Оглавление

От редактора перевода

Книга Криса Окасаки «Чисто функциональные структуры данных» даже спустя почти 20 лет после публикации остаётся единственным достаточно полным источником информации по разработке и анализу производительности различных структур данных в функциональном окружении. Разумеется, за прошедшее время было получено множество новых результатов*, однако работы, которая бы их обобщала, к сожалению, пока не появилось. Книга сочетает техническую точность в изложении основных результатов и академическую строгость их обоснования.

Русский перевод этой книги можно считать чрезвычайно удачным дополнением к выпущенной издательством ДМК Пресс в 2013 году книги Ричарда Бёрда «Жемчужины проектирования алгоритмов: функциональный подход», вместе они обеспечивают читателя богатым арсеналом средств для разработки эффективных алгоритмов и структур данных при работе с функциональными языками программирования. Надеемся, что это издание будет способствовать распространению интереса к функциональному программированию в русскоязычном сообществе.

Перевод книги был выполнен Георгием Бронниковым. Благодарим также Романа Кашицына, Всеволода Опарина, Кирилла Заборского, Александра Карпича и Дмитрия Косарева за участие в работе по подготовке настоящего издания.

Виталий Брагилевский,
Институт математики, механики и компьютерных наук
Южного федерального университета, Ростов-на-Дону

*Подробный их перечень можно найти в ответе на вопрос Евгения Кирпичёва по адресу `http://cstheory.stackexchange.com/q/1539/`.

Предисловие

Я впервые познакомился с языком Стандартный ML в 1989 году. Мне всегда нравилось программировать эффективные реализации структур данных, и я немедленно занялся переводом некоторых своих любимых программ на Стандартный ML. Для некоторых структур перевод оказался достаточно простым и, к моему большому удовольствию, получался код значительно более краткий и ясный, чем предыдущие версии, написанные мной на C, Pascal или Ada. Однако не всегда результат оказывался столь приятным. Раз за разом мне приходилось использовать разрушающее присваивание, которое в Стандартном ML не приветствуется, а во многих других функциональных языках вообще запрещено. Я пытался обращаться к литературе, но нашёл лишь несколько разрозненных статей. Понемногу я стал понимать, что столкнулся с неисследованной областью, и начал искать новые способы решения задач.

Сейчас, восемь лет спустя, мой поиск продолжается. Всё ещё есть много примеров структур данных, которые я просто не знаю как эффективно реализовать на функциональном языке. Однако за это время я получил множество уроков о том, что в функциональных языках *работает*. Эта книга является попыткой записать выученные уроки, и я надеюсь, что она послужит справочником для функциональных программистов, а также как текст для тех, кто хочет больше узнать о структурах данных в функциональном окружении.

Стандартный ML. Несмотря на то, что структуры данных из этой книги можно реализовать практически на любом функциональном языке, я во всех примерах буду использовать Стандартный ML. У этого языка имеются следующие преимущества для моих целей: (1) энергичный порядок вычислений, что значительно упрощает рассуждения о том, сколько времени потребует тот или иной алгоритм, и (2) замечательная система модулей, идеально подходящая для выражения абстрактных типов данных. Однако пользователи других языков, например, Haskell или Lisp, смогут без труда адаптировать мои примеры к своим вычислительным окружениям. (В приложении я привожу переводы большинства примеров на Haskell.) Даже программисты на C или Java должны быть способны реализовать эти структуры данных, хотя в случае C отсутствие автоматической сборки мусора иногда будет доставлять неприятности.

Читателям, незнакомым со Стандартным ML, я рекомендую в качестве введения книги *ML для программиста-практика* Полсона [Pau96] или *Элементы программирования на ML* Ульмана [Ull94].

Прочие предварительные требования. Эта книга не рассчитана служить первоначальным общим введением в структуры данных. Я предполагаю, что читателю достаточно знакомы основные абстрактные структуры данных — стеки, очереди, кучи (приоритетные очереди) и конечные отображения (словари). Кроме того, я предполагаю знакомство с основами анализа алгоритмов, особенно с нотацией «большого О» (например, $O(n \log n)$). Обычно эти вопросы рассматриваются во втором курсе для студентов, изучающих информатику.

Благодарности. Моё понимание функциональных структур данных чрезвычайно обогатилось в результате дискуссий со многими специалистами на протяжении многих лет. Мне бы особенно хотелось поблагодарить Питера Ли, Генри Бейкера, Герта Бродала, Боба Харпера, Хаима Каплана, Грэма Мосса, Саймона Пейтона Джонса и Боба Тарьяна.

1. Введение

Когда программисту на C для решения определённой задачи требуется эффективная структура данных, он или она обычно могут просто найти подходящее решение в одном из многих учебников или справочников. К сожалению, для программистов на функциональных языках вроде Стандартного ML или Haskell такая роскошь недоступна. Хотя большинство справочников стараются быть независимы от языка, независимость эта получается только в смысле Генри Форда: программисты свободны выбрать любой язык, если язык этот императивный[1]. Чтобы несколько исправить этот дисбаланс, в этой книге я рассматриваю структуры данных с функциональной точки зрения. В примерах программ я использую Стандартный ML, однако эти программы нетрудно перевести на другие функциональные языки, например, Haskell или Lisp. Версии наших программ на Haskell можно найти в Приложении A.

1.1. Функциональные и императивные структуры данных

Методологические преимущества функциональных языков хорошо известны [Bac78, Hug89, HJ94], но тем не менее большинство программ попрежнему пишутся на императивных языках вроде C. Кажущееся противоречие легко объяснить тем, что исторически функциональные языки проигрывали в скорости своим более традиционным аналогам, однако этот разрыв сейчас сужается. По широкому фронту задач был достигнут впечатляющий прогресс, начиная от базовой техники построения компиляторов и заканчивая глубоким анализом и оптимизацией программ. Однако одну особенность функционального программирования не исправить никакими ухищрениями со стороны авторов компиляторов — использование слабых или несоответствующих задаче структур данных. К сожалению, имеющаяся литература содержит относительно мало рецептов помощи в этой области.

Почему оказывается, что функциональные структуры данных труднее спроектировать и реализовать, чем императивные? Здесь две основные

[1] Генри Форд однажды сказал о цветах автомобилей Модели T: «[Покупатели] могут выбрать любой цвет, при условии, что он чёрный».

проблемы. Во-первых, с точки зрения проектирования и реализации эффективных структур данных, запрет функционального программирования на деструктивное обновление (то есть присваивание) является существенным препятствием, подобно запрету для повара использовать ножи. Как и ножи, деструктивные обновления при неправильном употреблении опасны, но, будучи пущены в дело должным образом, чрезвычайно эффективны. Императивные структуры данных часто существенным образом полагаются на присваивание, так что в функциональных программах приходится искать другие подходы.

Второе затруднение состоит в том, что от функциональных структур ожидается большая гибкость, чем от их императивных аналогов. В частности, когда мы производим обновление императивной структуры данных, мы, как правило, принимаем как данность, что старая версия данных более недоступна, в то время как при обновлении функциональной структуры мы ожидаем, что как старая, так и новая версия доступны для дальнейшей обработки. Структура данных, поддерживающая несколько версий, называется *устойчивой* (persistent), в то время как структура данных, позволяющая иметь лишь одну версию в каждый момент времени, называется *эфемерной* (ephemeral) [DSST89]. Функциональные языки программирования обладают тем интересным свойством, что *все* структуры данных в них автоматически устойчивы. Императивные структуры данных, как правило, эфемерны. В тех случаях, когда требуется устойчивая структура, императивные программисты не удивляются, что она получается более сложной и, возможно, даже асимптотически менее эффективной, чем эквивалентная эфемерная структура.

Более того, теоретики установили нижние границы, которые показывают, что в некоторых ситуациях функциональные языки по своей природе менее эффективны, чем императивные [BAG92, Pip96]. В свете перечисленного, функциональные структуры данных иногда кажутся похожими на танцующего медведя, о котором говорится: «поразительна не красота его танца, а то, что он вообще танцует!» Однако на практике ситуация совсем не так безнадёжна. Как мы увидим, часто оказывается возможным построить функциональные структуры данных, асимптотически столь же эффективные, как лучшие императивные решения.

1.2. Энергичное и ленивое вычисление

Большинство (последовательных) функциональных языков программирования можно отнести либо к *энергичным* (strict), либо к *ленивым* (lazy), в зависимости от порядка вычислений. Какой из этих порядков

предпочтительнее — тема, обсуждаемая функциональными программистами подчас с религиозным жаром. Различие между двумя порядками вычисления наиболее ярко проявляется в подходах к вычислению аргументов функции. В энергичных языках аргументы вычисляются прежде тела функции. В ленивых языках вычисление аргументов управляется потребностью; изначально они передаются в функцию в невычисленном виде, и вычисляются только тогда, когда (и если!) их значение нужно для продолжения работы. Кроме того, после однократного вычисления значение аргумента кэшируется, так что если оно потребуется снова, его можно получить из памяти, а не перевычислять заново. Такое кэширование называется *мемоизация* (memoization) [Mic68].

Каждый из этих порядков имеет свои достоинства и недостатки, но энергичное вычисление явно удобнее по крайней мере в одном отношении: с ним проще рассуждать об асимптотической сложности вычислений. В энергичных языках то, какие именно подвыражения будут вычислены и когда, ясно по большей части уже из синтаксиса. Таким образом, рассуждения о времени выполнения каждой данной программы относительно просты. В то же время в ленивых языках даже эксперты часто испытывают сложности при ответе на вопрос, когда будет вычислено данное подвыражение и будет ли вычислено вообще. Программисты на таких языках часто вынуждены притворяться, что язык на самом деле энергичен, чтобы получить хотя бы грубые оценки времени работы.

Оба порядка вычисления влияют на проектирование и анализ структур данных. Как мы увидим, энергичные языки могут описать структуры с жёсткой оценкой времени выполнения в худшем случае, но не с амортизированной оценкой, а в ленивых языках описываются амортизированные структуры данных, но не рассчитанные на худший случай. Чтобы описывать обе разновидности структур, требуется язык, поддерживающий оба порядка вычислений. Мы получаем такой язык, расширяя Стандартный ML примитивами для ленивого вычисления, как описано в главе 4.

1.3. Терминология

Любой разговор о структурах данных содержит опасность возникновения путаницы, поскольку у термина *структура данных* (data structure) есть по крайней мере четыре различных связанных между собой значения.

- *Абстрактный тип данных* (то есть *тип и набор функций над этим типом*). Для этого значения мы будем пользоваться словом *абстракция* (abstraction).

- *Конкретная реализация абстрактного типа данных.* Для этого значения мы используем слово *реализация* (implementation). Однако от реализации мы не требуем воплощения в коде — достаточно детального проекта.

- *Экземпляр типа данных, например, конкретный список или дерево.* Для такого экземпляра мы будем использовать слово *объект* (object) или *версия* (version). Впрочем, для конкретных типов часто бывает свой термин. Например, стеки и очереди мы будем называть просто стеками и очередями.

- *Сущность, сохраняющая свою идентичность при изменениях.* Например, в интерпретаторе, построенном на основе стека, мы часто говорим о «стеке», как если бы это был один объект, а не различные версии в различные моменты времени. Для этого значения мы будем использовать выражение *устойчивая сущность* (persistent identity). Нужда в этом возникает прежде всего при разговоре об устойчивых структурах данных; когда мы говорим о различных версиях одной и той же структуры, мы имеем в виду, что они все имеют одну и ту же устойчивую сущность.

Грубо говоря, абстракциям в Стандартном ML соответствуют сигнатуры, реализациям — структуры или функторы, а объектам или версиям — значения. Хорошего аналога понятию устойчивой сущности в Стандартном ML нет[2].

Термин *операция* (operation) перегружен подобным же образом; он обозначает и функции, предоставляемые абстрактным типом данных, и конкретные применения этих функций. Мы пользуемся словом *операция* только во втором значении, а для первого употребляем слова *функция* (function) или *оператор* (operator).

1.4. Наш подход

Вместо того, чтобы каталогизировать структуры данных, подходящие для каждой возможной задачи (безнадёжное предприятие!), мы сосредоточим внимание на нескольких общих методиках проектирования эффективных функциональных структур данных, и каждую такую методику будем иллюстрировать одной или несколькими реализациями базовых абстрак-

[2]Устойчивая сущность эфемерной структуры данных может быть реализована как ссылочная ячейка, но для моделирования устойчивой сущности устойчивой структуры данных такого подхода недостаточно.

ций, таких, как последовательность, куча (очередь с приоритетами) или структуры для поиска. Когда читатель овладел этими методиками, он может с лёгкостью их приспособить к собственным нуждам, или даже спроектировать новые структуры с нуля.

1.5. Обзор книги

Книга состоит из трёх частей. Первая (главы 2 и 3) служит введением в функциональные структуры данных.

- В главе 2 обсуждается, как функциональные структуры данных добиваются устойчивости.
- Глава 3 описывает три хорошо известных структуры данных — левоориентированные кучи, биномиальные кучи и красно-чёрные деревья, — и показывает, как их можно реализовать на Стандартном ML.

Вторая часть (главы 4–7) посвящена соотношению между ленивым вычислением и амортизацией.

- В главе 4 кратко рассматриваются основные понятия ленивого вычисления и вводится синтаксис, которым мы пользуемся для описания ленивых вычислений в Стандартном ML.
- Глава 5 служит введением в основные методы амортизации. В ней объясняется, почему эти методы не работают при анализе устойчивых структур данных.
- Глава 6 описывает связующую роль, которую ленивое вычисление играет при сочетании амортизации и устойчивости, и даёт два метода анализа амортизированной стоимости структур данных, реализованных через ленивое вычисление.
- В главе 7 демонстрируется, какую выразительную мощь даёт сочетание энергичного и ленивого вычисления в одном языке. Мы показываем, как во многих случаях можно получить структуру данных с жёсткими характеристиками производительности из структуры с амортизированными характеристиками, если систематически запускать преждевременное вычисление ленивых компонент структуры.

В третьей части книги (главы 8–11) исследуется несколько общих методик построения функциональных структур данных.

- В главе 8 описывается *ленивая перестройка* (lazy rebuilding), вариант идеи *глобальной перестройки* (global rebuilding) [Ove83]. Ленивая перестройка значительно проще глобальной, но в результате получаются структуры с амортизированными, а не с жёсткими характеристиками. Сочетание ленивой перестройки с методиками планирования из главы 7 часто позволяет восстановить жёсткие характеристики.

- В главе 9 исследуются *числовые представления* (numerical representations) — представления данных, построенные по аналогии с представлениями чисел (как правило, двоичных чисел). В этой модели нахождение эффективных процедур вставки и изъятия соответствует выбору таких вариантов двоичных чисел, где добавление или вычитание единицы занимает константное время.

- Глава 10 рассматривает *развёртку структур данных* (data-structural bootstrapping) [Buc93]. Эта методика существует в трёх вариантах: *структурная декомпозиция* (structural decomposition), когда решения без ограничений строятся на основе ограниченных решений, *структурная абстракция* (structural abstraction), когда эффективные решения строятся на основе неэффективных, и *развёртка до составных типов* (bootstrapping to aggregate types), когда реализации с атомарными элементами развёртываются до реализаций с составными элементами.

- В главе 11 описывается *неявное рекурсивное замедление* (implicit recursive slowdown), ленивый вариант метода *рекурсивного замедления* (recursive slowdown) Каплана и Тарьяна [KT95]. Подобно ленивой перестройке, неявное рекурсивное замедление значительно проще обычного рекурсивного замедления, но вместо жёстких характеристик даёт лишь амортизированные. Как и в случае ленивой перестройки, жёсткие характеристики зачастую можно восстановить с помощью расписаний.

Наконец, Приложение A включает в себя перевод большинства программных реализаций этой книги на Haskell.

2. Устойчивость

Отличительной особенностью функциональных структур данных является то, что они всегда *устойчивы* (persistent) — обновление функциональной структуры не уничтожает старую версию, а создаёт новую, которая с ней сосуществует. Устойчивость достигается путём *копирования* затронутых узлов структуры данных, и все изменения проводятся на копии, а не на оригинале. Поскольку узлы никогда напрямую не модифицируются, все незатронутые узлы могут *совместно использоваться* (be shared) между старой и новой версией структуры данных без опасения, что изменения одной версии непроизвольно окажутся видны другой.

В этой главе мы исследуем подробности копирования и совместного использования для двух простых структур данных: списков и двоичных деревьев.

2.1. Списки

Мы начинаем с простых связанных списков, часто встречающихся в императивном программировании и вездесущих в функциональном. Основные функции, поддерживаемые списками, в сущности те же, что и для абстракции стека, описанной в виде сигнатуры на Стандартном ML на рис. 2.1. Списки и стеки можно тривиально реализовать либо с помощью встроенного типа «список» (рис. 2.2), либо как отдельный тип (рис. 2.3).

Замечание. Сигнатура на рис. 2.1 использует терминологию списков (`cons`, `head`, `tail`), а не стеков (`push`, `top`, `pop`), потому что мы рассматриваем списки как частный случай общего класса последовательностей. Другими примерами этого класса являются *очереди*, *двусторонние очереди* и *списки с конкатенацией*. Для функций во всех этих абстракциях мы используем одинаковые соглашения по именованию, чтобы можно было заменять одну реализацию другой с минимальными трудностями.

К этой сигнатуре мы могли бы добавить ещё одну часто встречающуюся операцию на списках: $+\!\!+$, которая конкатенирует (то есть соединяет) два списка. В императивной среде эту функцию нетрудно реализовать за время $O(1)$, если сохранять указатели и на первый, и на последний элемент списка. Тогда $+\!\!+$ просто изменяет последнюю ячейку первого списка

```
signature STACK =
sig
  type α Stack
  val empty   : α Stack
  val isEmpty : α Stack → bool
  val cons    : α × α Stack → α Stack
  val head    : α Stack → α Stack
      (* возбуждает EMPTY для пустого стека *)
  val tail    : α Stack → α Stack
      (* возбуждает EMPTY для пустого стека *)
end
```

Рис. 2.1: сигнатура стеков.

```
structure List: STACK =
struct
  type α Stack = α list
  val empty = []
  fun isEmpty s = null s
  fun cons (x, s) = x :: s
  fun head s = hd s
  fun tail s = tl s
end
```

Рис. 2.2: реализация стека с помощью встроенного типа списков.

```
structure CustomStack: STACK =
struct
  datatype α Stack = NIL | CONS of α × α Stack
  val empty = NIL
  fun isEmpty NIL = true | isEmpty _ = false
  fun cons (x,s) = CONS (x, s)
  fun head NIL = raise EMPTY
    | head (CONS (x,s)) = x
  fun tail NIL = raise EMPTY
    | tail (CONS (x,s)) = s
end
```

Рис. 2.3: реализация стека в виде отдельного типа.

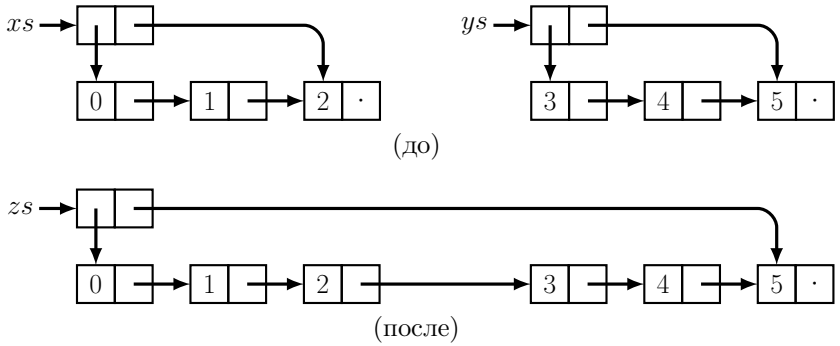

Рис. 2.4: выполнение zs = xs ++ ys в императивном окружении (эта операция уничтожает списки-аргументы xs и ys).

так, чтобы она указывала на первую ячейку второго списка. Результат этой операции графически показан на рис. 2.4. Обратите внимание, что эта операция *уничтожает* оба своих аргумента — после выполнения xs ++ ys ни xs, ни ys использовать больше нельзя.

В функциональном окружении мы не можем деструктивно модифицировать последнюю ячейку первого списка. Вместо этого мы копируем эту ячейку и модифицируем хвостовой указатель в ячейке-копии. Затем мы копируем предпоследнюю ячейку и модифицируем её хвостовой указатель, указывая на копию последней ячейки. Такое копирование продолжается, пока не окажется скопирован весь список. Процесс в общем виде можно реализовать как

fun xs ++ ys = **if** isEmpty xs **then** ys **else** cons (head xs, tail xs ++ ys)

Если у нас есть доступ к реализации нашей структуры (например, в виде встроенных списков Стандартного ML), мы можем переписать эту функцию через сопоставление с образцом:

```
fun  [] ++ ys = ys
   | (x :: xs) ++ ys = x :: (xs ++ ys)
```

На рис. 2.5 изображён результат конкатенации двух списков. Обратите внимание, что после выполнения операции мы можем продолжать использовать два исходных списка, xs и ys. Таким образом, мы добиваемся устойчивости, но за счёт копирования ценой $O(n)$[1].

[1] В главах 10 и 11 мы увидим, как можно реализовать ++ за время $O(1)$ без потери устойчивости.

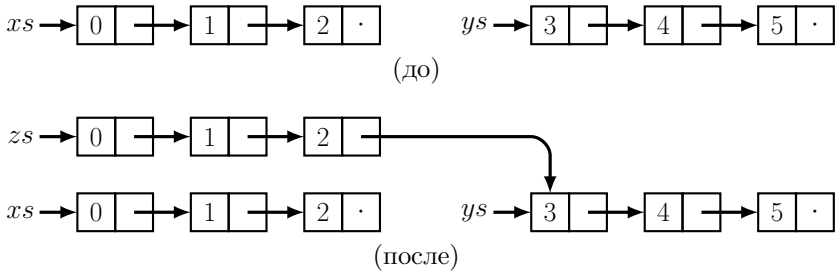

Рис. 2.5: выполнение zs = xs ++ ys в функциональном окружении (заметим, что списки-аргументы xs и ys не затронуты операцией).

Хотя объём копирования довольно большой, тем не менее второй список, ys, нам копировать не пришлось. Эти узлы теперь общие между ys и zs. Ещё одна функция, иллюстрирующая парные понятия копирования и общности подструктур — update, изменяющая значение узла списка по данному индексу. Эту функцию можно реализовать как

```
fun update ([], i, y) = raise Subscript
  | update (x :: xs, 0, y) = y :: xs
  | update (x :: xs, i, y) = x :: update(xs, i−1, y)
```

Здесь мы не копируем весь список-аргумент. Копировать приходится только сам узел, подлежащий модификации (узел i) и узлы, содержащие прямые или косвенные указатели на i. Другими словами, чтобы изменить один узел, мы копируем все узлы на пути от корня к изменяемому. Все узлы, не находящиеся на этом пути, используются как исходной, так и обновлённой версиями. На рис. 2.6 показан результат изменения третьего узла в пятиэлементном списке: первые три узла копируются, а последние два используются совместно.

Замечание. Такой стиль программирования очень сильно упрощается при наличии автоматической сборки мусора. Очень важно освободить память от тех копий, которые больше не нужны, однако многочисленные совместно используемые узлы делают ручную сборку мусора нетривиальной задачей.

Упражнение 2.1. Реализуйте функцию suffixes : α list $\to \alpha$ list list, которая принимает как аргумент список xs и возвращает список всех его суффиксов в порядке убывания длины. Например,

```
suffixes [1,2,3,4] = [[1,2,3,4],[2,3,4],[3,4],[4],[]]
```

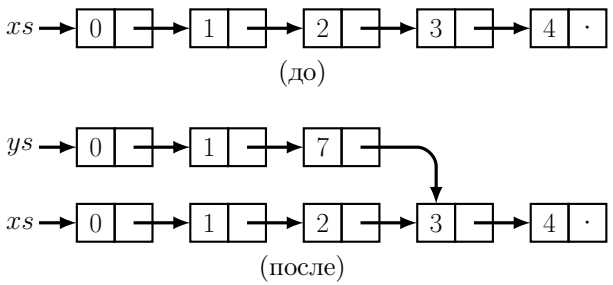

Рис. 2.6: выполнение ys = update(xs, 2, 7) (обратите внимание на совместное использование структуры списками xs и ys).

Покажите, что список суффиксов можно породить за время $O(n)$, используя при этом $O(n)$ памяти.

2.2. Двоичные деревья поиска

Если узел структуры содержит более одного указателя, оказываются возможны более сложные сценарии совместного использования памяти. Хорошим примером совместного использования такого вида служат двоичные деревья поиска.

Двоичные деревья поиска — это двоичные деревья, в которых элементы хранятся во внутренних узлах в *симметричном* (symmetric) порядке, то есть элемент в каждом узле больше любого элемента в левом поддереве этого узла и меньше любого элемента в правом поддереве. В Стандартном ML мы представляем двоичные деревья поиска при помощи следующего типа:

datatype Tree = E | T **of** Tree × Elem × Tree

где Elem — какой-либо фиксированный полностью упорядоченный тип элементов.

Замечание. Двоичные деревья поиска не являются полиморфными относительно типа элементов, поскольку в качестве элементов не может выступать любой тип — подходят только типы, снабжённые отношением полного порядка. Однако это не значит, что для каждого типа элементов мы должны заново реализовывать деревья двоичного поиска. Вместо этого мы делаем тип элементов и прилагающиеся к нему функции сравнения пара-

```
signature SET =
sig
  type Elem
  type Set
  val empty  : Set
  val insert : Elem × Set → Set
  val member : Elem × Set → bool
end
```

Рис. 2.7: сигнатура множеств.

метрами *функтора* (functor), реализующего двоичные деревья поиска (см. рис. 2.9).

Мы используем это представление для реализации множеств. Однако оно легко адаптируется и для других абстракций (например, конечных отображений) или поддержки более оригинальных функций (скажем, найти i-й по порядку элемент), если добавить в конструктор T дополнительные поля.

На рис. 2.7 показана минимальная сигнатура для множеств. Она содержит значение «пустое множество», а также функции добавления нового элемента и проверки на членство. В более практической реализации, вероятно, будут присутствовать и многие другие функции, например, для удаления элемента или перечисления всех элементов.

Функция member ищет в дереве, сравнивая запрошенный элемент с находящимся в корне дерева. Если запрошенный элемент меньше корневого, мы рекурсивно ищем в левом поддереве. Если он больше, рекурсивно ищем в правом поддереве. Наконец, в оставшемся случае запрошенный элемент равен корневому, и мы возвращаем значение «истина». Если мы когда-либо натыкаемся на пустое дерево, значит, запрашиваемый элемент не является членом множества, и мы возвращаем значение «ложь». Эта стратегия реализуется так:

```
fun member (x, E) = false
  | member (x, T (a, y, b)) =
      if x <y then member (x, a)
      else if x >y then member (x, b)
      else true
```

Замечание. Простоты ради, мы предполагаем, что функции сравнения называются $<$ и $>$. Однако если эти функции передаются в качестве пара-

метров функтора, как на рис. 2.9, часто оказывается удобнее называть их именами вроде lt или leq, а символы < и > оставить для сравнения целых и других элементарных типов.

Функция insert проводит поиск в дереве по той же стратегии, что и member, но только по пути она копирует каждый элемент. Когда оказывается достигнут пустой узел, он заменяется на узел, содержащий новый элемент.

```
fun insert (x, E) = T (E, x, E)
  | insert (x, s as T (a, y, b)) =
      if x <y then T (insert (x, a),y, b)
      else if x >y then T (a, y, insert (x, b))
      else s
```

На рис. 2.8 показана типичная вставка. Каждый скопированный узел использует одно из поддеревьев совместно с исходным деревом, а именно то, которое не оказалось на пути поиска. Для большинства деревьев путь поиска содержит лишь небольшую долю узлов в дереве. Громадное большинство узлов находятся в совместно используемых поддеревьях.

На рис. 2.9 показано, как двоичные деревья поиска можно реализовать в виде функтора на Стандартном ML. Функтор принимает тип элементов и связанные с ним функции сравнения как параметры. Поскольку часто те же самые параметры будут использоваться и другими функторами (см., например, упражнение 2.6), мы упаковываем их в структуру с сигнатурой ORDERED.

Упражнение 2.2. (Андерсон [And95]) В худшем случае member производит $2d$ сравнений, где d — глубина дерева. Перепишите её так, чтобы она делала не более $d+1$ сравнений, сохраняя элемент, который *может* оказаться равным запрашиваемому (например, последний элемент, для которого операция < вернула значение «истина» или $\leqslant$ — «ложь»), и производя проверку на равенство только по достижении дна дерева.

Упражнение 2.3. Вставка уже существующего элемента в двоичное дерево поиска копирует весь путь поиска, хотя скопированные узлы неотличимы от исходных. Перепишите insert так, чтобы она избегала копирования с помощью исключений. Установите только один обработчик исключений для всей операции поиска, а не по обработчику на итерацию.

Упражнение 2.4. Совместите улучшения из предыдущих двух упражнений, и получите версию insert, которая не делает ненужного копирования и использует не более $d+1$ сравнений.

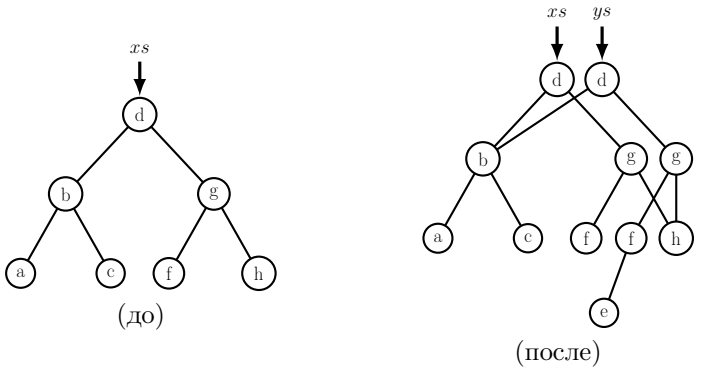

Рис. 2.8: выполнение ys = insert ("e", xs) (как и прежде, обратите внимание на совместное использование структуры деревьями xs и ys).

```
signature ORDERED =
  (* полностью упорядоченный тип и его функции сравнения *)
sig
  type T
  val eq  : T × T → bool
  val lt  : T × T → bool
  val leq : T × T → bool
end

functor UnbalancedSet (Element: ORDERED): SET =
struct
  type Elem = Element.T
  datatype Tree = E | T of Tree × Elem × Tree
  type Set = Tree
  val empty = E
  fun member (x, E) = false
    | member (x, T (a, y, b)) =
        if Element.lt (x, y) then member (x, a)
        else Element.lt (y, x) then member (x, b)
        else true
  fun insert (x, E) = T (E, x, E)
    | insert (x, s as T (a, y, b)) =
        if Element.lt (x, y) then T (insert (x, a), y, b)
        else Element.lt (y, x) then T (a, y, insert (x, b))
        else s
end
```

Рис. 2.9: реализация двоичных деревьев поиска в виде функтора.

```
signature FiniteMap =
sig
  type Key
  type α Map
  val empty : α Map
  val bind  : Key × α × α Map → α Map
  val lookup : Key × α Map → α (* NotFound, если ключ не найден *)
end
```

Рис. 2.10: сигнатура для конечных отображений.

Упражнение 2.5. Совместное использование может быть полезно и внутри одного объекта, не обязательно между двумя различными. Например, если два поддерева одного дерева идентичны, их можно представить одним и тем же деревом.

(а) Воспользуйтесь этой идеей, написав такую функцию complete типа Elem × Int → Tree, что complete (x,d) создаёт полное двоичное дерево глубины d, где в каждом узле содержится x. (Разумеется, такая функция бессмысленна для абстракции множества, но она может оказаться полезной для какой-либо другой абстракции, например, мультимножества.) Функция должна работать за время $O(d)$.

(б) Расширьте свою функцию, чтобы она строила сбалансированные деревья произвольного размера. Эти деревья не всегда будут полны, но они должны быть как можно более сбалансированными: для любого узла размеры поддеревьев должны различаться не более чем на единицу. Функция должна работать за время $O(\log n)$. (Подсказка: воспользуйтесь вспомогательной функцией create2, которая, получая размер m, создаёт пару деревьев — одно размера m, а другое размера $m + 1$.)

Упражнение 2.6. Измените функтор UnbalancedSet так, чтобы он служил реализацией не множеств, а *конечных отображений* (finite maps). На рис. 2.10 приведена минимальная сигнатура для конечных отображений. (Заметим, что исключение NotFound не является встроенным в Стандартный ML — вам придётся определить его самостоятельно. Это исключение можно было бы сделать частью сигнатуры FiniteMap, чтобы каждая реализация определяла собственное исключение NotFound, но удобнее, если все конечные отображения будут использовать одно и то же исключение.)

2.3. Примечания

Майерс [Mye82, Mye84] использовал копирование и совместное использование при реализации двоичных деревьев поиска (в его случае это были AVL-деревья). Для общего метода реализации устойчивых структур данных путём копирования затронутых узлов Сарнак и Тарьян [ST86a] выбрали термин *копирование путей* (path copying). Существуют также другие методы реализации устойчивых структур данных, предложенные Дрисколлом, Сарнаком, Слейтором и Тарьяном [DSST89] и Дитцем [Die89], но эти методы не являются чисто функциональными.