разные задачи требуют использования разных инструментов, одного инструмента на всё просто не хватает. Обработка изображений естественным образом выражается через параллельные операции с массивами, тогда как потоки хорошо ложатся на конкурентные веб-серверы.

Так и в Haskell для каждой конкретной задачи мы хотим предоставить наиболее подходящий инструмент, причём сделать это желательно для как можно большего числа типов задач. Если оказывается, что для некоторой задачи инструмента в Haskell нет, мы подыскиваем пути, чтобы его создать. Неизбежное следствие такого разнообразия состоит в том, что требуется многое изучать, и именно для этого написана эта книга. Здесь рассматриваются подходы к написанию параллельных и конкурентных программ от простых примеров применения параллелизма в целях ускорения ориентированных на вычисления программ до использования легковесных потоков в реализации высокопроизводительных конкурентных сетевых серверов. По пути мы также увидим, как на языке Haskell пишутся программы, предназначенные для работы на мощных процессорах современных графических карт (GPU), и программы, запускаемые на отдельных машинах компьютерной сети (распределённое программирование).

Речь не идёт о том, чтобы рассказать о каждой когда-либо возникшей экспериментальной модели программирования. Если вы внимательно посмотрите на пакеты с Hackage, то обнаружите массу библиотек для па-

раллельного и конкурентного программирования, многие из которых написаны исключительно ради удовлетворения любопытства их авторов, не говоря уже о многочисленных исследовательских проектах, пока ещё не готовых к использованию в реальном мире. В этой книге я собираюсь сосредоточиться на интерфейсах, которые уже сейчас можно применять для решения ваших задач, они достаточно стабильны, чтобы на них можно было положиться. Кроме того, я хотел бы обеспечить глубокое понимание принципов работы нижних уровней, чтобы на них можно было бы основывать собственные абстракции, если вдруг таковые понадобятся.

Терминология: параллелизм и конкурентность

Во многих областях знаний термины *параллельный* (*parallel*) и *конкурентный* (*concurrent*) являются синонимами, однако в программировании это не так. Концепции, которым они соответствуют, совершенно разные.

Параллельная программа эксплуатирует множественность вычислительного оборудования (например, ядер процессора) для более быстрого решения вычислительных задач. Цель состоит в сокращении времени получения ответа, и достигается она передачей отдельных фрагментов задачи на разные процессоры, работающие в одно и то же время.

Напротив, *конкурентность* является приёмом структурирования программ, в которых имеются множественные *потоки управления*. Концептуально потоки управления выполняются «одновременно», то есть пользователь может наблюдать переплетение эффектов, возникающих в результате их деятельности. Действительно ли они выполняются одновременно или же нет — деталь реализации, конкурентная программа вполне может исполняться как на одном процессоре с переключением контекстов, так и на множестве физических процессоров.

Если параллельное программирование заботится только об эффективности, то конкурентное ориентируется на структурирование программы, взаимодействующей со множеством независимых внешних агентов (таких как пользователь, сервер баз данных, клиенты извне). Конкурентность способствует *модульности* таких программ: поток, общающийся с пользователем, отделяется от потока, работающего с базой данных. В отсутствие конкурентности такие программы пришлось бы писать посредством циклов обработки событий и функций обратного вызова, что обычно приводит к крайне запутанным монолитным программам.

Понятие «потоков управления» не имеет смысла в чисто функциональных программах, потому что в них нет эффектов, за которыми можно было бы наблюдать, а порядок вычислений не является существенным.

Поэтому конкурентность оказывается способом структурирования кода с эффектами, что в случае языка Haskell означает код в монаде `IO`.

Похожее соотношение имеется между *детерминированной* и *недетерминированной* моделями программирования. В детерминированных моделях каждая программа может выдать только один результат, тогда как недетерминированные допускают программы, приводящие к различным результатам в зависимости от некоторых аспектов выполнения. Конкурентные модели программирования изначально недетерминированные, поскольку в них происходит взаимодействие с внешними агентами, то есть события в программе происходят в непредсказуемые моменты времени. Недетерминизм, однако, имеет значительный недостаток: существенно затрудняются тестирование и формальные рассуждения о корректности программ.

В параллельном программировании мы предпочитаем использовать детерминированные модели, пока это возможно. Поскольку цель состоит просто в более скором получении ответа, мы бы не хотели по ходу усложнять отладку программ. Детерминированное параллельное программирование есть лучший из миров: тестирование, отладка и доказательство корректности могут быть проведены на последовательной версии программы, а с добавлением большего числа процессоров программы просто начинают работать быстрее. На самом деле большинство процессоров сами по себе реализуют детерминированный параллелизм в виде конвейерной обработки и множества исполнительных блоков.

Хотя параллельно программировать можно и с помощью конкурентности, это обычно плохое решение, так как конкурентность приводит к отказу от детерминированности. В языке Haskell большинство моделей параллельного программирования детерминировано. Важно, правда, отметить, что не все виды параллельных алгоритмов можно выразить посредством моделей параллельного программирования, некоторые из них полагаются на внутренний недетерминизм, это в первую очередь задачи, использующие поиск в пространстве решений. Более того, мы часто хотим распараллеливать программы с побочными эффектами, а в этом случае использованию недетерминированного параллельного или конкурентного программирования альтернативы нет.

Наконец, вполне допустимо совмещать параллелизм и конкурентность в рамках одной программы. Большинство интерактивных программ вынуждены использовать конкурентность для реализации пользовательского интерфейса, своевременно реагирующего на действия пользователя, выполняя при этом в фоновом режиме вычислительные задачи.

Инструменты и документация

Чтобы попробовать поработать с программами и упражнениями из этой книги, необходимо установить Haskell Platform[1], включающую компилятор GHC и все важные библиотеки, в том числе и используемые в этой книге библиотеки для параллельного и конкурентного программирования. Код в книге проверялся на версии 2012.4.0.0, но примеры программ будут обновляться с выходом новых версий.

В некоторых главах потребуется установить дополнительные пакеты. Инструкции по установке необходимых зависимостей приводятся в разделе «Примеры программ» на с. 20.

Я также рекомендую установить программу ThreadScope, это средство для визуализации исполнения программ на языке Haskell, оно особенно полезно для понимания сути поведения параллельного и конкурентного кода на Haskell. В Linux-системах утилита ThreadScope, скорее всего, доступна напрямую из репозиториев дистрибутива, и это простейший способ её установить. К примеру, в Ubuntu для этого достаточно выполнить следующую команду:

```
$ sudo apt-get install threadscope
```

Инструкции по установке ThreadScope на других системах можно получить на странице этой программы на сайте языка Haskell[2].

При чтении данной книги я рекомендую иметь под рукой следующую документацию:

- Руководство пользователя GHC[3].
- Документация к библиотекам Haskell Platform[4], тут можно найти информацию по любым функциям, используемым, но не описанным в этой книге.
- Документация для пакетов с Hackage, не входящих в состав Haskell Platform. Для поиска документации по конкретной функции или типу используйте Hoogle[5].

Стоит отметить, что большая часть используемых в этой книге программных интерфейсов не является частью стандарта Haskell 2010. Они поддерживаются дополнительными пакетами, некоторые из которых входят в Haskell Platform, остальные же доступны на Hackage.

[1] `http://www.haskell.org/platform/`.
[2] `http://www.haskell.org/haskellwiki/ThreadScope`.
[3] `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.
[4] `http://lambda.haskell.org/platform/doc/current/frames.html`.
[5] `http://www.haskell.org/hoogle/`.

Примеры программ

Примеры всех программ из этой книги собраны в виде опубликованного на Hackage пакета `parconc-examples`. Чтобы загрузить и распаковать его, выполните:

```
$ cabal unpack parconc-examples
```

Затем установите все зависимости:

```
$ cd parconc-examples
$ cabal install --only-dependencies
```

Наконец, запустите конфигурирование и скомпилируйте все примеры программ:

```
$ cabal configure
$ cabal build
```

Пакет `parconc-examples` будет при необходимости обновляться, следуя будущим изменениям Haskell Platform и других программных интерфейсов.

Часть I

Parallel Haskell

Сейчас, когда производители процессоров по большей части сдались в своём стремлении увеличивать быстродействие отдельных процессоров и взамен этого сфокусировались на предоставлении пользователям большего их числа, наибольший прирост производительности программ приходится достигать с помощью приёмов параллельного программирования, использующих дополнительные вычислительные ядра. Параллельное программирование на Haskell предназначено именно для предоставления доступа ко множеству процессоров наиболее естественным и надёжным образом.

Вам, вероятно, интересно, не может ли компилятор распараллеливать программы автоматически. В конце концов, в чисто функциональном языке, вычисления в котором зависят друг от друга только по входным и выходным данным, а эти зависимости легко увидеть и проанализировать, автоматическое распараллеливание должно быть проще. Тем не менее даже в чисто функциональном языке ему препятствует старая как мир проблема: для ускорения программы нужно сделать так, чтобы выгода от использования параллелизма превзошла накладные расходы, возникающие при его добавлении, тогда как анализ во время компиляции не способен выявить, будет ли это иметь место. Альтернативный подход состоит в динамическом профилировании с целью поиска подходящих для распараллеливания кандидатов и передаче этой информации обратно компилятору. Однако даже это пока не получилось достаточно успешно реализовать на практике.

Полностью автоматическое распараллеливание пока остаётся несбыточной мечтой. Тем не менее модели параллельного программирования, реализуемые в языке Haskell, достаточно успешны в преодолении некоторых неприятных и способствующих возникновению ошибок аспектов, традиционно связываемых с параллельным программированием:

- Параллельное программирование на языке Haskell *детерминировано*. Параллельная программа всегда возвращает один и тот же результат независимо от числа использованных при её выполнении процессоров. Это, в частности, означает, что параллельные программы можно отлаживать, не запуская их параллельно. Более того, программист может быть уверен, что добавление параллелизма не привнесёт в программу условия гонок или взаимные блокировки, избавиться от которых тестированием довольно трудно.

- Параллельные программы на Haskell являются высокоуровневыми и декларативными, в них не приходится напрямую иметь дело с такими механизмами, как *синхронизация* или *передача сообщений*. Программист просто указывает, где должен быть параллелизм, а все детали того, как именно следует исполнять программу, поручаются систе-

ме времени исполнения (*runtime management system*, RTS). В таком подходе есть место как благословению, так и проклятию:

- Чем меньше деталей по исполнению программ, тем более программы абстрактны, а значит, проще для выполнения на широком спектре параллельного аппаратного обеспечения.
- Параллельные программы на Haskell могут пользоваться всеми преимуществами хорошо отлаженных и чётко настроенных технологий системы времени исполнения, например параллельной сборкой мусора. Более того, программа может выиграть от дальнейших улучшений этой системы без каких бы то ни было дополнительных усилий.
- Сокрытие множества деталей исполнения затрудняет выявление сути проблем с производительностью. Более того, у программиста гораздо меньше способов проконтролировать выполнение, нежели при использовании низкоуровневого языка программирования, а значит, сложнее решить проблему. Собственно, эти сложности связаны не только с параллельным программированием на Haskell. Любой, кто пытался оптимизировать программу на Haskell, с ними сталкивался. В этой книге я надеюсь показать, как выявлять и обходить наиболее часто встречающиеся на практике проблемы.

Основное, над чем приходится задумываться параллельному программисту на Haskell, — это деление решаемой задачи на задания, которые можно вычислять параллельно. В идеале хочется иметь достаточное количество заданий, чтобы загрузить непрерывной работой все имеющиеся процессоры. Однако на этом пути подстерегают два препятствия:

Степень детализации
: Если сделать задания слишком маленькими, то накладные расходы на управление ими превысят все выгоды от параллельного исполнения. Поэтому степень детализации должна быть достаточно велика, чтобы нивелировать влияние накладных расходов, но при этом не настолько, чтобы рисковать наличием объёма работы, необходимого для загрузки всех процессоров, что особенно существенно в конце выполнения программы, когда заданий становится меньше.

Зависимости по данным
: Если одно задание зависит от другого, они должны исполняться последовательно. В первых двух моделях программирования, рассмат-

риваемых в этой книге, предпринимаются различные подходы к зависимостям по данным: в третьей главе эти зависимости остаются неявными, тогда как в четвёртой — явно обозначаются. Программирование с явно указанными зависимостями менее лаконично, однако проще для понимания и исправления ошибок.

В следующих главах будут рассматриваться различные модели параллельного программирования, реализованные в языке Haskell:

- Во второй и третьей главах вводятся монада `Eval` и стратегии вычислений, которые можно использовать для описания параллелизма в задачах, где не требуются масштабные численные расчёты и не используются массивы. Эти модели хорошо известны, есть много отличных примеров, которые можно распараллелить с их использованием.

- Четвёртая глава посвящена монаде `Par`, реализующей относительно новую модель параллельного программирования, которая также предназначена для распараллеливания обычного кода на языке Haskell, но с другим уклоном: она позволяет программисту получить большую степень контроля за счёт меньшей лаконичности и модульности, свойственных стратегиям из третьей главы.

- В пятой главе рассматривается библиотека `Repa`, обеспечивающая большой набор комбинаторов для построения параллельных вычислений с массивами. Сложные алгоритмы обработки массивов можно выразить композицией нескольких более простых операций, а композиция, в свою очередь, будет автоматически оптимизирована библиотекой в однопроходный алгоритм с помощью техники, известной как *слияние* (*fusion*). Затем реализация библиотеки автоматически распараллеливает вычисления на имеющиеся процессоры.

- В шестой главе обсуждается программирование графических сопроцессоров (GPU) с помощью библиотеки `Accelerate`, модель программирования которой похожа на реализованную в `Repa`, но позволяет выполнять вычисления непосредственно на GPU.

Распараллеливание кода на Haskell может приносить радость, когда добавление к программе маленькой аннотации вдруг приводит к ускорению в несколько раз на многоядерной машине. Иногда оно может разочаровывать: как будет видно на протяжении нескольких последующих глав, нас поджидают многочисленные ловушки, некоторые связаны исключительно с Haskell, другие присущи параллельному программированию на любом