Чисто функциональные структуры данных

Крис Окасаки

22 марта 2015 г.

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 нашел 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» (напр., $O(n \log n)$). Обычно эти вопросы рассматриваются во втором курсе для студентов, изучающих информатику.

Благодарности. Мое 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 Пейтон Джонса и Боба Тарждана.

Глава 1

Введение

Когда программисту на C для решения определенной задачи требуется эффективная структура данных, часто он или она 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 по-прежнему 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 несоответ-

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

ствующих 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 проблемы. Во-первых, с точки зрения проектирования и реализации эффективных структур данных, запрет функционального программирования на деструктивное обновление (т., е., 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 безнадежна. 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; исходно 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 асимптотичееской 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 (безнадежное предприятие!), мы сосредоточим внимание на нескольких общих методиках проектирования эффективных функциональных структур данных, и каждую такую методику будем иллюстрировать одной или несколькими реализациями базовых абстракций, таких, как последовательность, куча (очередь с приоритетами) или структуры для поиска. Когда читатель овладел этими методиками, он может с легкостью их приспособить к собственным нуждам, или даже спроектировать новые структуры с нуля.

1.5 Обзор книги

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

- В Главе 2 обсуждается, как функциональные структуры данных добиваются устойчивости.

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

- Глава 3 описывает три хорошо известных структуры данных — левоориентированные кучи, биномиальные кучи и красно-черные 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. 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 дает два метода анализа амортизированной стоимости структур данных, реализованных через ленивое вычисление.

- В Главе 7 демонстрируется, какую выразительную мощь дает 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 трех 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 дает лишь амортизированные. Как и в случае ленивой перестройки, часто жёсткие характеристики можно восстановить с помощью расписаний.

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

Глава 2

Устойчивость

Отличительной особенностью функциональных структур данных является то, что они всегда *устойчивы* (persistent) — обновление функциональной структуры не уничтожает старую версию, а создает новую, которая с ней сосуществует. Устойчивость достигается путем *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 2.1 *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.*

```
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 для стеков.

```
structure List : Stack = structure
  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 = structure
  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.

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

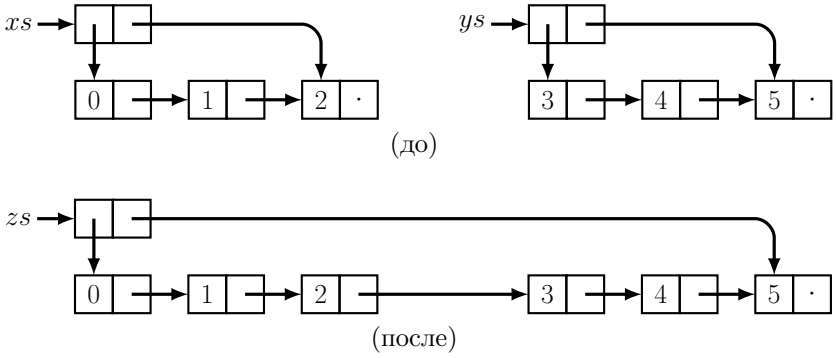


Рис. 2.4: Выполнение xs++ys в императивной среде. Эта операция уничтожает списки-аргументы xs и ys.

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 ее 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 изображен 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 счет копирования ценой $O(n)$.[1]

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

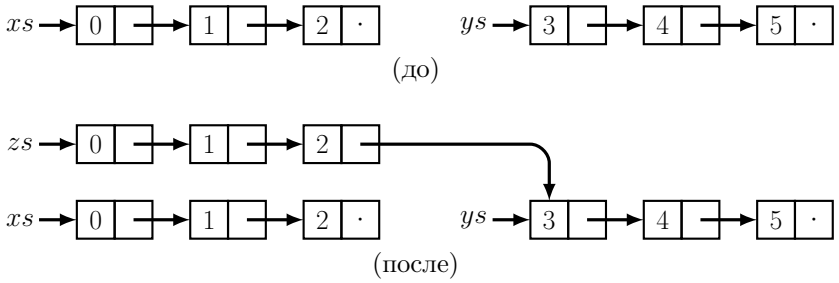


Рис. 2.5: Выполнение zs = xs ++ ys в функциональной среде. Заметим, что списки-аргументы xs и ys не затронуты операцией.

Несмотря на большой объем копирования, заметим, что 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 обновленной версиями. На Рис. 2.6 показан результат изменения третьего узла в пятиэлементном списке: первые три узла копируются, а последние два используются совместно.

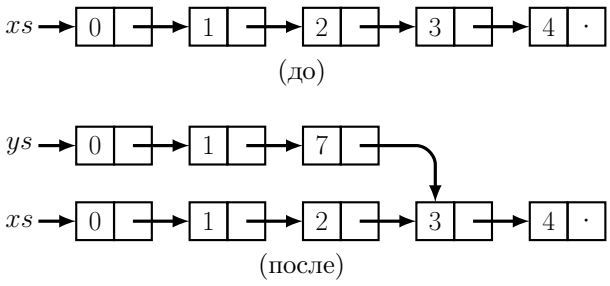


Рис. 2.6: Выполнение ys = update(xs, 2, 7). Обратите внимание на совместное использование структуры списками xs и ys.

Замечание 2.2 *Такой стиль программирования очень сильно упрощается при наличии автоматической сборки мусора. Очень важно освободить память от тех копий, которые больше не нужны, но многочисленные совместно используемые узлы делают ручную сборку мусора нетривиальной задачей.*

Упражнение 2.1 *Напишите функцию suffixes типа* α *list* $\to$ α *list list, которая принимает как аргумент список xs и возвращает список всех его суффиксов в убывающем порядке длины. Например,*

$$suffixes\ [1,2,3,4] = [[1,2,3,4],[2,3,4],[3,4],[4],[]]$$

Покажите, что список суффиксов можно породить за время $O(n)$ *и занять 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 $\times$ Elem $\times$ Tree

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

Замечание 2.3 *Двоичные деревья поиска не являются полиморфными относительно типа элементов, поскольку в качестве элементов не может выступать любой тип — подходят только типы, снабженные 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 параметрами 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 изысканных функций (скажем, найти i-й по порядку элемент), если добавить в конструктор T дополнительные поля.

```
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 для множеств.

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 2.4 *Простоты ради, мы предполагаем, что функции сравнения называются < и >. Однако если эти функции передаются в качестве параметров 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 наконец 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; речь о том поддереве, 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.

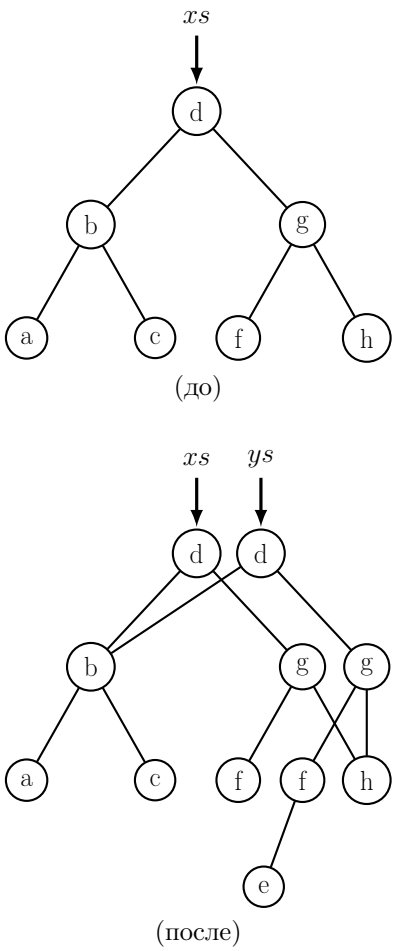


Рис. 2.8: Выполнение ys = insert ("e", xs). Как и прежде, обратите внимание на совместное использвание структуры деревьями xs и ys.

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

```
signature ORDERED = sig
  /* Полностью упорядоченный тип и его функции сравнения */
  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 на Стандартном ML.

Ordered.

Упражнение 2.2 ***Андерсон [And95]*** *В худшем случае member производит $2d$ сравнений, где d — глубина дерева. Перепишите ее 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 $\leq$ — «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.*

```
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 /* возбуждает 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.*

1. *Используя эту идею, напишите функцию complete типа Elem × Int → Tree, такую, что complete(x,d) создает 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 $O(d)$.*

2. *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, создает 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 придется его определить 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 путем 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.

Глава 3

Некоторые известные структуры данных в функциональном окружении

Хотя реализовать в функциональной среде многие императивные структуры данных трудно или невозможно, есть и такие, которые реализуются без особых усилий. В этой главе мы рассматриваем три структуры данных, которым обычно учат в императивном контексте. Первая из них, левоориентированные кучи, просто устроена и в том, и в другом окружении. Однако две других, биномиальные очереди и красно-чёрные деревья, часто считаются сложными для понимания, поскольку их императивные реализации быстро превращаются в мешанину манипуляций с указателями. Напротив, функциональные реализации этих структур данных абстрагируются от действий с указателями и прямо отражают высокоуровневые представления. Дополнительное преимущество функциональной реализации этих структур состоит в том, что мы бесплатно получаем устойчивость.

3.1 Левоориентированные кучи

Как правило, множества и конечные отображения поддерживают эффективный доступ к произвольным элементам. Однако иногда требуется эффективный доступ только к *минимальному* элементу. Структура данных, поддерживающая такой режим доступа, называется *очередь с приоритетами* (priority queue) или *куча* (heap). Чтобы избежать путаницы с очередями FIFO, мы будем использовать второй из этих терминов. На Рис. 3.1 приведена простая сигнатура для кучи.

Замечание 3.1 *Сравнивая сигнатуру кучи с сигнатурой множества (Рис. 2.7), мы видим, что для кучи отношение порядка на элементах включено в сигнатуру, а для множества нет. Это различие вытекает из того, что*

```
signature HEAD = sig
  structure Elem: ORDERED
  type Heap

  val empty     : Heap
  val isEmpty   : Heap → bool
  val insert    : Elem.T × Heap → Heap
  val merge     : Heap × Heap → Heap

  val findMin   : Heap → Elem.T /* возбуждает EMPTY при пустой куче */
  val deleteMin : Heap → Elem.T /* возбуждает EMPTY при пустой куче */
end
```

Рис. 3.1: Сигнатура для кучи (очереди с приоритетами).

отношение порядка играет важную роль в семантике кучи, а в семантике множества не играет. С другой стороны, можно утверждать, что в семантике множества большую роль играет отношение равенства*, и оно должно быть включено в сигнатуру.*

Часто кучи реализуются через деревья *с порядком кучи* (heap-ordered), т. е., в которых элемент при каждой вершине не больше элементов в поддеревьях. При таком упорядочении минимальный элемент дерева всегда находится в корне.

Левоориентированные кучи [Cra72, Knu73a] представляют собой двоичные деревья с порядком кучи, обладающие свойством *левоориентированности* (leftist property): ранг любого левого поддерева не меньше ранга его сестринской правой вершины. Ранг узла определяется как длина его *правой периферии* (right spine) (т. е., самого правого пути от данного узла до пустого). Простым следствием свойства левоориентированности является то, что правая периферия любого узла — кратчайший путь от него к пустому узлу.

Упражнение 3.1 *Докажите, что правая периферия левоориентированной кучи размера n всегда содержит не более $\lfloor \log(n+1) \rfloor$ элементов. (В этой книге все логарифмы, если не указано обратного, берутся по основанию 2.)*

Если у нас есть некоторая структура упорядоченных элементов Elem, мы можем представить левоориентированные кучи как двоичные деревья, снабженные информацией о ранге.

```
datatype Heap = E | T of int × Elem.T × Heap × Heap
```

Заметим, что элементы правой периферии левоориентированной кучи (да и любого дерева с порядком кучи) расположены в порядке возрастания. Главная идея левоориентированной кучи заключается в том, что для слияния двух куч достаточно слить их правые периферии как упорядоченные

списки, а затем вдоль полученного пути обменивать местами поддеревья при вершинах, чтобы восстановить свойство левоориентированности. Это можно реализовать следующим образом:

```
fun merge (h, E) = h
  | merge (E, h) = h
  | merge (h1 as T(_, x, a1, b1), h2 as T(_, y, a2, b2)) =
      if Elem.leq (x, y) then makeT (x, a1, merge (b1, h2))
      else makeT (y, a2, merge (h1, b2))
```

где makeT — вспомогательная функция, вычисляющая ранг вершины T и, если необходимо, меняющая местами ее поддеревья.

```
fun rank (E) = 0
  | rank (T (r, _, _, _)) = r
fun makeT (x, a, b) = if rank a ≥ rank b then T (rank b +1, x, a, b)
                                         else T (rank a + 1, x, b, a)
```

Поскольку длина правой периферии любой вершины в худшем случае логарифмическая, merge выполняется за время $O(\log n)$.

Теперь, когда у нас есть эффективная функция merge, оставшиеся функции не представляют труда: insert создает одноэлементную кучу и сливает ее с существующей, findMin возвращает корневой элемент, а deleteMin отбрасывает корневой элемент и сливает его поддеревья.

```
fun insert (x, h) = merge (T (1, x, E, E), h)
fun findMin (T (_, x, a, b)) = x
fun deleteMin (T (_, x, a, b)) = merge (a, b)
```

Поскольку merge выполняется за время $O(\log n)$, столько же занимают и insert с deleteMin. Очевидно, что findMin выполняется за $O(1)$. Полная реализация левоориентированных куч приведена на Рис. 3.2 в виде функтора, принимающего в качестве параметра структуру упорядоченных элементов.

Замечание 3.2 *Чтобы не перегружать примеры мелкими деталями, мы обычно в фрагментах кода пропускаем варианты, ведущие к ошибкам. Например, приведенные выше фрагменты не показывают поведение findMin и deleteMin на пустых кучах. Когда дело доходит до полной реализации, как на Рис. 3.2, мы всегда включаем в нее разбор ошибок.*

Упражнение 3.2 *Определите insert напрямую, а не через обращение к merge.*

Упражнение 3.3 *Реализуйте функцию fromList типа Elem.T list → Heap, порождающую левоориентированную кучу из неупорядоченного списка элементов путем преобразования каждого элемента в одноэлементную кучу, а затем слияния получившихся куч, пока не останется одна. Вместо того, чтобы сливать кучи проходом слева направо или справа налево при помощи foldr или foldl, слейте кучи за $\lceil \log n \rceil$ проходов, где на каждом проходе сливаются пары соседних куч. Покажите, что fromList требует всего $O(n)$ времени.*

```
functor LeftistHeap(Element: ORDERED) : Heap = struct
  structure Elem = Element

  datatype Heap = E | T of int × Elem.T × Heap × Heap

  fun rank E = 0
    | rank (T(r,_,_,_)) = r
  fun makeT (x,a,b) = if rank a ≥ rank b then T(rank b+1, x,a,b)
                      else T(rank a + 1, x,b,a)

  val empty = E
  fun isEmpty E = true
    | isEmpty _ = false

  fun merge (h,E) = h
    | merge (E,h) = h
    | merge (h1 as T(_,x,a1,b1), h_2 as T(_,y,a2,b2)) =
      if Elem.leq (x,y) then makeT(x,a1, merge(b1,h2)
      else makeT(y,a2,merge(h1,b2))

  fun insert (x,h) = merge (T(1,x,E,E),h)
  fun findMin E = raise EMPTY
    | findMin (T(_,x,a,b)) = x
  fun deleteMin E = raise EMPTY
    | deleteMin (T(_,x,a,b)) = merge(a,b)
end
```

Рис. 3.2: Левоориентированные кучи.

Упражнение 3.4 ***(Чо и Сахни [CS96])*** *Левоориентированные кучи со сдвинутым весом — альтернатива левоориентированным кучам, где вместо свойства левоориентированности используется свойство левоориентированности, сдвинутой по весу (weight-biased leftist property): размер любого левого поддерева всегда не меньше размера соответствующего правого поддерева.*

1. *Докажите, что правая периферия левоориентированной кучи со сдвинутым весом содержит не более* $\lfloor \log(n+1) \rfloor$ *элементов.*
2. *Измените реализацию на Рис. 3.2, чтобы получились левоориентированные кучи со сдвинутым весом.*
3. *Функция* merge *сейчас выполняется в два прохода: сверху вниз, с вызовами* merge*, и снизу вверх, с вызовами вспомогательной функции* makeT*. Измените* merge *для левоориентированных куч со сдвинутым весом так, чтобы она работала за один проход сверху вниз.*
4. *Каковы преимущества однопроходной версии* merge *в условиях ленивого вычисления? В условиях параллельного вычисления?*

3.2 Биномиальные кучи

Биномиальные очереди [Vui78, Bro78], которые мы, чтобы избежать путаницы с очередями FIFO, будем называть *биномиальными кучами* (binomial heaps) — ещё одна распространенная реализация куч. Биномиальные кучи устроены сложнее, чем левоориентированные, и, на первый взгляд, не возмещают эту сложность никакими преимуществами. Однако в последующих главах мы увидим, как в различных вариантах биномиальных куч можно заставить insert и merge выполняться за время $O(1)$.

Биномиальные кучи строятся из более простых объектов, называемых биномиальными деревьями. Биномиальные деревья индуктивно определяются так:

- Биномиальное дерево ранга 1 представляет собой одиночный узел.
- Биномиальное дерево ранга $r+1$ получается путем *связывания* (linking) двух биномиальных деревьев ранга r, так что одно из них становится самым левым потомком второго.

Из этого определения видно, что биномиальное дерево ранга r содержит ровно 2^r элементов. Существует второе, эквивалентное первому, определение биномиальных деревьев, которым иногда удобнее пользоваться: биномиальное дерево ранга r представляет собой узел с r потомками $t_1 \ldots t_r$, где каждое t_i является биномиальным деревом ранга $r-i$. На Рис. 3.3 показаны биномиальные деревья рангов от 0 до 3.

Мы представляем вершину биномиального дерева в виде элемента и списка его потомков. Для удобства мы также помечаем каждый узел его рангом.

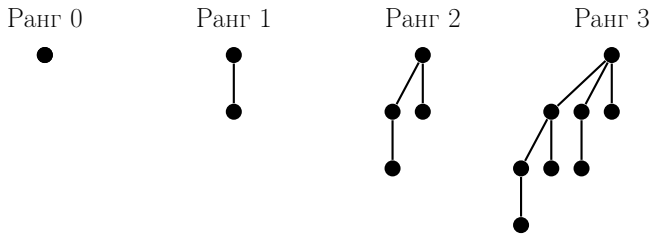


Рис. 3.3: Биномиальные деревья рангов 0–3.

```
datatype Tree = Node of int × Elem.T × Tree list
```

Каждый список потомков хранится в убывающем порядке рангов, а элементы хранятся с порядком кучи. Чтобы сохранять этот порядок, мы всегда привязываем дерево с большим корнем к дереву с меньшим корнем.

```
fun link (t1 as Node (r, x1, c1), t2 as Node (_, x2, c2)) =
      if Elem.leq (x1, x2) then Node (r+1, x1, t2 :: c1)
      else Node (r+1, x2, t1 :: c2
```

Связываем мы всегда деревья одного ранга.

Теперь определяем биномиальную кучу как коллекцию биномиальных деревьев, каждое из которых имеет порядок кучи, и никакие два дерева не совпадают по рангу. Мы представляем эту коллекцию в виде списка деревьев в возрастающем порядке ранга.

```
Type Heap = Tree list
```

Поскольку каждое биномиальное дерево содержит 2^r элементов, и никакие два дерева по рангу не совпадают, деревья размера n в точности соответствуют единицам в двоичном представлении n. Например, число 21 в двоичном виде выглядит как 10101, и поэтому биномиальная куча размера 21 содержит одно дерево ранга 0, одно ранга 2, и одно ранга 4 (размерами, соответственно, 1, 4 и 16). Заметим, что так же, как двоичное представление n содержит не более $\lfloor log(n+1) \rfloor$ единиц, биномиальная куча размера n содержит не более $\lfloor log(n+1) \rfloor$ деревьев.

Теперь мы готовы описать функции, действующие на биномиальных деревьях. Начинаем мы с insert и merge, которые определяются примерно аналогично сложению двоичных чисел. (Мы укрепим эту аналогию в Главе 9.) Чтобы внести элемент в кучу, мы сначала создаем одноэлементное дерево (т. е., биномиальное дерево ранга 0), затем поднимаемся по списку существующих деревьев в порядке возрастания рангов, связывая при этом одноранговые деревья. Каждое связывание соответствует переносу в двоичной арифметике.

```
fun rank (Node (r, x, c)) = r
fun insTree (t,[]) = [t]
  | insTree (t, ts as t' :: ts') =
      if rank t < rank t' then t :: ts else insTree (link (t, t'), ts')
```

```
fun insert (x, ts) = insTree (Node (0, x, []), ts)
```

В худшем случае, при вставке в кучу размера $n = 2^k - 1$, требуется k связываний и $O(k) = O(\log n)$ времени.

При слиянии двух куч мы проходим через оба списка деревьев в порядке возрастания ранга и связываем по пути деревья равного ранга. Как и прежде, каждое связывание соответствует переносу в двоичной арифметике.

```
fun merge (ts1, []) =ts1
  | merge ([], ts2) =ts2
  | merge (ts1 as t1 :: ts'1, ts2 as t2 :: ts'2) =
      if rank t1 <rank t2 then t1 :: merge (ts'1, ts2)
      else if rank t2 <rank t1 then merge (ts1, ts'2)
      else insTree (link (t1, t2), merge (ts'1, ts'2))
```

Функции findMin и deleteMin вызывают вспомогательную функцию removeMinTree, которая находит дерево с минимальным корнем, исключает его из списка и возвращает как это дерево, так и список оставшихся деревьев.

```
fun removeMinTree [t] = (t, [])
  | removeMinTree (t :: ts) =
      let val (t', ts') = removeMinTree ts
      in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
```

Функция findMin просто возвращает корень найденного дерева

```
fun findMin ts = let val (t, _) = removeMinTree ts in root t end
```

Функция deleteMin устроена немного похитрее. Отбросив корень найденного дерева, мы ещё должны вернуть его потомков в список остальных деревьев. Заметим, что список потомков *почти* уже соответствует определению биномиальной кучи. Это коллекция биномиальных деревьев с неповторяющимися рангами, но только отсортирована она не по возрастанию, а по убыванию ранга. Таким образом, обратив список потомков, мы преобразуем его в биномиальную кучу, а затем сливаем с оставшимися деревьями.

```
fun deleteMin ts = let val (Node (_, x, ts1), ts2) =removeMinTree ts
                   in merge (rev ts1, ts2) end
```

Полная реализация биномиальных куч приведена на Рис. 3.4. Все четыре основные операции в худшем случае требуют $O(\log n)$ времени.

Упражнение 3.5 *Определите findMin напрямую, без обращения к removeMinTree.*

Упражнение 3.6 *Большая часть аннотаций ранга в нашем представлении биномиальных куч излишня, потому что мы и так знаем, что дети узла ранга r имеют ранги $r-1, \ldots, 0$. Таким образом, можно исключить поле-аннотацию ранга из узлов, а вместо этого помечать ранг корня каждого дерева, т. е.,*

```
datatype Tree = Node of Elem × Tree list
type Heap = (int × Tree) list
```

```
functor BinomialHeap(Element: ORDERED) : Heap = struct
  structure Elem = Element
  datatype Tree = Node of int × Elem.T × Tree list
  datatype Heap = Tree list

  val empty = []
  val isEmpty ts = null ts

  fun rank (Node(r,x,c)) = r
  fun root (Node(r,x,c)) = x

  fun link (t1 as Node (r1,x1,c1), t2 as Node (r2,x2,c2)) =
    if Elem.leq (x1,x2) then Node(r+1, x1, t2::c1)
    else Node(r+1, x2, t1::c2)

  fun insTree (t,[]) = [t]
    | insTree (t, ts as t'::ts') =
      if rank t < rank t' then t::ts else insTree(link(t,t'), ts')

  fun insert (x,ts) = insTree(Node(0,x,[]), ts)

  fun merge (ts1, []) = ts1
    | merge ([], ts2) = ts2
    | merge (ts1 as t1::ts1', ts2 as t2::ts2') =
      if rank t1 <rank t2 then t1 :: merge(ts1,ts2')
      else if rank t2 <rank t1 then t2 :: merge(ts2, ts1')
      else insTree(link(t1,t2), merge(ts1',ts2')

  fun removeMinTree [] = raise EMPTY
    | removeMinTree [t] = (t,[])
    | removeMinTree (t::ts) =
      let val (t', ts') = removeMinTree ts
      in if Elem.leq (root t, root t') then (t,ts) else (t', t::ts') end

  fun findMin ts = let val (t,_) = removeMinTree ts in root t end

  fun deleteMin ts =
    let val (Node(_, x,ts1), ts2) = removeMinTree ts
    in merge (rev ts1,ts2) end
end
```

Рис. 3.4: Биномиальные кучи.

Реализуйте биномиальные кучи в таком представлении.

Упражнение 3.7 *Одно из основных преимуществ левоориентированных куч над биномиальными заключается в том, что findMin занимает в них $O(1)$ времени, а не $O(\log n)$. Следующая заготовка функтора улучшает время findMin до $O(1)$, сохраняя минимальный элемент отдельно от остальной кучи.*

```
functor ExplicitMin (H : Heap) : Heap =
struct
    structure Elem = H.Elem
    datatype Heap = E | NE of Elem.t × H.Heap
  ◦ ..
end
```

Заметим, что этот функтор не ограничен биномиальными кучами, а принимает любую реализацию куч в качестве параметра. Закончите этот функтор так, чтобы findMin требовал время $O(1)$, а функции insert, merge и deleteMin каждая по $O(\log n)$. Предполагается, что нижележащая реализация H для всех операций занимает $O(\log n)$.

3.3 Красно-чёрные деревья

В разделе 2.2 мы описали двоичные деревья поиска. Такие деревья хорошо ведут себя на случайных или неупорядоченных данных, однако на упорядоченных данных их производительность резко падает, и каждая операция может занимать до $O(n)$ времени. Решение этой проблемы состоит в том, чтобы каждое дерево поддерживать в приблизительно сбалансированном состоянии. Тогда каждая операция выполняется не хуже, чем за время $O(\log n)$. Одним из наиболее популярных семейств сбалансированных двоичных деревьев поиска являются красно-чёрные [GS78].

Красно-чёрное дерево представляет собой двоичное дерево поиска, в котором каждый узел окрашен либо красным, либо чёрным. Мы добавляем поле цвета в тип двоичных деревьев поиска из раздела 2.2.

```
datatype Color = R | B
datatype Tree = E | T of Color × Tree × Elem × Tree
```

Все пустые узлы считаются чёрными, поэтому пустой конструктор E в поле цвета не нуждается.

Мы требуем, чтобы всякое красно-чёрное дерево соблюдало два инварианта:

- **Инвариант 1.** У красного узла не может быть красного ребёнка.
- **Инвариант 2.** Каждый путь от корня дерева до пустого узла содержит одинаковое количество чёрных узлов.

Вместе эти два инварианта гарантируют, что самый длинный возможный путь по красно-чёрному дереву, где красные и чёрные узлы чередуются, не более чем вдвое длиннее самого короткого, состоящего только из чёрных узлов.

Упражнение 3.8 *Докажите, что максимальная глубина узла в красно-чёрном дереве размера n не превышает $2\lfloor \log(n+1) \rfloor$.*

Функция member для красно-чёрных деревьев не обращает внимания на цвета. За исключением заглушки в варианте для конструктора T, она не отличается от функции member для несбалансированных деревьев.

```
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
```

Функция insert более интересна, поскольку она должна поддерживать два инварианта баланса.

```
fun insert (x, s) =
    let fun ins E = T (R, E, x, E)
          | ins (s as T (color, a, y, b)) =
              if x < y then balance (color, ins a, y, b)
              else if x > y then balance (color, a, y, ins b)
              else s
        val T (_, a, y, b) = ins s   (* гарантированно непустое *)
    in T (B, a, y, b)
```

Эта функция содержит три существенных изменения по сравнению с insert для несбалансированных деревьев поиска. Во-первых, когда мы создаем новый узел в ветке ins E, мы сначала окрашиваем его в красный цвет. Во-вторых, независимо от цвета, возвращаемого ins, в окончательном результате мы корень окрашиваем чёрным. Наконец, в ветках x <y и x >y мы вызовы конструктора T заменяем на обращения к функции balance. Функция balance действует подобно конструктору T, но только она переупорядочивает свои аргументы, чтобы обеспечить выполнение инвариантов баланса.

Если новый узел окрашен красным, мы сохраняем Инвариант 2, но в случае, если отец нового узла тоже красный, нарушается Инвариант 1. Мы временно позволяем существовать одному такому нарушению, и переносим его снизу вверх по мере перебалансирования. Функция balance обнаруживает и исправляет красно-красные нарушения, когда обрабатывает чёрного родителя красного узла с красным ребёнком. Такая чёрно-красно-красная цепочка может возникнуть в четырёх различных конфигурациях, в зависимости от того, левым или правым ребёнком является каждая из красных вершин. Однако в каждом из этих случаев решение одно и то же: нужно преобразовать чёрно-красно-красный путь в красную вершину с двумя чёрными детьми, как показано на Рис. 3.5. Это преобразование можно записать так:

```
fun balance (B,T (R,T (R,a,x,b),y,c),z,d) = T (R, T (B,a,x,b),T (B,c,z,d))
  | balance (B,T (R,a,x,T (R,b,y,c)),z,d) = T (R, T (B,a,x,b),T (B,c,z,d))
  | balance (B,a,x,T (R,T (R,b,y,c),z,d)) = T (R, T (B,a,x,b),T (B,c,z,d))
  | balance (B,a,x,T (R,b,y,T (R,c,z,d))) = T (R, T (B,a,x,b),T (B,c,z,d))
  | balance body = T body
```

Нетрудно проверить, что в получающемся поддереве будут соблюдены оба инварианта красно-чёрного баланса.

Замечание 3.3 *Заметим, что в первых четырех строках balance правые части одинаковы. В некоторых реализациях Стандартного ML, в частности, в Нью-Джерсийском Стандартном ML (Standard ML of New Jersey), поддерживается расширение, называемое или-образцы (or-patterns), позволяющее слить несколько вариантов с одинаковыми правыми сторонами в один [FB97]. С использованием или-образцов можно переписать функцию balance так:*

```
fun balance ( (B,T (R,T (R,a,x,b),y,c),z,d)
            | (B,T (R,a,x,T (R,b,y,c)),z,d)
            | (B,a,x,T (R,T (R,b,y,c),z,d))
            | (B,a,x,T (R,b,y,T (R,c,z,d))) ) = T (R, T (B,a,x,b),T (B,c,z,d))
  | balance body = T body
```

После балансировки некоторого поддерева красный корень этого поддерева может оказаться ребёнком ещё одного красного узла. Таким образом, балансировка продолжается до самого корня дерева. На самом верху дерева мы можем получить красную вершину с красным ребёнком, но без чёрного родителя. С этим вариантом мы справляемся, всегда перекрашивая корень в чёрное.

Реализация красно-чёрных деревьев полностью приведена на Рис. 3.6.

Указание разработчикам 3.1 *Даже без дополнительных оптимизаций наша реализация сбалансированных двоичных деревьев поиска — одна из самых быстрых среди имеющихся. С оптимизациями вроде описанных в Упражнениях 2.2 и 3.10 она просто летает!*

Замечание 3.4 *Одна из причин, почему наша реализация выглядит настолько проще, чем типичное описание красно-чёрных деревьев (напр., Глава 14 в книге [CLR90]), состоит в том, что мы используем несколько другие преобразования перебалансировки. В императивных реализациях обычно наши четыре проблематичных случая разбиваются на восемь, в зависимости от цвета узла, соседствующего с красной вершиной с красным ребёнком. Знание цвета этого узла в некоторых случаях позволяет совершить меньше присваиваний, а в некоторых других завершить балансировку раньше. Однако в функциональной среде мы в любом случае копируем все эти вершины, и таким образом, не можем ни сократить число присваиваний, ни прекратить копирование раньше времени, так что для использования более сложных преобразований нет причины.*

Упражнение 3.9 *Напишите функцию fromOrdList типа Elem list → Tree, преобразующую отсортированный список без повторений в красно-чёрное дерево. Функция должна выполняться за время $O(n)$.*

Упражнение 3.10 *Приведенная нами функция balance производит несколько ненужных проверок. Например, когда функция ins рекурсивно вызывается для левого ребёнка, не требуется проверять красно-красные нарушения на правом ребёнке.*

1. *Разбейте balance на две функции lbalance и rbalance, которые проверяют, соответственно, нарушения инварианта в левом и правом ребёнке. Замените обращения к balance внутри ins на вызовы lbalance либо rbalance.*

2. *Ту же самую логику можно распространить ещё на шаг и убрать одну из проверок для внуков. Перепишите ins так, чтобы она никогда не проверяла цвет узлов, не находящихся на пути поиска.*

3.4 Примечания

Нуньес, Палао и Пенья [NPP95] и Кинг [Kin94] описывают подобные нашим реализации, соответственно, левоориентированных куч и биномиальных куч на Haskell. Красно-чёрные деревья до сих пор не были описаны в литературе по функциональному программированию, в отличие от некоторых других вариантов сбалансированных деревьев поиска, таких как AVL-деревья [Mye82, Mye84, BW88, NPP95], 2-3-деревья [Rea92] и деревья, сбалансированные по весу [Ada93].

Левоориентированные кучи были изобретены Кнутом [Knu73a] как упрощение структуры данных, введенной Крейном [Cra72]. Виллемин [Vui78] изобрел биномиальные кучи; Браун [Bro78] исследовал многие свойства этой изящной структуры данных. Гибас и Седжвик [GS78] предложили красно-чёрные деревья в качестве обобщающего описания для многих других разновидностей сбалансированных деревьев.

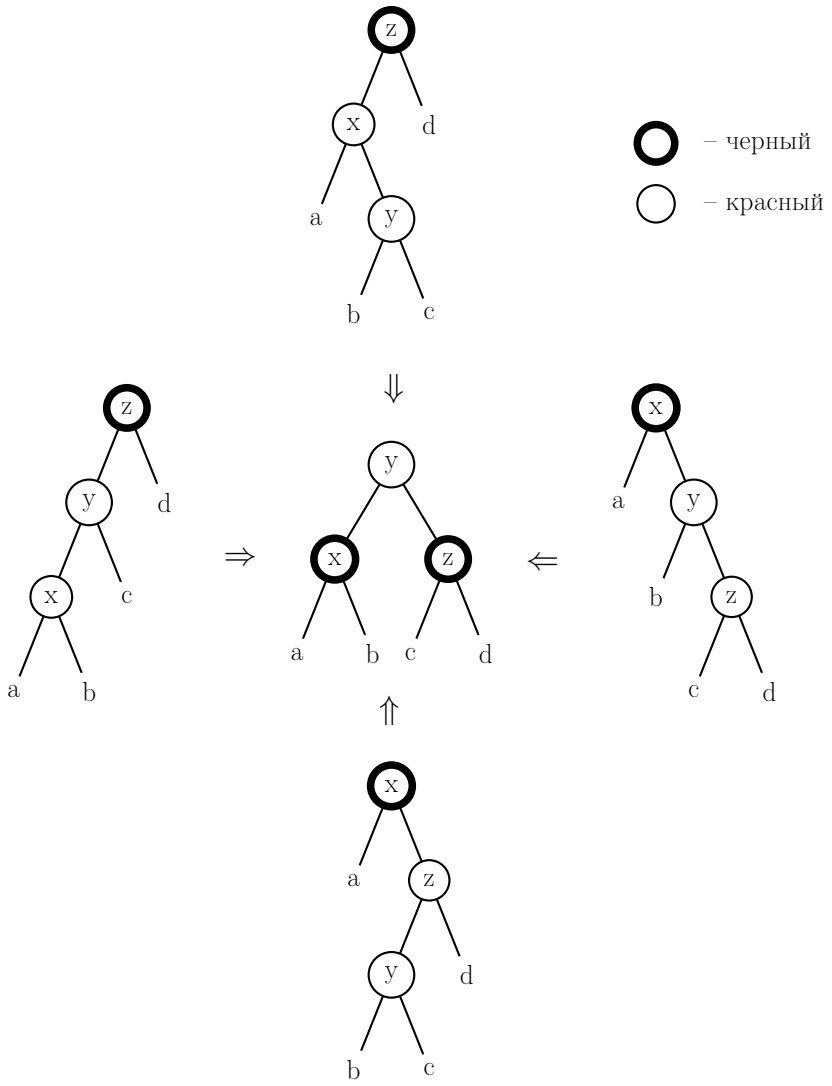


Рис. 3.5: Избавление от красных узлов с красными родителями.

```
functor RedBlackSet(Element: ORDERED) : SET =
  type Elem = Element.T
  datatype Color = R | B
  datatype Tree = E | T of Color × 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 if Element.lt (y,x) then member (x, b)
        else true

  fun balance (B,T (R,T (R,a,x,b),y,c),z,d) = T (R, T (B,a,x,b),T (B,c,z,d))
    | balance (B,T (R,a,x,T (R,b,y,c)),z,d) = T (R, T (B,a,x,b),T (B,c,z,d))
    | balance (B,a,x,T (R,T (R,b,y,c),z,d)) = T (R, T (B,a,x,b),T (B,c,z,d))
    | balance (B,a,x,T (R,b,y,T (R,c,z,d))) = T (R, T (B,a,x,b),T (B,c,z,d))
    | balance body = T body

  fun insert (x, s) =
        let fun ins E = T (R, E, x, E)
              | ins (s as T (color, a, y, b)) =
                  if Element.lt (x,y) then balance (color, ins a, y, b)
                  else if Element.lt (y,x) then balance (color, a, y, ins b)
                  else s
            val T (_, a, y, b) = ins s   /* гарантированно непустое */
        in T (B, a, y, b)
end
```

Рис. 3.6: Красно-чёрные деревья.

Глава 4

Ленивое вычисление

Ленивое вычисление является основной стратегий вычисления во многих функциональных языках программирования (но не в Стандартном ML). У этой стратегии есть два существенных свойства. Во-первых, вычисление всякого выражения задерживается, или *подвешивается* (suspend), пока не потребуется его результат. Во-вторых, когда задержанное выражение вычисляется в первый раз, результат вычисления запоминается (*мемоизируется* (memoize)), так что, если он потребуется снова, можно его просто извлечь из памяти, а не вычислять заново. Оба этих свойства ленивого вычисления оказываются алгоритмически полезными.

В этой главе мы вводим удобные обозначения для ленивых вычислений и, в качестве иллюстрации, строим при помощи этой нотации простую библиотеку потоков. В последующих главах мы будем активно пользоваться как ленивыми вычислениями вообще, так и потоками в частности.

4.1 $-запись

К сожалению, определение Стандартного ML [MTHM97] не включает поддержки ленивого вычисления, так что каждая реализация может предоставлять свой собственный набор элементарных операций. Мы представляем здесь один такой набор, называемый $-записью. Перевод программ, использующих $-запись, в другие варианты примитивов ленивого вычисления не должен представлять трудности.

В $-записи мы вводим новый тип α susp, представляющий задержки (задержанные вычисления). У этого типа имеется один одноместный конструктор $. В первом приближении α susp и $ ведут себя так, как будто они введены при помощи обыкновенного объявления типа

datatype α susp =$ **of** α

Новая задержка типа τ susp создается при помощи конструкции $$e$, где e — выражение типа τ. Подобным же образом, содержимое задержки можно

извлечь через сопоставление с образцом $\$p$. Если образец p сопоставляется со значениями типа τ, то $\$p$ сопоставляется с задержками типа τ susp.

Основное различие между \$ и обыкновенными конструкторами состоит в том, что \$ не вычисляет свой аргумент немедленно. Вместо этого он запоминает информацию, необходимую для того, чтобы вычислить выражение-аргумент позже. (Как правило, эта информация состоит из указателя на код, а также значений свободных переменных выражения.) Выражение-аргумент не вычисляется до тех пор, когда (и если) оно не сопоставится с образцом вида $\$p$. В этот момент выражение вычисляется, а его результат запоминается. Затем результат сопоставляется с образцом p. Если задержанное выражение потом сопоставляется с другим образцом вида $\$p'$, запомненное значение извлекается и сопоставляется с образцом p'.

Кроме того, конструктор \$ отличается от прочих конструкторов синтаксически. Во-первых, его область действия распространяется направо как можно дальше. Таким образом, например, выражение \$f x равнозначно \$(f x), а не (\$f) x; образец \$Cons (x, xs) обозначает то же, что \$(Cons (x, xs)), а не (\$ Cons) (x, xs). Во-вторых, \$ не является правильно построенным выражением сам по себе — он всегда должен сочетаться с аргументом.

В качестве примера \$–записи рассмотрим следующий фрагмент программы:

```
val s = $primes 1000000        (* быстро *)
- ..
 val $x =s                     (* медленно *)
- ..
 val $y =s                     (* быстро *)
- ..
```

Программа вычисляет миллионное простое число. Первая строка, которая просто создает новую задержку, выполняется очень быстро. Вторая строка выполняет задержанное вычисление и находит простое число. В зависимости от алгоритма поиска простых чисел, она может потребовать значительного времени. Третья строка обращается к мемоизированному значению и также выполняется очень быстро.

В качестве второго примера рассмотрим фрагмент

```
let val s = $primes 1000000
in 15 end
```

В этой программе содержимое задержки никогда не требуется, и, значит, выражение primes 1000000 не выполняется.

Хотя все примеры ленивого вычисления в этой книге можно было бы выразить только через выражения и образцы со знаком \$, удобно оказывается ввести два элемента синтаксического сахара. Первый из них — оператор force («вынудить»), определяемый как

```
fun force ($x) =x
```

Он полезен, чтобы извлечь содержимое задержки посредине выражения, где было бы неудобно вставлять конструкцию сопоставления с образцом.

Второй элемент синтаксического сахара полезен при написании некоторых разновидностей ленивых функций. Рассмотрим, например, следующую функцию для сложения задержанных целых:

```
fun plus ($m, $n) = $m+n
```

Несмотря на то, что определение функции выглядит совершенно разумно, скорее всего, это не та функция, которую мы хотели написать. Проблема состоит в том, что оба ее задержанных аргумента выполняются слишком рано. Они вынуждаются в момент применения функции plus, а не тогда, когда требуется выполнить задержку, создаваемую ей. Один из способов получить нужное поведение — явным образом задержать сопоставление с образцом

```
fun plus (x, y) = $case (x, y) of ($m, $n) ⇒ m+n
```

Подобные конструкции встречаются достаточно часто, поэтому мы введём для них синтаксический сахар

```
fun lazy f p = e
```

что равносильно

```
fun f x = $case x of p ⇒ force e
```

При помощи дополнительного force мы добиваемся того, что ключевое слово lazy никак не влияет на тип функции (если предположить, что он уже был α susp), так что эту аннотацию можно добавлять и убирать, никак не меняя остальной текст. Теперь требуемую нам функцию для сложения задержанных целых можно написать просто как

```
fun lazy plus ($m, $n) = $m+n
```

Раскрытие синтаксического сахара дает

```
fun plus (x, y) = $case (x, y) of ($m, $n) ⇒ force ($m+n)
```

что совпадает с ранее вручную написанной версией с точностью до дополнительных force и $ вокруг m+n. Хороший компилятор уберет эти force и $ при оптимизации, поскольку для любого e выражения e и force ($e) эквивалентны.

В функции plus аннотация lazy используется для задержки сопоставления с образцом, чтобы $-образцы не были сопоставлены раньше времени. Однако аннотация lazy полезна также, когда правая сторона определения функции возвращает задержку в результате вычисления, которое может оказаться долгим и сложным. В такой ситуации использование lazy сдвигает выполнение дорогого вычисления от того момента, когда функция применяется к аргументу, на тот, когда вынуждается возвращаемая ею задержка. В следующем разделе мы увидим несколько примеров такого использования lazy.

Синтаксис и семантика $-записи формально определены в [Oka96a].

4.2 Потоки

В качестве расширенного примера ленивых вычислений и $-записи в Стандартном ML мы представляем простой пакет для работы с потоками. Потоки будут использоваться в нескольких структурах данных из последующих глав. Потоки (известные также как ленивые списки) подобны обыкновенным спискам, за исключением того, что вычисление каждой их ячейки задерживается. Тип потоков выглядит так:

```
datatype α StreamCell =Nil | Cons of α × α Stream
withtype α Stream =α StreamCell susp
```

Простой поток, содержащий элементы 1, 2 и 3, можно записать как

```
$Cons (1, $Cons (2, $Cons (3, $Nil)))
```

Полезно сравнить потоки с задержанными списками типа α list susp. Вычисления, представленные последними, по существу *монолитны* — единожды начав вычислять задержанный список, мы вычисляем его до конца. Напротив, вычисления, представленные потоками, часто *пошаговы* — при обращении к потоку проводится только та часть вычисления, которая порождает его первый элемент, а остальное задерживается. Такое поведение часто встречается в типах, которые, подобно потокам, содержат вложенные задержки.

Чтобы яснее прочувствовать эту разницу в поведении, рассмотрим функцию конкатенации, записываемую s ++ t. Для задержанных списков ее можно записать как

```
fun s ++ t =$(force s @ force t)
```

что равносильно

```
fun lazy ($xs) ++ ($ys) =$(xs @ ys)
```

Задержка, порождаемая этой функцией, вынуждает оба аргумента, а затем конкатенирует полученные списки и возвращает результат целиком. Таким образом, задержка монолитна. Можно также сказать, что монолитна вся функция. Для потоков функция записывается как

```
fun lazy ($Nil) ++ t =t
       | ($Cons (x, s)) ++ t =$Cons (x, s ++ t)
```

Эта функция немедленно возвращает задержку, которая, будучи запущена, требует первую ячейку первого потока, сопоставляя ее с $-образцом. Если эта ячейка представляет собой Cons, мы строим результат из x и s ++ t. Вследствие аннотации lazy рекурсивный вызов просто порождает ещё одну задержку, не производя никакой дополнительной работы. Следовательно, эта функция описывает пошаговое вычисление: порождается первая ячейка результата, а остальное задерживается. Мы также говорим, что пошаговой является сама функция.

Ещё одна пошаговая функция — take, извлекающая первые n элементов потока.

```
fun lazy take (0, s) = $Nil
      | take (n, $Nil) =$Nil
      | take (n, $Cons (x, s)) =$Cons (x, take (n−1, s))
```

Как и в случае с ++, рекурсивный вызов take немедленно возвращает задержку, а не выполняет оставшуюся часть кода функции.

Рассмотрим, однако, функцию, уничтожающую первые n элементов потока, которую можно записать как

```
fun lazy drop (0, s) = s
      | drop (n, $Nil) =$Nil
      | drop (n, $Cons (x, s)) =drop (n−1, s)
```

или, более эффективно, как

```
fun lazy drop (n, s) = let fun drop' (0, s) = s
                             | drop' (n, $Nil) =$Nil
                             | drop' (n−1, $Cons (x, s)) =drop' (n−1, s)
                       in drop' (n, s) end
```

Эта функция монолитна, поскольку рекурсивные вызовы drop' никогда не задерживаются — вычисление первой же ячейки результата требует выполнения всей функции целиком. Здесь аннотация lazy используется, чтобы задержать исходный вызов drop', а не сопоставление с образцом.

Упражнение 4.1 *Покажите, используя эквивалентность force ($e) и e, что два определения drop эквивалентны.*

Ещё одна часто используемая монолитная функция над потоками — reverse.

```
fun lazy reverse s =
    let fun reverse' ($Nil, r) =r
          | reverse' ($Cons (x, s), r) =reverse' (s, $Cons (x, r))
    in reverse' (s, $Nil) end
```

Здесь рекурсивные вызовы reverse' никогда не задерживаются. Обратите внимание, однако, что каждый такой вызов создает задержку вида $Cons (x, r). Может показаться, что reverse на самом деле не производит всю работу за один раз. Однако задержки такого вида, где тело содержит лишь несколько конструкторов и переменных, называются *тривиальными* (trivial). Тривиальные задержки создаются не из каких-то алгоритмических соображений, а для того, чтобы удовлетворить систему типов. Можно считать, что тело тривиальной задержки выполняется в момент ее создания. На самом деле, при минимальной оптимизации компилятором подобные задержки создаются уже в мемоизированном виде. В любом случае, вынуждение тривиальной задержки никогда не занимает больше, чем $O(1)$ времени.

Несмотря на распространенность монолитных функций над потоками вроде drop и reverse, смыслом существования потоков являются пошаговые функции вроде ++ и take. Каждая задержка несет с собой небольшие, но

существенные расходы, поэтому для максимальной эффективности ленивость следует использовать только тогда, когда для этого есть серьезные основания. Если все операции над ленивыми списками в каком-то приложении монолитны, то в этом приложении лучше пользоваться обыкновенными ленивыми списками, а не потоками.

На Рис. 4.1 потоковые функции собраны в единый модуль на Стандартном ML. Заметим, что в модуле не экспортируются, как можно было бы ожидать, функции вроде isEmpty и cons. Вместо этого мы намеренно выставляем для обозрения внутреннее представление, чтобы поддержать для потоков сопоставление с образцом.

Упражнение 4.2 *Реализуйте сортировку вставками для потоков. Покажите, что извлечение первых k элементов sort xs требует лишь $O(n \cdot k)$ времени, где n — длина xs, а не $O(n^2)$, как можно было бы ожидать от сортировки вставками.*

4.3 Примечания

Ленивое вычисление. Ленивое вычисление было изобретено Уодсвортом [Wad71] как оптимизация нормального порядка редукции в лямбда-исчислении. Позже Виллемин [Vui74] показал, что при некоторым образом ограниченных условиях ленивое вычисление является оптимальной стратегией вычисления. Формальная семантика ленивого вычисления подробно исследовалась в [Jos89, Lau93, OLT94, AFM+95].

Потоки. Потоки изобрел Ландин [Lan65], но без мемоизации. Фридман и Уайз [FW76] и Хендерсон и Моррис [HM76] расширили потоки Ландина мемоизацией.

Мемоизация. Термин «мемоизация» придумал Мичи [Mic68], чтобы называть так кэширование пар аргумент-результат у функции. Поле аргумента можно отбросить при мемоизации задержек, если рассматривать задержки как нульместные функции, то есть функции с нулем аргументов. Позднее Хьюз [Hug85] применил мемоизацию в исходном смысле Мичи к функциональным программам.

Алгоритмика. Обе компоненты ленивых вычислений — задержка вычисления и мемоизация результатов, — имеют долгую историю в науке построения алгоритмов, хотя и не всегда в сочетании друг с другом. Идея задержки вычислений, которые могут оказаться дорогими (часто это уничтожение элементов) с пользой используется в хэш-таблицах [WV86], очередях с приоритетами [ST86b, FT87] и деревьях поиска [DSST89]. В свою очередь, мемоизация является основой таких методик, как динамическое программирование [Bel57] и сжатие путей [HU73, TvL84].

```
signature STREAM = sig
  datatype α StreamCell = Nil |Cons of α × α Stream
  withtype α Stream = α StreamCell susp

  val ++ : α Stream ×α Stream → α Stream /* Конкатенация потоков */
  val take : int ×α Stream → α Stream
  val drop : int ×α Stream → α Stream
  val reverse : α Stream → α Stream
end

structure Stream: STREAM = struct
  datatype α StreamCell = Nil |Cons of α × α Stream
  withtype α Stream = α StreamCell susp

  fun lazy ($Nil) ++t = t
         | ($Cons(x,s)) ++t = $Cons(x,s++t)

  fun lazy take (0,s) = $Nil
         | take (n, $Nil) = $Nil
         | take (n, $Cons(x,s)) = $Cons(x,take(n−1,s))

  fun lazy drop (n, s) =
        let fun drop' (0, s) = s
              | drop' (n, $Nil) = $Nil
              | drop' (n−1, $Cons (x, s)) = drop' (n−1, s)
        in drop' (n, s) en

  fun lazy reverse s =
        let fun reverse' ($Nil, r) = r
              | reverse' ($Cons(x, s), r) = reverse' (s, $Cons(x, r))
        in reverse' (s, $Nil) end
end
```

Рис. 4.1: Небольшой пакет потоков.

Глава 5

Основы амортизации

За последние пятнадцать лет амортизация стала мощным инструментом в построении и анализе структур данных. Реализации с амортизированными характеристиками производительности часто оказываются проще и быстрее, чем реализации со сравнимыми жёсткими характеристиками. В этой главе мы даем обзор основных методов амортизации и иллюстрируем эти идеи через простую реализацию очередей FIFO и несколько реализаций кучи.

К сожалению, простой подход к амортизации, рассматриваемый в этой главе, конфликтует с идеей устойчивости — эти структуры, будучи используемы как устойчивые, могут быть весьма неэффективны. Однако на практике многие приложения устойчивости не требуют, и часто для таких приложений реализации, представленные в этой главе, могут быть замечательным выбором. В следующей главе мы увидим, как можно совместить понятия амортизации и устойчивости при помощи ленивого вычисления.

5.1 Методы амортизированного анализа

Понятие амортизации возникает из следующего наблюдения. Имея последовательность операций, мы можем интересоваться временем, которое отнимает вся эта последовательность, однако при этом нам может быть безразлично время каждой отдельной операции. Например, имея n операций, мы можем желать, чтобы время всей последовательности было ограничено показателем $O(n)$, не настаивая, чтобы каждая из этих операций происходила за время $O(1)$. Нас может устраивать, чтобы некоторые из операций занимали $O(\log n)$ или даже $O(n)$, при условии, что общая стоимость всей последовательности будет $O(n)$. Такая дополнительная степень свободы открывает широкое пространство возможностей при проектировании, и часто позволяет найти более простые и быстрые решения, чем варианты с аналогичными жёсткими ограничениями.

Чтобы доказать, что соблюдается амортизированное ограничение, нуж-

но определить амортизированную стоимость для каждой операции, и доказать, что для любой последовательности операций общая амортизированная стоимость является верхней границей общей реальной стоимости, т. е.,

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

где a_i — амортизированная стоимость операции i, t_i — ее реальная стоимость, а m — общее число операций. Обычно доказывается несколько более сильный результат: что на любой промежуточной стадии в последовательности операций общая текущая амортизированная стоимость является верхней границей для общей текущей реальной стоимости, т. е.,

$$\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_i$$

для любого j. Разница между общей текущей амортизированной стоимостью и общей текущей реальной стоимостью называется *текущие накопления* (accumulated savings). Таким образом, общая текущая амортизированная стоимость является верхней границей для общей текущей реальной стоимости тогда и только тогда, когда текущие накопления неотрицательны.

Амортизация позволяет некоторым операциям быть дороже, чем их амортизированная стоимость. Такие операции называются *дорогими* (expensive). Операции, для которых амортизированная стоимость превышает реальную, называются *дешевыми* (cheap). Дорогие операции уменьшают текущие накопления, а дешевые их увеличивают. Главное при доказательстве амортизированных характеристик стоимости — показать, что дорогие операции случаются только тогда, когда текущих накоплений хватает, чтобы покрыть их дополнительную стоимость.

Тарьян [Tar85] описывает два метода для анализа амортизированных структур данных: *метод банкира* (banker's method) и *метод физика* (physicist's method). В методе банкира текущие накопления представляются как *кредит* (credits), привязанный к определенным ячейкам структуры данных. Этот кредит используется, чтобы расплатиться за будущие операции доступа к этим ячейкам. Амортизированная стоимость операции определяется как ее реальная стоимость плюс размер кредита, выделяемого этой операцией, минус размер кредита, который она расходует, т. е.,

$$a_i = t_i + c_i - \bar{c}_i$$

где c_i — размер кредита, выделяемого операцией i, а $\bar{c}_i$ — размер кредита, расходуемого операцией i. Каждая единица кредита должна быть выделена, прежде чем израсходована, и нельзя расходовать кредит дважды. Таким образом, $\sum c_i \geq \sum \bar{c}_i$, а следовательно, как и требуется, $\sum a_i \geq \sum t_i$. Как правило, доказательства с использованием метода банкира определяют *инвариант кредита* (credit invariant), регулирующий распределение кредита

так, чтобы при всякой дорогой операции достаточное его количество было выделено в нужных ячейках структуры для покрытия стоимости операции.

В методе физика определяется функция Φ, отображающая всякий объект d на действительное число, называемое его *потенциалом* (potential). Потенциал обычно выбирается так, чтобы изначально равняться нулю и оставаться неотрицательным. В таком случае потенциал представляет нижнюю границу текущих накоплений.

Пусть объект d_i будет результатом операции i и аргументом операции $i+1$. Тогда амортизированная стоимость операции i определяется как сумма реальной стоимости и изменения потенциалов между d_{i-1} и d_i, т. е.,

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

Текущая реальная стоимость последовательности операций равна

$$\begin{aligned} \textstyle\sum_{i=1}^{j} t_i &= \textstyle\sum_{i=0}^{j}(a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ &= \textstyle\sum_{i=1}^{j} a_i + \sum_{i=1}^{j}(\Phi(d_{i-1}) - \Phi(d_i)) \\ &= \textstyle\sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j) \end{aligned}$$

Суммы вроде $\sum_{i=1}^{j}(\Phi(d_{i-1})+\Phi(d_i))$, где чередующиеся отрицательные и положительные члены взаимно уничтожаются, называются *телескопическими последовательностями* (telescoping series). Если Φ выбран таким образом, что $\Phi(d_0)$ равен нулю, а $\Phi(d_j)$ неотрицателен, мы имеем $\Phi(d_j) \geq \Phi(d_0)$, так что, как и требуется, текущая общая амортизированная стоимость является верхней границей для текущей общей реальной стоимости.

Замечание 5.1 *Такое описание метода физика несколько упрощает картину. Часто при анализе оказывается трудно втиснуть реальное положение дел в указанные рамки. Например, что делать с функциями, которые порождают или возвращают более одного объекта? Однако даже упрощенное описание достаточно для демонстрации основных идей.*

Ясно, что два метода анализа весьма похожи. Можно преобразовать метод банкира в метод физика, если игнорировать распределение по ячейкам, и считать, что потенциал равен общему количеству единиц кредита в объекте, как указано в инварианте кредита. Подобным образом, можно преобразовать метод физика в метод банкира, если расположить весь кредит в корне объекта. Возможно, несколько удивляет то, что знание о расположении ячеек не дает никакой дополнительной мощности в доказательстве, но методы на самом деле эквивалентны [Tar85, Sch92]. Чаще всего метод физика оказывается проще, но иногда бывает удобно принять во внимание распределение по ячейкам.

Заметим, что кредит и потенциал являются лишь средствами анализа; ни то, ни другое не присутствует в тексте программы (разве что, возможно, в комментариях).

```
signature QUEUE =
sig
  type α Queue
  val empty : α Queue
  val isEmpty : α Queue → bool
  val snoc : α Queue × α → α Queue
  val head : α Queue → α          /* возбуждает исключение Empty, если очередь пуста */
  val tail : α Queue → α Queue /* возбуждает исключение Empty, если очередь пуста */
end
```

Рис. 5.1: Сигнатура для очередей. (Этимологическое замечание: snoc представляет собой перевернутое слово cons и означает «добавить справа».)

5.2 Очереди

Мы демонстрируем методы банкира и физика через анализ простой функциональной реализации FIFO-очередей, чья сигнатура приведена на Рис. 5.1.

Самая распространенная чисто функциональная реализация очередей представляет собой пару списков, f и r, где f содержит головные элементы очереди в правильном порядке, а r состоит из хвостовых элементов в обратном порядке. Например, очередь, содержащая целые числа $1 \ldots 6$, может быть представлена списками f=[1,2,3] и r=[6,5,4]. Это представление можно описать следующим типом:

```
type α Queue =α list × α list
```

В этом представлении голова очереди — первый элемент f, так что функции head и tail возвращают и отбрасывают этот элемент, соответственно.

```
fun head (x :: f, r) = x
fun tail (x :: f, r) = f
```

Подобным образом, хвостом очереди является первый элемент r, так что snoc добавляет к r новый элемент.

```
fun snoc ((f,r), x) = (f, x :: r)
```

Элементы добавляются к r и убираются из f, так что они должны как-то переезжать из одного списка в другой. Этот переезд осуществляется путем обращения r и установки его на место f всякий раз, когда в противном случае f оказался бы пустым. Одновременно r устанавливается в []. Наша цель — поддерживать инвариант, что список f может быть пустым только в том случае, когда список r также пуст (т. е., пуста вся очередь). Заметим, что если бы f был пустым при непустом r, то первый элемент очереди находился бы в конце r, и доступ к нему занимал бы $O(n)$ времени. Поддерживая инвариант, мы гарантируем, что функция head всегда может найти голову очереди за $O(1)$ времени.

Теперь snoc и tail должны распознавать ситуацию, которая может привести к нарушению инварианта, и соответствующим образом менять свое

Рис. 5.2: Распространенная реализация чисто функциональной очереди.

поведение.

```
fun snoc (([], _), x) = ([x], [])
  | snoc ((f,r), x) = (f, x :: r)
fun tail ([x], r) = (rev r, [])
  | tail (x :: f, r) = (f, r)
```

Заметим, что в первой ветке snoc используется образец-заглушка. В этом случае поле r проверять не нужно, поскольку из инварианта мы знаем, что если список f равен [], то r также пуст.

Чуть более изящный способ записать эти функции — вынести те части snoc и tail, которые поддерживают инвариант, в отдельную функцию checkf. Она заменяет f на rev r, если f пуст, а в противном случае ничего не делает.

```
fun checkf ([], r) = (rev r, [])
  | checkf q = q

fun snoc ((f,r), x) = checkf (f, x :: r)
fun tail (x :: f, r) = checkf (f, r)
```

Полный код реализации показан на Рис. 5.2. Функции snoc и head всегда завершаются за время $O(1)$, но tail в худшем случае отнимает $O(n)$ времени. Однако, используя либо метод банкира, либо метод физика, мы можем показать, что как snoc, так и tail занимают амортизированное время $O(1)$.

В методе банкира мы поддерживаем инвариант, что каждый элемент в хвостовом списке связан с одной единицей кредита. Каждый вызов snoc для непустой очереди занимает один реальный шаг и выделяет одну единицу кредита для элемента хвостового списка; таким образом, общая амортизированная стоимость равна двум. Вызов tail, не обращающий хвостовой список, занимает один шаг, не выделяет и не тратит никакого кредита, и, таким образом, имеет амортизированную стоимость 1. Наконец, вызов tail, обращающий хвостовой список, занимает $m+1$ реальный шаг, где m — длина хвостового списка, и тратит m единиц кредита, содержащиеся в этом списке, так что амортизированная стоимость получается $m + 1 - m = 1$.

В методе физика мы определяем функцию потенциала Φ как длину хвостового списка. Тогда всякий snoc к непустой очереди занимает один реальный шаг и увеличивает потенциал на единицу, так что амортизированная стоимость равна двум. Вызов tail без обращения хвостовой очереди занимает один реальный шаг и не изменяет потенциал, так что амортизированная стоимость равна одному. Наконец, вызов tail с обращением очереди занимает $m + 1$ реальный шаг, но при этом устанавливает хвостовой список равным [], уменьшая таким образом потенциал на m, так что амортизированная стоимость равна $m + 1 - m = 1$.

В этом простом примере доказательства почти одинаковы. Но даже при этом метод физика оказывается чуть проще по следующей причине. Ис-

```
(* вставка, просмотр и уничтожение головного элемента *)
  (* возбуждает исключение Empty, если очередь пуста *)
  (* возбуждает исключение Empty, если очередь пуста *)
(* вставка, просмотр и уничтожение хвостового элемента *)
  (* возбуждает исключение Empty, если очередь пуста *)
  (* возбуждает исключение Empty, если очередь пуста *)
```

Рис. 5.3: Сигнатура двусторонней очереди.

пользуя метод банкира, мы должны сначала выбрать инвариант кредита, а затем для каждой функции решить, когда она должна выделять или расходовать кредит. Инвариант кредита подсказывает нам, как это сделать, но решение все же не принимается автоматически. Например, должен ли snoc выделить одну единицу кредита и израсходовать ноль, или выделить две и одну израсходовать? Общий результат оказывается один и тот же, так что дополнительная свобода оказывается лишь дополнительным возможным источником путаницы. С другой стороны, в методе физика от нас требуется принять только одно решение — выбрать функцию потенциала. После этого анализ сводится к простым вычислениям; никакой свободы выбора не остается.

Указание разработчикам 5.1 *Эта реализация очередей идеальна в приложениях, где не требуется устойчивости и где приемлемы амортизированные показатели производительности.*

Упражнение 5.1 ***Хогерворд [Hoo92].*** *Идея этих очередей легко может быть расширена на абстракцию двусторонней очереди (double-ended queue), или дека (deque), где чтение и запись разрешены с обоих концов очереди (см. Рис. 5.3). Инвариант делается симметричным относительно f и r: если очередь содержит более одного элемента, оба списка должны быть непустыми. Когда один из списков становится пустым, мы делим другой пополам и одну из половин обращаем.*

1. *Реализуйте эту версию деков.*
2. *Докажите, что каждая операция занимает $O(1)$ амортизированного времени, используя функцию потенциала $\Phi(f, r) = abs(|f| - |r|)$, где abs — функция модуля.*

5.3 Биномиальные кучи

В Разделе 3.2 мы показали, что вставка в биномиальную кучу проходит в худшем случае за время $O(\log n)$. Здесь мы доказываем, что на самом деле амортизированное ограничение на время вставки составляет $O(1)$.

Мы пользуемся методом физика. Определим потенциал биномиальной кучи как число деревьев в ней. Заметим, что это число равно количеству

единиц в двоичном представлении n, числа элементов в куче. Вызов insert занимает $k+1$ шаг, где k — число обращений к link. Если изначально в куче было t деревьев, то после вставки окажется $t-k+1$ деревьев. Таким образом, изменение потенциала составляет $(t-k+1)-t = 1-k$, а амортизированная стоимость вставки $(k+1)-(1-k) = 2$.

Упражнение 5.2 *Повторите доказательство с использованием метода банкира.*

Для полноты картины нам нужно показать, что амортизированная стоимость операций merge и deleteMin по-прежнему составляет $O(\log n)$. deleteMin не доставляет здесь никаких трудностей, но в случае merge требуется небольшое расширение метода физика. До сих пор мы определяли амортизированную стоимость операции как

$$a = t + \Phi(d_{вых}) - \Phi(d_{вх})$$

где $d_{вх}$ — структура на входе операции, а $d_{вых}$ — структура на выходе. Однако если операция принимает либо возвращает более одного объекта, это определение требуется обобщить до

$$a = t + \sum_{d \in Вых} \Phi(d) - \sum_{d \in Вх} \Phi(d)$$

где $Вх$ — множество входов, а $Вых$ — множество выходов. В этом правиле мы рассматриваем только входы и выходы анализируемого типа.

Упражнение 5.3 *Докажите, что амортизированная стоимость операций merge и deleteMin по-прежнему составляет $O(\log n)$.*

5.4 Расширяющиеся кучи

Расширяющиеся деревья (splay trees) [ST85] — возможно, самая известная и успешно применяемая амортизированная структура данных. Расширяющиеся деревья являются ближайшими родственниками двоичных сбалансированных деревьев поиска, но они не хранят никакую информацию о балансе явно. Вместо этого каждая операция перестраивает дерево при помощи некоторых простых преобразований, которые имеют тенденцию увеличивать сбалансированность. Несмотря на то, что каждая конкретная операция может занимать до $O(n)$ времени, амортизированная стоимость ее, как мы покажем, не превышает $O(\log n)$.

Важное различие между расширяющимися деревьями и сбалансированными двоичными деревьями поиска вроде красно-чёрных деревьев из Раздела 3.3 состоит в том, что расширяющиеся деревья перестраиваются даже во время запросов (таких, как member), а не только во время обновлений (таких, как insert). Это свойство мешает использованию расширяющихся

деревьев для реализации абстракций вроде множеств или конечных отображений в чисто функциональном окружении, поскольку приходилось бы возвращать в запросе новое дерево наряду с ответом на запрос[1]. Однако в некоторых абстракциях операции-запросы достаточно ограничены, чтобы эту проблему можно было обойти. Хорошим примером служит абстракция кучи, поскольку здесь единственным интересным запросом является findMin. Как мы увидим, расширяющиеся деревья дают нам отличную реализацию кучи.

Представление расширяющихся деревьев идентично представлению несбалансированных двоичных деревьев поиска.

```
datatype Tree = E | T of Tree × Elem.T × Tree
```

Однако в отличие от несбалансированных двоичных деревьев поиска из Раздела 2.2, мы позволяем дереву содержать повторяющиеся элементы. Эта разница не является фундаментальным различием расширяющихся деревьев и несбалансированных двоичных деревьев поиска; она просто отражает отличие абстракции множества от абстракции кучи.

Рассмотрим следующую стратегию реализации для insert: разобьем существующее дерево на два поддерева, чтобы одно содержало все элементы, меньше или равные новому, а второе все элементы, большие нового. Затем породим новый узел из нового элемента и двух этих поддеревьев. В отличие от вставки в обыкновенное двоичное дерево поиска, эта процедура добавляет элемент как корень дерева, а не как новый лист. Код для insert выглядит просто как

```
fun insert (x, t) = T (smaller (x, t), x, bigger (x, t))
```

где smaller выделяет дерево из элементов, меньше или равных x, а bigger дерево из элементов, больших x. По аналогии с фазой разделения быстрой сортировки, назовем новый элемент *границей* (pivot).

Можно наивно реализовать bigger как

```
fun bigger (pivot, E) = E
  | bigger (pivot, T (a, x, b)) =
      if x ≤ pivot then bigger (pivot, b)
      else T (bigger (pivot, a), x, b)
```

однако при таком решении не делается никакой попытки перестроить дерево, добиваясь лучшего баланса. Вместо этого мы применяем простую эвристику для перестройки: каждый раз, пройдя по двум левым ветвям подряд, мы проворачиваем два пройденных узла.

```
fun bigger (pivot, E) = E
  | bigger (pivot, T a, x, b)) =
      if x ≤ pivot then bigger (pivot, b)
      else case a of
```

[1] В Стандартном ML можно было бы хранить корень расширяющегося дерева в ссылочной ячейке и обновлять значение по ссылке при каждом запросе, но такое решение не является чисто функциональным.

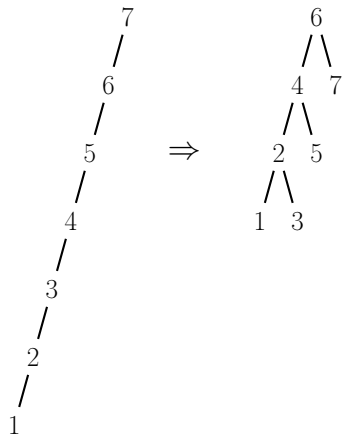


Рис. 5.4: Вызов функции bigger с граничным элементом 0.

```
  E ⇒ T (E, x, b)
| T (a1, y, a2) ⇒
    if y ≤ pivot then T (bigger (pivot, a2), x, b)
    else T (bigger (pivot. a1), y, T (a2, x, b))
```

На Рис. 5.4 показано, как bigger действует на сильно несбалансированное дерево. Несмотря на то, что результат по-прежнему не является сбалансированным в обычном смысле, новое дерево намного сбалансированнее исходного; глубина каждого узла уменьшилась примерно наполовину, от d до $\lfloor d/2 \rfloor$ или $\lfloor d/2 \rfloor + 1$. Разумеется, мы не всегда можем уполовинить глубину каждого узла в дереве, но мы можем уполовинить глубину каждого узла, лежащего на пути поиска. В сущности, в этом и состоит принцип расширяющихся деревьев: нужно перестраивать путь поиска так, чтобы глубина каждого лежащего на пути узла уменьшалась примерно вполовину.

Упражнение 5.4 *Реализуйте операцию smaller. Не забудьте, что smaller должна сохранять элементы, равные границе (однако устраивать отдельную проверку на равенство не следует!).*

Заметим, что smaller и bigger вегда проходят по одному и тому же пути поиска. Вместо того, чтобы повторять это прохождение дважды, можно соединить smaller и bigger в единую функцию с названием partition, которая вернет оба результата в виде пары. Написание этой функции не представляет труда, но несколько утомительно.

```
fun partition (pivot, E) = (E, E)
  | partition (pivot, t as T (a, x, b)) =
    if x ≤ pivot then
      case b of
        E ⇒ (t, E)
      | T (b1, y, b2) ⇒
```

Рис. 5.5: Реализация кучи через расширяющиеся деревья.

```
        if y ≤ pivot then
            let val (small, big) = partition (pivot, b2)
            in (T (T (a, x, b1), y, small), big) end
        else
            let val (small, big) = partition (pivot, b1)
            in (T (a, x, small), T (big, y, b2)) end
  else
    case a of
      E ⇒ (E, t)
    | T (a1, y, a2) ⇒
        if y ≤ pivot then
            let val (small, big) = partition (pivot, a2)
            in (T (a1, y, small), T (big, x, b)) end
        else
            let val (small, big) = partition (pivot, a1)
            in (small, T (big, y, T (a2, x, b))) end
```

Замечание 5.2 *Эта функция не является точным эквивалентом smaller и bigger из-за расхождения фаз: partition всегда обрабатывает узлы парами, а smaller и bigger иногда проходят по одному узлу. Поэтому иногда smaller и bigger оборачивают не те же самые узлы, что partition. Однако ни к каким важным последствиям это расхождение не приводит.*

Рассмотрим теперь findMin и deleteMin. Минимальный элемент расширяющегося дерева хранится в самой левой его вершине типа T. Найти эту вершину несложно.

```
fun findMin (T (E, x, b)) = x
  | findMin (T (a, x, b)) = findMin a
```

Функция deleteMin должна уничтожить самый левый узел и одновременно перестроить дерево таким же образом, как это делает bigger. Поскольку мы всегда рассматриваем только левую ветвь, сравнения не нужны.

```
fun deleteMin (T (E, x, b)) = b
  | deleteMin (T (T (E, x, b), y, c)) = T (b, y, c)
  | deleteMin (T (T (a, x, b), y, c)) = T (deleteMin a, x, T (b, y, c))
```

На Рис. 5.5 реализация расширяющихся деревьев приведена целиком. Для полноты мы включили в нее функцию слияния merge, хотя она довольно неэффективна и для многих входов занимает $O(n)$ времени.

Теперь мы хотим показать, что insert выполняется за время $O(\log n)$. Пусть $\#t$ обозначает размер дерева t плюс один. Заметим, что если $t = \mathrm{T}(a, x, b)$, то $\#t = \#a + \#b$. Пусть потенциал вершины $\phi(t)$ равен $\log(\#t)$, а потенциал всего дерева равен сумме потенциалов его вершин. Нам требуется следующее элементарное утверждение, касающееся логарифмов:

Лемма 5.1 *Для всех положительных x, y, z, таких, что $y + z \leq x$,*

$$1 + \log y + \log z < 2 \log x$$

Доказательство. Без потери общности предположим, что $y \leq z$. Тогда $y \leq x/2$ и $z \leq x$, так что $1 + \log y \leq \log x$ и $\log z < \log x$

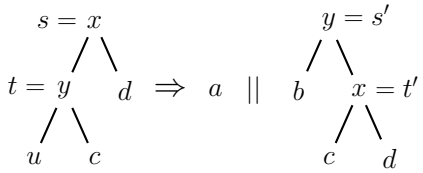
Пусть $\mathcal{T}(t)$ обозначает реальную стоимость вызова partition для дерева t, что определяется как число рекурсивных вызовов partition. Пусть $\mathcal{A}(t)$ — амортизированная стоимость такого вызова, определяемая как

$$\mathcal{A}(t) = \mathcal{T}(t) + \Phi(a) + \Phi(b) - \Phi(t)$$

где a и b — возвращаемые функцией partition поддеревья.

Теорема 5.2 $\mathcal{A}(t) \leq 1 + 2\phi(t) = 1 + 2\log(\#t)$
Доказательство. Требуется рассмотреть два нетривиальных случая, называемые зиг-зиг и зиг-заг, в зависимости от того, проходит ли вызов partition по двум левым ветвям (или, симметрично, по двум правым), либо по левой ветке, а затем правой (или, симметрично, по правой, а затем по левой).

Для случая зиг-зиг предположим, что исходное и результирующее дерево имеют формы



где a и b являются результатами вызова partition (pivot, u). Тогда

$$\begin{array}{ll}
 & \mathcal{A}(s) \\
= & \quad \{ \text{ по определению } \mathcal{A} \} \\
 & \mathcal{T}(s) + \Phi(a) + \Phi(s') - \Phi(s) \\
= & \quad \{ \mathcal{T}(s) = 1 + \mathcal{T}(u) \} \\
 & 1 + \mathcal{T}(u) + \Phi(a) + \Phi(s') - \Phi(s) \\
= & \quad \{ \mathcal{T}(u) = \mathcal{A}(u) - \Phi(a) - \Phi(b) + \Phi(u) \} \\
 & 1 + \mathcal{A}(u) - \Phi(a) - \Phi(b) + \Phi(u) + \Phi(a) + \Phi(s') - \Phi(s) \\
= & \quad \{ \text{ раскрываем } \Phi(s) \text{ и } \Phi(s'), \text{ упрощаем } \} \\
 & 1 + \mathcal{A}(u) + \phi(s') + \phi(t') - \phi(s) - \phi(t) \\
\leq & \quad \{ \text{ по предположению индукции, } \mathcal{A}(u) \leq 1 + 2\phi(u) \} \\
 & 2 + 2\phi(u) + \phi(s') + \phi(t') - \phi(s) - \phi(t) \\
< & \quad \{\phi(u) < \phi(t), \text{ а } \phi(s') \leq \phi(s)\} \\
 & 2 + \phi(u) + \phi(t') \\
< & \quad \{ \#u + \#t' < \#s, \text{ а также Лемма 5.1 } \} \\
 & 1 + 2\phi(s)
\end{array}$$

Доказательство случая зиг-заг мы оставляем как упражнение для читателя.

Упражнение 5.5 *Докажите случай зиг-заг.*

Дополнительная стоимость операции insert по сравнению с partition составляет один реальный шаг плюс разница потенциалов между двумя поддеревьями-результатами partition и деревом-окончательным результатом insert. Это изменение потенциала равно просто ϕ от нового корня. Поскольку амортизированная стоимость partition ограничена $1+2\log(\#t)$, амортизированная стоимость insert ограничена $2+2\log(\#t)+\log(\#t+1) \approx 2+3\log(\#t)$.

Упражнение 5.6 *Докажите, что стоимость deleteMin также составляет $O(\log n)$.*

Какова ситуация с findMin? Если дерево сильно несбалансированно, findMin может занять до $O(n)$ времени. Причем поскольку findMin не проводит никакой перестройки и, следовательно, никак не изменяет потенциал, амортизировать эту стоимость негде! Однако раз время findMin пропорционально времени deleteMin, мы можем увеличить стоимость, взимаемую за deleteMin, вдвое, и один раз на каждый ее вызов бесплатно звать findMin. Этого достаточно для тех приложений, которые всегда зовут findMin и deleteMin вместе. Однако в некоторых приложениях findMin может вызываться по несколько раз на каждый вызов deleteMin. Для этих приложений мы не будем напрямую вызывать функтор SplayHeap, а будем его использовать в комбинации с функтором ExplicitMin из Упражнения 3.7. Напомним, что задачей функтора ExplicitMin было обеспечить выполнение findMin за время $O(1)$. Функции insert и deleteMin по-прежнему будут выполняться за время $O(\log n)$.

Указание разработчикам 5.2 *Расширяющиеся деревья, дополняемые при необходимости функтором ExplicitMin, — самая быстрая из известных реализаций кучи для большинства приложений, не требующих устойчивости данных и не вызывающих функцию merge.*

Особенно приятным свойством расширяющихся деревьев является то, что они естественным образом подстраиваются под любой порядок, присутствующий во входных данных. Например, при использовании расширяющихся деревьев для сортировки уже сортированного заранее списка тратится всего $O(n)$ времени, а не $O(n \log n)$ [MEP96]. Тем же свойством обладают левоориентированные кучи, но только для уменьшающихся последовательностей. Расширяющиеся кучи отлично себя ведут как на растущих, так и на уменьшающихся последовательностях, а также на последовательностях, отсортированных лишь частично.

Упражнение 5.7 *Напишите функцию сортировки, которая складывает элементы в расширяющееся дерево, а затем обходит его по порядку, выводя элементы в список. Покажите, что на уже отсортированном списке она работает за время всего $O(n)$.*

Рис. 5.6: Парные кучи.

5.5 Парные кучи

Парные кучи (pairing heaps) [FSST86] — одна из тех структур, которые сводят специалистов с ума. С одной стороны, их легко реализовать и они весьма хорошо показали себя на практике. С другой стороны, провести их полный анализ не удается уже более 10 лет!

Парные кучи представляют собой упорядоченные по принципу кучи деревья с переменной степенью ветвления; их можно определить следующим типом данных:

```
datatype Heap = E | T of Elem.T × Heap list
```

Мы считаем правильными только такие деревья, где E никогда не встречается в качестве ребенка вершины T.

Поскольку деревья упорядочены по принципу кучи, функция findMin тривиальна:

```
fun findMin (T (x, hs)) = x
```

Функции merge и insert ненамного сложнее. merge добавляет то дерево, чей корень больше, в качестве первого ребенка того дерева, чей корень меньше. insert сначала создает новое дерево с одним элементом, а затем зовет merge.

```
fun merge (h, E) = h
  | merge (E, h) = h
  | merge (h1 as T (x, hs1), h2 as T (y, hs2)) =
      if Elem.leq (x, y) then T (x, h2 :: hs1) else T (y, h1 :: hs2)
fun insert (x, h) = merge (T(x, []), h)
```

Парные деревья называются именно так благодаря операции deleteMin. Эта операция отбрасывает корень, а затем сливает деревья в два прохода. Первый проход сливает деревья парами слева направо (т. е., первое дерево сливается со вторым, третье с четвертым и т. д.). При втором проходе получившиеся деревья сливаются справа налево. Эти два прохода можно кратко выразить так:

```
fun mergePairs [] = E
  | mergePairs [h] = h
  | mergePairs (h1 :: h2 :: hs) =merge (merge (h1, h2), mergePairs hs)
```

После этого deleteMin выглядит совсем просто:

```
fun deleteMin (T (x, hs)) = mergePairs hs
```

Полная реализация приведена на Рис. 5.6

Легко видеть, что findMin, insert и merge занимают каждая по $O(1)$ времени. Однако в худшем случае deleteMin может отнять до $O(n)$. По аналогии с расширяющимися деревьями (см. Упражнение 5.8) мы можем показать,

что insert , merge и deleteMin каждая отнимает по $O(\log n)$ амортизированного времени. Существует предположение, что insert и merge на самом деле работают за амортизированное время $O(1)$ [FSST86], но его до сих пор никому не удалось ни доказать, ни опровергнуть.

Указание разработчикам 5.3 *В приложениях, где не требуется функция merge, парные кучи работают почти так же быстро, как расширяющиеся кучи, а если merge требуется, то они значительно быстрее. Подобно расширяющимся кучам, их следует применять только в тех приложениях, где устойчивость не требуется.*

Упражнение 5.8 *Часто проще оказывается работать с двоичными деревьями, чем с деревьями с произвольным ветвлением. К счастью, любое дерево с произвольным ветвлением легко представить в виде двоичного. Достаточно преобразовать каждый узел со списком детей в двоичный узел, где левый ребенок представляет самого левого ребенка исходного узла, а правый потомок представляет его сестринский узел непосредственно справа. Если отсутствуют либо левый узел, либо правый сосед, то соответствующий узел двоичного дерева оказывается пустым. (Заметим, что таким образом в двоичном представлении правый потомок корневого узла всегда оказывается пуст.) Применив такое преобразование к парной куче, мы получаем полуупорядоченные двоичные деревья, где элемент в каждом узле не больше любого элемента в своем левом дочернем поддереве.*

1. *Напишите функцию toBinary, преобразующую парные кучи из исходного представления в тип*

$$\textbf{\textit{datatype}}\ BinTree = E' \mid T'\ \textbf{\textit{of}}\ Elem.T \times BinTree \times BinTree$$

2. *Заново реализуйте парные кучи, используя это новое представление.*

3. *Модифицируйте анализ расширяющихся деревьев и докажите, что deleteMin и merge работают за амортизированное время $O(\log n)$ в этом новом представлении (а следовательно, и в старом тоже). Следует использовать ту же самую функцию потенциала, как и в расширяющихся деревьях.*

5.6 Плохая новость

Как мы могли убедиться, амортизированные структуры могут быть чрезвычайно эффективны на практике. К сожалению, все рассуждения в этой главе неявно предполагают, что анализируемые структуры данных используются эфемерным образом (то есть, только одной нитью последовательных операций). Что произойдет, если мы попытаемся с теми же самыми структурами обращаться как с устойчивыми?

Рассмотрим очереди из Раздела 5.2. Пусть q будет очередь, получаемая вставкой n элементов в изначально пустую очередь, так что головной список q содержит один элемент, а хвостовой $n-1$ элементов. Теперь предположим, что мы считаем очередь устойчивой и n раз удаляем первый элемент. Каждый из этих вызовов отнимет n реальных шагов. Общая реальная стоимость этой последовательности операций, включая изначальное построение q, равна n^2+n. Если бы операции на самом деле отнимали только по $O(1)$ амортизированного времени, общая реальная стоимость была бы всего $O(n)$. Таким образом, ясно, что использование наших очередей как устойчивой структуры нарушает установленные в Разделе 5.2 амортизированные ограничения стоимости $O(1)$. Где же ошибка в доказательствах?

В обоих случаях одно из основных предположений доказательства оказывается нарушенным при рассмотрении структуры как устойчивой. В методе банкира требуется, чтобы каждая единица кредита тратилась не более одного раза, а метод физика требует, чтобы результат одной операции служил аргументом следующей (или, в более общей формулировке, чтобы всякий результат операции использовался как аргумент другой не более одного раза). Рассмотрим второе обращение к `tail q` в вышеописанном примере. Первое обращение тратит весь кредит, накопленный в хвостовом списке q, и оказывается нечем оплатить второй и последующие вызовы, так что метод банкира терпит неудачу. Кроме того, второе обращение к `tail q` повторно использует q, а не результат первого вызова, так что метод физика тоже не работает.

Обе неудачных попытки доказательства отражают слабость всякой системы подсчета, основанной на накоплениях — то, что эти накопления можно потратить лишь один раз. Традиционные методы амортизации работают путем накопления единиц работы (либо кредита, либо потенциала) для дальнейшего использования. Это отлично работает при эфемерном использовании, когда у каждой операции лишь одно логическое будущее. Но у операции над устойчивой структурой может быть сколько угодно логических будущих, и в каждом из них структура может пытаться потратить одни и те же накопления.

В следующей главе мы разъясним, что имеется в виду под «логическим будущим» операции, и как можно совместить амортизацию и устойчивость через ленивое вычисление.

Упражнение 5.9 *Приведите примеры последовательности операций, где биномиальные кучи, расширяющиеся кучи и парные кучи отнимают намного больше времени, чем указывают амортизированные границы их стоимости.*

5.7 Примечания

Методы амортизации, обсуждаемые в этой главе, были разработаны Слейтором и Тарьяном [ST85, ST86b]. Они стали популярны благодаря Тарьяну [Tar85]. Схунмакерс [Sch92] показывает, как систематическим образом

получать амортизированные оценки стоимости при функциональном программировании без использования устойчивости.

Кучи из Раздела 5.2 были предложены Грисом [Gri81, с. 250-251], а также Худом и Мелвиллом [HM81]. Бёртон [Bur82] предложил похожую реализацию, однако без ограничения, чтобы у непустой кучи головной список всегда был непуст. У Бёртона head и tail объединены в одну функцию, и, таким образом, нет требования, чтобы head по отдельности была эффективна.

В нескольких экспериментальных исследованиях было показано, что расширяющиеся кучи [Joh86] и парные кучи [MS91, Lia92] — одни из самых быстрых реализаций для этой абстракции. Стаско и Виттер [SV87] подтвердили для варианта парных куч предполагаемое амортизированное ограничение $O(1)$ на вставку.

Глава 6

Сочетание амортизации и устойчивости через ленивое вычисление

В предыдущей главе мы представили понятие амортизации и привели несколько примеров структур данных с хорошими амортизированными показателями производительности. Однако все эти показатели для всех этих структур перестают быть применимы, если их использовать как устойчивые. В этой главе мы покажем, как ленивое вычисление может разрешить конфликт между амортизацией и устойчивостью, и модифицируем методы банкира и физика, чтобы они учитывали особенности ленивого вычисления. Затем мы демонстрируем применение наших методов к нескольким амортизированным структурам данных, использующим ленивое вычисление в своей реализации.

6.1 Трассировка вычисления и логическое время

В предыдущей главе мы заметили, что традиционные методы амортизации ломаются при наличии устойчивости, поскольку они предполагают наличие у структуры единственного будущего, где накопленные сбережения будут потрачены только один раз. Однако в устойчивой структуре несколько будущих логических историй могут одновременно пытаться использовать одни и те же сбережения. Однако что же мы имеем в виду, говоря о «логическом будущем» операции?

Мы моделируем логическое время при помощи *трассировок вычисления* (execution traces), которые представляют абстракцию истории выполнения программы. Трассировка вычисления представляет собой направленный граф, вершины которого соответствуют операциям, которые нас инте-

ресуют; как правило, это только операции модификации над рассматриваемым типом данных. Дуга от вершины v к вершине v' означает, что операция v' использует результат операции v. *Логической историей* (logical history) операции v (обозначается $\hat{v}$) называется множество всех операций, от которых зависит её результат (включая и саму операцию v). Другими словами, $\hat{v}$ — множество вершин w, таких, что существует путь (возможно, длины 0) от w до v. *Логическим будущим* (logical future) вершины v называется любой путь от v до конечной вершины (т. е., вершины с числом исходящих дуг 0). Если таких путей больше одного, значит, вершина v имеет несколько логических будущих. Иногда мы говорим о логической истории или логическом будущем объекта, имея при этом в виду логическую историю или будущее операции, создавшей этот объект.

Упражнение 6.1 *Нарисуйте трассировку вычисления для следующей последовательности операций. Пометьте каждую вершину в графе количеством её логических будущих.*

```
val a = snoc (empty, 0)
val b = snoc (a, 1)
val c = tail b
val d = snoc (b, 2)
val e = c ++ d
val f = tail c
val g = snoc (d, 3)
```

Понятие трассировки вычисления обобщает *графы версий* (version graphs) [DSST89], часто используемые для моделирования историй устойчивых структур данных. В графе версий вершины представляют различные версии единой устойчивой структуры, а дуги соответствуют зависимостям между этими версиями. Таким образом, графы версий моделируют результаты операций, а трассировки вычисления — операции сами по себе. Трассировки вычисления часто оказываются удобнее, если надо совместить истории нескольких устойчивых объектов (возможно, даже разных типов), а также для рассуждений об операциях, не изменяющих версию объекта (например, о запросах) либо возвращающих несколько результатов (скажем, разбивающих список на два подсписка).

Для эфемерных структур данных, как правило, число исходящих дуг в графе версий или в трассировке вычисления должно быть не более единицы; это отражает ограничение, что каждая структура может модифицироваться не более одного раза. Для моделирования различных вариантов устойчивости графы версий могут позволять числу исходящих дуг вершины быть каким угодно, но вводить другие ограничения. Например, часто требуют, чтобы графы версий были деревьями (или лесами), говоря, что число входящих дуг для каждой вершины не может превышать 1. Или же разрешается больше одной входящей дуги у вершины, но запрещаются циклы, и таким образом, граф оказывается направленным ациклическим графом. Мы никаких таких ограничений для трассировок выполнения устойчивых

структур данных не накладываем. Вершины с числом входящих дуг более одной соответствуют операциям, принимающим более одного аргумента, например, конкатенации списков или объединению множеств. Циклы возникают для рекурсивно определенных объектов, которые поддерживаются во многих ленивых языках. Разрешено даже иметь несколько дуг между одними и теми же вершинами, например, когда список конкатенируется сам с собой.

Трассировки вычисления будут использоваться в Разделе 6.3.1, где мы расширяем метод банкира для работы с устойчивыми структурами.

6.2 Сочетание амортизации и устойчивости

В этом разделе мы показываем, как можно исправить методы банкира и физика, заменив понятие текущих накоплений понятием текущего долга, который представляет стоимость невыполненных ленивых вычислений. Интуиция здесь состоит в том, что в то время, как накопления можно тратить только один раз, нет никакого вреда в многократном выплачивании долга.

6.2.1 Роль ленивого вычисления

Напомним, что *дорогой* (expensive) называется операция, чья реальная стоимость превышает её (желательную) амортизированную стоимость. Предположим, к примеру, что некоторый вызов функции f x является дорогим. При наличии устойчивости вредоносный противник может вызывать f x сколь угодно часто. (Заметим, что каждый такой вызов образует новое логическое будущее x.) Если каждая такая операция занимает одно и то же время, амортизированные ограничения на время вычисления деградируют до наихудших ограничений. Следовательно, нам надо добиться того, чтобы даже если первое вычисление f x окажется дорогим, последующие вызовы таковыми не были.

При программировании без побочных эффектов такая цель недостижима ни при вызове по значению (т. е., при энергичном порядке вычислений), ни при вызове по имени (т. е., при ленивом вычислении без мемоизации), поскольку всякое применение функции f к аргументу x занимает одно и то же время. Следовательно, амортизацию невозможно выгодно совместить с устойчивостью в языках, поддерживающих только эти два порядка вычисления.

Однако рассмотрим теперь вызов по необходимости (т. е., ленивое вычисление с мемоизацией). Если x содержит задержанный компонент, необходимый для вычисления f, первое применение f к x вынудит (возможно, дорогое) вычисление этого компонента и запомнит результат. Последующие операции смогут обращаться к результату напрямую. Ровно это нам и требовалось!

Замечание 6.1 *Будучи однажды обнаруженной, связь ленивого вычисления и амортизации кажется естественной. Ленивое вычисление мож-*

но рассматривать как разновидность самомодификации, а самомодификация часто используется при амортизации [ST85, ST86b]. Однако ленивое вычисление является особым образом ограниченной разновидностью самомодификации — не все виды самомодификации, используемые в амортизированных эфемерных структурах данных, могут быть выражены при помощи ленивого вычисления. В частности, расширяющиеся деревья, по-видимому, этому методу неподвластны.

6.2.2 Общая методика анализа ленивых структур данных

Как мы только что показали, ленивое вычисление необходимо для чисто функциональной реализации амортизированных структур данных. Но программы с ленивым вычислением знамениты тем, что анализ времени их работы чрезвычайно сложен. Наиболее обычный способ анализа ленивых программ состоит в том, чтобы притвориться, что они на самом деле используют энергичный порядок. Однако для анализа амортизированных структур данных этот способ совершенно непригоден. Ниже мы описываем базовую методику, позволяющую проводить такой анализ. В оставшейся части главы мы с помощью этой методики модифицируем методы банкира и физика. В результате мы получаем первые в истории методы анализа устойчивых амортизированных структур данных и первые практически применимые методы анализа нетривиальных ленивых программ.

Стоимость каждой операции мы разбиваем на несколько категорий. Во-первых, *нераздельная* (unshared) стоимость операции — это время, требуемое операции в предположении, что все задержки в системе уже вынуждены и мемоизированы ко времени её начала (т. е., в предположении, что force всегда занимает время $O(1)$, за исключением тех задержек, которые создаются и вынуждаются в процессе выполнения самой операции). *Разделяемая* (shared) стоимость операции — это время, требуемое для выполнения всех задержек, создаваемых, но не вынуждаемых операцией (при тех же предположениях, что и раньше). Наконец, *полная* (complete) стоимость операции есть сумма её нераздельной и разделяемой стоимости. Заметим, что полная стоимость операции равна её стоимости, если бы ленивое вычисление было заменено на энергичное.

Кроме того, мы разбиваем разделяемую стоимость последовательности операций на реализованную и нереализованную. *Реализованная* (realized) стоимость есть стоимость задержек, которые вынуждаются в процессе полного вычисления. *Нереализованная* (unrealized) стоимость — стоимость задержек, которые так и остаются невыполненными. *Общая реальная* (total actual) стоимость последовательности операций равняется сумме общей нераздельной стоимости и реализованной разделяемой стоимости — нереализованные вычисления не влияют на общую стоимость. Заметим, что доля каждой операции в общей реальной стоимости не меньше её нераздельной стоимости и не больше её полной стоимости, в зависимости от того, какая доля разделяемой стоимости реализуется.

Мы будем учитывать разделяемую стоимость с помощью понятия *текущего долга* (accumulated debt). В начале вычисления долг равен нулю, но каждый раз, когда создается задержка, он увеличивается на разделяемую стоимость этой задержки (а также вложенных в неё задержек). Впоследствии каждая операция выплачивает часть текущего долга. *Амортизированная стоимость* (amortized cost) операции равна сумме её нераздельной стоимости и количества выплаченного этой операцией долга. Нам запрещается вынуждать задержку, прежде чем полностью выплачен связанный с ней долг.

Замечание 6.2 *Амортизационный анализ на основе понятия текущего долга во многом работает как отложенная покупка (layaway plan). В случае отложенной покупки вы находите в магазине некоторый товар — например, кольцо с бриллиантом, — который вы не можете позволить себе немедленно. Вы договариваетесь с магазином о цене и просите персонал отложить для вас кольцо. Затем вы производите регулярные платежи, и получаете кольцо только тогда, когда его цена полностью выплачена.*

При анализе ленивой структуры данных вы имеете вычисление, которое пока что не можете позволить себе немедленно. Вы создаете для этого вычисления задержку и присваиваете ей размер долга, пропорциональный её разделяемой стоимости. Затем вы выплачиваете долг небольшими порциями. Наконец, когда долг полностью выплачен, вам позволено произвести вычисление задержки.

В жизненном цикле задержки есть три важных момента: когда она создается, когда её стоимость полностью оплачена, и когда она выполняется. Мы обязаны доказать, что второй из этих моментов предшествует третьему. Если каждая задержка до своего вынуждения полностью оплачена, то общее количество выплаченного долга является верхней границей для реализованной разделяемой стоимости, а следовательно, общая амортизированная стоимость (т. е., общая нераздельная стоимость плюс общее количество выплаченного долга) является верхней границей для общей реальной стоимости (т. е., общей нераздельной стоимости плюс реализованная разделяемая стоимость). Мы сделаем этот аргумент формальным в Разделе 6.3.1.

Одна из наиболее трудных проблем при анализе времени выполнения ленивых программ — взаимодействие множественных логических будущих. Мы избегаем этой проблемы, рассуждая о каждом из этих будущих *как если бы оно было единственным.* С точки зрения операции, создающей задержку, каждое логическое будущее, эту задержку вынуждающее, обязано само её оплатить. Если два логических будущих желают вынудить одну и ту же задержку, каждое из них платит за неё по отдельности. Сговориться и выплатить долг по частям не разрешается. Альтернативный взгляд на это ограничение состоит в том, что задержку разрешается вынуждать *только тогда, когда ее стоимость оплачена в рамках логической истории текущей операции.* При использовании этого метода иногда мы будем выплачивать долг более одного раза, и следовательно, переоценивать общее

время, необходимое для некоторых вычислений. Однако такая переоценка безвредна, и её цена невелика по сравнению с простотой получаемого анализа.

6.3 Метод банкира

Чтобы приспособить метод банкира к использованию понятия текущего долга вместо текущих накоплений, мы заменяем кредит дебетом. Каждая единица долга представляет определенное количество отложенной работы. Когда мы вначале задерживаем какое-то вычисление, мы создаём дебет, равный разделяемой стоимости этого вычисления, и распределяем долг по узлам созданного объекта. Выбор места, с которым связывается каждая единица долга, зависит от природы вычисления. Если оно *монолитно* (monolithic) (то есть, будучи однажды запущено, сработает до завершения), обычно весь долг присваивается корневому узлу результата. С другой стороны, если мы имеем дело с *пошаговым* (incremental) вычислением (то есть, оно разбивается на фрагменты, которые можно выполнить независимо друг от друга), то долг может распределяться по корневым узлам частичных результатов.

Амортизированная стоимость операции равна её нераздельной стоимости плюс количество единиц долга, освобождаемых этой операцией. Обратите внимание, что единицы долга, создаваемые операцией, в ее амортизированную стоимость *не включаются.* Порядок, в котором высвобождаются единицы долга, зависит от того, как предполагается в будущем обращаться к узлам объекта; долг на тех узлах, к которым мы обратимся раньше, следует выплачивать в первую очередь. Чтобы доказать ограничение амортизированной стоимости операции, нам нужно показать, что всякий раз, как мы обращаемся к некоторой ячейке (возможно, при этом вынуждая некоторую задержку), все единицы долга, привязанные к этой ячейке, уже высвобождены (а следовательно, задержанная операция полностью оплачена). Таким образом, мы гарантируем, что общее количество долга, высвобожденное последовательностью операций, является верхней границей реализованной разделяемой стоимости этих операций. Следовательно, общая амортизированная стоимость является верхней границей общей реальной стоимости. Долг, сохраняющийся в конце вычислений, соответствует нереализованной разделяемой стоимости, и не влияет на общую реальную стоимость.

Пошаговые функции играют важную роль в методе банкира, поскольку они позволяют распределить долг по различным ячейкам структуры данных, каждая из которых соответствует вложенной задержке. Впоследствии доступ к каждой ячейке может быть открыт по мере высвобождения долга, не ожидая выплаты долга по другим ячейкам. На практике это означает, что можно очень быстро оплатить начальные частичные результаты пошагового вычисления, а последующие частичные результаты оплачиваются по мере необходимости. Однако монолитные функции дают намного мень-

шую гибкость. Программист вынужден предсказывать, когда понадобится результат дорогого монолитного вычисления, и обустроить высвобождение долга достаточно рано, чтобы ко времени, когда результат понадобится, он был полностью выплачен.

6.3.1 Обоснование метода банкира

В этом разделе мы доказываем утверждение, что общая амортизированная стоимость является верхней границей общей реальной стоимости. Общая амортизированная стоимость равна сумме общей нераздельной стоимости и общего количества высвобожденного долга (включая повторные высвобождения); общая реальная стоимость равна общей нераздельной стоимости плюс реализованная разделяемая стоимость. Следовательно, нам надо показать, что общее количество высвобожденного долга является верхней границей для реализованной разделяемой стоимости.

Можно абстрактно рассматривать метод банкира как задачу разметки графа трассировки вычисления из Раздела 6.1. Задача состоит в том, чтобы каждую вершину графа пометить тремя (мульти)множествами $s(v)$, $a(v)$ и $r(v)$, так что

$$\begin{array}{ll} (I) & v \neq v' \Rightarrow s(v) \cap s(v') = \emptyset \\ (II) & a(v) \subseteq \bigcup_{w \in \hat{v}} s(w) \\ (III) & r(v) \subseteq \bigcup_{w \in \hat{v}} a(w) \end{array}$$

$s(v)$ является множеством, но $a(v)$ и $r(v)$ могут быть мультимножествами (т. е., могут содержать повторения). Условия II и III не учитывают эти повторения.

$s(v)$ — это множество дебетов, выделяемых операцией v. Условие I утверждает, что никакая единица долга не выделяется более одного раза. $a(v)$ — множество дебетов, высвобождаемых операцией v. Условие II требует, чтобы всякая высвобождаемая единица долга была заранее выделена; точнее, операция может высвобождать только единицы долга, которые были выделены в её логической истории. Наконец, $r(v)$ — это единицы долга, *реализуемые* (realized) операцией v, то есть, мультимножество единиц долга, соответствующее задержкам, которые вынуждаются операцией v. Условие III требует, чтобы всякая единица долга была высвобождена, прежде чем она реализована, или, точнее, что нельзя реализовать единицу долга, если она не высвобождена в логической истории текущей операции.

Почему $a(v)$ и $r(v)$ являются не просто множествами, а мультимножествами? Потому что одна и та же операция может высвободить одни и те же единицы долга более одного раза или реализовать их более одного раза (многократно вынудив одни и те же задержки). Несмотря на то, что мы никогда не имеем намерения высвободить одни и те же единицы долга многократно, при сочетании объекта с самим собой это может произойти. Предположим, например, что в анализе функции конкатенации списков мы высвобождаем какое-то количество единиц долга из первого аргумента и какое-то из второго. Если мы будем строить конкатенацию списка с самим

собой, возможно, одни и те же единицы долга окажутся высвобожденными дважды.

При таком абстрактном взгляде на метод банкира мы легко можем измерить различные показатели стоимости вычисления. Пусть V будет множество всех вершин в трассировке вычисления. В таком случае общая разделяемая стоимость равна $\sum_{v \in V} |s(v)|$, а общее количество высвобожденного долга равно $\sum_{v \in V} |a(v)|$. Поскольку имеется мемоизация, реализованная разделяемая стоимость равна не $\sum_{v \in V} |r(v)|$, а $\bigcup_{v \in V} |r(v)|$, где операция $\bigcup$ отбрасывает повторения. Таким образом, многократно вынужденная задержка участвует в реальной стоимости только один раз. Согласно Условию III, мы знаем, что $\bigcup_{v \in V} r(v) \subseteq \bigcup_{v \in V} a(v)$. Следовательно,

$$|\bigcup_{v \in V} r(v)| \leq |\bigcup_{v \in V} a(v)| \leq \sum_{v \in V} |a(v)|$$

Таким образом, реализованная разделяемая стоимость ограничена сверху общим количеством высвобожденного долга, а общая реальная стоимость ограничена общей амортизированной стоимостью, что и требовалось доказать.

Замечание 6.3 *В этом рассуждении ещё раз подчеркивается важность мемоизации. Без мемоизации (то есть, при вызове по имени вместо вызова по необходимости) общая реализованная стоимость была бы равна* $\sum_{v \in V} |r(v)|$*, и не было бы никаких причин считать, что она меньше, чем* $\sum_{v \in V} |a(v)|$.

6.3.2 Пример: очереди

В этом подразделе мы разрабатываем эффективную устойчивую реализацию очередей и доказываем методом банкира, что все операции занимают амортизированное время $O(1)$.

Как видно из обсуждения в предыдущем разделе, мы должны каким-то образом внести в устройство нашей структуры данных ленивое вычисление, так что мы заменяем пару списков из простых очередей (Раздел 5.2) на пару потоков[1]. Для упрощения дальнейшей работы мы также явно отслеживаем длину обоих этих потоков.

type α Queue =int $\times$ α Stream $\times$ int $\times$ α Stream

Первое целое число здесь — длина головного потока, а второе — длина хвостового потока. Заметим, что, в качестве приятного побочного эффекта, благодаря явному хранению информации о длине мы тривиальным образом можем поддержать функцию size (размер) с константным временем выполнения.

Если мы будем, прежде чем обратить хвостовой список, ждать, пока головной опустеет, у нас не окажется достаточно времени, чтобы заплатить за

[1]Было бы достаточно заменить потоком только список-голову, однако для простоты мы заменяем оба списка.

Рис. 6.1: Амортизированные очереди с использованием метода банкира.

обращение. Вместо этого мы периодически *проворачиваем* (rotate) очередь, заменяя f на f ++ reverse r, и делая хвостовой поток пустым. Заметим, что это преобразование не меняет относительный порядок элементов.

Когда следует проворачивать очередь? Напомним, что функция reverse монолитна. Следовательно, её вычисление должно быть спланировано достаточно сильно заранее, чтобы ко времени, когда оно понадобится, весь её долг был высвобожден. Вычисление reverse занимает $|r|$ шагов, так что, чтобы учесть её цену, мы выделяем $|r|$ единиц долга (пока что игнорируя цену операции ++). Задержка reverse может быть вынуждена самое раннее после $|f|$ применений tail, так что, если мы провернем очередь в момент, когда $|r| \approx |f|$ и высвободим по одной единице долга на каждое применение tail, ко времени исполнения reverse долг будет выплачен. Так что мы проворачиваем очередь, когда r становится на единицу длиннее f, и поддерживаем инвариант $|f| > |r|$. Между прочим, это гарантирует нам, что f может быть пустым только при пустом r, как и в простых очередях из Раздела 5.2. Основные функции работы с очередями теперь записываются так:

```
fun snoc ((lenf, f, lenr, r), x) = check (lenf, f, lenr+1, $Cons (x, r))
fun head (lenf, $Cons (x, f'), lenr, r) =x
fun tail (lenf, $Cons (x, f'), lenr, r) =check (lenf-1, f', lenr, r)
```

где вспомогательная функция check обеспечивает $|f| \geq |r|$.

```
fun check (q as (lenf, f, lenr, r) =
      if lenr ≤ lenf then q else (lenf+lenr, f ++ reverse r, 0, $Nil)
```

Полный программный код этой реализации приведен на Рис. 6.1.

Чтобы лучше объяснить, как эта реализация эффективно справляется с устойчивостью, рассмотрим следующий сценарий. Пусть имеется очередь q_0, чьи головной и хвостовой потоки каждый имеют длину m, и пусть $q_i = \text{tail}\, q_{i-1}$ для $0 < i \leq m+1$. Очередь проворачивается при первом вызове tail, а задержка reverse, созданная при провороте, вынуждается при последнем вызове tail. Обращение занимает m шагов, и его стоимость амортизируется по цепочке $q_1 \ldots q_m$. (Пока что мы заботимся только о стоимости reverse и игнорируем стоимость ++.)

Выберем теперь какую-нибудь точку ветвления k и повторим вычисление от q_k до q_{m+1}. (Заметим, что q используется как устойчивая структура.) Сделаем это d раз. Как часто выполняется reverse? Зависит от того, находится ли точка ветвления k до или после проворачивания очереди. Допустим, k расположена после проворота. Допустим даже, что $k = m$, так что каждая из повторяющихся ветвей представляет собой простое взятие хвоста очереди. Каждая из ветвей выполнения вынуждает задержку reverse, но все они вынуждают *одну и ту же* задержку, так что функция reverse выполняется только один раз. Здесь ключевую роль играет мемоизация — без неё

вычисление reverse повторялось бы каждый раз, и общая стоимость составила бы $m(d+1)$ шагов, при том, что для её амортизации у нас есть всего лишь $m+1+d$ операций. При большом d получилась бы амортизированная стоимость операции $O(1)$, но мемоизация дает нам амортизированную стоимость операции всего лишь $O(1)$.

Возможна, однако, и ситуация, когда вычисление reverse будет повторяться. Надо только взять $k=0$ (т. е., расположить точку ветвления прямо перед проворотом очереди.) В этом случае первый вызов tail на каждой ветке выполнения повторяет проворот и создает новую задержку reverse. Эта новая задержка вынуждается при последнем вызове tail на каждой ветке, и функция reverse вычисляется заново. Поскольку все задержки различны, мемоизация здесь не приносит никакой пользы. Полная стоимость всех обращений равна m · d, но теперь у нас для амортизации этой стоимости есть $(m+1)(d+1)$ операций, и опять амортизированная стоимость каждой операции получается $O(1)$. Главное, что здесь следует заметить — это что мы повторяем работу только в том случае, если повторяем и последовательность операций, по которым мы распределяем амортизированную стоимость этой работы.

Это неформальное рассуждение показывает, что наши очереди требуют амортизированное время $O(1)$ на каждую операцию, даже если они используются как устойчивая структура. Мы переводим это рассуждение в формальное доказательство с помощью метода банкира.

При простом просмотре кода легко убедиться, что нераздельная стоимость каждой операции над очередью равна $O(1)$. Следовательно, чтобы показать, что амортизированная стоимость каждой операции равна $O(1)$, нужно доказать, что высвобождение $O(1)$ единиц долга на каждой операции достаточно, чтобы выплатить стоимость каждой задержки ко времени её вынуждения. Высвобождение долга происходит только на операциях tail и snoc.

Пусть размер долга на i-ом узле головного списка будет $d(i)$, и пусть $D(i) = \sum_{j=0}^{i} d(j)$ — кумулятивный размер долга на узлах от начала очереди до i включительно. Мы будем соблюдать следующий *инвариант долга*:

$$D(i) \leq \min(2i, |f| - |r|)$$

Подвыражение $2i$ гарантирует нам, что весь долг на первом узле головного потока уже высвобожден (поскольку $d(0) = D(0) \leq 2 \cdot 0 = 0$), так что этот узел можно спокойно вынуждать (операциями head или tail). Подвыражение $|f| - |r|$ гарантирует, что весь долг во всей очереди высвобожден к моменту, когда потоки имеют одинаковую длину, что бывает перед следующим проворотом очереди.

Теорема 6.1 *Операции snoc и tail сохраняют инвариант долга, высвобождая, соответственно, одну и две единицы.*
Доказательство. Операция snoc, не вызывающая проворот, просто добавляет один элемент к хвостовому потоку, и таким образом, увеличивает

на один величину $|r|$ и уменьшает на один $|f| - |r|$. При этом инвариант оказывается нарушен в узлах, где до сих пор мы имели $D(i) = |f| - |r|$. Мы можем восстановить этот инвариант, высвободив первую единицу долга в очереди; при этом кумулятивный долг на всех последующих узлах уменьшится на единицу. Подобным образом, если операция tail не вызывает проворота очереди, она просто отбрасывает элемент из головного потока. При этом $|f|$ уменьшается на единицу (а следовательно, и $|f|-|r|$), однако, что более важно, индекс i всех остающихся узлов уменьшается на один, а следовательно, $2i$ уменьшается на два. Высвобождение первых двух единиц долга в очереди восстанавливает инвариант. Наконец, рассмотрим операцию snoc или tail, которая вызывает проворот очереди. Перед самым проворотом мы знаем, что весь долг в очереди высвобожден, так что после него единственные невысвобожденные единицы долга порождены самим этим проворотом. Если на момент проворота $|f| = m$ и $|r| = m + 1$, то мы создаём m единиц долга для конкатенации и $m + 1$ для операции reverse. Конкатенация — пошаговая операция, так что мы распределяем ее долг по единице на каждый из первых m узлов. С другой стороны, функция reverse монолитна, так что все её $m + 1$ единиц долга мы помещаем в узел m, первый узел обращенного потока. Таким образом, долг теперь распределен так, что

$$d(i) = \begin{cases} 1 & \text{если } i < m \\ m+1 & \text{если } i = m \\ 0 & \text{если } i > m \end{cases} \quad \text{и} \quad D(i) = \begin{cases} i+1 & \text{если } i < m \\ 2m+1 & \text{если } i \geq m \end{cases}$$

Это распределение нарушает инвариант в узлах 0 и m. Однако после высвобождения единицы долга в узле 0 инвариант в обоих этих узлах оказывается восстановлен.

Это рассуждение имеет стандартную структуру. Единицы долга распределяются по нескольким узлам для пошаговых функций и концентрируются в одном узле для монолитных. Инварианты долга измеряют не только количество его единиц в каждой конкретной вершине, но и размер долга на пути от корневого узла к этой вершине. Это отражает наблюдение, что для доступа к любому узлу мы должны сначала пройти через всех его предков. Следовательно, к этому времени долг на всех этих узлах тоже должен быть равен нулю.

Эта структура данных демонстрирует также тонкую деталь, касающуюся вложенных задержек: долг для вложенной задержки может быть выделен и даже высвобожден прежде, чем задержка физически создается. Рассмотрим, например, как работает операция $+\!\!+$. Задержка для второго узла потока физически создается только при вынуждении первого. Однако из-за мемоизации задержка для второго узла будет разделяться между ветвями вычисления всегда, когда разделяется задержка для первого. Следовательно, мы считаем, что неявно вложенная задержка создается тогда, когда создается задержка, в которую она вложена. Более того, когда мы рассуждаем о долге или вообще о структуре объекта, нас не интересует,

создан ли узел физически или нет. Мы рассуждаем так, будто бы все узлы были созданы в своем окончательном виде, т. е., как будто все задержки в объекте уже были вынуждены.

Упражнение 6.2 *Допустим, мы изменим инвариант очереди с формулы* $|f| > |r|$ *на* $2|f| > |r|$.

1. *Докажите, что амортизированные ограничения* $O(1)$ *по-прежнему выполняются.*

2. *Сравните производительность двух реализаций при последовательной вставке ста элементов через* snoc, *а затем их последовательном удалении через* tail .

6.3.3 Наследование долга

Часто мы создаём задержки, чьи тела вынуждают другие, существующие задержки. В таких случаях мы говорим, что новая задержка *зависит* (depends) от старой. В примере с очередью задержка, создаваемая операцией reverse r, зависит от r, а задержка, создаваемая f ++ reverse r, зависит от f. Каждый раз, когда мы вынуждаем задержку, следует быть уверенными, что высвобожден не только долг, относящийся к ней самой, но и долг всех задержек, от которых она зависит. В примере с очередью инвариант долга гарантирует, что мы создаём задержки через ++ и reverse только в случаях, когда ранее существующие задержки уже полностью оплачены. Однако так будет не всегда.

Когда мы создаём задержку, зависящую от существующей задержки с невысвобожденным долгом, мы переносим этот долг на новую задержку и говорим, что новая задержка *наследует* (inherits) долги старой. Мы не имеем права вынуждать новую задержку, пока мы не высвободили как её собственные долги, так и унаследованные от старой задержки. В методе банкира не делается никакого различия между этими двумя разновидностями долга; считается, что весь долг принадлежит новой задержке. Наследование долга будет применяться при анализе структур данных из Глав 9, 10 и 11.

Замечание 6.4 *Наследование долга предполагает, что не существует способа доступа к более старой задержке внутри текущего объекта в обход новой. К примеру, наследование долга нельзя использовать при анализе следующей функции, применяемой к паре потоков:*

```
fun reverseSnd (xs, ys) = (reverse ys, ys)
```

Здесь ys *может быть вынуждена как через первый компонент пары, так и через второй. В таких случаях мы либо удваиваем долг, связанный с* ys, *и новая задержка наследует копию долга, либо сохраняем одну копию каждой единицы долга и отслеживаем зависимости явно.*

6.4 Метод физика

Подобно методу банкира, метод физика тоже можно адаптировать для работы с понятием текущего долга вместо текущих накоплений. В традиционном методе физика описывается функция потенциала Φ, представляющая нижнюю границу текущих накоплений. Чтобы работать с долгом вместо накоплений, мы заменяем Φ на функцию Ψ, которая сопоставляет объектам потенциалы, представляющие верхнюю границу текущего долга (или, по крайней мере, долю общего долга, относящуюся к рассматриваемому объекту). Грубо говоря, после этого амортизированная стоимость операции определяется как её полная стоимость (т. е., разделенная стоимость плюс нераздельная) минус изменение потенциала. Напомним, что простейший способ рассчитать полную стоимость операции — притвориться, что все вычисление работает энергично.

Всякое изменение текущего долга отражается в изменении потенциала. Если операция не выплачивает никакую долю разделенной стоимости, то изменение потенциала равно её разделенной стоимости, и, следовательно, амортизированная стоимость операции равна её нераздельной стоимости. С другой стороны, если операция выплачивает часть своей разделяемой стоимости, или разделяемой стоимости предыдущих операций, тогда изменение потенциала будет меньше, чем её разделяемая стоимость (т. е., текущий долг увеличивается меньше, чем на её разделяемую стоимость), и амортизированная стоимость операции окажется больше, чем нераздельная. Однако амортизированная стоимость операции не может быть меньше, чем её нераздельная стоимость, так что мы не позволяем потенциалу изменяться больше, чем на разделяемую стоимость операции.

Обосновать метод физика можно путём сведения его к методу банкира. Напомним, что в методе банкира амортизированная стоимость операции равна её нераздельной стоимости плюс размер высвобождаемого долга. В методе физика амортизированная стоимость равна полной стоимости минус изменение потенциала или, другими словами, нераздельной стоимости плюс разница между разделяемой стоимостью и изменением потенциала. Если единицу потенциала мы считаем равной единице долга, то разделяемая стоимость равна количеству единиц, на которое мог бы увеличиться текущий долг, а изменение потенциала равно количеству единиц, на которое текущий долг увеличился на самом деле. Разница должна была быть покрыта путем высвобождения части долга. Следовательно, амортизированная стоимость в методе физика также может рассматриваться как нераздельная стоимость плюс количество высвобождаемых единиц долга.

Мы иногда хотим вынудить задержку в объекте, чей потенциал не равен нулю. В таком случае мы добавляем потенциал этого объекта к амортизированной стоимости операции. Как правило, такое случается в операциях-запросах, поскольку там стоимость вынуждения задержки нельзя отразить как изменение потенциала: такая операция не возвращает нового объекта.

Главное различие между методами банкира и физика состоит в том, что при использовании метода банкира мы можем вынудить задержку, как

Рис. 6.2: Ленивые биномиальные кучи.

только её собственный долг выплачен, не ожидая выплаты долга по другим задержкам, в то время как в методе физика разделяемая задержка может быть вынуждена только после того, как весь текущий долг объекта, измеряемый его потенциалом, обращен в ноль. Поскольку потенциал измеряет только накопившийся долг всего объекта и не делает различия между его ячейками, нам приходится делать пессимистическое предположение, что весь текущий долг привязан к той конкретной задержке, которую мы сейчас хотим вынудить. Из-за этого метод физика кажется менее мощным, чем метод банкира. Однако когда он применим, как правило, метод физика значительно упрощает рассуждения.

Поскольку метод физика не может воспользоваться частичным выполнением вложенных задержек, нет никаких причин предпочитать пошаговые функции монолитным. В сущности, если все или большинство задержек монолитны, это может служить подсказкой о применимости метода физика.

6.4.1 Пример: биномиальные кучи

В Главе 5 мы показали, что биномиальные кучи из Раздела 3.2 поддерживают операцию insert за амортизированное время $O(1)$. Однако если кучи используются как устойчивая структура, этот показатель для худшего случая деградирует до $O(\log n)$. С помощью ленивого вычисления мы можем восстановить амортизированное ограничение по времени $O(1)$ вне зависимости от того, используются ли кучи как устойчивая структура.

Основная идея состоит в том, чтобы заменить в представлении кучи список деревьев на задержанный список деревьев.

```
type Heap = Tree list susp
```

При этом мы можем переписать insert в виде

```
fun lazy insert (x, $ts) =$insTree (Node (0, x, []), ts)
```

или, эквивалентным образом, в виде

```
fun insert (x, h) = $insTree (Node (0, x, []), force h)
```

Остальные функции столь же просты в написании; они показаны на Рис. 6.2.

Проанализируем амортизированное время работы insert. Поскольку это монолитная операция, мы можем использовать метод физика. Сначала определяем функцию потенциала как $\Psi(h) = Z(|h|)$, где $Z(n)$ — число нулей в двоичном представлении n (минимальной длины). Затем мы покажем, что амортизированная стоимость вставки элемента в биномиальную кучу размера n равна двум. Допустим, что k младших разрядов в двоичном представлении n равны единице. Тогда полная стоимость операции insert пропорциональна $k + 1$, поскольку включает k вызовов операции link. Рассмотрим теперь изменение потенциала. Младшие k разрядов изменяются с

единиц на нули, а одна следующая цифра изменяется с нуля на единицу, так что изменение потенциала равно $k - 1$. Амортизированная стоимость получается $(k + 1) - (k - 1) = 2$.

Замечание 6.5 *Заметим, что наше доказательство двойственно по отношению к доказательству из Раздела 5.3. Тогда потенциал равнялся количеству единиц в двоичном представлении n, теперь это количество нулей. Такая зеркальность отражает двойственность между понятиями текущих накоплений и текущего долга.*

Упражнение 6.3 *Докажите, что findMin, deleteMin и merge также выполняются за амортизированное время* $O(\log n)$.

Упражнение 6.4 *Допустим, мы уберем ключевое слово lazy из определений функций merge и deleteMin, так что эти функции будут вычислять свои аргументы немедленно. Покажите, что обе они по-прежнему выполняются за время* $O(\log n)$.

Упражнение 6.5 *Задержка списка деревьев имеет неприятное последствие: время работы isEmpty деградирует от* $O(1)$ *в худшем случае до амортизированного* $O(\log n)$. *Восстановите время работы* $O(1)$ *для isEmpty путем явного хранения размера каждой кучи. Вместо того, чтобы явно модифицировать нашу теперешнюю реализацию, напишите функтор SizedHeap, подобный ExplicitMin из Упражнения 3.7. Он должен преобразовывать произвольную реализацию кучи в реализацию, которая явно хранит размер.*

6.4.2 Пример: очереди

В этом разделе мы приспосабливаем под метод физика нашу реализацию очередей. Как и раньше, мы показываем, что все операции занимают амортизированное время $O(1)$.

Поскольку теперь нет никакого смысла предпочитать пошаговые задержки монолитным, вместо потоков мы используем задержанные списки. В сущности, хвостовой список даже задерживать не надо, поэтому его мы представляем как обыкновенный список. Как и раньше, мы явно храним длины списков и гарантируем, что головной список имеет длину не меньше хвостового.

Поскольку головной список задержан, мы не можем получить доступ к его первому элементу, не выполнив всю задержку целиком. Поэтому для ответов на запросы head мы держим рабочую копию некоторого префикса головного списка. Ради эффективности доступа эта рабочая копия хранится в виде обычного списка. Если головной список непуст, эта копия также непуста. Итоговый тип выглядит так:

```
type α Queue = α list × int × α list × int × α list
```

Теперь мы можем записать основные функции над очередями:

Рис. 6.3: Амортизированные кучи с использованием метода физика.

```
fun snoc ((w, lenf, f, lenr, r), x) = check (w, lenf, f, lenr+1, x :: r)
fun head (x :: w, lenf, f, lenr, r) = x
fun tail (x :: w, lenf, f, lenr, r) = check (w, lenf−1, $tl (force f), lenr, r)
```

Вспомогательная функция check обеспечивает два инварианта: что r не может быть длиннее, чем f, и что при непустом f не может быть пуст w.

```
fun checkw ([], lenf, f, lenr, r) = (force f, lenf, f, lenr, r)
  | checkw q = q
fun check (q as (w, lenf, f, lenr, r)) =
      if lenr ≤ lenf then checkw q
      else let val f' = force f
           in checkw (f', lenf+lenr, $(f' @ rev r), 0, []) end
```

Полная реализация очередей приведена на Рис. 6.3.

Для анализа очередей методом физика мы выбираем функцию потенциала Ψ так, чтобы при вынуждении задержанного списка потенциал всегда был равен нулю. Такое может произойти в двух ситуациях: когда w оказывается пустым, и когда r оказывается длиннее f. Поэтому мы выбираем такое Ψ:

$$\Psi(\mathrm{q}) = \min(2|\mathrm{w}|, |\mathrm{f}| - |\mathrm{r}|)$$

Теорема 6.2 *Амортизированная стоимость операций snoc и tail равна, соответственно, двум и четырем.*

Доказательство. *snoc, не вызывающий проворота, просто добавляет новый элемент к хвостовому списку. При этом $|r|$ увеличивается на единицу, а $|f|-|r|$ уменьшается на единицу. Полная стоимость snoc равна одному, а уменьшение потенциала не больше одного, так что амортизированная стоимость равна максимум $1-(-1)=2$. Вызов tail, не приводящий к повороту очереди, убирает элемент из рабочего списка и лениво убирает тот же самый элемент из головного списка. При этом $|w|$ уменьшается на единицу, и на столько же уменьшается $|f|-|r|$, так что потенциал уменьшается максимум на два. Полная стоимость tail равна двум — один как нераздельная стоимость (включая отбрасывание первого элемента w) и один как разделяемая стоимость ленивого отбрасывания головы f. Амортизированная стоимость получается $2-(-2)=4$.*

Наконец, рассмотрим вызов snoc или tail, приводящий к повороту очереди. В начале операции $|f|=|r|$, так что $\Psi=0$. Перед самым поворотом $|f|=m$, а $|r|=m+1$. Разделяемая стоимость проворота равна $2m+1$, а потенциал получающейся очереди $2m$. Таким образом, амортизированная стоимость snoc равна $1+(2m+1)-2m=2$. Амортизированная стоимость tail равна $2+(2m+1)-2m=3$. (Разница получается потому, что в случае tail нам нужно ещё учесть стоимость удаления первого элемента f.)

Рис. 6.4: Сигнатура сортируемых коллекций.

Упражнение 6.6 *Покажите, почему каждая из следующих «оптимизаций» уничтожает амортизированное ограничение времени $O(1)$. Эти примеры показывают типичные ошибки при проектировании устойчивых амортизированных структур данных.*

1. *Заметим, что check при повороте вынуждает f, а затем записывает результат в w. Разве не было бы более ленивым поведением, а следовательно, более выгодным, не вынуждать f, пока w не окажется пустым?*

2. *Заметим, что во время операции tail мы заменяем f на $tl (force f). Создание и вынуждение задержек приводит к заметным расходам, которые, хотя и сохраняют стоимость константной, могут сделать константу слишком большой. Разве не было бы ленивее, а следовательно, лучше, не изменять f, а просто уменьшать lenf, показывая таким образом, что элемент удален?*

6.4.3 Сортировка слиянием снизу вверх с совместным использованием

Большинство примеров в оставшихся главах использует метод банкира, а не физика. Поэтому здесь мы приводим ещё один пример на метод физика.

Допустим, что вы хотите отсортировать несколько похожих списков, например, x и x :: xs, или xs @ zs и ys @ zs. Из соображений эффективности вам хотелось бы использовать то, что хвосты списков совпадают, чтобы не повторять работу по сортировке хвостов. Назовем абстрактный тип данных для решения этой задачи *сортируемая коллекция* (sortable collection). Сигнатура сортируемых коллекций приведена на Рис. 6.4.

Теперь если мы из списка xs сделаем сортируемую коллекцию xs', добавив к пустой коллекции все элементы xs по очереди, то сможем отсортировать xs и x :: xs, вызвав, соответственно, sort xs' и sort (add (x, xs')).

Сортируемые коллекции можно реализовать как сбалансированные двоичные деревья поиска. Тогда add и sort будут иметь, соответственно, ограничения по времени в худшем случае $O(\log n)$ и $O(n)$. Здесь мы достигаем тех же самых ограничений, но только амортизированных, используя *сортировку слиянием снизу вверх* (bottom-up mergesort).

Сортировка слиянием снизу вверх сначала разбивает список на n упорядоченных сегментов (на первом этапе каждый сегмент содержит по одному элементу). Затем она попарно сливает сегменты одинакового размера, пока для каждого размера не останется только один. Наконец, сливаются сегменты неодинакового размера.

Возьмем состояние данных непосредственно перед последним шагом. Размеры сегментов в этот момент равны степеням двойки, соответствую-

щим единичным битам в n. Именно это представление мы будем использовать для наших сортируемых коллекций. Похожие коллекции будут совместно использовать работу сортировки снизу вверх с точностью до последней фазы, когда сливаются сегменты разного размера. Полностью данные будут представлены в виде задержанного списка сегментов, каждый из которых является списком элементов, плюс целое число — размер коллекции.

```
type Sortable = int × Elem.T list list susp
```

Отдельные сегменты хранятся в порядке возрастания размера, а элементы каждого сегмента хранятся в порядке возрастания согласно функциям сравнения структуры Elem.

Основная операция над сегментами — слияние двух упорядоченных списков, mrg.

```
fun mrg ([], ys) = ys
  | mrg (xs, []) = xs
  | mrg (xs as x :: xs', ys as y :: ys') =
      if Elem.leq (x, y) then x :: mrg (xs', ys) else y :: mrg (xs, ys')
```

При добавлении нового элемента мы создаём одноэлементный сегмент. Если наименьший из существующих сегментов тоже одноэлементен, мы эти два сегмента сливаем, и продолжаем слияние до тех пор, пока новый сегмент не окажется меньше наименьшего существующего. Это слияние управляется битами в поле размера. Если младший бит size равен нулю, то мы просто прицепляем новый сегмент к списку сегментов. Если бит равен единице, мы сливаем два сегмента и повторяем операцию. Разумеется, все это происходит в ленивом режиме.

```
fun add (x, (size, segs)) =
    let fun addSeg (seg, segs, size) =
              if size mod 2 = 0 then seg :: segs
              else addSeg (mrg (seg, hd segs), tl segs, size div 2)
    in (size+1, $addSeg([x], force segs, size)) end
```

Наконец, чтобы отсортировать коллекцию, мы сливаем сегменты от меньшего к большему.

```
fun sort (size, segs) =
    let fun mrgAll ([], []) = xs
          | mrgAll (xs, seg :: segs) = mrgAll (mrg (xs, seg), segs)
    in mrgAll ([], force segs) end
```

Замечание 6.6 *Можно рассматривать mrgAll как вычисление*

$$[] \bowtie s_1 \bowtie \ldots \bowtie s_m$$

где s_i — i-й сегмент, а $\bowtie$ — инфиксное лево-ассоциативное обозначение для операции mrg. Это частный случай весьма распространенного программного шаблона, который можно записать как

$$c \oplus x_1 \oplus \ldots \oplus x_m$$

Рис. 6.5: Сортируемые коллекции на основе сортировки слиянием снизу вверх.

для любого c и лево-ассоциативной $\oplus$. В качестве других примеров этого шаблона можно привести суммирование списка целых ($c = 0$ и $\oplus = +$) или нахождение максимума в списке натуральных чисел ($c = 0$ и $\oplus = \max$). Одна из самых сильных черт функциональных языков — способность определять шаблоны подобного рода в виде функций высших порядков (higher-order functions) (т. е., функций, которые принимают другие функции в качестве аргументов или возвращают функции как результат). Например, вышеприведенный шаблон можно записать как

```
fun foldl (f, c, []) = c
  | foldl (f, c, x :: xs) = foldl (f, f (c, x), xs)
```

Тогда sort выглядит как

```
fun sort (size, segs) = foldl (mrg, [], force segs)
```

Полный программный код к нашей реализации сортируемых коллекций приведен на Рис. 6.5.

Покажем, используя метод физика, что операция add занимает амортизированное время $O(\log n)$, а операция sort амортизированное время $O(n)$. Вначале зададим функцию потенциала Ψ, которая полностью определяется размером коллекции.

$$\Psi(n) = 2n - 2\sum_{i=0}^{\infty} b_i(n \mod 2^i + 1)$$

где b_i — i-й бит n. Заметим, что $\Psi(n)$ ограничен сверху величиной $2n$, и что $\Psi(n) = 0$ в точности тогда, когда $n = 2^k - 1$ для некоторого k.

Замечание 6.7 *Наша функция потенциала может показаться немного сложноватой. Она возникает из желания считать, что каждый сегмент имеет потенциал, пропорциональный его собственному размеру минус размер всех более мелких сегментов. Интуиция здесь заключается в том, что у всякого сегмента потенциал сначала велик, но он уменьшается по мере добавления новых элементов в коллекцию, и обращается в ноль непосредственно перед тем, как наш сегмент сливается с другим сегментом. Однако для того, чтобы проводить вычисления с функцией, необязательно знать, какими соображениями мотивировано ее определение.*

Сначала вычислим полную стоимость операции add. Её нераздельная стоимость равна единице, а разделяемая равна стоимости слияний, проводимых внутри addSeg. Допустим, что младшие k бит числа n равны единице

(т. е., $b_i = 1$ для $i < k$ и $b_k = 0$). В этом случае addSeg проводит k слияний. Первое из них сливает два списка длиной 1, второе два списка длиной 2, и так далее. Поскольку слияние двух списков размера m занимает $2m$ шагов, addSeg занимает

$$(1+1) + (2+2) + \cdots + (2^{k-1} + 2^{k-1}) = 2(\sum_{i=0}^{k-1} 2^i) = 2(2^k - 1)$$

шагов. Следовательно, полная стоимость add равна $2(2^k - 1) + 1 = 2^{k+1} - 1$.

Вычислим теперь изменение потенциала. Пусть $n' = n+1$, а b'_i — i-й бит числа n'. Тогда

$$\begin{array}{l} \Psi(n') - \Psi(n) \\ = 2n' - 2\sum_{i=0}^{\infty} b'_i(n \mod 2^i + 1) - (2n - 2\sum_{i=0}^{\infty}(n \mod 2^i + 1) \\ = 2 + 2\sum_{i=0}^{\infty}(b_i(n \mod 2^i + 1) - b'_i(n' \mod 2^i + 1)) \\ = 2 + 2\sum_{i=0}^{\infty} \delta(i) \end{array}$$

где $\delta(i) = b_i(n \mod 2^i+1) - b'_i(n \mod 2^i+1)$. Рассмотрим три случая: $i < k$, $i = k$ и $i > k$.

- $(i < k)$: поскольку $b_i = 1$, а $b'_i = 0$, $\delta(i) = n \mod 2^i + 1$. Но $n \mod 2^i = 2^i - 1$, так что $\delta(i) - 2^i$.
- $(i = k)$: поскольку $b_k = 0$, а $b'_k = 1$, $\delta(k) = -(n' \mod 2^k + 1)$. Но $n' \mod 2^k = 0$, так что $\delta(k) = -1 = -b'_k$.
- $(i > k)$: поскольку $b'_i = b_i$, $\delta(i) = b'_i(n \mod 2^i - n' \mod 2^i)$. Но $n' \mod 2^i = (n+1) \mod 2^i = n \mod 2^i + 1$, так что $\delta(i) = b'_i(-1) = -b'_i$.

Следовательно,

$$\begin{array}{rcl} \Psi(n') - \Psi(n) & = & 2 + 2\sum_{i=0}^{\infty} \delta(i) \\ & = & 2 + 2\sum_{i=0}^{k-1} 2^i + 2\sum_{i=k}^{\infty}(-b'_i) \\ & = & 2 + 2(2^k - 1) - 2\sum_{i=k}^{\infty} b'_i \\ & = & 2^{k+1} - 2B' \end{array}$$

где B' — число единичных битов в n'. Тогда амортизированная стоимость операции add равна

$$(2^{k+1} - 1) - (2^{k+1} - 2B') = 2B' - 1$$

Поскольку B' пропорционален $O(\log n)$, такую же оценку имеет и амортизированная стоимость add.

Наконец, вычисляем амортизированную стоимость операции sort. Первое её действие — вынудить задержанный список сегментов. Поскольку потенциал не обязательно равен нулю, это добавляет $\Psi(n)$ к амортизированной стоимости операции. Затем sort сливает сегменты, двигаясь от меньших

к большим. В худшем случае $n = 2^k - 1$, так что есть по сегменту каждого размера от 1 до 2^{k-1}. Слияние сегментов занимает в общей сложности

$$(1+2)+(1+2+4)+(1+2+4+8)+\cdots+(1+2+\cdots+2^{k-1})$$
$$=\sum_{i=1}^{k-1}\sum_{j=0}^{i}2^j=\sum_{i=1}^{k-1}(2^{i+1}-1)=(2^{k+1}-4)-(k-1)=2n-k-1$$

шагов. Следовательно, амортизированная стоимость равна $O(n) + \Psi(n) = O(n)$.

Упражнение 6.7 *Замените в нашей реализации задержанный список списков на список потоков.*

1. *докажите ограничения стоимости для add и sort с помощью метода банкира.*
2. *Напишите функцию для извлечения наименьших k элементов из сортируемой коллекции. Докажите, что ваша функция работает за амортизированное время не хуже $O(k \log n)$.*

6.5 Ленивые парные кучи

В завершение этой главы мы модифицируем парные кучи из Раздела 5.5 для работы в условиях устойчивости. К сожалению, анализ получающейся структуры данных оказывается столь же сложен, как и для исходной. Однако мы предполагаем, что асимптотически наша новая реализация столь же эффективна в условиях устойчивости, как исходная реализация эффективна в эфемерных условиях.

Напомним, что в предыдущей реализации парных куч дети каждого узла представлялись как список структур Heap. При уничтожении минимального элемента корень отбрасывался, а затем дети сливались попарно при помощи функции

```
fun mergePairs [] = E
  | mergePairs [h] = h
  | mergePairs (h1 :: h2 :: hs) =merge (merge (h1, h2), mergePairs hs)
```

Если уничтожить корневой элемент одной и той же кучи дважды, mergePairs будет также вызвана дважды. При этом работа будет повторяться, а всякая надежда на эффективное амортизированное использование будет потеряна. Чтобы справиться с задачей устойчивости, нужно предотвратить повторение этой работы. Очередной раз мы используем для этого ленивое вычисление. Вместо списка куч Heap list, мы представляем детей узла как задержанную кучу Heap susp. Значение этой задержки равно $mergePairs cs. Поскольку mergePairs работает с парами элементов списка детей, мы будем расширять нашу задержку двумя элементами сразу. Следовательно, нам понадобится дополнительное поле типа Heap в каждом узле для хранения непарных потомков. Если непарных потомков нет (т. е., число детей чётно), это дополнительное поле будет пустым. Поскольку это поле используется

Рис. 6.6: Устойчивые парные кучи с использованием ленивого вычисления.

только тогда, когда число детей нечетно, мы будем называть его *нечётным полем* (odd field). Таким образом, наш новый тип данных имеет вид

```
datatype Heap = E | T of Elem.T × Heap × Heap susp
```

Операции insert и findMin почти не требуют изменений.

```
fun insert (x, a) = merge (T (x, E, $E), a)
fun findMin (T (x, a, m)) = x
```

Раньше у нас операция merge была простой, а операция deleteMin — сложной. Теперь ситуация обратная — вся сложность функции mergePairs оказалась перенесена в merge, которая устанавливает все необходимые задержки. Функция deleteMin просто вынуждает задержку кучи и сливает её с нечётным полем.

```
fun deleteMin (T (x, a, $b)) =merge (a, b)
```

Функцию merge мы определяем в два шага. Первый шаг проверяет, что аргументы непусты, и если это так, выясняет, у которого из двух аргументов меньше корневой элемент.

```
fun merge (a, E) = a
  | merge (E, a) = a
  | merge (a as T (x, _, _), b as T (y, _, _)) =
      if Elem.leq (x, y) then link (a, b) else link (b, a)
```

Второй шаг, воплощенный во вспомогательной функции link, добавляет к куче новый элемент. Если нечётное поле пусто, новый ребёнок добавляется туда.

```
fun link (T (x, E, m), a) = T (x, a, m)
```

В противном случае новый ребёнок спаривается с ребёнком из нечётного поля, и оба они добавляются к задержке. Другими словами, мы превращаем задержку m =$mergePairs cs в $mergePairs (a :: b :: cs). Заметим, что

```
$mergePairs (a :: b :: cs)
  ≡ $merge (merge (a, b), mergePairs cs)
  ≡ $merge (merge (a, b), force ($mergePairs cs))
  ≡ $merge (merge (a, b), force m)
```

так что, вторую ветвь функции link можно записать как

```
fun link (T (x, b, m), a) = T (x, E, $merge (merge (a, b), force m))
```

Полный код этой реализации приведен на Рис. 6.6.

Указание разработчикам 6.1 *Несмотря на то, что эта реализация парных куч хорошо работает в условиях устойчивости, на практике она оказывается довольно медленной из-за высокой стоимости ленивого вычисления. Однако в условиях активного использования устойчивости эта реализация ведёт себя прекрасно — мы получаем максимальную пользу от*

мемоизации. Кроме того, эта реализация конкурентоспособна в ленивых языках, где дополнительную стоимость ленивого вычисления платят все структуры данных, независимо от того, есть им от этого выгода или нет.

6.6 Примечания

Долг Некоторые разновидности анализа с использованием традиционного метода банкира, например, анализ сжатия путей Тарьяном [Tar83], работают и с понятием кредита, и с понятием дебета. Когда операции требуется кредит больше, чем имеется в данный момент, она создает пару кредит-дебет и немедленно тратит кредит. Дебет остается как обязательство, подлежащее исполнению. Позже избыток кредита можно использовать для выплаты долга[2]. Дебет, остающийся в конце вычисления, добавляется к общей реальной стоимости. Несмотря на некоторое сходство между двумя понятиями долга, есть и явные различия. Например, долг, как он введен в этой главе, оставшийся в конце вычисления, тихо уничтожается.

Интересно заметить, что дебет введен Тарьяном при анализе сжатия путей; ведь сжатие путей, в сущности, является применением мемоизации к функции поиска.

Амортизация и устойчивость До публикации этой работы считалось, что амортизация несовместима с устойчивостью. Несколько исследователей [DST94, Ram92] замечали, что амортизированные структуры невозможно сделать эффективно устойчивыми с помощью существующих методик добавления устойчивости к эфемерным структурам данных, подобных описанным в [DSST89, Die89], по причинам вроде указанных нами в Разделе 5.6. Интересно, что эти методики порождают структуры с амортизированными показателями производительности, при том, что показатели нижележащей структуры должны быть жёсткими. (У этих методик есть и другие ограничения. Прежде всего, они не работают для структур данных, имеющих функции, применимые более, чем к одной версии. Примерами таких запретных операций являются конкатенация списков и объединение множеств.)

Идея, что ленивое вычисление может помирить амортизацию и устойчивость, впервые, в рудиментарной форме, появилась в [Oka95c]. Теория и практика этого подхода были развиты в [Oka95a, Oka96b].

Амортизация и функциональные структуры данных Схунмакерс [Sch93] в своей диссертации исследует амортизированные структуры данных в энергичном функциональном языке, в основном исследуя формальный вывод амортизированных ограничений с помощью метода физика. Он избегает проблем устойчивости, настаивая, чтобы все структуры данных использовались только в однопоточном режиме.

[2]Здесь напрашивается явная аналогия со спонтанным порождением и аннигиляцией пар частица-античастица в физике. В сущности, для этого дебета более подходящим названием было бы «антикредит».

Очереди и биномиальные кучи Очереди из Раздела 6.3.2 и ленивые биномиальные кучи из Раздела 6.4.1 впервые появились в [Oka96b]. Анализ ленивых биномиальных куч применим также к реализации Кинга [Kin94].

Анализ времени выполнения ленивых программ Существует несколько теоретических формализмов для анализа временного поведения ленивых программ [BH89, San90, San95, Wad88]. Однако эти формализмы недостаточно пока разработаны, чтобы быть применимыми на практике. Одна из сложностей состоит в том, что они, в некотором смысле, чрезмерно общие. В каждой из этих систем стоимость программы вычисляется по отношению к некоторому контексту, который представляет собой описание, как результат программы будет использоваться. Однако этот подход часто неприменим в методологии разработки программ, где структуры данных проектируются как абстрактные типы, чье поведение, включая сложность операций, описывается в изоляции. В противоположность этим подходам наш способ анализа дает независимые от контекста результаты (т. е., они верны безотносительно того, как структуры данных будут использоваться).

Глава 7

Избавление от амортизации

Чаще всего нас не интересует, являются ли ограничения, соблюдаемые структурой, жёсткими или амортизированными; основные критерии для выбора одной структуры данных вместо другой — общая эффективность и простота реализации (возможно, ещё наличие исходного кода). Однако в некоторых прикладных областях оказывается важно ограничить время выполнения отдельных операций, а не их последовательностей. В этих случаях структура с жёсткими ограничениями часто предпочтительнее структуры с амортизированными характеристиками, даже если амортизированная структура в целом проще и быстрее. Раман [Ram92] перечисляет несколько таких прикладных областей, в том числе

- **Системы реального времени.** В системах реального времени предсказуемость важнее голой скорости [Sta88]. Если из-за дорогой операции система пропустит жёсткий предельный срок, неважно будет, сколько дешёвых операций завершилось раньше назначенного времени.
- **Параллельные системы.** Если один процессор в синхронной системе выполняет дорогую операцию в то время, как остальные выполняют дешёвые, то остальным процессорам придётся ждать, пока закончит работу самый медленный.
- **Диалоговые системы.** Диалоговые системы подобны системам реального времени — для пользователей предсказуемость часто важнее, чем чистая скорость [But83]. Например, пользователи могут предпочесть 100 ответов с задержкой 1 секунда варианту с 99 ответами при задержке 0.25 секунд и одним ответом с задержкой 25 секунд, даже при том, что второй из этих сценариев вдвое быстрее.

Замечание 7.1 *Раман упоминает ещё одну область приложений — устойчивые структуры данных. Как указано в предыдущей главе, долгое время считалось, что амортизация несовместима с устойчивостью. Однако, разумеется, теперь мы знаем, что это не так.*

Означает ли это, что для программистов в этих областях амортизированные структуры данных не представляют интереса? Вовсе нет. Поскольку часто амортизированные структуры данных устроены проще, чем структуры с жёсткими ограничениями, иногда оказывается легче сначала разработать амортизированную структуру, а затем преобразовать ее в жёсткую, чем разработать структуру с жёсткими ограничениями с нуля.

В этой главе мы описываем *расписания* (scheduling) — метод для преобразования амортизированных структур данных в структуры с жёсткими ограничениями путем систематического вынуждения задержек, так что ни одна из них не выполняется слишком долго. При использовании этого метода к каждому объекту добавляется дополнительная компонента — *расписание* (schedule), управляющая порядком вынуждения задержек внутри этого объекта.

7.1 Расписания

Амортизированные структуры данных и структуры данных с жёсткими ограничениями различаются в основном временем, когда происходит вычисление, входящее в стоимость какой-либо операции. В структуре с жёсткими ограничениями для наихудшего случая все вычисления, составляющие стоимость операции, происходят во время самой этой операции. В амортизированной структуре данных некоторые вычисления, входящие в стоимость операции, могут на самом деле производиться во время более поздних операций. Отсюда мы видим, что почти все структуры данных, считающиеся жёсткими, будучи реализованы на чисто ленивом языке, превращаются в амортизированные, поскольку многие вычисления оказываются задержаны без особой нужды. Следовательно, для описания структур с жёсткими ограничениями нам нужен энергичный язык. Если же нам нужно описывать как амортизированные, так и жёсткие структуры данных, требуется язык, поддерживающий как ленивый порядок вычисления, так и энергичный. Имея такой язык, мы можем рассмотреть любопытный гибридный подход: структуры с жёсткими характеристиками, использующие внутри своей реализации ленивое вычисление. Мы получаем такие структуры данных, беря за основу ленивые амортизированные структуры и модифицируя их так, чтобы каждая операция укладывалась в отведенное ей время.

В ленивой амортизированной структуре данных каждая конкретная операция может занять дольше, чем её заявленное ограничение. Однако такое происходит только в том случае, если эта операция вынуждает задержку, которая уже была оплачена, но требует большого времени для выполнения. Чтобы уложиться в жёсткие ограничения, мы должны гарантировать, что всякая задержка выполняется не дольше отведенного ей времени.

Определим *собственную стоимость* (intrinsic cost) задержки как время, которое уходит на вынуждение задержки в предположении, что все другие задержки, от которых она зависит, уже были вынуждены и мемоизированы, и, следовательно, занимают по $O(1)$ времени каждая. (Это опреде-

ление похоже на определение нераздельной стоимости операции.) Первым шагом в преобразовании амортизированной структуры в жёсткую будет уменьшение собственной стоимости каждой задержки до размеров меньше желаемого ограничения. Обычно при этом требуется переписать дорогие монолитные функции и сделать их пошаговыми — либо путем небольших изменений в алгоритмах, либо путем перехода от представления, поддерживающего только монолитные функции, скажем, задержанных списков, к такому, которое также поддерживает пошаговые функции, скажем, к потокам.

Даже если каждая задержка имеет маленькую собственную стоимость, некоторые задержки по-прежнему могут занимать долгое время. Это происходит, когда одна задержка зависит от другой, эта вторая от третьей, и так далее. Если ни одна из этих задержек заранее не была выполнена, то вынуждение первой задержки приводит к каскаду других вынуждений. Рассмотрим, например, следующее вычисление:

$$(\cdots((s_1 +\!\!+\, s_2) +\!\!+\, s_3) +\!\!+\, \cdots) +\!\!+\, s_k$$

Вынуждение задержки, возвращаемой самым внешним $+\!\!+$, вызывает цепную реакцию, в ходе которой каждая из $+\!\!+$ производит по шагу. Несмотря на то, что собственная стоимость внешней задержки составляет $O(1)$, общее время на её вынуждение равно $O(k)$ (или даже больше, если первая ячейка s_1 является дорогой по каким-либо ещё причинам).

Замечание 7.2 *Случалось ли вам ставить костяшки домино в ряд, чтобы каждая из них сбивала следующую? Несмотря на то, что собственная стоимость опрокидывания каждой костяшки равна $O(1)$, реальная стоимость опрокидывания первой костяшки может быть намного, намного больше.*

Второй шаг при преобразовании амортизированной структуры данных в жёсткую — избежать каскадирования вынуждений, устроив так, чтобы всякий раз при вынуждении задержки все другие задержки, от которых она зависит, были уже вынуждены и мемоизированы. Тогда ни одна задержка не занимает при выполнении больше, чем её собственная стоимость. Мы этого добиваемся, строя систематическое *расписание* (schedule) выполнений каждой задержки, чтобы все они были готовы к тому времени, как нам понадобятся. Трюк здесь состоит в том, чтобы рассматривать выплату долга буквально, и вынуждать каждую задержку в момент, когда она оплачивается.

Замечание 7.3 *Работа с расписанием подобна опрокидыванию ряда костяшек начиная с хвоста, так чтобы всякий раз, когда одна костяшка падает на другую, эта другая была уже заранее сбита. Тогда реальная стоимость опрокидывания каждой костяшки будет мала.*

Мы добавляем к каждому объекту новую компоненту, называемую *расписание* (schedule). Она содержит, по крайней мере, концептуально, ссылки

на все невычисленные задержки внутри объекта. Некоторые из этих задержек, возможно, уже были вычислены в рамках другого логического будущего, но вынуждение их по второму разу безвредно, поскольку алгоритм от этого становится только ещё быстрее, чем ожидалось, а не медленнее. Всякая операция, в дополнение к любым другим действиям, которые она производит с объектом, вынуждает несколько первых задержек в расписании. Точное количество вынуждаемых задержек управляется амортизационным анализом; как правило, каждая задержка для выполнения требует время $O(1)$, так что мы вынуждаем задержки пропорционально амортизированной стоимости операции. В зависимости от конкретной структуры данных, ведение расписания может быть нетривиальной задачей. Чтобы наша методика была применима, добавление новой задержки к расписанию, а также нахождение следующей задержки, подлежащей вынуждению, не должно требовать больше времени, чем желаемые жёсткие ограничения.

7.2 Очереди реального времени

Как пример нашей методики мы преобразуем амортизированные очереди по методу банкира из Раздела 6.3.2 в очереди с жёсткими ограничениями. Очереди вроде этих, поддерживающие все операции за время $O(1)$ в худшем случае, называются *очередями реального времени* (real time queues) [HM81].

В исходной структуре данных очереди проворачиваются с помощью $+\!\!+$ и reverse. Поскольку операция reverse монолитна, первая наша задача состоит в том, чтобы научиться делать проворот пошагово. Этого можно добиться, если проводить по шагу reverse на каждый шаг $+\!\!+$. Определим функцию rotate, такую, что

$$\text{rotate}\ (\text{xs},\ \text{ys},\ \text{a}) \equiv \text{xs} +\!\!+ \text{reverse ys} +\!\!+ \text{a}$$

Тогда

$$\text{rotate}\ (\text{f},\ \text{r},\ \$\text{Nil}) \equiv \text{f} +\!\!+ \text{reverse r}$$

Дополнительный аргумент a называется *аккумулирующим параметром* (accumulating parameter) и служит для хранения частичных результатов обращения r. Изначально он пуст.

Проворот происходит, когда $|\text{r}| = |\text{f}| + 1$, так что в начале $|\text{xs}| = |\text{ys}| + 1$. Это соотношение сохраняется на протяжении всего проворота, так что когда xs становится пустым, ys содержит единственный элемент. Следовательно, основание рекурсии выглядит как

$$\begin{aligned}&\text{rotate}\ (\$\text{Nil}, \$\text{Cons}\ (\text{y}, \$\text{Nil}), \text{a})\\ &\quad \equiv (\$\text{Nil}) +\!\!+ \text{reverse}\ (\$\text{Cons}\ (\text{y}, \$\text{Nil})) +\!\!+ \text{a}\\ &\quad \equiv \$\text{Cons}\ (\text{y}, \text{a})\end{aligned}$$

Шаг рекурсии таков:

```
rotate ($Cons (x, xs), $Cons (y, ys), a)
    ≡ ($Cons (x, xs)) ++ reverse ($Cons (y, ys)) ++ a
    ≡ $Cons (x, xs ++ reverse ($Cons (y, ys) ++ a)
    ≡ $Cons (x, xs ++ reverse ys ++ $Cons (y, a))
    ≡ $Cons (x, rotate (xs, ys, $Cons (y, a)))
```

Объединяя два уравнения, получаем

```
fun rotate ($Nil, $Cons (y, _), a) =$Cons (y, a)
  | rotate ($Cons (x, xs), $Cons (y, ys), a) =
      $Cons (x, rotate (xs, ys, $Cons (y, a)))
```

Заметим, что собственная стоимость каждой задержки, создаваемой функцией rotate, равна $O(1)$.

Упражнение 7.1 *Покажите, что замена f ++ reverse r на rotate (f, r, $Nil) в очередях по методу банкира из Раздела 6.3.2 уменьшает худшее возможное время операций snoc, head и tail с $O(n)$ до $O(\log n)$. (Подсказка: докажите, что самая длинная цепочка зависимостей между задержками имеет длину $O(\log n)$.) Если это упростит ваш анализ, можете задержать сопоставление с образцом в функции rotate, написав* ***fun*** *lazy вместо* ***fun****.*

Теперь мы добавим к нашему типу данных расписание. Исходный тип имел вид

```
type α Queue =int × α Stream × int × α Stream
```

Мы его расширяем новым полем s типа α Stream, которое представляет расписание вынуждения узлов f. Можно думать об s двумя способами: либо рассматривать его как суффикс f, либо как указатель на первую невычисленную задержку внутри f. Чтобы вычислить следующую задержку в расписании, мы просто вынуждаем s.

Помимо добавления s, мы производим в нашем типе данных ещё два изменения. Во-первых, чтобы подчеркнуть, что к элементам r расписание не относится, мы превращаем r из потока в список. Это влечет небольшие изменения в rotate. Во-вторых, мы больше не храним явно длины списков. Как мы скоро увидим, они нам теперь не нужны, чтобы определить, когда r становится длиннее f — мы можем получить эту информацию из расписания. Таким образом, новый тип данных имеет вид

```
type α Queue =α Stream × α list × α Stream
```

Замечание 7.4 *Выигрыш в памяти при замене четырехчленных кортежей трехчленными может оказаться достаточным, чтобы оправдать переход с одного представления на другое, даже если жёсткие ограничения по времени нас не волнуют.*

С новым представлением основные операции над очередями выглядят весьма просто:

Рис. 7.1: Очереди реального времени на основе расписания.

```
fun snoc ((f, r, s), x) = exec (f, x :: r, s)
fun head ($Cons (x, f), r, s) =x
fun tail ($Cons (x, f), r, s) =exec (f, r, s)
```

Вспомогательная функция exec выполняет следующую задержку и поддерживает инвариант $|s| = |f| - |r|$ (что, между прочим, гарантирует $|f| \geq |r|$, поскольку $|s|$ не может быть отрицательной). Функция snoc увеличивает $|r|$ на единицу, а tail уменьшает $|f|$ на единицу, так что к моменту вызова exec имеем $|s| = |f| - |r| + 1$. Если s непуст, для восстановления инварианта мы просто берем хвост s. Если же s пуст, то r на единицу длиннее, чем f, так что мы проворачиваем очередь. В любом из этих случаев сопоставление s с образцом, когда мы выясняем, пуст ли он, само по себе вынуждает и мемоизирует следующую задержку в расписании.

```
fun exec (f, r, $Cons (x, s)) =(f, r, s)
  | exec (f, r, $Nil) =let val f' = rotate (f, r, $Nil) in (f', [], f') end
```

Полный код этой реализации очередей приведен на Рис. 7.1.

Путем просмотра кода можно убедиться, что каждая операция над очередью занимает всего лишь $O(1)$ помимо вынуждения задержек, и что ни одна операция не вынуждает более трех задержек. Следовательно, чтобы показать, что все операции в худшем случае занимают время $O(1)$, остается доказать, что ни одна задержка при вычислении не занимает время больше $O(1)$.

Операции над очередями создают только три различных вида задержек:

- $Nil создается функциями empty и exec (при первом вызове rotate). Эта задержка тривиальна и всегда выполняется за время $O(1)$, независимо от того, была ли она вынуждена и мемоизирована раньше.

- $Cons (y, a) создается в обеих строках rotate. Эта задержка также тривиальна.

- $Cons (x, rotate (xs, ys, $Cons (y, a))) создается во второй строке rotate. Эта задержка выделяет ячейку Cons, создает новую задержку, и рекурсивно вызывает rotate, которая производит сопоставление образца с первой ячейкой xs и немедленно создает ещё одну задержку. Из всех этих действий только вынуждение, связанное с сопоставлением с образцом, в принципе могло бы занимать больше, чем время $O(1)$. Заметим, однако, что xs — суффикс головного потока, который существовал перед предыдущим проворотом очереди. Наш режим работы с расписанием s гарантирует, что *все* ячейки этого потока были вынуждены и мемоизированы, прежде чем запустился проворот, так что новое вынуждение этой ячейки занимает только $O(1)$ времени.

Поскольку все задержки выполняются за время $O(1)$, наихудшее время выполнения любой из операций над очередью также $O(1)$.

Указание разработчикам 7.1 *Эти очереди намного проще всех других реализаций реального времени. Кроме того, это одна из самых быстрых реализаций — с жёсткими ограничениями или амортизированными, — для приложений, где активно используется устойчивость.*

Упражнение 7.2 *Вычислите размер очереди на основе размеров s и r. Насколько быстрее будет работать такая функция по сравнению с вычислением на основе размеров f и r?*

7.3 Биномиальные кучи

Мы возвращаемся к биномиальным кучам из Раздела 6.4.1, и с помощью расписания обеспечиваем вставку за время $O(1)$ в худшем случае. Напомним, что в предыдущей реализации представление кучи выглядело как Tree list susp, и поэтому функция insert по необходимости была монолитной. Первая наша цель — сделать её пошаговой.

Сначала заменим задержанные списки в типе данных кучи на потоки. Операция insert зовет вспомогательную функцию insTree, которую теперь можно записать как

```
fun lazy insTree (t, $Nil) =$Cons (t, $Nil)
       | insTree (t, ts as $Cons (t', ts')) =
           if rank t < rank t' then $Cons (t, ts)
           else insTree (link (t, t'), ts')
```

Эта функция по-прежнему монолитна, поскольку она не может вернуть первое дерево, пока не выполнены все связывания. Чтобы сделать функцию пошаговой, нужно как-то заставить insTree возвращать после каждой итерации частичный результат. Можно этого добиться, если сделать связь между биномиальными кучами и двоичными числами более явной. Деревья в куче соответствуют единицам в двоичном представлении размера кучи. Мы расширяем это соответствие, вводя явное представление для нулей.

```
datatype Tree = Node of Elem.T × Tree list
datatype Digit = Zero | One of Tree
type Heap = Digit stream
```

Заметим, что мы исключили поле ранга из конструктора Node, поскольку ранг каждого дерева полностью определяется его позицией: дерево, хранящееся в i-й цифре, имеет ранг i, а дети ячейки ранга r имеют ранги $r-1,\ldots,0$. Кроме того, мы требуем, чтобы всякий непустой поток заканчивался единицей-One.

Теперь можно написать insTree:

```
fun lazy insTree (t, $Nil) =$Cons (One t, $Nil)
       | insTree (t, $Cons(Zero, ds)) =$Cons (One t, ds)
```

```
  | insTree (t, $Cons (One t', ds)) =
      $Cons (Zero, insTree (link (t, t'), ds))
```

Эта функция пошаговая, поскольку каждый ее промежуточный шаг возвращает ячейку Cons, содержащую ноль-Zero и задержку для остального вычисления. Последний шаг возвращает единицу.

На следующем шаге мы добавляем к нашему типу данных расписание. Расписание выглядит как список заданий, а каждое задание — поток Digit Stream, представляющий не выполненный пока полностью вызов insTree.

```
type Schedule = Digit Stream list
type Heap = Digit Stream × Schedule
```

Когда нам нужно выполнить шаг в расписании, мы вынуждаем голову первого задания. Если в результате получается One, значит, это задание выполнено, и мы его из расписания изымаем. Если же получается Zero, мы кладем остаток задания обратно в расписание.

```
fun exec [] = []
  | exec (($Cons (One t, _)) :: sched) =sched
  | exec (($Cons (Zero, job) :: sched) =job :: sched
```

Наконец, мы меняем код функции insert, чтобы она поддерживала расписание. Поскольку амортизированная стоимость insert равнялась двум, мы предполагаем, что выполнение двух шагов при каждом insert будет достаточно, чтобы ко времени, когда каждая задержка потребуется, она была уже вынуждена.

```
fun insert (x, (ds, sched)) =
      let val ds' = insTree (Node (x, []), ds)
      in (ds', exec (exec (ds' :: sched))) end
```

Чтобы показать, что insert занимает время $O(1)$ в худшем случае, надо сначала показать, что exec занимает $O(1)$ в худшем случае. А именно, надо показать, что всякий раз, как exec вынуждает некоторую задержку (сопоставляя её с образцом), все другие задержки, от которых зависит данная, уже вычислены и мемоизированы.

Если мы развернем конструкцию **fun** lazy в определении insTree и немного упростим результат, мы увидим, что insTree порождает задержку, эквивалентную такому коду:

```
$case ds of
      $Nil ⇒ Cons (One t, $Nil)
    | $Cons (Zero, ds') ⇒ Cons (One t, ds')
    | $Cons (One t', ds') ⇒ Cons (Zero, insTree (link (t, t'), ds'))
```

Задержка для каждой порождаемой insTree цифры зависит от задержки для предыдущей цифры того же индекса. Мы доказываем, что никогда не бывает более одной ожидающей задержки на индекс потока цифр, и, следовательно, что никакая невыполненная задержка не зависит от другой невыполненной задержки.

Пусть *область* (range) задания в расписании будет набор цифр, порожденных соответствующим вызовом insTree. Каждая такая область содержит последовательность нулей (возможно, пустую) с единицей в конце. Мы говорим, что две области *перекрываются* (overlap), если какие-то их цифры имеют один и тот же индекс в потоке цифр. Каждая невычисленная цифра находится в области некоторого задания в расписании, так что нам надо доказать, что никакие две области не перекрываются.

Мы доказываем даже несколько более сильное утверждение. Назовем *завершенным нолем* (completed zero) ноль, чья ячейка в потоке уже вычислена и мемоизирована.

Теорема 7.1 *В каждой правильно построенной куче есть по крайней мере два завершенных ноля перед первой областью в расписании, и по крайней мере по одному завершенному нолю между любыми двумя соседними областями в расписании.*

Доказательство. *Пусть r_1 и r_2 — первые две области в расписании. Пусть z_1 и z_2 — два завершенных ноля перед r_1, а z_3 — завершенный ноль между r_1 и r_2. Функция insert добавляет новую область r_0 в начало расписания, а затем немедленно дважды вызывает exec. Заметим, что r_0 заканчивается единицей-One, замещающей z_1. Пусть m будет количество нолей в r_0. Возможны три случая.*

Случай 1. $m = 0$. *Единственная цифра в r_0 — единица, так что r_0 уничтожается первым же exec. Второй exec вынуждает первую цифру r_1. Если это ноль, то он оказывается вторым завершенным нолем (помимо z_2) перед первой областью. Если же цифра — единица, то r_1 уничтожается, и новой первой областью становится r_2. Перед r_2 имеется два завершенных ноля z_2 и z_3.*

Случай 2. $m = 1$. *В r_0 содержится две цифры, ноль и единица. Эти две цифры немедленно вынуждаются двумя вызовами exec, и r_0 уничтожается. Ведущий ноль заменяет z_1 как один из двух завершенных нолей перед r_1.*

Случай 3. $m \geq 2$. *Первые две цифры r_0 — ноли. После двух вызовов exec они оказываются двумя завершенными нолями перед тем областью, которая теперь первая (остаток r_0). z_2 оказывается (единственным) завершенным нолем между r_0 и r_1.*

Упражнение 7.3 *Покажите, что аннотацию lazy в определении insTree можно удалить без всякого вреда для времени работы insert.*

Адаптация остальных функций к новым типам данных не представляет особого труда. Полная реализация приведена на Рис. 7.2. По поводу этого кода имеет смысл сделать ещё четыре небольших замечания. Во-первых, вместо того, чтобы пытаться производить какие-либо ухищрения с расписанием, функции merge и deleteMin выполняют все задержки в системе (путем

Рис. 7.2: Биномиальные кучи с расписанием.

вызова функции normalize), а расписание устанавливают в []. Во-вторых, как следует из Теоремы 7.1, в куче всегда содержится не более $O(\log n)$ невычисленных задержек, так что их вынуждение при нормализации или поиске минимального корневого элемента не влияет на асимптотическое время выполнения merge, findMin или deleteMin, поскольку каждая из них и так работает в худшем случае за $O(\log n)$. В-третьих, вспомогательная функция removeMinTree иногда дает в результате потоки цифр, завершающиеся нолями, однако эти потоки либо отбрасываются в findMin, либо сливаются со списком единиц внутри deleteMin. Наконец, deleteMin должна теперь производить больше работы, чем в предыдущих реализациях, преобразуя список детей в правильно построенную кучу. В дополнение к обращению списка, deleteMin должна добавить по единице к каждому из деревьев, а затем преобразовать список в поток. Если c — список детей, весь процесс можно записать как

```
listToStream (map One (rev c))
```

где

```
fun listToStream [] = $Nil
  | listToStream (x :: xs) = $Cons (x, listToStream xs)

fun map f [] = []
  | map f (x :: xs) = (f x) :: (map f xs)
```

Здесь map — стандартная функция, применяющая другую функцию (в нашем случае, конструктор One) ко всем элементам списка.

Упражнение 7.4 *Напишите эффективную специализированную версию mrg, называемую mrgWithList, чтобы deleteMin мог вызывать*

```
mrgWithList (rev c, ds')
```

вместо

```
mrg (listToStream (map One (rev c)), ds')
```

7.4 Сортировка снизу вверх с расписанием

В качестве третьего примера расписаний, мы изменяем сортируемые коллекции из Раздела 6.4.3, чтобы add работала в худшем случае за время $O(\log n)$, а sort в худшем случае за $O(n)$.

Единственное место, где в амортизированной реализации используется ленивое вычисление —задержанный вызов addSeg в функции add. Задержка монолитна, так что нашей первой задачей будет пошаговое выполнение этого вычисления. В сущности, достаточно сделать пошаговой только

функцию mrg: поскольку addSeg требует всего лишь $O(\log n)$ шагов, мы можем себе позволить вычислять ее энергично. Следовательно, мы представляем сегменты в виде потоков, а не списков, и отказываемся от задержки при коллекции сегментов. Новый тип для коллекции сегментов будет Elem.T Stream list, а не Elem.T list list susp.

Модификация функций mrg, add и sort под это новое представление не составляет труда; разве что sort должна окончательный отсортированный результат перевести обратно в список. Для этого используется функция преобразования streamToList.

```
fun streamToList ($Nil) =[]
  | streamToList ($Cons (x, xs)) =x :: streamToList xs
```

Новая версия mrg, приведенная на Рис. 7.3, производит слияние по шагу за раз, и собственная стоимость такого шага $O(1)$. Вторая наша цель состоит в том, чтобы проводить достаточное количество шагов слияния при каждом add и гарантировать, что любая сортируемая коллекция содержит не более $O(n)$ невычисленных задержек. Тогда sort будет проводить не более $O(n)$ вынуждений в дополнение к своей собственной работе ценой также $O(n)$. Вынуждение невычисленных задержек отнимает максимум $O(n)$ времени, так что общая стоимость sort оказывается не более $O(n)$.

При амортизационном анализе амортизированная стоимость add составляла приблизительно $2B'$, где B' — число единичных битов в $n' = n + 1$. Это наводит на мысль, что add должна выполнять две задержки на каждый бит, или, что то же самое, две задержки на сегмент. Мы храним отдельное расписание для каждого сегмента. Каждое расписание является списком потоков, которые представляют невыполненные пока полностью вызовы mrg. Таким образом, полный тип выглядит так:

```
type Schedule = Elem.T Stream list
type Sortable = int × (Elem.T Stream × Schedule) list
```

Чтобы выполнить один шаг расписания, мы зовем функцию exec1.

```
fun exec1 [] = []
  | exec1 (($Nil):: sched) =exec1 sched
  | exec1 (($Cons (x, xs) :: sched)) =xs :: sched
```

Во второй строке этой функции мы достигаем конца одного потока и выполняем первый шаг в следующем потоке. Здесь не может образоваться цикл, поскольку только первый поток в списке может быть пуст. Функция exec2 берет сегмент и дважды применяет exec1 к его расписанию.

```
fun exec2 (xs, sched) = (xs, exec1 (exec1 sched))
```

Функция add вызывает exec2 для каждого сегмента, но кроме этого она отвечает за построение расписания для нового сегмента. Если младшие k битов размера n равны единице, то добавление нового элемента приведет к k слияниям вида

$$((s_0 \bowtie s_1) \bowtie s_2) \cdots \bowtie s_k$$

Рис. 7.3: Сортировка слиянием снизу вверх с расписанием.

где s_0 — новый одноэлементный сегмент, а $s_1 \ldots s_k$ — первые k сегментов существующей коллекции. Частичными результатами этого вычисления будут $s'_1 \ldots s'_k$, где $s'_1 = s_0 \bowtie s_1$, а $s'_i = s'_{i-1} \bowtie s_i$. Поскольку задержки в s'_i зависят от задержек в s'_{i-1}, нам нужно спланировать выполнение s'_{i-1} прежде выполнения s'_i. Кроме этого, задержки в s'_i зависят от задержек в s_i, но тут мы гарантируем, что ко времени вызова add значения $s_1 \ldots s_k$ будут уже полностью вычислены.

Окончательная версия функции add, создающая новое расписание и выполняющая по две задержки на сегмент, выглядит так:

```
fun add (x, (size, segs)) =
    let fun addSeg (xs, segs, size, rsched) =
            if size mod 2 = 0 then (xs, rev rsched) :: segs
            else let val ((xs', []) :: segs') = segs
                     val xs'' = mrg (xs, xs')
                 in addSeg (xs'', segs', size div 2, xs'' :: rsched) end
        val segs' = addSeg ($Cons (x, $Nil), segs, size, [])
    in (size+1, map exec2 segs') end
```

Аккумулирующий параметр rsched собирает свежеслитые потоки в обратном порядке. Поэтому мы обращаем их список в конце и получаем правильный порядок. Сопоставление с образцом в четвертой строке требует, чтобы старое расписание для текущего сегмента было пустым, т. е., чтобы оно уже было полностью выполнено. Мы вскоре увидим, почему это так.

Полный код нашей реализации приведен на Рис. 7.3. Функция add имеет нераздельную стоимость $O(\log n)$, а sort нераздельную стоимость $O(n)$, так что, чтобы доказать ограничения для худшего случая, мы должны доказать, что $O(\log n)$ задержек, вынуждаемых add, занимают время $O(1)$ каждая, и что $O(n)$ невычисленных задержек, вынуждаемых sort, вместе требуют $O(n)$.

Каждый шаг слияния, вынуждаемый изнутри add (через exec2 и exec1) зависит от двух других потоков. Если текущий шаг является частью потока s'_i, то он зависит от потоков s'_{i-1} и s_i. Поток s'_{i-1} находился в расписании раньше s'_i, так что ко времени начала вычисления s'_i он уже был полностью вычислен. Кроме того, s_i был уже полностью вычислен ко времени вызова add, который создал s'_i. Поскольку собственная стоимость каждого шага слияния равна $O(1)$, а задержки, вынуждаемые каждым шагом, уже были вынуждены и мемоизированы, каждый шаг слияния, вынуждаемый add, отнимает время только $O(1)$ для наихудшего случая.

Следующая лемма доказывает, что каждый сегмент, вовлеченный в слияние через addSeg, уже полностью вычислен, и что коллекция в целом содержит не более $O(n)$ невычисленных задержек.

Лемма 7.2 *Во всякой сортируемой коллекции размера n расписание для сегмента размера $m = 2^k$ содержит не более $2m - 2(n \mod m+1)$ элементов.*

Доказательство. *Рассмотрим сортируемую коллекцию размера n, где k младших битов n равны единице (т. е., n можно записать как $c2^{k+1}+2^k-1$ для некоторого целого c). Тогда add порождает новый сегмент размера $m = 2^k$, и его расписание содержит потоки размеров $2, 4, 8, \ldots 2^k$. Общий размер этого расписания $2^{k+1} - 2 = 2m - 2$. После выполнения двух шагов размер расписания оказывается $2m-4$. Размер новой коллекции равен $n' = n + 1 = c2^{k+1} + 2^k$. Поскольку $2m - 4 < 2m - 2(n' \mod m+1) = 2m - 2$, для этого сегмента лемма выполняется.*

В важдом сегменте размера m', большего m, при операции add выполняются только два шага расписания, и больше ничего не происходит. Размер нового расписания ограничен

$$2m' - 2(n \mod m' + 1) - 2 = 2m' - 2(n' \mod m' + 1),$$

так что и для этих сегментов лемма также выполняется.

Теперь всякий раз, когда младшие k битов n равны единице (т. е., когда следующий add собирается слить первые k сегментов), мы из Леммы 7.2 знаем, что для каждого сегмента размера $m = 2^i$, где $i < k$, число элементов в расписании этого сегмента не больше

$$2m - 2(n \mod m + 1) = 2m - 2((m - 1) + 1) = 0$$

Другими словами, этот сегмент уже полностью вычислен.

Наконец, общий размер расписаний для всех сегментов не может быть больше

$$2\sum_{i=0}^{\infty} b_i(2^i - (n \mod 2^i + 1)) = 2n - 2\sum_{i=0}^{\infty} b_i(n \mod 2^i + 1)$$

элементов, где b_i — i-й бит числа n. Заметим, что это очень похоже на функцию потенциала из анализа методом физика в Разделе 6.4.3. Поскольку этот общий размер ограничен числом $2n$, вся коллекция содержит только $O(n)$ невыполненных задержек, а следовательно, sort выполняется в худшем случае за время $O(n)$.

7.5 Примечания

Избавление от амортизации Дитц и Раман [DR91, DR93, Ram92] разработали методику избавления от амортизации на основе *игр с камнями* (pebble games), где порождаемые алгоритмы с жёсткими ограничениями соответствуют выигрышным стратегиям в некоторой игре. Другие исследователи использовали частные методы, похожие на расписание, для избавления от амортизации в конкретных структурах данных, например, в

неявных биномиальных кучах (implicit binomial queues) [CMP88] и *расслабленных кучах* (relaxed heaps) [DGST88]. Расписания, подобные описанным в этой главе, впервые были применены к очередям в [Oka95c], а затем обобщены в [Oka96b].

Очереди Реализация кучи, описанная в Разделе 7.2, впервые появилась в [Oka95c]. Худ и Мелвилл [HM81] представили первую чисто функциональную реализацию очередей реального времени, на основе метода, известного как *глобальная перестройка* (global rebuilding) [Ove83], который будет обсуждаться в следующей главе. Их реализация сложнее нашей и не использует ленивое вычисление.

Глава 8

Ленивая перестройка

В оставшихся четырех главах мы описываем общие методы проектирования функциональных структур данных. Первый из них, рассматриваемый в этой главе — *ленивая перестройка* (lazy rebuilding), разновидность *глобальной перестройки* (global rebuilding) [Ove83].

8.1 Порционная перестройка

Во многих структурах данных соблюдаются инварианты баланса, благодаря которым гарантируется эффективный доступ. Каноническим примером могут служить сбалансированные деревья поиска, улучшающие время работы в худшем случае для многих операций с $O(n)$ у несбалансированных деревьев до $O(\log n)$. Один из подходов к соблюдению инварианта баланса — перебалансировка структуры после каждой её модификации. Для большинства сбалансированных структур существует понятие *идеального баланса* (perfect balance), то есть, конфигурация, минимизирующая стоимость последующих действий. Однако, поскольку, как правило, восстанавливать идеальный баланс после каждого изменения оказывается слишком дорого, в большинстве реализаций считается достаточным поддерживать некоторое приближение к нему, ухудшающее показатели не более чем на константный множитель. Примерами такого подхода являются AVL-деревья [AVL62] и красно-чёрные деревья [GS78].

Однако если каждое отдельное обновление не слишком сильно влияет на баланс, привлекательным альтернативным подходом будет отложить перестройку, пока не пройдёт некоторая серия операций, а затем перебалансировать всю структуру и восстановить идеальный баланс. Назовем этот подход *порционной перестройкой* (batched rebuilding). Порционная перестройка дает хорошие амортизированные ограничения, если выполняются два условия: (1) глобальная структура перестраивается не слишком часто, и (2) отдельные модифицирующие действия ухудшают показатели последующих операций не слишком сильно. Выражаясь более точно, условие

(1) говорит, что, если мы надеемся достичь амортизированного показателя $O(f(n))$ на операцию, а преобразование перебалансировки занимает время $O(g(n))$, запускать это преобразование нельзя чаще, чем раз в $c \cdot g(n)/f(n)$ операций, для некоторой константы c. Рассмотрим, например, двоичные деревья поиска. Перестройка дерева с полной балансировкой занимает время $O(n)$, так что, если мы хотим, чтобы наши операции занимали амортизированное время не больше $O(n)$, структуру данных нельзя перестраивать чаще, чем раз в $c \cdot n/\log n$ операций, для некоторой константы c.

Допустим, что структура данных будет перестраиваться раз в $c \cdot g(n)/f(n)$ операций, и что отдельная операция над перестроенной структурой отнимает время $O(f(n))$ (ограничение может быть жёстким или амортизированным). В этом случае условие (2) утверждает, что, сделав не более $c \cdot g(n)/f(n)$ обновлений непосредственно после перестройки, мы по-прежнему будем тратить время не более $O(f(n))$. Другими словами, стоимость каждой отдельной операции должна ухудшиться максимум на константный множитель. Функции обновления, удовлетворяющие условию (2), называются *операциями слабого обновления* (weak updates).

Рассмотрим, например, следующий подход к реализации функции delete на двоичных деревьях поиска. Вместо того, чтобы физически уничтожать указанный узел дерева, оставляем его в дереве с пометкой «стёрто». Затем, когда стёртыми оказываются половина узлов, делаем глобальный проход, уничтожая стёртые узлы и восстанавливая идеальный баланс. Удовлетворяет ли этот подход нашим двум условиям, если мы хотим, чтобы уничтожение элемента занимало амортизированное время $O(\log n)$?

Допустим, дерево содержит n узлов, из которых не более половины помечено как стёртые. Уничтожение стёртых узлов и восстановление идеального баланса в дереве занимает время $O(n)$. Мы выполняем это преобразование раз в $\frac{1}{2}n$ операций уничтожения, так что условие (1) выполнено. На самом деле, условие (1) позволяет нам перестраивать структуру даже чаще, раз в $c \cdot n/\log n$ операций. Наивный алгоритм уничтожения ищет нужный узел и помечает его как стёртый. Это отнимает время $O(\log n)$, даже если половина узлов уже помечена как стёртые, так что условие (2) выполнено. Заметим, что даже если половина узлов в дереве помечена, средняя глубина активного узла больше всего на единицу по сравнению со случаем, когда они физически уничтожены. Дополнительная глубина ухудшает стоимость операции всего лишь на аддитивную константу, в то время как условие (2) позволяет времени каждой операции ухудшаться на константный множитель. Следовательно, условие (2) позволяет нам перестраивать нашу структуру данных даже ещё реже.

В этом рассуждении мы говорили только об уничтожении узлов. Разумеется, как правило, в двоичных деревьях поддерживается также операция вставки элемента. К сожалению, вставка не является слабым обновлением, поскольку вставками можно очень быстро создать длинную цепочку вершин. Возможен, однако, гибридный подход, когда при каждой вставке мы проводим локальную перебалансировку, как в AVL или красно-чёрных деревьях, а уничтожение элемента обрабатывается методом порционной пе-

рестройки.

Упражнение 8.1 *Добавьте к красно-чёрным деревьям из Раздела 3.3 функцию delete на основе описанного здесь подхода. Добавьте к конструктору T булевское поле, и поддерживайте счётчики-оценки числа активных и неактивных элементов в дереве. Для этих счётчиков предполагайте, что каждая вставка создает новый элемент, а каждая операция уничтожения делает какой-то активный элемент неактивным. Обновляйте значение этих счётчиков при перестройке. Для перестройки воспользуйтесь решением Упражнения 3.9.*

В качестве второго примера порционной перестройки рассмотрим порционные очереди из Раздела 5.2. Преобразование перестройки переносит обращённый хвостовой список в головной, и очередь переходит в идеально сбалансированное состояние, когда все элементы содержатся в головном списке. Как мы уже видели, порционные очереди имеют хорошие показатели эффективности, но только при эфемерном использовании. Если их использовать как устойчивую структуру, амортизированные характеристики деградируют до стоимости операции перестройки, поскольку эта операция может срабатывать сколь угодно часто. Это наблюдение верно для всех структур с порционной перестройкой.

8.2 Глобальная перестройка

Овермарс [Ove83] описывает метод избавления от амортизации, основанный на порционной перестройке. Он называет этот метод *глобальная перестройка* (global rebuilding). Основная идея состоит в том, чтобы проводить трансформацию перестройки постепенно, по несколько шагов при каждой нормальной операции. Полезно рассматривать это как выполнение преобразования в сопрограмме. Сложность в том, чтобы запустить сопрограмму достаточно рано, чтобы она завершилась ко времени, когда понадобится перестроенная структура.

Более конкретно, при глобальной перестройке поддерживаются две копии каждого объекта. Первичная, или *рабочая копия* (working copy) — это исходная структура. Вторичная копия — та, которая постепенно перестраивается. Все запросы и операции обновления обращаются к рабочей копии. Когда построение вторичной копии завершено, она становится новой рабочей копией, а старая уничтожается. При этом либо сразу же запускается новая вторичная копия, либо некоторое время объект может работать без вторичной структуры, прежде чем начнётся новая фаза перестройки.

Отдельную сложность представляет обработка обновлений, происходящих, пока ведется перестройка вторичной копии. Рабочая копия обновляется обычным образом, но должна быть обновлена и вторичная копия, иначе, когда она станет рабочей, эффект обновления будет потерян. Однако в общем случае вторичная копия представлена не в такой форме, которую можно эффективно обновить. Таким образом, обновления вторичной копии

буферизуются и выполняются, по несколько за раз, после того, как вторичная копия перестроена, но до того, как она становится рабочей.

Глобальную перестройку можно реализовать в чисто функциональном стиле, и несколько таких реализаций существуют. Например, очереди реального времени Худа и Мелвилла [HM81] основаны именно на этом методе. В отличие от порционной перестройки, при глобальной перестройке не возникает проблем с устойчивостью. Поскольку ни одна из операций не является особенно дорогой, произвольное повторение операций не влияет на временные характеристики. К сожалению, часто глобальная перестройка дает очень сложные структуры. В частности, представление вторичной копии, которое сводится к хранению промежуточного состояния сопрограммы, может быть довольно неприятным.

8.2.1 Пример: очереди реального времени по Худу-Мелвиллу

Реализация очередей реального времени Худа и Мелвилла [HM81] во многом похожа на очереди реального времени из Раздела 7.2. В обеих реализациях поддерживается два списка, представляющие головную и хвостовую части очереди соответственно, и ведется пошаговый процесс переноса элементов из хвостового списка в головной, начиная с того момента, когда хвостовой список становится на единицу длиннее, чем головной. Разница состоит в деталях этого пошагового проворота.

Рассмотрим сначала, как можно провести пошаговое обращение списка путем хранения двух списков и постепенного переноса элементов из одного в другой.

```
datatype α ReverseState =Working of α list × α list |Done of α list

fun startReverse xs = Working (xs, [])

fun exec (Working (x :: xs, xs')) = Working (xs, x :: xs')
  | exec (Working ([], xs')) = Done xs'
```

Чтобы обратить список xs, мы сначала создаем новое состояние Working (xs, []), а затем многократно вызываем exec, пока не получим состояние Done с обращенным списком. Всего требуется $n + 1$ вызовов exec, где n — длина исходного списка xs.

Можно провести пошаговую конкатенацию двух списков, применив этот прием дважды. Сначала мы обращаем xs, получая xs', а затем обращаем xs', добавляя его к ys.

```
datatype α AppendState =
      Reversing of α list × list × α list × α list
    | Appending of α list × α list
    | Done of α list

fun startAppend (xs, ys) = Reversing (xs, [], ys)

fun exec (Reversing (x :: xs, xs', ys)) = Reversing (xs, x :: xs', ys)
```

```
  | exec (Reversing ([], xs', ys)) = Appending (xs', ys)
  | exec (Appending (x :: xs', ys)) = Appending (xs', x :: ys)
  | exec (Appending ([], ys)) = Done ys
```

Всего требуется $2m + 2$ вызова exec, если длина исходного списка xs равна m.

Наконец, чтобы добавить f к обращенному r, мы проводим три обращения. Сначала мы в параллель обращаем f и r, получая f' и r', а затем приписываем обращенный f' к r'. Нижеследующий код предполагает, что длина r на единицу больше длины f.

```
datatype α RotationState =
    Reversing of α list × α list × α list × α list
  | Appending of α list × α list
  | Done of α list

fun startRotation (f, r) = Reversing (f, [], r, [])

fun exec (Reversing (x :: f, f', y :: r, r')) = Reversing (f, x :: f', r, y :: r')
  | exec (Reversing ([], f', [y], r')) = Appending (f', y :: r')
  | exec (Appending (x :: f', r')) = Appending (f', x :: r')
  | exec (Appending ([], r') = Done r'
```

Как и раньше, процедура завершается после $2m + 2$ вызовов exec, где m — исходная длина списка f.

К сожалению, у этого способа проворота есть большой недостаток. Если мы просто зовем exec по несколько раз при каждом вызове snoc или tail, то ко времени, когда проворот закончится, ответ может быть уже не тот, который нам нужен! В частности, если за время проворота было k вызовов tail, то k первых элементов получившегося списка уже не актуальны. Эту проблему можно решить двумя основными способами. Во-первых, можно хранить счетчик устаревших элементов, и добавить к процедуре проворота третье состояние Deleting, которое уничтожает элементы по несколько за раз, пока устаревшие элементы не кончатся. Этот подход точнее всего соответствует определению глобальной перестройки. Однако ещё лучше просто не включать устаревшие элементы в окончательный список. Мы отслеживаем, сколько живых элементов осталось в f', и перестаем копировать элементы из f' в r', когда счетчик достигает нуля. Каждый вызов tail во время проворота уменьшает число живых элементов.

```
datatype α RotationState =
    Reversing of int × α list × α list × α list × α list
  | Appending of int × α list × α list
  | Done of α list

fun startRotation (f, r) = Reversing (0, f, [], r, [])

fun exec (Reversing (ok, x :: f, f', y :: r, r')) =
      Reversing (ok+1, f, x :: f', r, y :: r')
  | exec (Reversing (ok, [], f', [y], r')) = Appending (ok, f', y :: r')
```

Рис. 8.1: Очереди реального времени на основе глобальной перестройки.

```
  | exec (Appending (0, f', r')) = Done r'
  | exec (Appending (ok, x :: f', r')) = Appending (ok−1, f', x ::
r')

fun invalidate (Reversing (ok, f, f', r, r')) = Reversing (ok−1, f, f', r, r')
  | invalidate (Appending (0, f', x :: r')) = Done r'
  | invalidate (Appending (ok, f', r')) = Appending (ok−1, f', r')
```

Этот процесс завершается после $2m+2$ обращений к exec или invalidate, где m — исходная длина f.

Требуется рассмотреть ещё три нетривиальных мелких вопроса. Во-первых, во время проворота несколько начальных элементов очереди оказываются в конце поля f' структуры-состояния проворота. Как нам при этом отвечать на запрос head? Решение этой дилеммы состоит в том, чтобы хранить рабочую копию старого головного списка. Нужно только добиться того, чтобы новая копия головного списка оказалась готова к тому времени, как исчерпается старая. Во время проворота поле lenf измеряет длину создаваемого списка, а не рабочей копии f. Однако между проворотами поле lenf содержит длину f.

Во-вторых, надо решить, сколько именно обращений к exec надо делать при каждом вызове snoc и tail, чтобы гарантировать, что проворот закончится к тому времени, когда либо нужно будет начать следующий проворот, либо будет израсходована рабочая копия головного списка. Допустим, что в начале проворота длина списка f равна m, а длина списка r равна $m+1$. Тогда следующий проворот начнется после $2m+2$ вставок или извлечений (в любом соотношении), однако рабочая копия головного списка окажется израсходованной уже через m извлечений. Всего проворот заканчивается через $2m+2$ шагов. Если при каждой операции мы зовем exec два раза, включая операцию, которая запускает проворот, то проворот завершится самое большее через m операций после своего начала.

В-третьих, поскольку каждый проворот заканчивается задолго до того, как начинается следующий, требуется добавить к типу RotationState состояние Idle (неактивное), так что exec Idle = Idle. После этого мы можем спокойно звать exec, не заботясь о том, находимся мы в процессе проворота или нет.

Оставшиеся детали должны уже быть знакомы читателю. Полная реализация приведена на Рис. 8.1.

Упражнение 8.2 *Докажите, что если звать exec дважды при начале каждого проворота и один раз при каждой вставке или извлечении элемента, этого будет достаточно, чтобы проворот завершался вовремя. Соответствующим образом измените код.*

Упражнение 8.3 *Замените поля lenf и lenr одним полем diff, которое хранит разницу между длинами списков f и r. Поле diff не обязательно должно хранить точное значение в процессе проворота, но к концу проворота должно быть точным.*

8.3 Ленивая перестройка

Реализация очередей по методу физика из Раздела 6.4.2 очень похожа на версию с глобальной перестройкой, но имеется и существенное различие. Как и при глобальной перестройке, в этой реализации поддерживаются две копии головного списка, рабочая копия w и вторичная копия f, причем все запросы обращаются к рабочей копии. Операции обновления f (т. е., операции tail) буферизуются и выполняются по окончании проворота через выражение

```
$tl (force f)
```

Кроме того, эта реализация заботится о том, чтобы начать (или, по крайней мере, спланировать) проворот задолго до того, как понадобится его результат. Однако, в отличие от глобальной перестройки, эта реализация не занимается *выполнением* преобразования перестройки (т. е., проворота) в параллель с нормальными операциями; вместо этого она *оплачивает* преобразование перестройки одновременно с нормальными операциями, но затем, когда вся стоимость преобразования выплачена, оно выполняется целиком. В сущности, мы заменили сложности явного или неявного переноса перестройки в сопрограмму более простым механизмом ленивого вычисления. Этот вариант глобальной перестройки мы называем *ленивой перестройкой* (lazy rebuilding).

Реализация очередей по методу банкира из Раздела 6.3.2 показывает ещё одно упрощение, доступное нам при использовании ленивой перестройки. Внося вложенные задержки в исходную структуру данных — например, используя потоки вместо списков, — мы часто можем уничтожить различие между рабочей и вторичной копиями, и использовать единую структуру, обладающую свойствами их обеих. «Рабочая» часть этой структуры — это та часть, которая уже оплачена, а «вторичная» — та, за которую выплата ещё не произведена.

У глобальной перестройки есть два преимущества перед порционной: она годится для реализации устойчивых структур данных, а также соблюдает жёсткие ограничения вместо амортизированных. Ленивая перестройка также обладает первым из этих преимуществ, однако, по крайней мере, в простейшей своей форме дает амортизированные ограничения. Но если это требуется, часто можно восстановить жёсткие ограничения, используя расписания по методам из Главы 7. Например, очереди реального времени из Раздела 7.2 сочетают ленивую перестройку с расписанием, и получают в итоге реализацию с жёсткими характеристиками. В сущности, сочетание ленивой перестройки с расписаниями можно рассматривать как разновид-

Рис. 8.2: Сигнатура для двусторонних очередей.

ность глобальной перестройки, где сопрограммы реифицированы особенно простым образом через ленивое вычисление.

8.4 Двусторонние очереди

В качестве дальнейших примеров глобальной перестройки мы приведем несколько реализаций двусторонних очередей, или *деков* (deques). Деки отличаются от очередей FIFO тем, что элементы могут как добавляться, так и изыматься с любого конца очереди. Сигнатура для деков приведена на Рис. 8.2. Эта сигнатура расширяет сигнатуру очередей тремя новыми функциями: cons (добавить элемент к началу очереди), last (вернуть последний элемент) и init (изъять последний элемент).

Замечание 8.1 *Заметим, что сигнатура очередей является строгим подмножеством сигнатуры для деков — для типа и аналогичных функций были выбраны совпадающие имена. Поскольку деки являются строгим расширением очередей, Стандартный ML позволит нам использовать дек везде, где ожидается модуль, реализующий очередь.*

8.4.1 Деки с ограниченным выходом

Сначала заметим, что реализации очередей из Глав 6 и 7 можно тривиально расширить, добавив в дополнение к операции snoc операцию cons. Очередь, поддерживающая добавление элементов с обоих концов, но удаление только с одного, называется *дек с ограниченным выходом* (output-restricted deque).

Например, можно реализовать cons в очередях по методу банкира из Раздела 6.3.2 следующим образом:

```
fun cons (x, (lrnf, f, lenr, r)) = (lenf+1, $Cons (x, f), lenr, r)
```

Заметим, что нет никакой необходимости звать вспомогательную функцию check, поскольку добавление элемента к f никак не может сделать f короче, чем r.

Подобным же образом легко реализовать функцию cons для очередей реального времени из Раздела 7.2.

```
fun cons (x, (f, r,s)) = ($Cons (x, f), r, $Cons (x, s))
```

Мы добавлякм x к s только для того, чтобы поддержать инвариант $|s| = |f| - |r|$.

Упражнение 8.4 *К сожалению, очереди реального времени по Худу-Мелвиллу не так легко расширяются функцией cons, поскольку нет простого способа вставить элемент в структуру-состояние проворота. Напишите вместо*

этого функтор, который расширяет любую *реализацию очередей функцией cons, работающей за константное время, с использованием типа*

type α *Queue* $=\alpha$ *list* $\times$ α *Q.Queue*

где Q — параметр функтора. cons должен вставлять элементы в новый список, а head и tail должны удалять элементы из нового списка, когда он непуст.

8.4.2 Деки по методу банкира

Деки можно представлять так же, как очереди, в виде двух потоков (или списков), f и r, плюс некоторая дополнительная информация, помогающая поддерживать баланс. Для очередей идеально сбалансированная ситуация — когда все элементы находятся в головном потоке. Для деков идеально сбалансированное состояние — когда элементы поделены поровну между головным и хвостовым потоками. Поскольку мы не можем себе позволить восстанавливать идеальный баланс после каждой операции, мы удовольствуемся гарантией, что ни один из потоков не может быть длиннее другого более чем в c раз, для некоторой константы $c > 1$. А именно, мы поддерживаем следующий инвариант баланса:

$$|\mathrm{f}| \leq c|\mathrm{r}| + 1 \quad \wedge \quad |\mathrm{r}| \leq c|\mathrm{f}| + 1$$

Подвыражение «+1» в каждом из термов позволяет единственному элементу одноэлементного дека находиться в любом из двух потоков. Заметим, что если дек состоит по крайней мере из двух элементов, оба потока должны быть непусты. Каждый раз, когда инвариант грозит оказаться нарушенным, мы возвращаем дек в идеально сбалансированное состояние, перенося элементы из более длинного потока в более короткий, пока их длины не уравниваются.

На основе этих идей мы можем адаптировать либо очереди по методу банкира из Раздела 6.3.2, либо очереди по методу физика из Раздела 6.4.2, и получить дек, поддерживающий каждую операцию за амортизированное время $O(1)$. Поскольку банковские очереди немного проще, мы решили работать именно с ними.

Тип банковских деков в точности такой же, как у банковских очередей.

```
type α Queue =int × α Stream × int × α Stream
```

Функции, работающие с первым элементом, определены так:

```
fun cons (x, (lenf, f, lenr, r)) = check (lenf+1, $Cons (x, f), lenr, r)
fun head (lenf, $Nil, lenr, $Cons (x, _)) =x
  | head (lenf, $Cons (x, f'), lenr, r) =x
fun tail (lenf, $Nil, lenr, $Cons (x, _)) =empty
  | tail (lenf, Cons (x, f'), lenr, r) = check (lenf−1, f', lenr, r)
```

Первые варианты в определениях head и tail обрабатывают одноэлементные деки, чей единственный элемент хранится в хвостовом потоке. Функции,

Рис. 8.3: Реализация деков, основанная на ленивой перестройке и методе банкира.

работающие с последним элементом — snoc, last и init, — определяются симметричным образом.

Все интересное в этой реализации деков происходит во вспомогательной функции check, которая восстанавливает в деке идеальный баланс, когда один из потоков оказывается чрезмерно длинным, сначала обрезая более длинный поток так, чтобы его длина равнялась половине суммарной длины двух списков, а затем перенося оставшиеся элементы более длинного потока в конец более короткого. Например, если $|\mathrm{f}| > c|\mathrm{r}| + 1$, то check заменяет f на take (i, f), а r на r ++ reverse (drop (i, f)), где $\mathrm{i} = \lfloor(|\mathrm{f}| + |\mathrm{r}|)/2\rfloor$. Полное определение check выглядит так:

```
fun check (q as (lenf, f, lenr, r)) =
    if lenf > c*lenr + 1 then
        let val i = (lenf + lenr) div 2        val j = lenf + lenr - i
            val f' = take (i, f)               val r' = r ++ reverse (drop (i, f))
        in (i, f', j, r') end
    else if lenr > c*lenf + 1 then
        let val j = (lenf + lenr) div 2        val i = lenf + lenr - j
            val r' = take (j, r)               val f' = f ++ reverse (drop (j, r))
        in (i, f', j, r') end
    else q
```

Полностью эта реализация приведена на Рис. 8.3.

Замечание 8.2 *Поскольку наша реализация симметрична, мы можем обратить дек за время $O(1)$, попросту поменяв f и r ролями.*

```
fun reverse (lenf, f, lenr, r) = (lenr, r, lenf, f)
```

Это свойство разделяют многие другие реализации деков [Hoo92, CG93]. Вместо того, чтобы повторять весь код для функций над первым и последним элементами, можно определить функции для последнего элемента через reverse и функции для первого элемента. Например, init можно реализовать как

```
fun init q = reverse (tail (reverse q))
```

Разумеется, будучи реализована напрямую, init немного быстрее.

Для анализа наших деков мы снова обращаемся к методу банкира. Как для головного, так и для хвостового потока, пусть $d(i)$ будет число единиц долга, приписанных к i-му элементу потока, и пусть $D(i) = \sum_{j=0}^{i} d(j)$. Будем поддерживать инвариант, что как для головного, так и для хвостового потока

$$D(i) \leq \min(ci + i, cs + 1 - t)$$

где $s = \min(|\mathrm{f}|, |\mathrm{r}|)$, а $t = \max(|\mathrm{f}|, |\mathrm{r}|)$. Поскольку $d(0) = 0$, головные элементы обоих потоков не имеют долга, и к ним всегда можно обращаться функциями head и last.

Теорема 8.1 *cons и tail (и, симметрично к ним, snoc и init) поддерживают инвариант долга как на головном, так и на хвостовом потоке, высвобождая, соответственно, не более 1 и $c+1$ единиц долга на поток.*

Доказательство. *Подобно доказательству Теоремы 6.1 на стр. 68.*

Как теперь легко убедиться, у каждой операции нераздельная стоимость равна $O(1)$, и, по Теореме 8.1, каждая операция высвобождает не более $O(1)$ единиц долга. Следовательно, все операции работают за амортизированное время $O(1)$.

Упражнение 8.5 *Докажите Теорему 8.1.*

Упражнение 8.6 *Рассмотрите достоинства и недостатки при выборе различных значений константы c. Постройте последовательность операций, которая при $c = 4$ будет работать значительно быстрее, чем при $c = 2$. Затем постройте последовательность операций, которая будет значительно быстрее при $c = 2$, чем при $c = 4$.*

8.4.3 Деки реального времени

Дек реального времени (real-time deque) все операции выполняет за $O(1)$ в худшем случае. Мы получаем деки реального времени на основе деков из предыдущего раздела, снабжая головной и хвостовой потоки расписаниями.

Как всегда, первый шаг в применении метода расписаний состоит в том, чтобы преобразовать все монолитные функции в пошаговые. В предыдущей нашей реализации трансформация перестройки заменяла f и r на f ++ reverse (drop (j, r)) и take (j, r) (или наоборот). Функции take и ++ уже являются пошаговыми, но reverse и drop монолитны. Поэтому мы переписываем f ++ reverse (drop (j, r)) как rotateDrop (f, j, r). rotateDrop проводит c шагов операции drop на каждый шаг ++, а в конце зовет rotateRev, которая, в свою очередь, выполняет c шагов reverse на каждый остающийся шаг ++. rotateDrop можно реализовать как

```
fun rotateDrop (f, j, r) =
      if j < c then rotateRev (f, drop (j, r), $Nil)
      else let val ($Cons (x, xf')) =f
           in $Cons (x, rotateDrop (f', j - c, drop (c, r))) end
```

Вначале $|\mathrm{r}| = c|\mathrm{f}| + 1 + k$, где $1 \le k \le c$. При каждом вызове rotateDrop, кроме последнего, мы отбрасываем c элементов r и обрабатываем один элемент f. При последнем вызове мы отбрасываем $j \mod c$ элементов r, а f оставляем неизменным. Следовательно, при первом вызове rotateRev мы имеем $|\mathrm{r}| = c|\mathrm{f}|+1+k-(j \mod c)$. Удобно будет, если $|\mathrm{r}| \ge c|\mathrm{f}|$, так что мы требуем, чтобы $1+k-(j \mod c) \ge 0$. Это гарантировано только при $c < 4$. Поскольку c должно быть больше единицы, в качестве разрешённых значений c остаются только 2 и 3. Теперь мы можем реализовать rotateRev как

Рис. 8.4: Деки реального времени с ленивой перестройкой и расписаниями.

```
fun rotateRev ($Nil, r, a) =reverse r ++ a
  | rotateRev ($Cons (x, f), r, a) =
      $Cons (x, rotateRev (f, drop (c, r), reverse (take (c, r)) ++ a))
```

Заметим, что rotateDrop и rotateRev часто вызывают drop и reverse — те самые функции, которых мы хотели избежать. Однако теперь drop и reverse всегда зовутся с аргументами ограниченного размера, а следовательно, выполняются за $O(1)$ шагов.

После того, как монолитные функции преобразованы в пошаговые, следующим шагом мы устанавливаем расписания для задержек внутри f и r. Для каждого из этих потоков мы поддерживаем отдельное расписание, и на каждом шаге выполняем по несколько задержек из каждого расписания. Как и в очередях реального времени из Раздела 7.2, наша цель состоит в том, чтобы оба расписания были полностью выполнены ко времени следующего проворота, чтобы задержки, вынуждаемые внутри rotateDrop и rotateRev, были уже с гарантией мемоизированы.

Упражнение 8.7 *Покажите, что если выполнять по одной задержке на каждую вставку и по две задержки на каждое изъятие элемента, то мы можем гарантировать, что оба расписания будут полностью выполнены ко времени следующего проворота.*

Реализация полностью приведена на Рис. 8.4.

8.5 Примечания

Глобальная перестройка Глобальная перестройка была впервые предложена Овермарсом [Ove83]. С тех пор она использовалась во многих ситуациях, включая очереди реального времени [HM81], деки реального времени [Hoo82, GT86, Sar86, CG93], деки с конкатенацией [BT95] и в задаче поддержания порядка [DS87].

Деки Первым, кто адаптировал очереди реального времени из [HM81] и получил деки реального времени, был Худ [Hoo82]. Эта работа была повторена ещё несколькими исследователями [GT86, Sar86, CG93]. Все эти реализации похожи на методы, используемые для эмуляции машин Тьюринга с несколькими головками [Sto70, FMR72, LS81]. Хогерворд [Hoo92] предложил амортизированные деки на основе порционной перестройки, однако, как и всегда при порционной перестройке, его реализация неэффективна, будучи использованной в качестве устойчивой структуры. Деки реального времени с Рис. 8.4 впервые появились в [Oka95c].

Сопрограммы и ленивое вычисление Потоки (и другие ленивые структуры данных) часто использовались для реализации сопрограмм между источником данных в потоке и потребителем этих данных. Ландин [Lan65]

был первым, кто указал на связь между потоками и сопрограммами. Некоторые убедительные примеры использования этой конструкции можно найти у Хьюза [Hug89].

Глава 9

Числовые представления

Рассмотрим обыкновенные представления списков и натуральных чисел, а также несколько типичных функций над этими типами данных.

```
datatype α List =                    datatype Nat =
    Nil                                  Zero
  | Cons of α × α List                 | Succ of Nat

fun tail (Cons (x, xs)) = xs         fun pred (Succ n) = n

fun append (Nil, ys) = ys            fun plus (Zero, n) = n
  | append (Cons (x, xs), ys) =        | plus (Succ m, n) =
      Cons (x, append (xs, ys))            Succ (plus (m, n))
```

Помимо того, что списки содержат элементы, а натуральные числа нет, эти две реализации практически совпадают. Подобным же образом соотносятся биномиальные кучи и двоичные числа. Эти примеры наводят на сильную аналогию между представлениями числа n и представлениями объектов-контейнеров размером n. Функции, работающие с контейнерами, полностью аналогичны арифметическим функциям, работающим с числами. Например, добавление нового элемента похоже на увеличение числа на единицу, удаление элемента похоже на уменьшение числа на единицу, а слияние двух контейнеров похоже на сложение двух чисел. Можно использовать эту аналогию для проектирования новых представлений абстракций контейнеров — достаточно выбрать представление натуральных чисел, обладающее заданными свойствами, и соответствующим образом определить функции над объектами-контейнерами. Назовем реализацию, спроектированную при помощи этого приёма, *числовым представлением* (numerical representation).

В этой главе мы исследуем несколько числовых представлений для двух различных абстракций: *куч* (heaps) и *списков со свободным доступом* (random-access lists) (известных также как *гибкие массивы* (flexible arrays)). Эти две абстракции подчёркивают различные наборы арифметических операций. Для куч требуются эффективные функции увеличения на единицу и сложения, а для списков со свободным доступом требуются эффективные

функции увеличения и уменьшения на единицу.

9.1 Позиционные системы счисления

Позиционная система счисления (positional number system) [Knu73b] — способ записи числа в виде последовательности цифр $b_0 \dots b_{m-1}$. Цифра b_0 называется *младшим разрядом* (least significant digit), а цифра b_{m-1} *старшим разрядом* (most significant digit). Кроме обычных десятичных чисел, мы всегда будем записывать последовательности цифр в порядке от младшего разряда к старшему.

Каждый разряд b_i имеет вес w_i, так что значение последовательности $b_0 \dots b_{m-1}$ равно $\sum_{i=0}^{m-1} b_i w_i$. Для каждой конкретной позиционной системы счисления последовательность весов фиксирована, и фиксирован набор цифр D_i, из которых выбирается каждая b_i. Для единичных чисел $w_i = 1$ и $D_i = \{\mathtt{1}\}$ для всех i, а для двоичных чисел $w_i = 2^i$, а $D_i = \{\mathtt{0}, \mathtt{1}\}$. (Мы принимаем соглашение, по которому все цифры, кроме обычных десятичных, изображаются машинописным шрифтом.) Говорится, что число записано по основанию B, если $w_i = B^i$, а $D_i = \{\mathtt{0}, \dots, B-1\}$. Чаще всего, но не всегда, веса разрядов представляют собой увеличивающуюся степенную последовательность, а множество D_i во всех разрядах одинаково.

Система счисления называется *избыточной* (redundant), если некоторые числа могут быть представлены более, чем одним способом. Например, можно получить избыточную систему двоичного счисления, взяв $w_i = 2^i$ и $D_i = \{\mathtt{0}, \mathtt{1}, \mathtt{2}\}$. Тогда десятичное число 13 можно будет записать как `1011`, `1201` или `122`. Мы запрещаем нули в конце числа, поскольку иначе почти все системы счисления будут тривиально избыточны.

Компьютерные представления позиционных систем счисления могут быть *плотными* (dense) или *разреженными* (sparse). Плотное представление — это просто список (или какая-то другая последовательность) цифр, включая нули. Напротив, при разреженном представлении нули пропускаются. В таком случае требуется хранить информацию либо о ранге (т. е., индексе), либо о весе каждой ненулевой цифры. На Рис. 9.1 показаны два разных представления двоичных чисел в Стандартном ML, одно из которых плотное, второе разреженное, а также функции увеличения на единицу, уменьшения на единицу и сложения для каждого из них. Среди уже виденных нами числовых представлений биномиальные кучи с расписаниями (Раздел 7.3) используют плотное представление, а биномиальные кучи (Раздел 3.2) и ленивые биномиальные кучи (Раздел 6.4.1) — разреженное представление.

9.2 Двоичные числа

Имея позиционную систему счисления, мы можем реализовать числовое представление на её основе в виде последовательности деревьев. Количество

возрастающий порядок по старшинству
перенос
занятие
перенос
возрастающий порядок весов, каждый из которых степень двойки

Рис. 9.1: Два представления двоичных чисел.

и размеры деревьев, представляющих коллекцию размера n, определяются положением n в позиционной системе счисления. Для каждого веса w_i имеются b_i деревьев соответствующего размера. Например, двоичное представление числа 73 выглядит как `1001001`, так что коллекция размера 73 в двоичном числовом представлении будет содержать три дерева размеров 1, 8 и 64.

Как правило, деревья в числовых представлениях обладают весьма регулярной структурой. Например, в двоичных числовых представлениях все деревья имеют размер-степень двойки. Три часто встречающихся типа деревьев с такой структурой — *полные двоичные листовые деревья* (complete binary leaf trees) [KD96], *биномиальные деревья* (binomial trees) [Vui78] и *подвешенные деревья* (pennants) [SS90].

Определение 9.1 ***(Полные двоичные листовые деревья)*** *Полное двоичное листовое дерево ранга 0 — это лист; полное двоичное листовое дерево ранга $r > 0$ представляет собой узел с двумя поддеревьями, каждое из которых является полным двоичным листовым деревом ранга $r - 1$. Листовое дерево — это дерево, хранящее элементы только в листовых узлах, в отличие от обычных деревьев, где элементы содержатся в каждом узле. Полное двоичное дерево ранга r содержит $2^{r+1} - 1$ узлов, но только 2^r листьев. Следовательно, полное двоичное листовое дерево ранга r содержит 2^r элементов.*

Определение 9.2 ***(Биномиальные деревья)*** *Биномиальное дерево ранга r представляет собой узел с r дочерними деревьями $c_1 \ldots c_r$, где каждое c_i является биномиальным деревом ранга $r-i$. Можно также определить биномиальное дерево ранга $r > 0$ как биномиальное дерево ранга $r-1$, к которому в качестве самого левого поддерева добавлено другое биномиальное дерево ранга $r-1$. Из второго определения легко видеть, что биномиальное дерево ранга r содержит 2^r узлов.*

Определение 9.3 ***(Подвешенные деревья)*** *Подвешенное дерево ранга 0 представляет собой один узел, а подвешенное дерево ранга $r > 0$ представляет собой узел с единственным поддеревом — полным двоичным деревом ранга $r - 1$. Полное двоичное дерево содержит $2^r - 1$ элементов, так что подвешенное дерево содержит 2^r элементов.*

Три этих разновидности деревьев показаны на Рис. 9.2. Выбор разновидности для каждой структуры данных зависит от свойств, которыми эта

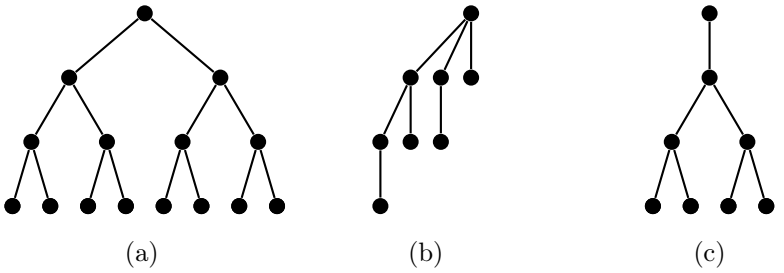


Рис. 9.2: Три дерева ранга 3: (a) полное двоичное листовое дерево, (b) биномиальное дерево и (c) подвешенное дерево.

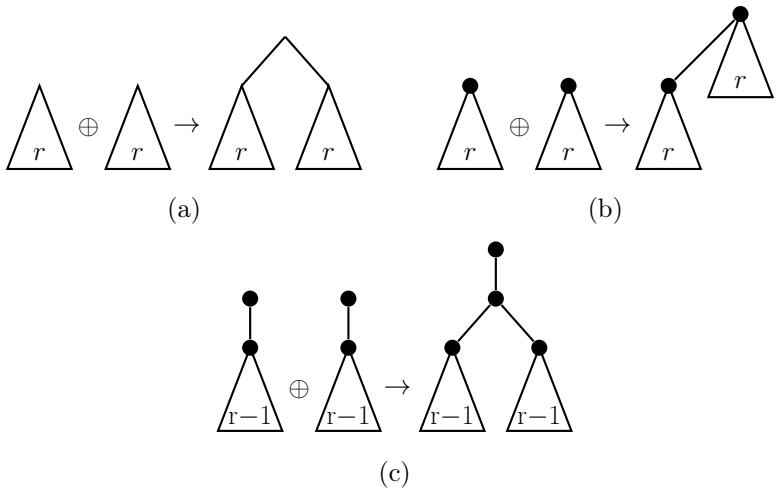


Рис. 9.3: Связывание двух деревьев ранга r в дерево ранга $r+1$ для (a) полных двоичных листовых деревьев, (b) биномиальных деревьев и (c) подвешенных деревьев.

структура должна обладать, например, от порядка, в котором требуется хранить элементы в деревьях. Важным вопросом при оценке соответствия разновидности деревьев для конкретной структуры данных будет то, насколько хорошо данная разновидность поддерживает функции, аналогичные переносу и занятию в двоичной арифметике. При имитации переноса мы *связываем* (link) два дерева ранга r и получаем дерево ранга $r+1$. Аналогично, при имитации занятия мы *развязываем* (unlink) дерево ранга $r > 0$ и получаем два дерева ранга $r-1$. На Рис. 9.3 показана операция связывания (обозначенная $\oplus$) для каждой из трех разновидностей деревьев. Если мы предполагаем, что элементы не переупорядочиваются, любая из разновидностей может быть связана или развязана за время $O(1)$.

В предыдущих главах мы уже видели несколько реализаций куч, основанных на двоичной арифметике и биномиальных деревьях. Теперь мы

Рис. 9.4: Сигнатура списков с произвольным доступом.

сначала рассмотрим простое числовое представление для списков с произвольным доступом. Затем мы исследуем насколько вариаций двоичной арифметики, позволяющих улучшить асимптотические показатели.

9.2.1 Двоичные списки с произвольным доступом

Список с произвольным доступом (random access list), называемый также односторонним гибким массивом — это структура данных, поддерживающая, подобно массиву, функции доступа и модификации любого элемента, а также обыкновенные функции для списков: cons, head и tail. Сигнатура списков с произвольным доступом приведена на Рис. 9.4.

Мы реализуем списки с произвольным доступом, используя двоичное числовое представление. Двоичный список с произвольным доступом размера n содержит по дереву на каждую единицу в двоичном представлении n. Ранг каждого дерева соответствует рангу соответствующей цифры; если i-й бит n равен единице, то список с произвольным доступом содержит дерево размера 2^i. Мы можем использовать любую из трех разновидностей деревьев и либо плотное, либо разреженное представление. Для этого примера мы используем простейшее сочетание: полные двоичные листовые деревья и плотное представление. Таким образом, тип RList выглядит так:

```
datatype α Tree =Leaf of α |Node of int × α Tree × α Tree
datatype α Digit =Zero | One of α Tree
datatype α RList =α Digit list
```

Целое число в каждой вершине — это размер дерева. Это число избыточно, поскольку размер каждого дерева полностью определяется размером его родителя или позицией в списке цифр, но мы всё равно его храним ради удобства. Деревья хранятся в порядке возрастания размера, а порядок элементов — слева направо, как внутри, так и между деревьями. Таким образом, головой списка с произвольным доступом является самый левый лист наименьшего дерева. На Рис. 9.5 показан двоичный список с произвольным доступом размера 7. Заметим, что максимальное число деревьев в списке размера n равно $\lfloor \log(n+1) \rfloor$, а максимальная глубина дерева равна $\lfloor \log n \rfloor$.

Вставка элемента в двоичный список с произвольным доступом (при помощи cons) аналогична увеличению двоичного числа на единицу. Напомним функцию увеличения для двоичных чисел:

```
fun inc [] = [One]
  | inc (Zero :: ds) = One :: ds
  | inc (One :: ds) = Zero :: inc ds
```

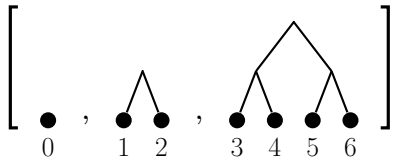


Рис. 9.5: Двоичный список с произвольным доступом, содержащий элементы 0...6.

Чтобы добавить новый элемент к началу списка, мы сначала преобразуем его в лист, а затем вставляем его в список деревьев с помощью вспомогательной функции consTree, которая следует образцу inc.

```
fun cons (x, ts) = consTree (Leaf x, ts)

fun consTree (t, []) = [One t]
  | consTree (t, Zero :: ts) = One t :: ts
  | consTree (t1, One t2 :: ts) =Zero :: consTree (link (t1, t2), ts)
```

Вспомогательная функция link порождает новое дерево из двух поддеревьев одинакового размера и автоматически вычисляет его размер.

Уничтожение элемента в двоичном списке с произвольным доступом (при помощи tail) аналогично уменьшению двоичного числа на единицу. Напомним функцию уменьшения для плотных двоичных чисел:

```
fun dec [One] = []
  | dec (One :: ds) = Zero :: ds
  | dec (Zero :: ds) = One :: dec ds
```

Соответствующая функция для списков деревьев называется unconsTree. Будучи примененной к списку, чья первая цифра имеет ранг r, unconsTree возвращает пару, состоящую из дерева ранга r и нового списка без этого дерева.

```
fun unconsTree [One t] = (t, [])
  | unconsTree (One t :: ts) = (t, Zero :: ts)
  | unconsTree (Zero :: ts) =
    let val (Node (_, t1, t2), ts') =unconsTree ts
    in (t1, One t2 :: ts') end
```

Функции head и tail удаляют самый левый элемент при помощи unconsTree, а затем, соответственно, либо возвращают этот элемент, либо отбрасывают.

```
fun head ts = let val (Leaf x, _) = unconsTree ts in x end
fun tail ts = let val (_, ts') = unconsTree ts in ts' end
```

Функции lookup и update не соответствуют никаким арифметическим операциям. Они просто пользуются организацией двоичных списков произвольного доступа в виде списков логарифмической длины, состоящих из деревьев логарифмической глубины. Поиск элемента состоит из двух этапов. Сначала в списке мы ищем нужное дерево, а затем в этом дереве ищем

Рис. 9.6: Двоичные списки с произвольным доступом.

требуемый элемент. Вспомогательная функция lookupTree использует поле размера в каждом узле, чтобы определить, находится ли i-й элемент в левом или правом поддереве.

```
fun lookup (i, Zero :: ts) = lookup (i, ts)
  | lookup (i, One t :: ts) =
      if i < size t then lookupTree (i, t) else lookup (i − size t, ts)

fun lookupTree (0, Leaf x) = x
  | lookupTree (i, Node (w, t1, t2)) =
      if i < w div 2 then lookupTree (i, t1
      else lookupTree (i − w div 2, t2)
```

update действует аналогично, но вдобавок копирует путь от корня до обновляемого листа.

```
fun update (i, y, Zero::ts) = Zero :: update (i, y, ts)
  | update (i, y, One t :: ts) =
      if i < size t then One (updateTree (i, y, t)) :: ts
      else One t :: update (i − size t, y, ts)

fun updateTree (0, y, Leaf x) = Leaf y
  | updateTree (i, y, Node (w, t1, t2)) =
      if i < w div 2 then Node (w, updateTree (i, y, t1), t2)
      else Node (w, t1, updateTree (i − w div 2, y, t2))
```

Полный код этой реализации приведен на Рис. 9.6.

Функции cons, head и tail производят не более $O(1)$ работы на цифру, так что общее время их работы $O(\log n)$ в худшем случае. lookup и update требуют не более $O(\log n)$ времени на поиск нужного дерева, а затем не более $O(\log n)$ времени на поиск нужного элемента в этом дереве, так что общее время их работы также $O(\log n)$ в худшем случае.

Упражнение 9.1 *Напишите функцию drop типа int* $\times$ α *RList* $\to$ α *RList, уничтожающую первые k элементов двоичного списка с произвольным доступом. Функция должна работать за время* $O(\log n)$.

Упражнение 9.2 *Напишите функцию create типа int* $\times$ $\alpha \to \alpha$ *RList, которая создает двоичный список с произвольным доступом, содержащий n копий некоторого значения x. Функция также должна работать за время* $O(\log n)$. *(Может оказаться полезным вспомнить Упражнение 2.5.)*

Упражнение 9.3 *Реализуйте BinaryRandomAccessList заново, используя разреженное представление*

```
datatype α Tree = Leaf of α | Node of int × α Tree × α Tree
type α RList = α Tree list
```

9.2.2 Безнулевые представления

В двоичных списках с произвольным доступом разочаровывает то, что списковые функции cons, head и tail требуют $O(\log n)$ времени вместо $O(1)$. В следующих трех подразделах мы исследуем варианты двоичных чисел, улучшающие время работы всех трех функций до $O(1)$. В этом подразделе мы начинаем с функции head.

Замечание 9.1 *Очевидное решение, позволяющее head выполняться за время $O(1)$ — хранить первый элемент отдельно от остального списка, подобно функтору ExplicitMin из Упражнения 3.7. Другое решение — использовать разреженное представление и либо биномиальные деревья, либо подвешенные деревья, так что головой списка будет корень первого дерева. Решение, которое мы исследуем в этом подразделе, хорошо тем, что оно также немного улучшает время работы lookup и update.*

Сейчас head у нас реализована через вызов unconsTree, которая выделяет первый элемент, а также перестраивает список без этого элемента. При таком подходе мы получаем компактный код, поскольку unconsTree поддерживает как head, так и tail, но теряется время на построение списков, не используемых функцией head. Ради большей эффективности имеет смысл реализовать head напрямую. В качестве особого случая, легко заставить head работать за время $O(1)$, когда первая цифра не ноль.

```
fun head (One (Leaf x) :: _) = x
```

Вдохновленные этим правилом, мы хотели бы устроить так, чтобы первая цифра *никогда* не была нулем. Есть множество простых трюков, достигающих именно этого, но более красивым решением будет использовать *безнулевое* (zeroless) представление, где ни одна цифра не равна нулю.

Безнулевые двоичные числа строятся из единиц и двоек, а не из единиц и нулей. Вес i-й цифры по-прежнему равен 2^i. Так, например, десятичное число 16 можно записать как `2111` вместо `00001`. Функция добавления единицы на безнулевых двоичных числах реализуется так:

```
datatype Digit = One | Two
type Nat = Digit list

fun inc [] = [One]
  | inc (One :: ds) = Two :: ds
  | inc (Two :: ds) = One :: inc ds
```

Упражнение 9.4 *Напишите функции уменьшения на единицу и сложения для безнулевых двоичных чисел. Заметим, что переноситься при сложении может как единица, так и двойка.*

Теперь если мы заменим тип цифр в двоичных списках с произвольным доступом на

```
datatype α Digit =One of α Tree |Two of α Tree × α Tree
```

то можем реализовать head как

```
fun head (One (Leaf x) :: _) = x
  | head (Two (Leaf x, Leaf y) :: _) = x
```

Ясно, что эта функция работает за время $O(1)$.

Упражнение 9.5 *Реализуйте оставшиеся функции для этого типа.*

Упражнение 9.6 *Покажите, что теперь функции lookup и update, примененные к элементу i, работают за время $O(\log i)$.*

Упражнение 9.7 *При некоторых дополнительных условиях красно-черные деревья (Раздел 3.3) можно рассматривать как числовое представление. Сопоставьте безнулевые двоичные списки с произвольным доступом и красно-черные деревья, в которых вставка разрешена только в самую левую позицию. Обратите особое внимание на функции cons и insert, а также на инварианты формы структур, порождаемых этими функциями.*

9.2.3 Ленивые представления

Допустим, мы представляем двоичные числа как потоки цифр, а не списки. Тогда функция увеличения на единицу получает вид

```
fun lazy inc ($Nil) =$Cons (One, $Nil)
       | inc ($Cons (Zero, ds)) =$Cons (One, ds)
       | inc ($Cons (One, ds)) =$Cons (Zero, inc ds)
```

Заметим, что функция эта пошаговая.

В Разделе 6.4.1 мы видели, как с помощью ленивого вычисления можно заставить вставку в биномиальные кучи работать за амортизированное время $O(1)$, так что нас не должно удивлять, что наша новая версия inc также работает за амортизированное время $O(1)$. Мы доказываем это по методу банкира.

Доказательство. Пусть каждая цифра ноль несет одну единицу долга, а цифра единица — ноль единиц долга. Допустим, ds начинается с k единиц (One), а затем имеет ноль (Zero). Тогда inc ds заменяет все эти One на Zero, а Zero на One. Выделим по одной единице долга на каждый из этих шагов. Теперь у каждого элемента Zero есть одна единица долга, а у One две: одна, унаследованная от исходной задержки в этом месте, и одна, созданная только что. Высвобождение этих двух единиц долга восстанавливает инвариант. Поскольку амортизированная стоимость функции равна ее нераздельной стоимости (здесь это $O(1)$) плюс число высвобождаемых единиц долга (здесь две), inc работает за амортизированное время $O(1)$.

Рассмотрим теперь функцию уменьшения.

```
fun lazy dec ($Cons (One, $Nil)) =$Nil
       | dec ($Cons (One, ds)) =$Cons (Zero, ds)
       | dec ($Cons (Zero, ds)) =$Cons (One, dec ds)
```

Поскольку эта функция подобна inc, но со сменой ролей цифр, можно ожидать, что при помощи подобного доказательства мы получим такое же ограничение. Так оно и есть, если мы не используем *обе* функции. Однако если используются как inc, так и dec, по крайней мере одной из них приходится приписывать амортизированное время $O(\log n)$. Чтобы понять, почему, представим последовательность увеличений и уменьшений, циклически переходящих от $2^k - 1$ к 2^k и обратно. Каждая операция при этом затрагивает каждую цифру, и общее время получается $O(\log n)$.

Но разве мы не доказали, что амортизированное время каждой из функций $O(1)$? Что здесь неверно? Проблема в том, что эти два доказательства требуют конфликтующих инвариантов долга. Чтобы доказать, что inc работает за амортизированное время $O(1)$, мы требовали, чтобы каждому Zero приписывалась одна единица долга, а каждому One ноль единиц. При доказательстве, что dec работает за амортизированное время $O(1)$, мы приписывали одну единицу долга каждому One и ноль единиц каждому Zero.

Главное свойство, которое как inc, так и dec по отдельности имеют, состоит в том, что по крайней мере половина операций, достигших какой-то позиции, на этой позиции останавливаются. А именно, каждый вызов inc или dec обрабатывает первую цифру, но только один вызов из двух затрагивает вторую. Третью цифру обрабатывает один вызов из четырех, и так далее. На интуитивном уровне, амортизированная стоимость каждой операции получается

$$O(1 + 1/2 + 1/4 + 1/8 + \ldots) = O(1)$$

Разделим возможные цифры-заполнители каждой позиции на *безопасные* (safe) и *опасные* (dangerous): функция, достигшая безопасной цифры, всегда на ней и завершается, а функция, добравшаяся до опасной цифры, может проследовать к следующей позиции. Чтобы доказать, что из двух последовательных операций никогда обе не добираются до следующей позиции, нам нужно показать, что каждый раз, когда операция обрабатывает опасную цифру, она заменяет её на безопасную. Тогда следующая операция, которая доберется до данной позиции, на ней и остановится. Формально мы доказываем, что каждая операция работает за амортизированное время $O(1)$, устанавливая инвариант долга, где каждой безопасной цифре приписывается одна единица долга, а опасной ноль.

Функция увеличения требует считать опасной самую большую цифру, а функция уменьшения считает опасной самую маленькую цифру. Чтобы поддержать их обе, нам нужна третья безопасная цифра. Таким образом, мы переключаемся на *избыточные* (redundant) двоичные числа, где каждая цифра может быть нулем, единицей или двойкой. Тогда inc и dec реализуются следующим образом:

```
datatype Digit = Zero | One | Two
type Nat = Digit Stream

fun lazy inc ($Nil) =$Cons (One, $Nil)
```

```
        | inc ($Cons (Zero, ds)) =$Cons (One, ds)
        | inc ($Cons (One, ds)) =$Cons (Two, ds)
        | inc ($Cons (Two, ds)) =$Cons (One, inc ds)

fun lazy dec ($Cons (One, $Nil) =$Nil
        | dec ($Cons (One, ds)) =$Cons (Zero, ds)
        | dec ($Cons (Two, ds)) =$Cons (One, ds)
        | dec ($Cons (Zero, ds)) =$Cons (One, dec ds)
```

Обратите внимание, что рекурсивные предложения в inc и dec — для Two (двойки) и Zero (ноля), соответственно — оба порождают One (единицу). При этом One — безопасная цифра, а Zero и Two — опасные. Чтобы увидеть, как нам здесь помогает избыточность, рассмотрим, как работает увеличение на единицу двоичного числа 222222, дающее 1111111. Для этой операции требуется семь шагов. Однако уменьшение этого значения не дает снова 222222, Вместо этого мы всего за один шаг получаем 0111111. Таким образом, чередование увеличений и уменьшений больше не является проблемой.

Ленивые двоичные числа могут служить моделью для построения многих других структур данных. В Главе 11 мы обобщим эту модель и получим метод проектирования под названием *неявное рекурсивное замедление* (implicit recursive slowdown).

Упражнение 9.8 *Докажите, что как inc, так и dec работают за амортизированное время $O(1)$ с помощью инварианта долга, присваивающего одну единицу долга цифре One и ноль цифрам Zero и Two.*

Упражнение 9.9 *Реализуйте cons, head и tail для списков с произвольным доступом на основе безнулевых избыточных двоичных чисел, используя тип*

```
datatype α Digit =
      One of α Tree
    | Two of α Tree × α Tree
    | Three of α Tree × α Tree × α Tree
type α RList =Digit Stream
```

Покажите, что все три функции работают за время $O(1)$.

Упражнение 9.10 *Как показано в Разделе 7.3 на биномиальных кучах с расписаниями, можно снабдить ленивые двоичные числа расписаниями и получить ограничение $O(1)$ в худшем случае. Заново реализуйте cons, head и tail из предыдущего упражнения, так, чтобы они работали за время $O(1)$ в худшем случае. Может оказаться полезным иметь два различных конструктора для цифры «два» (скажем, Two и Two'), чтобы различать рекурсивные и нерекурсивные варианты вызова cons и tail.*

9.2.4 Сегментированные представления

Ещё одна разновидность двоичных чисел, дающая показатели $O(1)$ в худшем случае — *сегментированные* (segmented) двоичные числа. Проблема

с обычными двоичными числами состоит в том, что переносы и занятия могут происходить каскадом. Например, увеличение $2^k - 1$ приводит в двоичной арифметике к k переносам. Аналогично, уменьшение 2^k ведет к k занятиям. Сегментированные двоичные числа решают эту проблему, позволяя нескольким переносам или занятиям выполняться за один шаг.

Заметим, что увеличение двоичного числа требует k шагов, когда число начинается с последовательности в k единиц. Подобным образом, уменьшение двоичного числа требует k шагов, когда число начинается с k нулей. Сегментированные двоичные числа объединяют непрерывные последовательности одинаковых цифр в блоки, так что мы можем применить перенос или занятие к целому блоку за один шаг. Мы представляем сегментированные двоичные числа как список чередующихся блоков из единиц и нулей согласно следующему объявлению типа:

```
datatype DigitBlock = Zeros of int | Ones of int
type Nat = DigitBlock list
```

Целое число в каждом DigitBlock представляет длину блока.

Мы добавляем новые блоки к началу списка блоков с помощью вспомогательных функций zeros (нули) и ones (единицы). Эти функции сливают идущие подряд блоки одинаковых цифр и отбрасывают пустые блоки. Кроме того, zeros отбрасывает нули в конце записи числа.

```
fun zeros (i, []) = []
  | zeros (0, blks) = blks
  | zeros (i, Zeros j :: blks) = Zeros (i+j) :: blks
  | zeros (i, blks) = Zeros i :: blks

fun ones (0, blks) = blks
  | ones (i, Ones j :: blks) = Ones (i+j) :: blks
  | ones (i, blks) = Ones i :: blks
```

Теперь при увеличении сегментированного двоичного числа мы смотрим на первый блок цифр (если он вообще есть). Если первый блок содержит нули, то мы заменяем первый из этих нулей на единицу, создавая новый единичный блок единиц, а длину блока нулей уменьшая на один. Если же первый блок содержит i единиц, то мы за один шаг проделываем i переносов, заменяя единицы на нули и увеличивая следующую цифру.

```
fun inc [] = [Ones 1]
  | inc (Zeros i :: blks) = ones (1, zeros (i-1, blks))
  | inc (Ones i :: blks) = Zeros i :: inc blks
```

В третьей строке функции мы знаем, что рекурсивный вызов inc не зациклится, поскольку если следующий блок присутствует, он будет содержать нули. Во второй строке вспомогательная функция позаботится об особом случае, когда первый блок содержит единственный ноль.

Уменьшение сегментированного двоичного числа выглядит почти точно так же, только роли единиц и нулей меняются.

```
fun dec (Ones i :: blks) = zeros (1, ones (i−1, blks))
  | dec (Zeros i :: blks) = Ones i :: dec blks
```

Здесь мы тоже знаем, что рекурсивный вызов не зациклится, потому что в следующем блоке должны быть единицы.

К сожалению, несмотря на то, что сегментированные двоичные числа поддерживают операции inc и dec за время $O(1)$ в худшем случае, числовые представления, основанные на них, оказываются слишком сложными, чтобы иметь какое-либо практическое значение. Проблема заключается в том, что понятие перевода целого блока единиц в нули и наоборот плохо переводится на язык операций с деревьями. Более практичные решения, однако, можно получить, если сочетать сегментацию с избыточными двоичными числами. При этом мы можем снова обрабатывать цифры (а следовательно, и деревья) по одной. Сегментация позволяет нам обрабатывать цифры в середине последовательности, а не только в начале.

Рассмотрим, например, избыточное представление, в котором блоки единиц рассматриваются как единый сегмент.

```
datatype Digits = Zero | Ones of int | Two
type Nat = Digits list
```

Определяем вспомогательную функцию ones, обрабатывающую блоки, идущие друг за другом, и уничтожающую пустые блоки.

```
fun ones (0, ds) = ds
  | ones (i, Ones j :: ds) = Ones (i+j) :: ds
  | ones (i, ds) = Ones i :: ds
```

Полезно рассматривать цифру Two (два) как незаконченный перенос. Чтобы не было каскадов переносов, нам надо гарантировать, что две двойки никогда не идут подряд. Будем поддерживать инвариант, что последняя не равная единице цифра перед каждой двойкой равна нулю. Этот инвариант можно записать как регулярное выражение $(0|1|01^*2)^*$ либо, если ещё учесть отсутствие нулей в конце, $(0^*1|0^+1^*2)^*$. Заметим, что двойка никогда не является первой цифрой. Таким образом, мы можем увеличить число на единицу за время $O(1)$ в худшем случае, просто увеличивая первую цифру.

```
fun simpleInc [] = [Ones 1]
  | simpleInc (Zero :: ds) = ones (1, ds)
  | simpleInc (Ones i :: ds) = Two :: ones (i−1, ds)
```

В третьей строке инвариант нарушается очевидным образом, поскольку Two оказывается в начале. Кроме этого, инвариант может быть нарушен во второй строке, если первая не равная единице цифра равна двойке. Мы восстанавливаем инвариант при помощи вспомогательной функции fixup, проверяющей, не является ли первая не равная единице цифра двойкой. Если это так, fixup заменяет двойку на ноль и увеличивает следующую цифру, которая, в свою очередь, двойкой быть не может.

```
fun fixup (Two :: ds) = Zero :: simpleInc ds
  | fixup (Ones i :: Two :: ds) = Ones i :: Zero :: fixup ds
  | fixup ds = ds
```

Во второй строке fixup мы пользуемся тем, что представление сегментировано, проскакивая блок единиц, за которыми следует двойка. Наконец, inc зовет сначала simpleInc, затем fixup.

```
fun inc ds = fixup (simpleInc ds)
```

Эта реализация может служить образцом для многих других структур данных. Такая структура представляет собой последовательность уровней, каждый из которых имеет признак *зелёный*, *жёлтый* или *красный*. Каждый уровень соответствует цифре в вышеописанной реализации. Зелёный соответствует нулю-Zero, жёлтый единице-One, а красный двойке-Two. Операция над любым объектом может перекрасить первый уровень из зеленого в жёлтый или из жёлтого в красный, но никогда из зелёного в красный. Инвариант состоит в том, что первый не-жёлтый уровень перед красным всегда зелёный. Процедура восстановления инварианта проверяет, не является ли первый не-жёлтый уровень красным и, если да, переводит этот уровень из красного в зелёный и, возможно, ухудшает цвет следующего уровня из зелёного в жёлтый или из жёлтого в красный. Последовательные жёлтые уровни собираются в блок, чтобы облегчить доступ к первому не-жёлтому. Каплан и Тарьян [KT95] называют эту общую методику *рекурсивное замедление* (recursive slowdown).

Упражнение 9.11 *Добавьте сегментацию к биномиальным кучам, чтобы операция insert работала за время $O(1)$ в худшем случае. Используйте тип*

```
datatype Tree = Node of Elem.T × Tree list
datatype Digit = Zero | Ones of Tree list | Two of Tree × Tree
type Heap = Digit list
```

Восстанавливайте инвариант после слияния куч, уничтожая все цифры Two.

Упражнение 9.12 *Пример реализации двоичных чисел на основе рекурсивного замедления поддерживает операцию inc за время $O(1)$ в худшем случае, но для dec может потребоваться $O(\log n)$. Реализуйте сегментированные избыточные двоичные числа, поддерживающие как inc, так и dec за время $O(1)$ в худшем случае, с цифрами `0`, `1`, `2`, `3` и `4`, причем `0` и `4` красные, `1` и `3` жёлтые, а `2` зелёная.*

Упражнение 9.13 *Реализуйте cons, head, tail и lookup для числового представления списков с произвольным доступом на основе системы счисления из предыдущего упражнения. Ваше представление должно поддерживать cons, head и tail за время $O(1)$ в худшем случае, а lookup за время $O(\log n)$ в худшем случае.*

9.3 Скошенные двоичные числа

При помощи ленивых двоичных чисел и сегментированных двоичных чисел мы получили два метода улучшения асимптотического поведения функций

увеличения на единицу и уменьшения на единицу с $O(\log n)$ до $O(1)$. В этом разделе мы рассмотрим третий метод; на практике он обычно приводит к более простым и быстрым программам, однако этот метод связан с более радикальным отходом от обыкновенных двоичных чисел.

В *скошенных двоичных числах* (skew binary numbers) [Mye83, Oka95b] вес i-й цифры w_i равен не 2^i, как в обыкновенных двоичных числах, а $2^i - 1$. Используются цифры ноль, один и два (т. е., $D_i = \{0, 1, 2\}$). Например, десятичное число 92 можно записать как `002101` (начиная с наименее значимой цифры).

Эта система счисления избыточна, однако мы можем вернуть уникальность представления, если введём дополнительное требование, что лишь самая младшая ненулевая цифра может быть двойкой. Будем говорить, что такое число записано в *каноническом виде* (canonical form). Начиная с этого момента, будем предполагать, что все скошенные двоичные числа записаны в каноническом виде.

Теорема 9.1 *(Майерс [Mye83]) Каждое натуральное число можно единственным образом записать в скошенном двоичном каноническом виде.*

Напомним, что вес i-й цифры равен $2^i - 1$, и заметим, что $1 + 2(2^{i+1} - 1) = 2^{i+2} - 1$. Отсюда следует, что мы можем добавить единицу к скошенному двоичному числу, чья младшая ненулевая цифра равна двойке, заменив эту двойку на ноль и увеличив следующую цифру с нуля до единицы или с единицы до двух. (Следующая цифра не может уже равняться двойке.) Увеличение на единицу скошенного двоичного числа, которое не содержит двойки, ещё проще — надо только увеличить младшую цифру с нуля до единицы или с единицы до двойки. В обоих случаях результат снова оказывается в каноническом виде. Если предположить, что мы можем найти младшую ненулевую цифру за время $O(1)$, в обоих случаях мы тратим не более $O(1)$ времени!

Мы не можем использовать для скошенных двоичных чисел плотное представление, потому что тогда поиск первой ненулевой цифры займет больше времени, чем $O(1)$. Поэтому мы выбираем разреженное представление и всегда имеем непосредственный доступ к младшей ненулевой цифре.

```
type Nat = int list
```

Целые числа представляют либо ранг, либо вес ненулевых цифр. Мы пока что используем веса. Веса хранятся в порядке возрастания, но два наименьших веса могут быть одинаковы, показывая, что младшая ненулевая цифра равна двум. При таком представлении мы реализуем inc следующим образом:

```
fun inc (ws as w1 :: w2 :: rest) =
      if w1 =w2 then (1 +w1 +w2) :: rest else 1 :: ws
  | inc ws = 1 :: ws
```

Первый вариант проверяет два первых веса на равенство, и либо сливает их в следующий больший вес (увеличивая таким образом следующую цифру),

либо добавляет новый вес 1 (увеличивая самую младшую цифру). Второй вариант обрабатывает случай, когда список ws пуст или содержит только один вес. Ясно, что эта процедура работает за время $O(1)$ в худшем случае.

Уменьшение скошенного двоичного числа на единицу столь же просто, как увеличение. Если младшая цифра не равна нулю, мы просто уменьшаем эту цифру с двух до единицы или с единицы до нуля. В противном случае мы уменьшаем самую младшую ненулевую цифру, а предыдущий ноль заменяем двойкой. Это реализуется так:

```
fun dec (1 :: ws) = ws
  | dec (w :: ws) = (w div 2) :: (w div 2) :: ws
```

Во второй строке нужно учитывать, что если $w = 2^{k+1}-1$, то $\lfloor w/2 \rfloor = 2^k - 1$. Ясно, что dec также работает за время $O(1)$ в худшем случае.

9.3.1 Скошенные двоичные списки с произвольным доступом

Теперь мы разработаем числовое представление для списков с произвольным доступом на основе скошенных двоичных чисел. Основа представления данных — список деревьев, одно дерево для каждой единицы и два дерева для каждой двойки. Деревья хранятся в порядке возрастания размера, но если младшая ненулевая цифра двойка, то два первых дерева будут одинакового размера.

Размеры деревьев соответствуют весам цифр в скошенных двоичных числах, так что дерево, представляющее i-ю цифру, имеет размер $2^{i+1} - 1$. До сих пор мы в основном рассматривали деревья размером степень двойки, но встречались и деревья нужного нам сейчас размера: полные двоичные деревья. Таким образом, мы представляем скошенные двоичные списки с произвольным доступом в виде списков полных двоичных деревьев.

Чтобы эффективно поддержать операцию head, мы должны сделать первый элемент списка с произвольным доступом вершиной первого дерева, так что элементы внутри каждого дерева мы будем хранить в предпорядке слева направо; элементы каждого дерева предшествуют элементам следующего дерева.

В предыдущих примерах мы хранили в каждой вершине её размер или ранг, даже когда эта информация была избыточна. В этом примере мы используем более реалистичный подход и храним размер только вместе с вершиной каждого дерева, а не для всех поддеревьев. Следовательно, тип данных для скошенных двоичных списков с произвольным доступом получается

```
datatype α Tree =Leaf of α |Node of α × α Tree × α Tree
type α RList =(int × α Tree) list
```

Теперь можно определить cons по аналогии с inc.

```
fun cons (x, ts as (w1, t1) :: (w2, t2) :: rest) =
      if w1 =w2 then (1+w1+w2, Node (x, t1, t2) :: rest)
```

Рис. 9.7: Скошенные двоичные списки с произвольным доступом.

```
      else (1, Leaf x) :: ts
  | cons (x, ts) = (1, Leaf x) :: ts
```

Функции head и tail работают с корнем первого дерева. tail возвращает дочерние узлы этого дерева (если они есть) обратно в начало списка, где они будут представлять новую цифру-двойку.

```
fun head ((1, Leaf x) :: ts) = x
  | head ((w, Node (x, t1, t2)) :: ts) =x
fun tail ((1, Leaf x) :: ts) = ts
  | tail ((w, Node (x, t1, t2)) :: ts) =(w div 2, t1) :: (w div 2, t2) :: ts
```

Чтобы найти элемент, мы сначала ищем нужное дерево, а затем нужный элемент в этом дереве. При поиске внутри дерева мы отслеживаем размер текущего дерева.

```
fun lookup (i, (w, t) :: ts) =
      if i < w then lookupTree (w, i ,t)
      else lookup (i–w, ts)

fun lookupTree (1, 0, Leaf x) = x
  | lookupTree (w, 0, Node (x, t1, t2)) =x
  | lookupTree (w, i, Node (x, t1, t2)) =
      if i < w div 2 then lookupTree (w div 2, i−1, t1)
      else lookupTree (w div 2, i − 1 − w div 2, t2)
```

Заметим, что в предпоследней строке мы отнимаем единицу от i, поскольку перескакиваем через x. В последней строке мы отнимаем $1 + \lfloor w/2 \rfloor$ от i, поскольку перескакиваем через x и через все элементы t_1. Функции update и updateTree определяются подобным же образом. Они приведены на Рис. 9.7 наряду со всеми остальными деталями реализации.

Нетрудно убедиться, что cons, head и tail работают за время $O(1)$ в худшем случае. Подобно двоичным спискам с произвольным доступом, скошенные двоичные списки с произвольным доступом представляют собой списки логарифмической длины, состоящие из деревьев логарифмической глубины, так что lookup и update работают за время $O(\log n)$ в худшем случае. На самом деле каждый неудачный шаг lookup или update отбрасывает по крайней мере один элемент, так что можно немного улучшить оценку до $O(\min(i, \log n))$.

Указание разработчикам 9.1 *Скошенные двоичные списки с произвольным доступом являются хорошим выбором для приложений, активно использующих как спископодобные, так и массивоподобные функции в списках с произвольным доступом. Существуют более производительные реализации списков и более производительные реализации (устойчивых) массивов, но ни одна реализация не превосходит нашу в обеих классах функций [Oka95b].*

Упражнение 9.14 *Перепишите структуру HoodMelvilleQueue из Раздела 8.2.1, чтобы она вместо обычных списков использовала скошенные двоичные списки с произвольным доступом. Реализуйте на получившейся структуре операции lookup и update.*

9.3.2 Скошенные биномиальные кучи

Наконец, рассмотрим гибридное числовое представление для куч, основанное как на скошенных двоичных числах, так и на обыкновенных двоичных числах. Реализация скошенного двоичного числа проста и быстра, и отлично подходит как образец для функции insert. К сожалению, сложение двух скошенных двоичных чисел весьма неудобно. Поэтому функцию merge мы порождаем на основе сложения обыкновенных двоичных чисел, а не сложения скошенных чисел.

Скошенное биномиальное дерево (skew binomial tree) представляет собой биномиальное дерево, в котором к каждому узлу приписан список длиной до r элементов, где r — ранг рассматриваемого узла.

```
datatype Tree = Node of int × Elem.T × Elem.T list × Tree list
```

В отличие от обыкновенных биномиальных деревьев, размер скошенного биномиального дерева не полностью определяется его рангом; ранг определяет диапазон возможных размеров.

Лемма 9.2 *Если t — скошенное биномиальное дерево ранга r, то $2^r \leq |t| \leq 2^{r+1} - 1$*

Упражнение 9.15 *Докажите Лемму 9.2*

Над скошенными биномиальными деревьями можно производить операцию *связывания* (linking) и *скошенного связывания* (skew linking). Функция связывания link сочетает два дерева ранга r и получает одно дерево ранга $r + 1$, делая дерево с большим корнем ребенком дерева с меньшим корнем.

```
fun link (t1 as Node (r, x1, xs1, c1), t2 as Node (_, x2, xs2, c2)) =
      if Elem.leq (x1, x2) then Node (r+1, x1, xs1, t2 :: c1)
      else Node (r+1, x2, xs2, t1 :: c2)
```

Функция скошенного связывания skewLink сочетает два дерева ранга r и дополнительный элемент, получая дерево ранга $r + 1$. Сначала она связывает два дерева, а затем сравнивает корень получившегося дерева с дополнительным элементом. Меньший из этих двух элементов становится корнем, а больший добавляется к дополнительному списку элементов.

```
fun skewLink (x, t1, t2) =
      let val Node (r, y, ys, c) = link (t1, t2)
      in
          if Elem.leq (x, y) then Node (r, x, y :: ys, c)
          else Node (r, y, x :: ys, c)
      end
```

Скошенная биномиальная куча представляет собой список скошенных биномиальных деревьев, упорядоченных в порядке кучи, отсортированных по возрастанию ранга, и только два первых дерева могут иметь одинаковый ранг. Поскольку скошенные биномиальные деревья одного ранга могут иметь различный размер, здесь уже нет прямого соответствия между деревьями в куче и цифрами скошенного двоичного числа. представляющего размер кучи. Например, хотя скошенное двоичное представление числа 4 равно `11`, скошенная биномиальная куча размера 4 может содержать либо одно дерево ранга 2 размера 4, либо два дерева ранга 1 размером 2, либо дерево ранга 1 размером 3 и дерево ранга 0, либо дерево ранга 1 размером 2 и два дерева ранга 0. Однако максимальное число деревьев в куче по-прежнему равно $O(\log n)$.

Большое преимущество скошенных биномиальных куч состоит в том, что новый элемент вставляется за время $O(1)$. Сначала мы сравниваем ранги двух наименьших деревьев. Если они совпадают, мы производим скошенное связывание нового элемента с этими деревьями. В противном случае мы создаем новое одноэлементное дерево и добавляем его к началу списка.

```
fun insert (x, ts as t1 :: t2 :: rest) =
      if rank t1 =rank t2 then skewLink (x, t1, t2) :: rest
      else Node (0, x, [], []) :: ts
  | insert (x, ts) = Node (0, x, [], []) :: ts
```

Оставшиеся функции почти такие же, как соответствующие функции обыкновенных биномиальных куч. Мы изменяем имя старой функции merge на mergeTrees. Она по-прежнему проходит оба списка деревьев, проводя связывание (не скошенное связывание!) каждый раз, когда видит два дерева одного ранга. Поскольку и mergeTrees, и её вспомогательная функция insTree ожидают списки деревьев строго возрастающего ранга, функция merge нормализует оба своих аргумента, убирая дубликаты из начала списков, прежде чем позвать mergeTrees.

```
fun normalize [] = []
  | normalize (t :: ts) = insTree (t, ts)
fun merge (ts1, ts2) =mergeTrees (normalize ts1, normalize ts2)
```

На функции findMin и removeMinTree переключение на скошенные биномиальные кучи никак не влияет, поскольку обе эти функции не заботятся о рангах, рассматривая только корень каждого дерева. Функция deleteMin изменяется лишь ненамного. Как и раньше, изымается дерево с минимальным корнем, список его детей обращается, и обращенный список детей сливается с оставшимися деревьями. Однако затем заново вставляются элементы дополнительного списка, прикрепленного к уничтоженному корню.

```
fun deleteMin ts =
      let val (Node (_, x, xs, ts1), ts2) =removeMinTree ts
          fun insertAll ([], ts) = ts
            | insertAll (x :: xs, ts) = insertAll (xs, insert (x,
            ts))
      in insertAll (xs, merge (rev ts1, ts2)) end
```

Рис. 9.8: Скошенные биномиальные кучи.

На Рис. 9.8 приведена полная реализация скошенных биномиальных куч.

Функция insert работает за время $O(1)$ в худшем случае, а merge, findMin и deleteMin работают за то же время, что и соответствующие функции для обыкновенных биномиальных куч, то есть, за $O(\log n)$ в худшем случае. Заметим, что каждая из различных фаз функции deleteMin — поиск дерева с минимальным корнем, обращение его детей, слияние детей с оставшимися деревьями и вставка дополнительных элементов, — занимает по $O(\log n)$.

Если нужно, мы можем улучшить время работы findMin до $O(1)$ при помощи функтора ExplicitMin из Упражнения 3.7. В Разделе 10.2.2 мы увидим, как улучшить также и время операции merge до $O(1)$.

Упражнение 9.16 *Допустим, нам нужна функция delete типа* $Elem.T \times Heap \to Heap$. *Напишите функтор, берущий реализацию куч H и порождающий реализацию куч, поддерживающую наряду с обычными операциями над кучей функцию delete. Используйте тип*

type $Heap = H.Heap \times H.Heap$

где одна из элементарных куч представляет положительные вхождения элементов, а вторая — отрицательные вхождения. Отрицательное вхождение элемента в кучу означает, что этот элемент был уже уничтожен, но физически ещё не удален из кучи. Положительные и отрицательные вхождения одного и того же элемента взаимоуничтожаются и физически удаляются из кучи, когда оба оказываются минимальными элементами своих куч. Поддерживайте инвариант, что минимальный элемент положительной кучи строго меньше, чем минимальный элемент отрицательной кучи. (У этой реализации есть забавное свойство: элемент можно уничтожить прежде, чем он вставлен в кучу; для многих приложений это свойство безвредно.)

9.4 Троичные и четверичные числа

В информатике мы настолько привыкли работать с двоичными числами, что иногда забываем о существовании других оснований. В этом разделе мы рассмотрим использование арифметики по основанию 3 и 4 в числовых представлениях.

Вес каждой цифры при основании k равен k^r, так что нам нужны семейства деревьев, имеющих такие размеры. Можно построить обобщения для каждого из семейств деревьев, используемых в двоичных числовых представлениях:

Определение 9.4 ***(Полные k-ичные листовые деревья)*** *Полное k-ичное дерево (complete k-ary tree) ранга 0 представляет собой лист, а полное k-*

ичное дерево ранга $r > 0$ представляет собой узел с k поддеревьями, каждое из которых является полным k-ичным деревом ранга $r - 1$. Полное k-ичное дерево ранга r содержит $(k^{r+1} - 1)/(k - 1)$ узлов и k^r листьев. Полное k-ичное листовое дерево — это полное k-ичное дерево, где элементы содержатся только в листьях.

Определение 9.5 ***(k-номиальные деревья)*** *k-номиальное дерево (k-nomial tree) ранга r представляет собой узел, у которого есть для каждого ранга q от $r - 1$ до 0 по $k - 1$ поддерева, имеющих ранг q. Иначе выражаясь, k-номиальное дерево ранга $r > 0$ представляет собой k-номиальное дерево ранга $r - 1$, к которому в качестве левых поддеревьев присоединены ещё $k - 1$ k-номиальных дерева ранга $r - 1$. Из второго определения легко увидеть, что k-номиальное дерево ранга r содержит k^r узлов.*

Определение 9.6 ***(k-ичные подвешенные деревья)*** *k-ичное подвешенное дерево (k-ary pennant) ранга 0 представляет собой единственную вершину, а k-ичное подвешенное дерево ранга $r > 0$ представляет собой вершину с $k-1$ поддеревьями, каждое из которых является полным k-ичным деревом ранга $r-1$. Каждое из этих поддеревьев содержит $(k^r-1)/(k-1)$ узлов, так что всё дерево целиком содержит k^r узлов.*

Преимущество при использовании оснований больше двойки заключается в том, что для представления каждого числа требуется меньше цифр. В то время как число по основанию 2 содержит примерно $\log_2 n$ цифр, число по основанию k содержит приблизительно $\log_k n = \log_2 n / \log_2 k$ цифр. Например, при основании 4 нужно примерно вдвое меньше цифр, чем при основании 2. С другой стороны, теперь для каждой цифры имеется больше возможных значений, так что обработка каждой цифры может отнимать больше времени. В числовых представлениях обработка одной цифры по основанию k часто требует примерно $k + 1$ шагов, так что операция, затрагивающая каждую цифру, должна отнимать примерно $(k + 1) \log_k n = \frac{k+1}{\log_2 k} \log n$ шагов. В следующей таблице приведены значения $(k + 1)/\log_2 k$ для $k = 2, \ldots, 8$.

k	2	3	4	5	6	7	8
$(k + 1)/\log_2 k$	3.00	2.52	2.50	2.58	2.71	2.85	3.0

По этой таблице можно заключить, что числовые представления, основанные на троичных или четверичных числах, могут выигрывать до 16% у числовых представлений на основе двоичных чисел. Другие факторы, например, размер кода, часто делают большие основания менее эффективными при увеличении k, так что настолько большие ускорения редко встречаются на практике. Более того, троичные и четверичные представления на маленьких объемах данных часто работают хуже, чем двоичные представления. Однако для больших объемов данных троичные и четверичные представления часто приносят ускорение от 5 до 10%.

Упражнение 9.17 *Реализуйте триномиальные кучи, используя тип*

```
datatype Tree = Node of Elem.T × (Tree × Tree) list
datatype Digit = Zero | One of Tree | Two of Tree × Tree
type Heap = Digit list
```

Упражнение 9.18 *Реализуйте безнулевые четверичные списки с произвольным доступом на основе типа*

```
datatype α Tree =Leaf of α |Node of α Tree vector.
datatype α RList =α Tree vector list
```

где каждый вектор в Node содержит по четыре дерева, а каждый вектор в списке содержит от одного до четырёх деревьев.

Упражнение 9.19 *Можно также приспособить к произвольному основанию понятие скошенного двоичного числа. В скошенных k-ичных числах i-я цифра имеет вес $(k^{i+1}-1)/(k-1)$. Цифры выбираются из набора $\{0,\ldots,k-1\}$, плюс младшая ненулевая цифра может равняться k. Реализуйте скошенные троичные списки с произвольным доступом на основе типа*

```
datatype α Tree =Leaf of α |Node of α × α Tree × α Tree × α Tree
type α RList =(int × α Tree) list
```

9.5 Примечания

Структуры данных, которые можно описать как числовые представления, встречаются на удивление часто, но явным образом связь с каким-либо вариантом системы счисления упоминают лишь изредка [GMPR77, Mye83, CMP88, KT96b]. Скошенные списки с произвольным доступом впервые появились в [Oka96b]. Скошенные биномиальные кучи описаны в [BO96].

Глава 10

Развёртка структур данных

Английское слово *bootstrapping** означает «процесс приподнимания самого себя за шнурки ботинок». Этот, казалось бы, бессмысленный образ описывает распространённую в информатике ситуацию, когда, чтобы решить задачу, нам нужно иметь готовое решение той же самой задачи (в какой-то её более простой разновидности).

Рассмотрим, например, процесс загрузки операционной системы в компьютер с диска или магнитной ленты. Без операционной системы компьютер не может даже обратиться к диску или ленте! Решением будет *начальный загрузчик* (bootstrap loader) — крошечная, неполная операционная система, чьей единственной целью является чтение немного более крупной и мощной операционной системы и передача ей управления. Та, в свою очередь, считывает настоящую операционную систему и передает управление уже ей. Можно рассматривать эту картину как пример развёртки полного решения на основе неполного.

Ещё одним примером может служить развёртка компилятора. Часто бывает, что компилятор с нового языка пишется на самом этом языке. Но как тогда скомпилировать этот компилятор? Одно из возможных решений — написать простой, неэффективный интерпретатор нового языка на каком-либо старом существующем языке. С помощью этого интерпретатора компилятор применяется к своему собственному коду, и получается эффективный компилятор в виде скомпилированного кода. Можно рассматривать это как пример развёртки эффективного решения на основе неэффективного.

Адам Бухсбаум в своей диссертации [Buc93] описывает две методики разработки алгоритмов, которые он вместе называет *развёртка структур данных* (data-structural bootstrapping). Первая методика, *структурная декомпозиция* (structural decomposition), предназначена для развёртки неполных структур данных, чтобы получить полные. Вторая методика, *структурная абстракция* (structural abstraction), используется для развёртки

*Я перевожу это слово как «развёртка». Каламбур при этом, к сожалению, теряется. — *прим. перев.*

неэффективных структур данных в эффективные. В этой главе мы заново рассматриваем обе этих методики и добавляем к ним третью, позволяющую развёртывать структуры данных с простейшими элементами и получать структуры с составными элементами.

10.1 Структурная декомпозиция

Структурная декомпозиция (structural decomposition) — это метод для получения полных структур данных на основе неполных. Как правило, берется реализация, способная работать только с объектами ограниченного размера (может быть, даже только нулевого размера), и расширяется так, чтобы работать с объектами неограниченного размера.

Рассмотрим типичные рекурсивные типы данных, например, списки и двоичные листовые деревья.

```
datatype α List =Nil | Cons of α × α List
datatype α Tree =Leaf of α |Node of α Tree × α Tree
```

При желании их можно рассматривать как примеры структурной декомпозиции. И то, и другое определение состоит из простой реализации для объектов ограниченного размера (ноль для списков и один для деревьев) плюс правило для рекурсивной декомпозиции более крупных объектов в более мелкие, пока, в конце концов, все объекты не окажутся достаточно маленькими, чтобы с ними справилось базовое правило.

Однако оба этих определения просты ещё и в том, что рекурсивная компонента каждого определения совпадает с определяемым типом. Например, рекурсивная компонента определения α List также является α List. Такой тип называется *гомогенно рекурсивным* (uniformly recursive).

Как правило, мы используем термин *структурная декомпозиция* для описания структур данных, которые являются *гетерогенными* (non-uniform). Рассмотрим, например, такое определение последовательностей:

```
datatype α Seq =Nil' | Cons' of α × (α × α) Seq
```

Здесь последовательность может быть либо пустой, либо состоять из элемента и последовательности пар элементов. Рекурсивная компонента $(\alpha \times \alpha)$ Seq отличается от α Seq, так что этот тип гетерогенен.

Почему мы можем предпочитать гетерогенное определение гомогенному? Часто гетерогенные типы за счёт своей более изощренной структуры поддерживают более эффективные алгоритмы, чем их гомогенные аналоги. Сравним, например, следующие функции определения размера для списков и последовательностей:

```
fun sizeL Nil = 0
  | sizeL (Cons (x, xs)) = 1 + sizeL xs

fun sizeS Nil' = 0
  | sizeS (Cons' (x, ps)) = 1 + 2 * sizeS ps
```

Функция для списков требует время $O(n)$, в то время как функция для последовательностей требует $O(\log n)$.

10.1.1 Гетерогенная рекурсия и Стандартный ML

К сожалению, как правило, мы не можем выразить структурную декомпозицию напрямую на Стандартном ML. Несмотря на то, что Стандартный ML позволяет определять гетерогенные рекурсивные типы данных, система типов запрещает большинство интересных функций на основе этих типов. Рассмотрим, например, функцию sizeS для последовательностей. Эта функция будет отвергнута компилятором Стандартного ML, поскольку система типов требует, чтобы все рекурсивные вызовы в теле рекурсивной функции имели тот же тип, что и объемлющая функция (т. е., рекурсивные определения функций должны быть гомогенны). Функция sizeS нарушает это ограничение, поскольку внешнее вхождение sizeS имеет тип α Seq $\to$ int, а внутреннее — тип $(\alpha \times \alpha)$ Seq $\to$ int.

Всегда можно преобразовать гетерогенный тип в гомогенный, введя новый тип данных, который сливает различные употребления в один тип. Например, сливая элементы и пары элементов, можно переписать тип Seq в виде

datatype α EP =Elem **of** α |Pair **of** α EP $\times$ α EP
datatype α Seq =Nil' | Cons' **of** α EP $\times$ α Seq

В таком случае sizeS оказывается совершенно законной в том виде, как она исходно была записана; как внешний вызов sizeS, так и внутренний имеют тип α Seq $\to$ int.

Поскольку гетерогенный тип всегда можно преобразовать в гомогенный, термин «структурная декомпозиция» относится скорее к способу нашего рассуждения о типе данных, чем к его реализации. Рассмотрим, например, модифицированное определение типа Seq, приведенное выше. Тип α Seq изоморфен двоичным листовым деревьям, так что модифицированная версия α Seq эквивалентна α Tree list. Однако, как правило, мы думаем о списке деревьев иначе, чем о последовательности пар — некоторые алгоритмы кажутся проще или естественнее в одном представлении, другие в другом. В следующем разделе мы увидим несколько примеров.

Есть также несколько практических соображений, заставляющих нас предпочитать гетерогенное определение α Seq гомогенному. Во-первых, оно короче; имеется один тип, а не два, и незачем всюду вручную расставлять конструкторы Elem и Pair. Во-вторых, в некоторых реализациях языка гетерогенное определение может быть эффективнее: нет необходимости проводить сопоставление с конструкторами Elem и Pair, и незачем во время выполнения строить в памяти представления этих конструкторов. В-третьих, и это самое важное соображение, гетерогенное определение позволяет системе типов отлавливать намного больше программистских ошибок. Тип в гетерогенном определении обеспечивает инвариант, что внешний конструктор Cons' содержит один элемент, второй пару элементов, третий пару пар,

и так далее. Тип гомогенного определения не гарантирует ни баланса в парах, ни увеличения глубины пар по одному на уровень. Эти ограничения должны обеспечиваться программистом как инварианты системы. Но если программист ненамеренно нарушит эти инварианты — например, использовав элемент там, где ожидается пара, — система типов не поможет ему поймать эту ошибку.

Исходя из этих соображений, мы часто представляем код так, как если бы Стандартный ML поддерживал гетерогенные рекурсивные определения функций, известные также как *полиморфная рекурсия* (polymorphic recursion) [Myc84]. Наш код будет невозможно напрямую выполнить, но он будет более читаемым. Его всегда можно преобразовать обратно в законный Стандартный ML, используя приёмы, описанные пару абзацев назад.

10.1.2 Снова двоичные списки с произвольным доступом

При всех своих достоинствах, обсуждаемый нами тип α Seq бесполезен для представления последовательностей. Проблема в том, что он может представлять только последовательности длиной $2^k - 1$. Если использовать терминологию числовых представлений, конструктор Cons' позволяет нам записывать биты-единицы, но не биты-нули. Это легко исправить, добавив в тип ещё один конструктор. Кроме того, мы переименовываем конструктор Cons', чтобы подчеркнуть аналогию с двоичными числами.

```
datatype α Seq =Nil | Zero of (α × α) Seq |One of α × (α × α) Seq
```

Теперь последовательность $0 \ldots 10$ можно представить как

```
One (0, One ((1,2), Zero (One ((((3,4),(5,6)),((7,8),(9,10))), Nil))))
```

Размер этой последовательности 11, что в двоичном виде записывается как `1101`.

Пары в этом типе всегда сбалансированы. В сущности, можно думать о парах элементов или о парах пар элементов и т. д. как о полных двоичных листовых деревьях. Таким образом, наш тип, по существу, эквивалентен типу двоичных списков с произвольным доступом из Раздела 9.2.1, но только с явно представленными инвариантами.

Давайте заново реализуем функции двоичных списков с произвольным доступом, на этот раз рассуждая в терминах элементов и последовательностей пар, а не в терминах списков полных двоичных листовых деревьев. Функции по-прежнему будут работать за время $O(\log n)$, однако, как мы сейчас увидим, новый способ мышления, как правило, даёт нам более короткие и понятные алгоритмы.

Начнём с функции cons. Первые два варианта не представляют трудности:

```
fun cons (x, Nil) = One (x, Nil)
  | cons (x, Zero ps) = One (x, ps)
```

Чтобы добавить элемент к последовательности, имеющей вид One (y, ps), мы строим пару из нового и существующего элементов, и добавляем её в последовательность пар.

```
fun cons (x, One (y, ps)) = Zero (cons ((x, y), ps))
```

Здесь требуется полиморфная рекурсия: внешний cons имеет тип

$$\alpha \times \alpha \text{ Seq} \to \alpha \text{ Seq}$$

а внутренний cons имеет тип

$$(\alpha \times \alpha) \times (\alpha \times \alpha) \text{ Seq} \to (\alpha \times \alpha) \text{ Seq}$$

Мы реализуем функции head и tail через вспомогательную функцию uncons, разбивающую последовательность на первый элемент и последовательность остальных элементов.

```
fun head xs = let val (x, _) = uncons xs in x end
fun tail xs = let val (_, xs') = uncons xs' in xs' end
```

Функция uncons получается путём прочтения всех строчек cons справа налево.

```
fun uncons (One (x, Nil)) = (x, Nil)
  | uncons (One (x, ps)) = (x, Zero ps)
  | uncons (Zero ps) = let val ((x, y), ps') = uncons ps
                       in (x, One (y, ps')) end
```

Рассмотрим теперь функцию lookup. Получив последовательность One (x, ps), мы либо возвращаем x, либо переадресуем запрос к Zero ps.

```
fun lookup (0, One (x, ps)) = x
  | lookup (i, One (x, ps)) = lookup (i-1, Zero ps)
```

Чтобы найти элемент по индексу i в списке пар, мы находим пару по индексу $\lfloor i/2 \rfloor$, а затем извлекаем нужный элемент из этой пары.

```
fun lookup (i, Zero ps) = let val (x, y) = lookup (i div 2, ps)
                          in if i mod 2 = 0 then x else y end
```

Наконец, рассмотрим функцию update. Варианты для конструктора One выглядят просто:

```
fun update (0, e, One (x, ps)) = One (e, ps)
  | update (i, e, One (x, ps)) = cons (x, update (i-1, e, Zero ps))
```

Однако пытаясь обновить элемент в последовательности пар, мы сталкиваемся с небольшой проблемой. Нам нужно обновить пару по индексу $\lfloor i/2 \rfloor$, но, чтобы создать новую пару, требуется второй элемент старой пары. Поэтому прежде, чем рекурсивно вызвать update, нам приходится звать lookup.

```
fun update (i, e, Zero ps) =
let val (x, y) = lookup (i div 2, ps)
    val p = if i mod 2 = 0 then (e, y) else (x, e)
in Zero (update (i-1, p, ps)) end
```

Рис. 10.1: Альтернативная реализация двоичных списков с произвольным доступом.

Упражнение 10.1 *Докажите, что эта версия update работает за время* $O(\log^2 n)$.

Чтобы восстановить ограничение $O(\log n)$ для функции update, надо избавиться от вызова lookup. Но как тогда мы получим второй элемент, который нам нужен для построения новой пары? Если мы не можем привести Магомета к горе, придётся вести гору к Магомету. А именно, вместо того, чтобы получать старую пару и локально конструировать новую, мы строим функцию для построения новой пары из старой, когда эта старая пара будет найдена. Используем вспомогательную функцию fupdate, принимающую в качестве аргумента функцию, которую требуется применить к i-му элементу последовательности. В этом случае update выглядит просто как

```
fun update (i, y, xs) = fupdate(fn x ⇒ y, i, xs)
```

Ключевым шагом в fupdate будет преобразование функции f на элементах в функцию f', принимающую пару элементов, и, в зависимости от чётности i, применяющую f либо к первому, либо ко второму элементу пары.

```
fun f' (x, y) = if i mod 2 = 0 then (f x, y) else (x, f y)
```

Имея это определение, уже нетрудно написать оставшуюся часть fupdate.

```
fun fupdate (f, 0, One (x, ps)) = One (f x, ps)
  | fupdate (f, i, One (x, ps)) = cons (x, fupdate (f, i−1, Zero ps))
  | fupdate (f, i, Zero ps) =
      let fun f' (x, y) = if i mod 2 = 0 then (f x, y) else (x, f y)
      in Zero (fupdate (f', i div 2, ps)) end
```

Полная реализация приведена на Рис. 10.1.

При сравнении Рис. 10.1 и Рис. 9.6 мы видим, что новая реализация намного короче и что все функции заметно проще; может быть, за исключением update. (А если мы не боимся функций высших порядков, то даже update выглядит проще.) Все эти преимущества получены потому, что мы представили структуру данных в виде гетерогенного типа, прямо отражающего искомые инварианты.

Упражнение 10.2 *Модифицируйте AltBinaryRandomAccessList, чтобы cons, head и tail работали за амортизированное время* $O(1)$*, используя тип*

```
datatype α RList =
    Nil
  | One of α × (α × α) RList susp
  | Two of α × α × (α × α) RList susp
  | Three of α × α × α × (α × α) RList susp
```

10.1.3 Развёрнутые очереди

Рассмотрим использование ++ в очередях по методу банкира из Раздела 6.3.2. Во время проворота очереди головной поток f заменяется потоком f ++ reverse r. После последовательности проворотов головной поток имеет вид

$$((\mathrm{f} \mathbin{+\!\!+} \mathrm{reverse}\ \mathrm{r}_1) \mathbin{+\!\!+} \mathrm{reverse}\ \mathrm{r}_2) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathrm{reverse}\ \mathrm{r}_k$$

Хорошо известно, что append в таких левоассоциативных контекстах неэффективен, поскольку он многократно обрабатывает элементы потоков, расположенных слева. Например, в этом случае элементы f будут обработаны k раз (по разу на каждое вхождение ++), а элементы r_i будут обработаны $k - i + 1$ раз (один раз при выполнении reverse, а затем по разу на каждое ++ справа). В общем случае левоассоциативная конкатенация легко ведет к квадратичному поведению. К счастью, в нашем случае общая стоимость всех конкатенаций по-прежнему линейна, поскольку каждый r_i по меньшей мере вдвое длиннее предыдущего. Однако повторная обработка иногда на практике делает эти очереди слишком медленными. Сейчас мы исправим этот недостаток с помощью структурной декомпозиции.

Считая, что передний поток имеет вышеописанную структуру, мы разбиваем его на две части: f и коллекцию $m = \{\mathrm{reverse}\ \mathrm{r}_1, \ldots, \mathrm{reverse}\ \mathrm{r}_k\}$. Мы можем теперь представлять f в виде списка, а каждый из reverse r_i как задержанный список. Кроме того, хвостовой поток r мы тоже заменяем на список. Эти преобразования уничтожают подавляющее большинство задержек, и нам удается избежать почти всех расходов, связанных с ленивым вычислением. Но как нам представить коллекцию m? Как мы увидим, доступ к этой коллекции происходит по правилу FIFO, так что, используя структурную декомпозицию, мы можем её представить как очередь задержанных списков. Как и в любом рекурсивном типе, нам нужно основание рекурсии, так что пустые очереди мы представляем через особый конструктор.[1] Следовательно, наше новое представление будет

```
datatype α Queue =
    E | Q of int × α list × α list susp Queue × int × α list
```

Первое целое число, lenfm, представляет собой совокупную длину f и всех задержанных списков в m (т. е., это величина, которая в старом представлении называлась просто lenf). Второе целое число, lenr — это, как обычно, просто длина r. Обычный наш инвариант баланса принимает вид lenr $\leq$ lenfm. Кроме того, мы требуем, чтобы список f был непустой. (В старом представлении f мог быть пуст, если пуста была вся очередь, но теперь мы этот случай представляем отдельно.)

Как обычно, функции, работающие с очередью, написать нетрудно.

```
fun snoc (E, x) = Q (`, [x], E, 0, [])
  | snoc (Q (lenfm, f, m, lenr, r), x) = checkQ (lenfm, f, m, lenr+1, x::r)
```

[1] Немного более эффективным вариантом было бы представлять очереди, меньшие определенного фиксированного размера, как обыкновенные списки.

Рис. 10.2: Развёрнутые очереди на основе структурной декомпозиции.

```
fun head (Q (lenfm, x :: f', m, lenr, r)) = x
fun tail (Q (lenfm, x :: f', m, lenr, r)) = checkQ (lenfm-1, f', m, lenr, r)
```

Всё самое интересное содержится во вспомогательной функции checkQ. Если список r слишком длинный, то checkQ создает задержку, которая должна развернуть r наоборот, а также оставляет задержку в m. После проверки длины r функция checkQ вызывает вторую вспомогательную функцию checkF, которая гарантирует, что список f непуст. Если пусты и f, и m, то вся очередь пуста. В противном случае, если пуст список f, мы изымаем первую задержку из m, вынуждаем её и запоминаем получившийся список как новую f.

```
fun checkF (lenfm, [], E, lenr, r) = E
  | checkF (lenfm, [], m, lenr, r) =
      Q (lenfm, force (head m), tail m, lenr, r)
  | checkF q = Q q
fun checkQ (q as (lenfm, f, m, lenr, r)) =
      if lenr ≤ lenfm then checkF q
      else checkF (lenfm+lenr, f, snoc (m, $rev r),0, [])
```

Обратите внимание, что checkQ и checkF вызывают snoc и tail, которые, в свою очередь, зовут checkQ. Следовательно, эти функции нужно определять через взаимную рекурсию. Полная реализация приведена на Рис. 10.2.

Эти очереди создают задержку, выполняющую обращение хвостового списка, в тот же самый момент, что и очереди по методу банкира, а вынуждают её на одну операцию раньше, чем очереди по методу банкира. Значит, поскольку операция обращения добавляет только $O(1)$ амортизированного времени к каждой операции в очередях по методу банкира, в наших развёрнутых очередях она тоже добавляет только $O(1)$ амортизированного времени. Однако время выполнения snoc и tail больше не является константой! Обратите внимание, что snoc вызывает checkQ, а эта функция, в свою очередь, может позвать snoc для m. Таким образом, может возникнуть каскад вызовов snoc, по одному на каждом уровне очереди. Но последовательные списки в m по крайней мере удваиваются в размере, так что длина m равна $O(\log n)$. Поскольку длина срединной очереди уменьшается по крайней мере на логарифмический множитель на каждом уровне, глубина всей очереди не больше $O(\log^* n)$. snoc на каждом уровне производит $O(1)$ амортизированной работы, так что всего snoc требует $O(\log^* n)$ амортизированного времени.

Подобным образом, вызов tail может привести к вызовам как snoc (из checkQ), так и tail (из checkF). Заметим, что когда такое бывает, tail применяется к результату snoc. Итак, snoc может рекурсивно себя вызвать, а tail может рекурсивно вызвать и snoc, и tail. Однако из Упражнения 10.3 мы знаем, что никогда snoc и tail не вызывают snoc рекурсивно подряд. Следовательно, и snoc, и tail зовутся максимум по разу на уровень. Поскольку на

каждом уровне и snoc, и tail выполняют $O(1)$ амортизированной работы, общая амортизированная стоимость tail равна $O(\log^* n)$.

Замечание 10.1 *На практике $O(\log^* n)$ является константой. Чтобы глубина достигла хотя бы пяти, очередь должна содержать не менее 2^{65536} элементов. Более того, если представлять очереди до размера четыре просто в виде списков, то очереди с числом элементов примерно до четырёх миллиардов будут содержать не более трёх уровней.*

Указание разработчикам 10.1 *На практике варианты этих очередей опережают все другие известные реализации в приложениях, где устойчивость используется умеренно, но требуется хорошее поведение даже в патологических случаях.*

Упражнение 10.3 *Рассмотрим выражение tail (snoc (q, x)). Покажите, что никогда не будет так, чтобы оба вызова, snoc и tail, рекурсивно обратились к snoc.*

Упражнение 10.4 *Реализуйте эти очереди без использования полиморфной рекурсии, при помощи типов*

```
datatype α EL =Elem of α | List of α EL list susp
datatype α Queue =E | Q of int × α EL list × α Queue × int × α EL list
```

Упражнение 10.5 *Ещё один способ избежать полиморфной рекурсии — представлять середину при помощи какой-либо другой реализации очередей. Тогда тип развёрнутых очередей будет*

```
datatype α Queue =
  E | Q of int × α list × α list susp PrimQ.Queue × int × α list
```

где PrimQ — другая реализация очередей.

1. *Реализуйте этот вариант развёрнутых очередей как функтор вида*

   ```
   functor BootstrappedQueue (PrimQ: Queue): Queue = ...
   ```

2. *Докажите, что если в качестве параметра PrimQ выступает какая-либо реализация очередей реального времени, то все операции на развёрнутых очередях выполняются за амортизированное время $O(1)$.*

10.2 Структурная абстракция

Второй разновидностью развёртки структур данных является *структурная абстракция* (structural abstraction). Как правило, она используется, чтобы расширить реализацию каких-либо коллекций, скажем, списков или куч, эффективной функцией слияния для сочетания двух коллекций. Во

многих реализациях нетрудно построить эффективную функцию insert, которая добавляет в коллекцию один новый элемент, но намного сложнее построить эффективную функцию слияния. Структурная абстракция создает коллекции, содержащие другие коллекции в качестве элементов. В таком случае, две коллекции можно слить, просто вставив одну в другую.

Идею структурной абстракции можно почти полностью описать на уровне типов. Допустим, имеется тип коллекций α C с элементами типа α, и этот тип поддерживает эффективную функцию insert с сигнатурой

val insert : $\alpha \times \alpha$ C $\to \alpha$ C

Назовём α C *элементарным типом* (primitive type). На основе этого типа мы хотим создать новый тип данных α B, называемый *развёрнутым типом* (bootstrapped type), чтобы α B эффективно поддерживал и операцию insert, и операцию join, с сигнатурами

val insert_B : $\alpha \times \alpha$ B $\to \alpha$ B
val join_B : α B $\times \alpha$ B $\to \alpha$ B

(С помощью нижнего индекса мы отличаем функции развёрнутого типа от функций элементарного типа.) Кроме того, развёрнутый тип должен поддерживать эффективную функцию unit для создания новой одноэлементной коллекции.

val unit_B : $\alpha \to \alpha$ B

Тогда можно реализовать insert_B просто как

fun insert_B (x, b) =join_B (unit_B x, b)

Основная идея структурной абстракции состоит в том, чтобы представлять развёрнутые коллекции как элементарные коллекции других развёрнутых коллекций. Тогда можно реализовать join_B через insert (а не insert_B!) приблизительно как

fun join_B (b_1, b_2) =insert (b_1, b_2)

Этот код вставляет b_1 в b_2 как элемент. Можно также вставить b_2 как элемент в b_1; фокус в том, что мы свели join к простой вставке.

Разумеется, всё не так просто. На основе приведенного описания, мы могли бы попытаться определить α B как

datatype α B =B **of** (α B) C

Можно считать, что это определение задает гомоморфизм

$$\alpha \text{ B} \cong (\alpha \text{ B}) \text{ C}$$

Раскрыв этот гомоморфизм несколько раз, мы легко увидим ошибку в определении.

$$\alpha \text{ B} \cong (\alpha \text{ B}) \text{ C} \cong ((\alpha \text{ B}) \text{ C}) \text{ C} \cong \cdots \cong ((\cdots \text{ C}) \text{ C}) \text{ C}$$

Тип α исчез, так что в нашей коллекции невозможно хранить никакие элементы! Можно справиться с этой проблемой, если сделать каждую развёрнутую коллекцию парой, состоящей из одного элемента и элементарной коллекции.

```
datatype α B =B of α × (α B) C
```

Тогда, например, unit_B можно определить как

```
fun unitB x =B (x, empty)
```

где empty — пустая элементарная коллекция.

Однако теперь возникает ещё одна проблема. Если каждая развёрнутая коллекция содержит по крайней мере один элемент, как мы представим пустую развёрнутую коллекцию? Следовательно, мы снова исправляем тип.

```
datatype α B =E | B of α × (α B) C
```

Замечание 10.2 *На самом деле, мы всегда устраиваем так, чтобы элементарная коллекция C содержала только непустые развёрнутые коллекции. Эту ситуацию можно более точно описать с помощью типов*

```
datatype α B+ =B+ of α × (α B+) C
datatype α B =E | NE of B+
```

К сожалению, определения такого вида ведут к более многословным программам, так что мы продолжаем использовать менее точное, но более короткое определение.

Теперь можно уточнить шаблоны функций insert_B и join_B:

```
fun insertB (x, E) =B (x, empty)
  | insertB (x, B (y, c)) =B (x, insert (unitB y, c))

fun joinB (b, E) =b
  | joinB (E, b) =b
  | joinB (B (x, c), b) =B (x, insert (b, c))
```

В этих шаблонах можно легко изменить различные детали. Например, во второй строке insert_B можно поменять местами x и y. Точно так же, в третьей строке join_B мы можем поменять ролями первый и второй аргументы.

Для каждого конкретного типа коллекций, как правило, существует некоторый выделенный элемент, к которому легче всего обратиться или который легче всего уничтожить, например, первый элемент или наименьший. Шаблоны insert_B и join_B нужно конкретизировать таким образом, чтобы выделенным элементом в коллекции B (x, c) был сам x. Творческой частью проектирования развёрнутой структуры данных методом структурной абстракции является реализация операции delete_B, стирающей выделенный элемент x. После уничтожения x у нас остается элементарная коллекция типа (α B) C, которую надо преобразовать в развёрнутую коллекцию типа α B. Подробности решения этой задачи различаются в зависимости от конкретной структуры.

Теперь мы собираемся конкретизировать шаблоны двумя способами. Сначала мы развёртываем очереди так, чтобы они эффективно поддерживали конкатенацию (т. е., операцию append). Затем мы развёртываем кучи, чтобы в них была эффективной операция слияния.

Рис. 10.3: Сигнатура списков с конкатенацией.

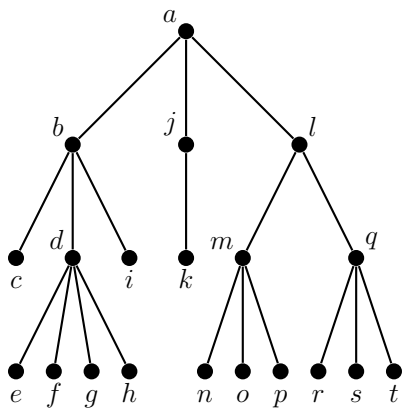


Рис. 10.4: Дерево, представляющее список $a \ldots t$.

10.2.1 Списки с эффективной конкатенацией

Первая структура данных, которую мы реализуем методом структурной абстракции — списки с конкатенацией, сигнатура которых представлена на Рис. 10.3. Списки с конкатенацией расширяют обычную сигнатуру списков эффективной функцией добавления одного списка к другому ($+\!\!+$). В качестве дополнительного удобства списки с конкатенацией также поддерживают операцию snoc, хотя мы могли бы легко имитировать snoc (xs, x) при помощи xs $+\!\!+$ cons (x, empty). Из-за этой способности списков с конкатенацией добавлять элементы к концу было бы правильно называть эту структуру данных деками с ограничением на вывод и конкатенацией.

Мы получим эффективную реализацию списков с конкатенацией, поддерживающую все операции за амортизированное время $O(1)$, применив развёртку к эффективной реализации очередей FIFO. Конкретный выбор реализации для элементарных очередей не имеет особого значения; годятся любые реализации устойчивых очередей с константным доступом, реального времени или амортизированные.

Если имеется реализация элементарных очередей Q, соответствующая сигнатуре Queue, по шаблону структурной абстракции списки с конкатенацией можно представить как

datatype α Cat = E | C **of** $\alpha \times \alpha$ Cat Q.Queue

Этот тип можно интерпретировать как дерево, где каждый узел содержит элемент, а непосредственные потомки каждого узла образуют очередь слева направо. Поскольку мы хотим иметь легкий доступ к первому элементу списка, мы его храним в корне дерева. На Рис. 10.4 изображен пример списка, хранящего элементы $a \ldots t$.

Функция head проста:

```
fun head (C (x, _)) = x
```

Чтобы сконкатенировать два непустых списка, мы связываем два дерева, делая второе из них последним ребёнком первого.

```
fun xs ++ E = xs
  | E ++ ys = ys
  | xs ++ ys = link xs ys
```

Вспомогательная функция link добавляет второй аргумент к очереди детей первого аргумента.

```
fun link (C (x, q), ys) = C (x, Q.snoc (q, ys))
```

Функции cons и snoc просто вызывают ++.

```
fun cons (x, xs) = C (x, Q.empty) ++ xs
fun snoc (xs, x) = xs ++ C (x, Q.empty)
```

Наконец, имея непустое дерево, функция tail должна отбросить корень и каким-то образом превратить очередь детей в единое дерево. Если очередь пуста, tail должна вернуть E. В противном случае мы связываем всех детей вместе.

```
fun tail (C (x, q)) = if Q.isEmpty q then E else linkAll q
```

Поскольку конкатенация ассоциативна, мы имеем право связывать детей в каком угодно порядке. Однако после небольшого размышления можно заключить, что связывание детей справа налево, как показано на Рис. 10.5, приведет к наименьшему повторению работы в последующих вызовах tail. Следовательно, мы реализуем linkAll как

```
fun linkAll q = let val t = Q.head q
                    val q' = Q.tail q
                in if Q.isEmpty q' then t else link (t, linkAll q') end
```

Замечание 10.3 *Функция linkAll является примером программной схемы foldr1.*

В этой реализации tail может отнимать до $O(n)$ времени. Мы надеемся уменьшить этот показатель до амортизированного $O(1)$, но чтобы добиться этого в условиях устойчивости, нужно как-то ввести в нашу структуру ленивое вычисление. Поскольку linkAll — единственная процедура, требующая более, чем $O(1)$ времени, она является естественным кандидатом. Мы переписываем linkAll, чтобы каждый рекурсивный вызов задерживался. Задержка вынуждается, когда дерево извлекается из очереди.

```
fun linkAll q = let val $t = Q.head q
                    val q' = Q.tail q
                in if Q.isEmpty q' then t else link (t, $linkAll q') end
```

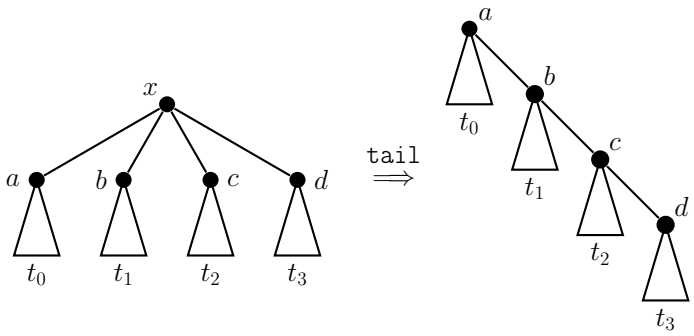


Рис. 10.5: Операция tail.

Рис. 10.6: Списки с конкатенацией.

Чтобы это определение имело смысл, нужно, чтобы в очередях содержались не просто деревья, а задержанные деревья, так что мы переопределяем тип как

datatype α Cat =E | C **of** α $\times$ α Cat susp Q.Queue

Чтобы соответствовать этому новому типу, операция ++ должна задерживать свой второй аргумент.

```
fun xs ++ E =xs
  | E ++ xs =xs
  | xs ++ ys =link (xs, $ys)
```

Полная реализация приведена на Рис. 10.6.

Очевидно, что head работает за время $O(1)$ в худшем случае, а cons и snoc имеют те же временные характеристики, что и ++. Мы доказываем, что ++ и tail работают за амортизированное время $O(1)$ методом банкира. Нераздельная стоимость этих операций $O(1)$, так что нам нужно только показать, что каждая из них высвобождает не более $O(1)$ единиц долга.

Пусть $d_t(i)$ будет количество единиц долга, приписанных к i-му узлу дерева t, а $D_t(i) = \sum_{j=0}^{i} d_t(j)$ — общая сумма долга на узлах t вплоть до i включительно. Пусть, наконец, D_t будет общая сумма долга на всех узлах t (т. е., $D_t = D_t(|t| - 1)$. Мы будем соблюдать два инварианта долга.

Во-первых, будем требовать, чтобы число единиц долга на каждом узле было ограничено сверху степенью этого узла (т. е., $d_t(i) \leq \mathit{degree}_t(i)$). Поскольку сумма степеней всех узлов непустого дерева на единицу меньше размера этого дерева, это означает, что общая сумма долга, приписанная к дереву, ограничена его размером (т. е., $D_t < |t|$). Этот инвариант мы будем поддерживать, увеличивая долг на узле дерева только одновременно с увеличением его степени.

Во-вторых, мы требуем, чтобы величина $D_t(i)$ была ограничена некоторой линейной функцией от i. Конкретная выбранная нами функция такова:

$$D_t(i) \leq i + depth_t(i)$$

где $depth_t(i)$ есть длина пути от корня дерева t до узла i. Этот инвариант называется *лево-линейный инвариант долга* (left-linear debit invariant). Заметим, что лево-линейный инвариант долга гарантирует нам, что $d_t(0) = D_t(0) \leq 0 + 0 = 0$, так что ко времени, когда узел оказывается корнем, весь долг на нём уже выплачен. (На самом деле, корень даже не является задержкой!) Единственное место, где мы вынуждаем задержки — когда задержанная вершина становится новым корнем.

Теорема 10.1 *Операции* $+\!\!+$ *и tail сохраняют оба инварианта долга, высвобождая, соответственно, одну и три единицы.*

Доказательство. *(*$+\!\!+$*) Единственная единица долга, создаваемая функцией* $+\!\!+$ *— для тривиальной задержки её второго аргумента. Поскольку степень этого узла не увеличивается, мы немедленно высвобождаем эту единицу. Предположим теперь, что* t_1 *и* t_2 *непусты, и что* $t = t_1 +\!\!+ t_2$*. Пусть* $n = |t_1|$*. Заметим, что индекс, глубина и общее количество единиц долга на всех вершинах* t_1 *не затрагиваются конкатенацией, так что для* $i < n$*,*

$$\begin{aligned} D_t(i) &= D_{t_1}(i) \\ &\leq i + depth_{t_1}(i) \\ &= i + depth_t(i) \end{aligned}$$

Индекс каждой вершины в t_2 *увеличивается на* n*, глубина увеличивается на единицу, а количество единиц долга увеличивается на общий долг* t_1*, так что*

$$\begin{aligned} D_t(n+i) &= D_{t_1} + D_{t_2}(i) \\ &< n + D_{t_2}(i) \\ &\leq n + i + depth_{t_2}(i) \\ &= n + i + depth_t(n+i) - 1 \\ &< n + i + depth_t(n+i) \end{aligned}$$

Таким образом, чтобы сохранить лево-линейный инвариант, больше никакие единицы долга высвобождать не требуется.

(tail) Пусть $t' = tail\, t$*. Отбросив корень* t*, мы связываем его детей* $t_0 \ldots t_{m-1}$ *справа налево. Пусть* t'_j *будет частичный результат связывания* $t_j \ldots t_{m-1}$*. Тогда* $t' = t'_0$*. Поскольку все операции связывания, кроме последней, задерживаются, мы присваиваем по одной единице долга корню каждого* t_j*,* $0 < j < m-1$*. Заметим, что степень каждого из этих узлов увеличивается на единицу. Кроме того, одну единицу долга мы присваиваем корню* t'_{m-1} *из-за того, что задерживается последний вызов linkAll, хотя он и не вызывает link. Поскольку степень этого узла не меняется, мы немедленно высвобождаем эту последнюю единицу долга.*

Предположим теперь, что i*-й узел дерева* t *оказывается в дереве* t_j*. Исходя из лево-линейного инварианта долга, мы знаем, что* $D_t(i) < i +$

$depth_t(i)$, однако рассмотрим теперь, как каждая из величин изменяется при применении операции tail . i уменьшается на единицу, поскольку отбрасывается первый элемент. Глубина каждого узла в t_j увеличивается на $j-1$ (см. Рис. 10.5), а общее число единиц долга на каждом узле t_j увеличивается на j. Таким образом,

$$\begin{aligned} D_{t'}(i-1) &= D_t(i) + j \\ &\leq i + depth_t(i) + j \\ &= i + (depth_{t'}(i-1) - (j-1)) + j \\ &= (i-1) + depth_{t'}(i-1) + 2 \end{aligned}$$

Высвобождение первых двух единиц долга восстанавливает инвариант, так что всего получается высвобождено три единицы.

Указание разработчикам 10.2 *Если имеется хорошая реализация очередей, то наши списки с конкатенацией — лучшая из известных устойчивых реализаций этой структуры, особенно для приложений, существенно опирающихся на устойчивость.*

Упражнение 10.6 *Напишите функцию flatten с типом α Cat list $\to \alpha$ Cat, конкатенирующую все элементы списка списков с конкатенацией. Покажите, что Ваша функция работает за амортизированное время $O(1+e)$, где e — число пустых списков с конкатенацией в исходном списке.*

10.2.2 Кучи с эффективным слиянием

В этом разделе мы используем структурную абстракцию для куч и получаем эффективную операцию слияния.

Допустим, у нас есть реализация куч, поддерживающая insert за время $O(1)$ в худшем случае, а merge, findMin и deleteMin за время $O(\log n)$ в худшем случае. Одна такая реализация — скошенные биномиальные кучи из Раздела 9.3.2; ещё одна — биномиальные кучи с расписанием из Раздела 7.3. При помощи структурной абстракции мы собираемся улучшить время работы операций merge и findMin до $O(1)$ в худшем случае.

Предположим пока что, что тип куч полиморфен относительно типа элементов, и что для любого типа элементов мы магическим образом знаем, какую функцию сравнения использовать. Позже мы учтём, что как тип элементов, так и функция сравнения на этих элементах задаются в момент применения функтора.

С учётом перечисленных предположений тип развёрнутых куч можно задать как

datatype α Heap =E | H **of** $\alpha \times (\alpha$ Heap$)$ PrimH.Heap

где PrimH — реализация элементарных куч. Элемент, хранимый в каждом узле H, будет минимальным элементом поддерева с корнем в этом узле. Элементами элементарных куч будут служить развёрнутые кучи. Внутри элементарных куч развёрнутые кучи упорядочены по своим минимальным

элементам (т. е., корням). Можно думать об этом типе как о типе деревьев с переменной степенью ветвления, причем дети каждого узла сами по себе хранятся в элементарных кучах.

Поскольку минимальный элемент хранится в корне, функция findMin проста:

```
fun findMin (H (x, _)) = x
```

Чтобы слить две развёрнутые кучи, мы помещаем кучу с большим корнем в кучу с меньшим корнем как элемент.

```
fun merge (E, h) = h
  | merge (h, E) = h
  | merge (h1 as H (x, p1), H2 as H (y, p2)) =
      if x < y then H (x, PrimH.insert (h2, p1))
      else H (y, H.insert (h1, p2))
```

(В выражении $x < y$ мы предполагаем, что функция $<$ — правильная функция сравнения для этих элементов.) insert определяется через merge.

```
fun insert (x, h) = merge (H (x, PrimH.empty), h)
```

Наконец, рассмотрим deleteMin, определённую как

```
fun deleteMin (H (x, p)) =
      if PrimH.isEmpty p then E
      else let val (H (y, p1)) =PrimH.findMin p
               val p2 =PrimH.deleteMin p
           in H (y, PrimH.merge (p1, p2)) end
```

Отбросив корень, сначала мы смотрим, пуста ли элементарная куча p. Если да, то новая куча также пуста. В противном случае мы находим и извлекаем минимальный элемент p, являющийся развернутой кучей с минимальным из всех элементом; этот элемент становится новым корнем. Наконец, мы сливаем p_1 и p_2 и получаем новую элементарную кучу.

Анализ этих куч не представляет сложности. Очевидно, что findMin работает за время $O(1)$ в худшем случае независимо от нижележащей реализации элементарных куч. Функции insert и merge зависят только от PrimH.insert. Поскольку мы предполагаем, что время работы PrimH.insert равно $O(1)$ в худшем случае, таково же и время работы insert и merge. Наконец, deleteMin вызывает PrimH.findMin, PrimH.deleteMin и PrimH.merge. Поскольку все они работают за $O(\log n)$ в худшем случае, такова же и характеристика deleteMin.

Замечание 10.4 *Можно также разворачивать кучи с амортизированными ограничениями производительности. Например, развёртка ленивых биномиальных куч из Раздела 6.4.1 даёт нам реализацию, поддерживающую findMin за время $O(1)$ в худшем случае, операции insert и merge за амортизированное время $O(1)$, а deleteMin за амортизированное время $O(\log n)$.*

До сиз пор мы предполагали, что тип куч полиморфен, но на самом деле сигнатура Heap указывает, что кучи мономорфны — как тип элементов, так и функция сравнения этих элементов фиксируются в момент применения функтора. Реализация кучи — это функтор, параметризованный типом элементов и функцией сравнения. Функтор, который мы используем для развёртки куч, отображает функторы куч в функторы куч, а не структуры куч в структуры куч. Мы можем выразить это с помощью функторов высших порядков [MT94]:

```
functor Bootstrap (functor MakeH (Element : Ordered)
                                 : Heap where type Elem.T = Element.T)
                  (Element : Ordered): Heap = ...
```

Функтор Bootstrap принимает функтор MakeH в качестве параметра. Функтор MakeH принимает структуру Element с сигнатурой Ordered, определяющей тип элементов и функцию сравнения, и возвращает структуру Heap. При заданном MakeH, Bootstrap возвращает функтор, который принимает структуру Element с сигнатурой Ordered и возвращает структуру Heap.

Замечание 10.5 *Ограничение where* ***type*** *в сигнатуре функтора MakeH необходимо, чтобы гарантировать, что функтор возвращает структуру кучи с необходимым типом элементов. Ограничения такого рода чрезвычайно часто встречаются в функторах высших порядков.*

Теперь, чтобы создать структуру элементарных куч с развёрнутыми кучами в качестве элементов, мы применяем функтор MakeH к структуре BootstrappedElem с сигнатурой Ordered, определяющей тип развёрнутых куч и функцию сравнения, упорядочивающую две развёрнутые кучи по их минимальным элементам. (Отношение порядка не определено на двух пустых кучах.) Это можно выразить при помощи следующих двух взаимно рекурсивных объявлений.

```
structure rec BootstrappedElem =
  struct
    datatype T = E | H of Elem.T × PrimH.Heap
    fun leq (H (x, _), H (Y, _)) = Elem.leq (x, y)
    ... Подобные же определения для eq и lt ...
  end
and PrimH = MakeH (BootstrappedElem)
```

где Elem — структура с сигнатурой Ordered, определяющая подлинные элементы развёрнутой кучи. Полная реализация функтора Bootstrapped приведена на Рис. 10.7.

Замечание 10.6 *В Стандартном ML запрещены рекурсивные определения структур, и этот запрет оправдан: это объявление не имеет смысла для функторов MakeH, имеющих эффекты. Однако функторы MakeH, к которым мы можем пожелать применить Bootstrap, например, SkewBinomialHeap*

```
(* Рекурсивные структуры не поддерживаются в Стандартном ML! *)
... Подобные же определения для  eq и lt ...
(* Экспортируются конструкторы  E и H *)
```

Рис. 10.7: Развёрнутые кучи.

Рис. 10.8: Альтернативная сигнатура для куч.

из Раздела 9.3.2, в этом отношении вполне безопасны, и рекурсивный шаблон, воплощаемый функтором Bootstrap, для них имеет смысл. Жаль, что Стандартный ML не дает нам выразить развёртку таким образом.

Развёрнутые кучи всё же можно реализовать в Стандартном ML, если явно подставить конкретную реализацию MakeH, скажем, SkewBinomialHeap или LazyBinomialHeap, а затем избавиться от отдельных структур BootstrappedElem и PrimH. Тогда рекурсия на структурах сводится к рекурсии на типах данных, которая Стандартным ML поддерживается.

Упражнение 10.7 *Подстановка функтора LazyBinomialHeap из Раздела 6.4.1, как указано выше, приводит к типам*

datatype *Tree* = *Node* ***of*** *int* × *Heap* × *Tree list*
datatype *Heap* = *E* | *NE* ***of*** *Elem.T* × *Tree list susp*

Завершите эту реализацию развёрнутых куч.

Упражнение 10.8 *Часто в элементах кучи содержится информация помимо приоритета. Для таких типов элементов часто бывает удобно использовать кучи, хранящие приоритет отдельно от остального содержимого элемента. На Рис. 10.8 приведена альтернативная сигнатура для такого типа куч.*

1. *Приспособьте либо LazyBinomialHeap, либо SkewBinomialHeap к этой сигнатуре.*

2. *Перепишите функтор Bootstrap как*

   functor *Bootstrap (PrimH : HeapWithInfo) : HeapWithInfo* = ...

   Вам не потребуются ни функторы высших порядков, ни рекурсивные структуры.

10.3 Развёртка до составных типов

Мы видели несколько примеров, где коллекции составных данных (например, кучи куч) оказывались полезными для реализации коллекций простых данных (например, кучи элементов). Однако коллекции составных данных

```
(* Если ключ не найден, возбудить  NotFound *)
```

Рис. 10.9: Сигнатура для конечных отображений.

часто бывают полезны сами по себе. Простой пример: строки (т. е., последовательности символов) часто служат типом элементов множеств или ключами в конечных отображениях. В этом разделе мы иллюстрируем развёртку конечных отображений, определённых для какого-то простого типа, до конечных отображений, определённых на списках или даже на деревьях, составленных из элементов этого типа.

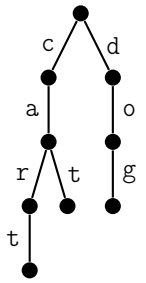
10.3.1 Префиксные деревья

Двоичные деревья поиска хорошо работают, когда операция сравнения для типа ключа или элемента дёшева. Это условие выполняется для простых типов вроде целых чисел и символов, но для составных типов вроде строк оно может оказаться неверным. Рассмотрим, например, представление телефонной книги с помощью двоичного дерева поиска. Обработка запроса «Кузнецов, Владислав» может потребовать множество сравнений с записями «Кузнецов, Владимир», и каждое из этих сравнений будет проверять десяток символов в каждой строке, прежде чем вернуть результат.

Для составных типов лучшим решением будет выбрать представление, которое использует структуру конкретного типа. Одним из таких представлений является *префиксное дерево* (trie), известное также как *цифровое дерево поиска* (digital search tree). В этой главе мы с помощью префиксного дерева реализуем абстракцию FiniteMap (конечное отображение), показанную на Рис. 10.9.

В последующем обсуждении мы будем предполагать, что ключи являются строками, и что представлены они как списки символов. Мы часто будем называть символы *базовым типом* (base type). Основные идеи легко распространить на другие типы последовательностей и другие базовые типы.

Префиксное дерево — это дерево с переменной степенью ветвления, и каждая дуга в нем помечена символом. Дуги, исходящие из корня дерева, представляют первый символ строки; дуги, исходящие из прямых потомков корня, представляют второй символ, и так далее. Чтобы найти узел, соответствующий данной строке, нужно начать с корня и двигаться по дугам, помеченным символами строки по порядку. Например, префиксное дерево, представляющее строки "cat", "dog", "car" и "cart", можно изобразить как



Заметим, что при вставке строки в префиксное дерево в него также попадают все префиксы этой строки. Только некоторые из этих префиксов будут соответствовать реальным записям. В нашем примере префиксами строки "cart" являются "c", "ca" и "car", но из них реальная запись соответствует только строке "car". Поэтому каждый узел требуется помечать как пустой или полный. В случае конечных отображений мы для этого используем встроенный тип данных option.

```
datatype α option =None | Some of α
```

Если узел пуст, мы помечаем его значением None. Если же узел полон, и соответствующая строка отображается в значение x, мы помечаем узел значением Some x.

Остаётся важный вопрос, как нам представлять дуги, исходящие из вершины. Обычно мы бы представляли непосредственных потомков вершины с переменной степенью ветвления в виде списка, однако здесь нам также требуется хранить метки дуг. В зависимости от выбора базового типа и ожидаемой плотности префиксного дерева, дуги, исходящие из узла, можно представлять в виде вектора, ассоциативного списка, двоичного дерева поиска или даже, если базовый тип сам по себе является списком или строкой, в виде ещё одного префиксного дерева! Однако все эти типы представляют из себя всего лишь конечные отображения из меток дуг в префиксные деревья. Мы абстрагируемся от конкретного представления отображений на дугах, предполагая, что нам дана некоторая структура M, реализующая конечные отображения для базового типа. В таком случае, представлением для префиксного дерева оказывается

```
datatype α Map =Trie of α option × α Map M.Map
```

Пустое дерево представлено как одиночная пустая вершина без потомков.

```
val empty = Trie (None, M.empty)
```

Чтобы найти строку, мы ищем каждый её символ в соответствующем отображении дуг. Дойдя до последнего узла, мы проверяем, пуст он или полон.

```
fun lookup ([], Trie (None, m)) = raise NotFound
  | lookup ([], Trie (Some x, m)) = x
  | lookup (k :: ks, Trie (v, m)) = lookup (ks, M.lookup (k, m))
```

Рис. 10.10: Простая реализация префиксных деревьев.

Заметим, что когда строка отсутствует в префиксном дереве, мы не всегда даже дойдём до последнего узла. Например, если в вышеприведённом примере мы будем искать слово "dark", мы найдём букву `d`, но не найдём `a`. При этом функция M.lookup возбудит исключение NotFound. Поскольку это исключение также является правильным ответом на lookup, мы его просто распространяем дальше.

Замечание 10.7 *Из-за такого поведения при неудачном поиске префиксные деревъя могут оказаться даже быстрее, чем хэш-таблицы. Неудачный поиск по префиксному дереву может завершиться, просмотрев только несколько первых символов, в то время как неудачный поиск по хэш-таблице требует просмотра всей строки только для того, чтобы вычислить хэш-функцию!*

Функция записи в дерево bind очень похожа на lookup, только здесь мы не позволяем M.lookup окончиться неудачно. Мы подставляем пустой узел каждый раз, когда она возбуждает исключение NotFound.

```
fun bind ([], x, Trie (_, m)) = Trie (Some x, m)
  | bind (k :: ks, x, Trie (v, m)) =
      let val t = M.lookup (k, m) handle NotFound ⇒ empty
          val t' = bind (ks, x, t)
      in Trie (v, M.bind (k, t', m)) end
```

Полная реализация приведена на Рис. 10.10.

Упражнение 10.9 *Очень часто множество ключей, которые нужно хранить в префиксном дереве, обладает свойством, что ни один ключ не может быть префиксом другого. Например, все ключи могут иметь одну и ту же длину, либо все ключи могут заканчиваться на один и тот же символ, который ни в одной другой позиции не встречается. Реализуйте префиксные деревья заново, предполагая, что это условие выполняется, и используя тип*

datatype α *Map* = *Entry* ***of*** α | *Trie* ***of*** α *Map M.Map*

Упражнение 10.10 *В префиксных деревъях часто встречаются длинные цепочки узлов, каждый из которых имеет только по одному потомку. Существует оптимизация, когда все эти узлы сливаются в один. Этого можно добиться, храня в каждом узле подстроку-наибольший общий префикс всех ключей, путь к которым пролегает через этот узел. В таком случае, тип префиксных деревъев будет*

datatype α *Map* = *Trie* ***of*** *M.key list* $\times$ α *option* $\times$ α *Map M.Map*

Реализуйте префиксные деревья, используя этот тип. Следует соблюдать инвариант, что узел не может одновременно быть пустым и являться единственным потомком. Можно предполагать, что структура M содержит функцию isEmpty.

Упражнение 10.11 ***(Швенке [Sch97])*** *Есть ещё одна структура, использующая многослойные конечные отображения — хэш-таблица (hash table). Завершите следующую реализацию абстрактных хэш-таблиц.*

```
functor HashTable (structure Approx : FiniteMap,
                   structure Exact  : FiniteMap,
                   val hash : Exact.Key → Approx.Key) : FiniteMap =
struct
   type Key = Exact.Key
   type α Map =α Exact.Map Approx.Map
   ...
   fun lookup (k, m) = Exact.lookup (k, Approx.lookup (hash k, m))
   ...
end
```

Преимущество этого представления состоит в том, что в Approx мы можем использовать эффективный тип ключа (скажем, целые числа), а в Exact можно использовать тривиальное представление (например, ассоциативные списки).

10.3.2 Обобщенные префиксные деревья

Идею префиксных деревьев можно обобщить со списков на другие составные типы, например, деревья [CM95]. Рассмотрим сначала, как отображения дуг в предыдущем разделе отражают тип конструктора cons. Отображения дуг представлены типом α Map M.Map. Внешнее отображение индексирует первое поле конструктора cons, а внутреннее отображение — второе поле конструктора cons. Поиск головы cons-ячейки во внешнем отображении даёт нам внутреннее отображение, и в нём мы ищем хвост ячейки cons.

Мы можем обобщить эту схему на двоичные деревья, имеющие три поля, добавив третий слой отображений. Например, если у нас есть двоичные деревья типа

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

то мы можем представлять отображения дуг в префиксных деревьях по этим деревьям как α Map Map M.Map. Внешнее отображение индексирует первое поле конструктора T, среднее отображение — второе поле, а внутреннее отображение индексирует третье поле. Поиск элемента в каком-либо узле во внешнем отображении даёт нам среднее отображение, где мы можем искать левое поддерево. Этот поиск, в свою очередь, даст нам внутреннее отображение, и там мы можем искать правое поддерево.

Более формально, мы представляем префиксные деревья над бинарными деревьями типом

(* предполагает полиморфную рекурсию! *)

Рис. 10.11: Обобщённые префиксные деревья.

```
datatype α Map =Trie of α option × α Map Map M.Map
```

Заметим, что здесь мы имеем гетерогенный рекурсивный тип, так что в функциях, работающих с этим типом, нам потребуется полиморфная рекурсия.

Функция lookup проводит три поиска на каждый конструктор T, соответствующих трём полям конструктора. Когда она достигает последнего узла, проводится проверка, полон ли этот узел.

```
fun lookup (E, Trie (None, m)) = raise NotFound
  | lookup (E, Trie (Some x, m)) = x
  | lookup (T (k, a, b), Trie (v, m)) =
      lookup (b, lookup (a, M.lookup (k, m)))
```

Функция bind работает подобным же образом. Она показана на Рис. 10.11, где приведена полная реализация префиксных деревьев поверх двоичных деревьев.

Упражнение 10.12 *Реализуйте функтор TrieOverTrees без использования полиморфной рекурсии, на основе типов*

```
datatype α Map =Trie of α EM option × α Map M.Map
datatype α EM =Elem of α | Map of α Map
```

Упражнение 10.13 *Реализуйте префиксные деревья, ключами в которых служат деревья с переменной степенью типа*

```
datatype α Tree =T of α × α Tree list
```

Имея эти примеры, мы можем обобщить понятие префиксного дерева до любого рекурсивного типа, включающего произведения и суммы. Требуется только несколько простых правил о том, как сконструировать конечное отображение для структурного типа, имея в качестве основы конечные отображения для его компонент. Пусть $\alpha\ \mathrm{Map}_\tau$ будет тип конечных отображений над типом τ.

В случае произведений мы уже знаем, что делать; чтобы найти пару в префиксном дереве, нужно сначала найти первый элемент этой пары и получить отображение, где следует искать второй элемент. Таким образом,

$$\tau = \tau_1 \times \tau_2 \Rightarrow \alpha\ \mathrm{Map}_\tau = \alpha\ \mathrm{Map}_{\tau_2}\ \mathrm{Map}_{\tau_1}$$

Что же делать с суммами? Вспомним тип деревьев и префиксных деревьев над ними:

```
datatype α Tree =E | T of α × α Tree × α Tree
datatype α Map =Trie of α option × α Map Map M.Map
```

Ясно, что тип α Map Map M.Map соответствует конструктору T, но что соответствует конструктору E? Тип α option представляет собой не что иное, как очень эффективную реализацию конечных отображений над типом unit, который, в свою очередь, эквивалентен отсутствующему телу конструктора E. Отсюда мы выводим общее правило для сумм:

$$\tau = \tau_1 + \tau_2 \Rightarrow \alpha \text{ Map}_\tau = \alpha \text{ Map}_{\tau_1} \times \alpha \text{ Map}_{\tau_2}$$

Упражнение 10.14 *Завершите следующие функторы, которые реализуют вышеописанные правила для произведений и сумм:*

```
functor ProductMap (M1 : FiniteMap) (M2 : FiniteMap) : FiniteMap =
struct
    type Key = M1.Key × M2.Key
    ...
end

datatype (α, β) Sum =Left of α | Right of β
functor SumMap (M1 : FiniteMap) (M2 : FiniteMap) : FiniteMap =
struct
    type Key = (M1.Key, M2 Key) Sum
    ...
end
```

Упражнение 10.15 *Пусть имеется структура M, реализующая конечные отображения над типом идентификаторов Id. Реализуйте префиксные деревья над типом лямбда-выражений Exp, где*

```
datatype Exp = Var of Id | Lam of Id × Exp | App of Exp × Exp
```

В процессе решения полезно будет расширить тип префиксных деревьев отдельным конструктором для пустого отображения.

10.4 Примечания

Развёртка структур данных Развёртка структур данных была распознана как общая методика проектирования структур данных в работах Бухсбаума и его коллег [Buc93, BT95, BST95]. Структурная декомпозиция и структурная абстракция использовались и раньше, соответственно в [Die82] и [DST94].

Списки с конкатенацией Несмотря на то, что построить альтернативное представление устойчивых списков, поддерживающих эффективную операцию конкатенации, относительно легко (см., например, [Hug86]), такие альтернативные представления, казалось, почти неизбежно приносили в жертву эффективность функций head или tail.

Майерс [Mye82] описывает представление, основанное на AVL-деревьях, поддерживающее все основные операции за время $O(\log n)$. Тарьян с коллегами [DST94, BT95, KT95] исследовали множество сублогарифмических

представлений. Кульминацией из работы стало представление, поддерживающее конкатенацию и все остальные обычные функции над списками за время $O(1)$ в худшем случае. Реализация списков с конкатенацией из Раздела 10.2.1 впервые появилась в [Oka95a]. Она намного проще, чем у Каплана и Тарьяна, но дает лишь амортизированные ограничения, а не жёсткие.

Кучи со слиянием Многие императивные реализации поддерживают операции insert , merge и findMin за амортизированное время $O(1)$, а операцию deleteMin за амортизированное время $O(\log n)$, включая биномиальные кучи [KL93], фибоначчиевы кучи [FT87], расслабленные кучи [DGST88], V-кучи [Pet87], скошенные снизу вверх кучи [ST86b] и парные кучи [FSST86]. Однако из всех этих структур, кажется, только парные кучи сохраняют свою амортизированную эффективность в сочетании с ленивым порядком вычисления и устойчивостью (см. Раздел 6.5), и, к сожалению, даже для парных куч скоростные характеристики являются только недоказанным предположением.

Бродал [Bro95, Bro96] достигает аналогичных ограничений в худшем случае. Его исходная структура данных [Bro95] может быть реализована чисто функциональным образом (и, таким образом, сделана устойчивой), если сочетать её с методикой рекурсивного замедления Каплана и Тарьяна [KT95], а также с чисто функциональной реализацией деков реального времени, вроде приведенной в Разделе 8.4.3. Однако такая реализация будет очень сложной и медленной. Бродал и Окасаки упрощают эту реализацию в [BO96], используя скошенные биномиальные кучи (Раздел 9.3.2) и структурную абстракцию (Раздел 10.2.2).

Полиморфная рекурсия Существует несколько попыток расширить Стандартный ML полиморфной рекурсией, например, [Mye84, Hen93, KTU93]. Одна из сложностей состоит в том, что при наличии полиморфной рекурсии вывод типов становится формально неразрешимым [Hen93, KTU93], хотя на практике он и работает. Язык Haskell обходит эту проблему, позволяя полиморфную рекурсию в случае, если программист явно указывает сигнатуру типа.

Глава 11

Неявное рекурсивное замедление

В Разделе 9.2.3 мы видели, что избыточное ленивое представление двоичных чисел может поддерживать как функцию увеличения, так и уменьшения за амортизированное время $O(1)$. В Разделе 10.1.2 мы видели, что гетерогенные типы и полиморфная рекурсия позволяют строить чрезвычайно простые реализации числовых представлений, например, двоичных списков с произвольным доступом. В этой главе мы сочетаем и расширяем эти идеи, получая в результате методику, называемую *неявное рекурсивное замедление* (implicit recursive slowdown).

Каплан и Тарждан [KT95, KT96b, KT96a] исследовали родственную методику под названием *рекурсивное замедление* (recursive slowdown), основанную, в отличие от нашей, не на ленивых двоичных числах, а на сегментированных двоичных числах (Раздел 9.2.4). Сходства и различия реализаций, основанных на рекурсивном замедлении и на неявном рекурсивном замедлении, в сущности, аналогичны сходствам и различиям между этими двумя системами счисления.

11.1 Очереди и деки

Напомним устройство двоичных списков с произвольным доступом из Раздела 10.1.2, имеющих тип

```
datatype α RList =
    Nil | Zero of (α × α) RList |One of α × (α × α) RList
```

Чтобы упростить дальнейшее обсуждение, давайте заменим этот тип на

```
datatype α Digit =Zero | One of α
datatype α RList =Shallow of α Digit |Deep of α Digit × (α × α) RList
```

Мелкий (Shallow) список содержит от нуля до одного элемента. Глубокий (Deep) список содержит ноль или один элемент, а также список пар. С этим

типом мы можем играть во многие из игр, освоенных нами при рассмотрении двоичных списков с произвольным доступом в Главе 9. Например, можно поддержать функцию head за время $O(1)$, переключившись на безнулевое представление вроде

```
datatype α Digit =Zero | One of α |Two of α × α
datatype α RList =Shallow of α Digit |Deep of α Digit × (α × α) RList
```

В этом представлении все цифры в глубоком (Deep) узле должны быть единицами или двойками. Конструктор ноль-Zero используется только в пустом списке Shallow Zero.

Подобным образом, задержав список пар в каждом глубоком узле, мы можем заставить либо cons, либо tail работать за амортизированное время $O(1)$, а вторую из этих операций за амортизированное время $O(\log n)$.

```
datatype α RList =
      Shallow of α Digit
    | Deep of α Digit × (α × α) RList susp
```

Позволив выбирать из трёх ненулевых цифр в каждом глубоком узле, мы можем заставить все три функции cons, head и tail работать за время $O(1)$.

```
datatype α Digit =
    Zero | One of α |Two of α × α |Three of α × α × α
```

Как и прежде, конструктор Zero используется только в пустом списке.

Чтобы расширить эту схему для поддержки очередей и деков, достаточно добавить вторую цифру в каждый глубокий узел.

```
datatype α Queue =
      Shallow of α Digit
    | Deep of α Digit × (α × α) Queue susp × α Digit
```

Первая цифра представляет первые несколько элементов очереди, а вторая — последние несколько элементов. Оставшиеся элементы хранятся в задержанной очереди пар, которую мы называем *срединной очередью* (middle queue).

Выбор типа цифры зависит от того, какие функции мы хотим поддерживать на каждом конце очереди. В следующей таблице приведены разрешённые значения для головной цифры очереди, поддерживающей каждое данное сочетание функций.

поддерживаемые функции	разрешённые цифры
cons	Zero, One
cons/head	One, Two
head/tail	One, Two
cons/head/tail	One, Two, Three

Те же правила выбора относятся и к хвостовой цифре.

В качестве конкретного примера давайте разработаем реализацию очередей, поддерживающую snoc на хвостовом конце и head и tail на головном

Рис. 11.1: Очереди на основе неявного рекурсивного замедления.

(т. е., обыкновенных очередей-FIFO). Обратившись к таблице, мы решаем, что головная цифра глубокого узла может быть единица-One или двойка-Two, а хвостовая цифра может быть ноль-Zero или единица-One. Цифра в мелком узле может быть Zero или One.

Чтобы добавить к глубокой очереди новый элемент y через snoc, мы смотрим на хвостовую цифру. Если это ноль (Zero), мы заменяем хвостовую цифру на единицу-One y. Если это One x, то мы заменяем её на Zero и добавляем пару (x, y) к срединной очереди. Кроме того, требуется выписать несколько особых случаев для добавления элементов к мелкой очереди.

```
fun snoc (Shallow Zero, y) = Shallow (One y)
  | snoc (Shallow (One x), y) = Deep (Two (x, y), $empty, Zero)
  | snoc (Deep (f, m, Zero), y) = Deep (f, m, One y)
  | snoc (Deep (f, m, One x), y) =
      Deep (f, $snoc (force m, (x, y)), Zero)
```

Чтобы удалить элемент из глубокой очереди через tail, мы смотрим на головную цифру. Если это Two (x, y), мы отбрасываем x и устанавливаем головную цифру в One y. Если это One x, мы «занимаем» в срединной очереди пару (y, z) и устанавливаем головную цифру в Two (y, z). Опять же, нужно учесть ещё несколько особых случаев для работы с мелкими очередями.

```
fun tail (Shallow (One x)) = empty
  | tail (Deep (Two (x, y), m, r)) = Deep (One y, m, r)
  | tail (Deep (One x, $q, r)) =
      if isEmpty q then Shallow r
      else let val (y, z) = head q
           in Deep (Two (y, z), $tail q, r) end
```

Заметим, что в последнем варианте tail мы вынуждаем срединную очередь. Полный код приведен на Рис. 11.1.

Теперь мы хотим показать, что snoc и tail работают за амортизированное время $O(1)$. Заметим, что snoc никак не обращается к головной цифре, а tail — к хвостовой цифре. Если мы рассматриваем каждую из функций по отдельности, то snoc оказывается аналогичен функции inc для ленивых двоичных чисел, а tail оказывается аналогичен функции dec для безнулевых ленивых двоичных чисел. Модифицируя доказательства для inc и dec, мы легко можем показать, что snoc и tail работают за амортизированное время $O(1)$, если каждая из них используется отдельно от другой.

Основная идея неявного рекурсивного замедления состоит в том, что когда функции вроде snoc и tail *почти* независимы друг от друга, мы можем сочетать их доказательства, просто сложив долги, используемые в каждом из доказательств. Доказательство для snoc использует одну единицу долга, если хвостовая цифра равна Zero, и ноль единиц, если хвостовая цифра равна One. Доказательство для tail использует одну единицу долга, если

головная цифра равна Two и ноль единиц, если головная цифра равна One. Нижеследующее доказательство сочетает эти два понятия долга.

Теорема 11.1 *Функции snoc и tail работают за амортизированное время* $O(1)$.

Доказательство. *Мы анализируем реализацию очередей, используя метод банкира. Долг присваивается каждой задержке; задержки у нас всегда находятся в среднем поле какой-либо глубокой очереди. Мы принимаем инвариант долга, позволяющий каждой задержке иметь размер долга, зависящий от цифр в головном и хвостовом поле. Среднее поле глубокой очереди может иметь до* $|f| - |r|$ *единиц долга, где* $|f|$ *равно одному или двум, а* $|r|$ *равно нулю или одному.*

Нераздельная стоимость каждой из функций равна $O(1)$*, так что нам остаётся показать, что ни одна из функций не высвобождает больше, чем* $O(1)$ *единиц долга. Мы приводим только доказательство для tail . Доказательство для snoc немного проще.*

Мы проводим рассуждения методом передачи долга, который близкородствен методу наследования долга. Каждый раз, когда вложенная задержка получает больше долга, чем ей разрешено иметь, мы передаём этот долг объемлющей задержке, которая служит средним полем предыдущего узла Deep. Передача долга является безопасной операцией, поскольку объемлющая задержка всегда вынуждается раньше вложенной. Передача ответственности за высвобождение долга от вложенной задержки к объемлющей гарантирует, что этот долг будет высвобожден прежде, чем будет вынуждена объемлющая задержка, а следовательно, и раньше, чем может быть вынуждена внутренняя.

Мы показываем, что каждый вызов tail передает одну единицу долга в объемлющую задержку, кроме самого внешнего вызова, у которого объемлющей задержки нет. Этот вызов просто высвобождает лишний долг.

Каждый каскад вызовов tail заканчивается на вызове, заменяющем Two на One. (Для простоты описания, мы сейчас не учитываем возможность добраться до мелкой очереди.) Это уменьшает разрешённый размер долга для m *на один, так что мы передаём эту лишнюю единицу в объемлющую задержку.*

Всякий промежуточный вызов tail заменяет f с единицы-One на двойку-Two и вызывает tail рекурсивно. Есть два подслучая:

- *r равно Zero. Очередь* m *имеет одну единицу долга, и эту единицу требуется высвободить, прежде чем мы можем вынудить* m*. Мы передаём эту единицу в объемлющую задержку. Кроме того, создаём единицу долга, чтобы покрыть нераздельную стоимость рекурсивного вызова. Наконец, нашей задержке передаётся одна единицы долга из рекурсивного вызова. Поскольку нашей задержке разрешено иметь до двух единиц долга, баланс оказывается в порядке.*

- *r равно One. Очередь* m *не имеет долга, так что мы бесплатно можем её вынудить. Создаём одну единицу долга, чтобы покрыть нераз-*

```
(* Возбуждает Empty, если дек пуст *)
(* Возбуждает Empty, если дек пуст *)
(* Возбуждает Empty, если дек пуст *)
(* Возбуждает Empty, если дек пуст *)
```

Рис. 11.2: Сигнатура для двусторонних очередей с конкатенацией.

дельную стоимость рекурсивного вызова. Кроме того, из рекурсивного вызова нам передаётся ещё одна единица долга. Поскольку разрешённый размер долга для текущей задержки равен одному, мы одну единицу долга оставляем у себя, а другую передаём в объемлющую задержку.

Упражнение 11.1 *Реализуйте для этих очередей функции lookup и update. Эти функции должны работать за амортизированное время $O(\log i)$. Может быть полезно снабдить каждую очередь полем, содержащим её размер.*

Упражнение 11.2 *С помощью методик, описанных в этом разделе, реализуйте двусторонние очереди.*

11.2 Двусторонние очереди с конкатенацией

Наконец, мы реализуем с помощью неявного рекурсивного замедления двусторонние очереди с конкатенацией, чья сигнатура приведена на Рис. 11.2. Сначала мы описываем относительно простую реализацию, поддерживающую $+\!\!+$ за амортизированное время $O(\log n)$, а остальные операции за амортизированное время $O(1)$. Затем мы строим намного более сложную реализацию, которая улучшает время работы $+\!\!+$ до $O(1)$.

Рассмотрим следующую реализацию двусторонних очередей с конкатенацией, или с-деков. С-дек является либо *мелким* (shallow), либо *глубоким* (deep). Мелкий с-дек — это просто обыкновенный дек, например, дек по методу банкира из Раздела 8.4.2. Глубокий с-дек состоит из трёх частей: *front* (голова), *середина* (middle) и *хвост* (rear). Голова и хвост являются обыкновенными деками, содержащими не меньше двух элементов каждый. Середина является с-деком с обыкновенными деками в качестве элементов, каждый из которых не короче двух. Мы предполагаем, что есть реализация D, реализующая сигнатуру Deque, и все её функции работают за время $O(1)$ (амортизированное или жёсткое).

```
datatype α Cat =
    Shallow of α D.Queue
  | Deep of α D.Queue × α D.Deque Cat susp × α D.Queue
```

Заметим, что это определение предполагает полиморфную рекурсию.

Чтобы добавить элемент к какому-либо концу, мы просто добавляем его в головной или хвостовой дек. Например, cons реализован как

```
fun cons (x, Shallow d) = Shallow (D.cons (x, d))
  | cons (x, Deep (f, m, r)) = Deep (D.cons (x, f), m, r)
```

Чтобы уничтожить элемент на каком-либо конце, мы уничтожаем элемент из головного либо хвостового дека. Если при этом длина этого дека падает ниже двух, мы извлекаем следующий дек из середины и делаем его новой головой либо хвостом. С добавлением остающегося элемента из старого дека новый дек содержит по крайней мере три элемента. Например, код tail выглядит как

```
fun tail (Shallow d) = Shallow (D.tail d)
  | tail (Deep (f, m, r) =
      let f' = D.tail f
      in
        if not (tooSmall f') then Deep (f', m, r)
        else if isEmpty (force m) then Shallow (dappendL (f', r))
        else Deep (dappendL (f', head (force m)), $tail (force m), r)
      end
```

где функция tooSmall возвращает истину, если длина дека меньше двух, а dappendL добавляет дек длины один или два к деку произвольной длины.

Заметим, что вызовы tail распространяются на следующий уровень с-дека только в том случае, когда длина головного дека равна двум. В терминах из Раздела 9.2.3 мы можем сказать, что дек длиной три или более *безопасен* (safe), а дек длиной два *опасен* (dangerous). Каждый раз, когда tail рекурсивно себя вызывает на следующем уровне, он переводит головной дек из опасного состояния в безопасное, так что ни на каком уровне с-дека два последовательных вызова tail не могут распространиться на следующий уровень. Мы легко можем доказать, что tail работает за амортизированное время $O(1)$, позволив безопасному деку иметь одну единицу долга, а опасному ноль.

Упражнение 11.3 *Докажите, что tail и init вместе работают за амортизированное время $O(1)$, сочетая их правила накопления долга согласно методике неявного рекурсивного замедления.*

Как реализовать конкатенацию? Чтобы сконкатенировать два глубоких с-дека c_1 и c_2, мы сохраняем голову c_1 как новую голову, хвост c_2 как новый хвост, а из оставшихся элементов собираем новую середину: хвост c_1 вставляем в середину c_1, голову c_2 в середину c_2, а затем конкатенируем результаты.

```
fun (Deep (f1, m1, r1)) ++ (Deep (f2, m2, r2)) =
      Deep (f1, $(snoc (force m1, r1) ++ cons (f2, force m2)), r2)
```

(Разумеется, есть ещё варианты, когда c_1 или c_2 являются мелкими.) Заметим, что глубина рекурсии ++ равна глубине более мелкого с-дека. Кроме того, ++ создаёт $O(1)$ долга на каждом уровне, и весь этот долг нужно немедленно высвободить, чтобы восстановить инвариант долга для tail и

(* предполагает полиморфную рекурсию! *)
... snoc, last и init определяются симметричным образом ...

Рис. 11.3: Простые деки с конкатенацией.

init. Следовательно, время работы $+\!\!+$ равно $O(\min(\log n_1, \log n_2))$, где n_i — длина c_i.

Полный код этой реализации с-деков приведён на Рис. 11.3.

Чтобы улучшить время работы $+\!\!+$ до $O(1)$, мы изменяем представление с-деков так, чтобы операция $+\!\!+$ не вызывала сама себя рекурсивно. Основная идея состоит в том, чтобы $+\!\!+$ каждого уровня обращалась на следующем уровне только к cons и snoc. Вместо трёх сегментов мы теперь заставляем глубокие с-деки содержать пять сегментов: (f, a, m, b, r). f, m и r представляют собой обыкновенные деки; f и r содержат при этом не менее трёх элементов каждый, а m не менее двух элементов. a и b представляют собой с-деки *составных элементов* (compound elements). Вырожденный составной элемент является обыкновенным деком, содержащим не менее двух элементов. Полный составной элемент содержит три сегмента: (f, c, r), где f и r — обыкновенные деки, содержащие не меньше чем по два элемента каждый, а m — с-дек составных элементов. Этот тип данных может быть записан на Стандартном ML (с полиморфной рекурсией) так:

```
datatype α Cat =
    Shallow of α D.Queue
  | Deep of α D.Queue                          (* ≥ 3 *)
          × α CmpdElem Cat susp
          × α D.Queue                          (* ≥ 2 *)
          × α CmpdElem Cat susp
          × α D.Queue                          (* ≥ 3 *)
and α CmpdElem =
      Simple of α D.Queue                      (* ≥ 2 *)
    | Cmpd of α D.Queue                        (* ≥ 2 *)
          × α CmpdElem Cat susp
          × α D.Queue                          (* ≥ 2 *)
```

Если нам даны глубокие с-деки c_1 =Deep (f_1, a_1, m_1, b_1, r_1) и c_2 =Deep (f_2, a_2, m_2, b_2, r_2), их конкатенация вычисляется следующим образом: прежде всего, f_1 сохраняется как голова результата, а r_2 как хвост результата. Затем мы строим новый срединный дек из последнего элемента r_1 и первого элемента f_2. Затем мы порождаем составной элемент из m_1, b_1 и остатка r_1, и прицепляем его к концу a_1 через snoc. Это будет сегмент a результата. Наконец, мы порождаем составной элемент из остатка f_2, a_2 и m_2, и присоединяем его к началу b_2. Это будет сегмент b результата. Вся реализация выглядит как

```
fun (Deep (f1, a1, m1, b1, r1)) ++ (Deep (f2, a2, m2, b2, r2)) =
    let val (r1', m, f2') =share (r1, f2)
        val a1' – $snoc (force a1, Cmpd (m1, b1, r1'))
        val b2' =$cons (Cmpd (f2', a2, m2), force b2)
    in Deep (f1, a1', m, b2', r2) end
```

```
(* Предполагается, что D поддерживает функцию size *)
... snoc и last определяются симметричным образом ...
```

Рис. 11.4: Деки с конкатенацией, использующие неявное рекурсивное замедление (часть I).

```
... replaceLast и init определяются симметричным образом ...
```

Рис. 11.5: Деки с конкатенацией, использующие неявное рекурсивное замедление (часть II).

где

```
fun share (f, r) =
      let val m = D.cons (D.last f, D.cons (D.head r, D.empty))
      in (D.init f, m, D.tail r)
fun cons (x, Deep (f, a, m, b, r)) = Deep (D.cons (x, f), a, m, b, r)
fun snoc (Deep (f, a, m, b, r), x) = Deep (f, a, m, b, D.snoc (r, x))
```

(Ради простоты описания мы опускаем варианты с участием мелких с-деков.)

К сожалению, tail и init в этой реализации устроены весьма коряво. Поскольку эти две функции симметричны, мы описываем только tail. Если у нас есть с-дек Deep (f, a, m, b, r), возможны шесть вариантов:

- $|\mathrm{f}| > 3$
- $|\mathrm{f}| = 3$
  - a непуст.
    - Первый составной элемент a вырожден.
    - Первый составной элемент a невырожден.
  - a пуст, a b непуст.
    - Первый составной элемент b вырожден.
    - Первый составной элемент b невырожден.
  - a и b оба пусты.

Мы описываем поведение tail с в первых трёх случаях. Код для оставшихся случаев можно найти в полной реализации, приведенной на Рис. 11.4 и 11.5. Если $|\mathrm{f}| > 3$, мы просто заменяем f на D.tail f. Если $|\mathrm{f}| = 3$, то уничтожение первого элемента f сделает его размер меньше разрешённого. Следовательно, нам нужно вынуть новый головной дек из a и состыковать его с остающимися в f двумя элементами. Новый f содержит не меньше четырёх элементов, так что следующий вызов tail пойдёт по ветке $|\mathrm{f}| > 3$.

Когда мы извлекаем первый составной элемент из a, чтобы построить новый головной дек, этот составной элемент может быть вырожденным или невырожденным. Если он вырожденный (т. е., обыкновенный дек), новым

значением a будет \$tail (force a). Если же мы получаем полный составной элемент Cmpd (f', c', r'), то f' оказывается новым значением f (вместе с остающимися элементами старого f), а новое значение a будет

```
$(force c' ++ cons (Simple r', tail (force a)))
```

Заметим, однако, что в результате комбинации cons и tail мы просто заменяем первый элемент a. Можно сделать это напрямую, избежав тем самым ненужного вызова tail, с помощью функции replaceHead.

```
fun replaceHead (x, Shallow d) = Shallow (D.cons (x, D.tail d))
  | replaceHead (x, Deep (f, a, m, b, r) =
      Deep (D.cons (x, D.tail f), a, m, b, r)
```

Оставшиеся варианты tail устроены похожим образом; каждый из них производит $O(1)$ работы, а затем делает максимум один вызов tail.

Замечание 11.1 *Этот код можно записать намного короче и намного понятнее с использованием языковой конструкции, называемой взгляды (views) [Wad87, BC93, PPN96], позволяющей устраивать сопоставление с образцом на абстрактных типах данных. Детали можно найти в [Oka97]. В Стандартном ML взгляды не поддерживаются.*

В функциях cons, snoc, head и last ленивое вычисление не используется, и легко видеть, что все они работают за время $O(1)$. Остальные функции мы анализируем методом банкира с использованием передачи долга.

Как всегда, мы присваиваем долг каждой задержке. Задержки содержатся в сегментах a и b глубокого c-дека, а также в средних сегментах (c) составных элементов. Каждому полю c мы разрешаем иметь до четырёх единиц долга, а полям a и b мы позволяем иметь от нуля до пяти единиц, в зависимости от длины полей f и r. Базовый лимит полей a и b равен нулю. Если в поле f содержится более трёх элементов, то лимит поля a увеличивается на четыре, а лимит поля b на одну единицу. Подобным образом, если поле r содержит более трёх элементов, то лимит поля b увеличивается на четыре, а лимит поля a на одну единицу.

Теорема 11.2 *Функции ++, tail и init работают за амортизированное время $O(1)$.*

Доказательство. *(++) Интересный случай — конкатенация двух c-деков Deep (f_1, a_1, m_1, b_1, r_1) и Deep (f_2, a_2, m_2, b_2, r_2). В этом случае ++ производит $O(1)$ нераздельной работы и высвобождает не более четырёх единиц долга. Во-первых, мы создаём две единицы долга для задержанных вызовов snoc и cons для a и b соответственно. Эти две единицы мы всегда высвобождаем. Кроме того, если на b_1 или a_2 висит пять единиц долга, нам нужно высвободить одну единицу, когда этот сегмент становится серединой составного элемента. Наконец, если в f_1 содержится только три элемента, а в f_2 более трёх элементов, нам нужно высвободить единицу долга из b_2, поскольку он становится новым b; и то же самое справедливо*

для r_1 и r_2. Заметим, однако, что если на b_1 висит пять единиц долга, то f_1 содержит более трёх элементов, а если на a_2 висит пять единиц долга, то r_2 содержит более трёх элементов. Следовательно, всего нам нужно высвободить не больше четырёх единиц долга, или, по крайней мере, передать этот долг объемлющей задержке.

(tail и init) Поскольку функции tail и init симметричны, мы приводим рассуждение только для tail. Простым просмотром можно убедиться, что tail совершает $O(1)$ нераздельной работы, так что нам остаётся показать, что она высвобождает не более $O(1)$ долга. Мы покажем, что размер высвобождаемого долга не превышает пять единиц.

Поскольку tail может вызывать сама себя рекурсивно, нам нужно учитывать возможность каскада вызовов tail. Мы используем в рассуждениях передачу долга. Пусть у нас есть глубокий с-дек Deep (f, a, m, b, r). Нужно рассмотреть каждый вариант поведения tail.

Если $|f| > 3$, мы находимся в конце каскада. Нового долга не создаётся, но извлечение элемента из f может уменьшить разрешённый размер долга для a на четыре единицы, а b на одну единицу, так что мы передаём этот долг объемлющей задержке.

Если $|f| > 3$, то предположим, что a непуст. (Случаи с пустым a не содержат принципиальных отличий.) Если $|r| > 3$, то a может иметь одну единицу долга, которую мы передаём в объемлющую задержку. В противном случае a не должен иметь долга. Если голова a является вырожденным составным элементом (т. е., простым деком элементов), то он становится новым значением f вместе с оставшимися элементами старого f. Новое значение a представляет собой задержку от результата применения tail к старому a. Эта задержка получает до пяти единиц долга из рекурсивного вызова tail. Поскольку новый разрешённый размер долга для a не меньше четырёх, мы передаём не более одной единицы долга в объемлющую задержку, и всего передаваемых единиц долга получается не больше двух. (На самом деле, размер передаваемого долга не больше одного, поскольку здесь мы передаём одну единицу в точности в тех случаях, когда нам не нужно было передавать одну единицу долга из исходного a.)

Если же голова a является невырожденным составным элементом Cmpd (f', c', r'), то f' становится новым значением f вместе с остающимися элементами старого f. Вычисление нового a требует вызовов ++ и replaceHead. Полное число получаемых при этом единиц долга равно девяти: четыре от c', четыре от ++ и одна свежесозданная единица долга от replaceHead. Разрешённый размер долга для нового a равен либо четырём, либо пяти, так что либо четыре, либо пять единиц долга мы передаём объемлющей задержке. Поскольку четыре единицы требуется передавать ровно в тех случаях, когда одну единицу нужно было передать из старого значения a, всего требуется передать не более пяти единиц долга.

Упражнение 11.4 *Пусть имеется реализация D деков без конкатенации. Реализуйте списки с конкатенацией, используя тип*

datatype α *Cat* =

```
        Shallow of α D.Queue
      | Deep of α D.Queue × α CmpdElem Cat susp × α D.Queue
and α CmpdElem =Cmpd of α D.Queue × α CmpdElem Cat susp
```

причём как головной дек глубокого (Deep) узла, так и дек в узле Cmpd должны содержать не менее двух элементов. Докажите, что все функции в вашей реализации работают за амортизированное время $O(1)$ при условии, что все функции в D работают за время $O(1)$ (ограничение может быть жёстким или амортизированным).

11.3 Примечания

Рекурсивное замедление Понятие рекурсивного замедления было введено Капланом и Тарьяном в [KT95] и снова использовано ими же в [KT96b], но оно близкородственно ограничениям регулярности у Гибаса и др. [GMPR77]. Бродал [Bro95] пользовался похожим методом при реализации куч.

Деки с конкатенацией Бухсбаум и Тарьян [BT95] представляют чисто функциональную реализацию деков с конкатенацией, которая поддерживает tail и init за время $O(\log^* n)$ в худшем случае, а все остальные операции за $O(1)$ в худшем случае. Наша реализация улучшает этот показатель до $O(1)$ для всех операций, но ограничения получаются амортизированными, а не жёсткими. Независимо от нас, Каплан и Тарьян разработали похожую реализацию с жёсткими показателями $O(1)$. Однако детали их реализации весьма сложны.

Приложение A

Код на языке Haskell

A.1 Очереди

```
module Queue (Queue(..)) where

import Prelude hiding (head, tail)

class Queue q where
  empty   :: q a
  isEmpty :: q a → Bool

  snoc    :: q a → a → q a
  head    :: q a → a
  tail    :: q a → q a
```

```
module BatchedQueue (BatchedQueue) where

import Prelude hiding (head, tail)
import Queue

data BatchedQueue a = BQ [a] [a]

check [] r = BQ (reverse r) []
check f  r = BQ f r

instance Queue BatchedQueue where
  empty = BQ [] []
  isEmpty (BQ f r) = null f

  snoc (BQ f r) x = check f (x:r)

  head (BQ []    _) = error "empty queue"
  head (BQ (x:f) r) = x
```

```
  tail (BQ []     _) = error "empty queue"
  tail (BQ (x:f) r) = check f r
```

```
module BankersQueue (BankersQueue) where

import Prelude hiding (head, tail)
import Queue

data BankersQueue a = BQ Int [a] Int [a]

check lenf f lenr r =
  if lenr ≤ lenf
  then BQ lenf f lenr r
  else BQ (lenf + lenr) (f ++ reverse r) 0 []

instance Queue BankersQueue where
  empty = BQ 0 [] 0 []
  isEmpty (BQ lenf f lenr r) = (lenf == 0)

  snoc (BQ lenf f lenr r) x = check lenf f (lenr + 1) (x:r)

  head (BQ lenf []     lenr r) = error "empty queue"
  head (BQ lenf (x:f') lenr r) = x

  tail (BQ lenf []     lenr r) = error "empty queue"
  tail (BQ lenf (x:f') lenr r) = check (lenf - 1) f' lenr r
```

```
module PhysicistsQueue (PhysicistsQueue) where

import Prelude as P
import Queue

data PhysicistsQueue a = PQ [a] Int [a] Int [a]

check w lenf f lenr r =
  if lenr ≤ lenf
  then checkw w lenf f lenr r
  else checkw f (lenf + lenr) (f ++ reverse r) 0 []

checkw [] lenf f lenr r = PQ f lenf f lenr r
checkw  w lenf f lenr r = PQ w lenf f lenr r

instance Queue PhysicistsQueue where
  empty = PQ [] 0 [] 0 []
  isEmpty (PQ w lenf f lenr r) = (lenf == 0)

  snoc (PQ w lenf f lenr r) x = check w lenf f (lenr+1) (x:r)
```

```
  head (PQ []     lenf f lenr r) = error "empty_queue"
  head (PQ (x:w)  lenf f lenr r) = x

  tail (PQ []     lenf f lenr r) = error "empty_queue"
  tail (PQ (x:w)  lenf f lenr r) = check w (lenf-1) (P.tail f) lenr r
```

```
module HoodMelvilleQueue (HoodMelvilleQueue) where

import Prelude hiding (head, tail)
import Queue

data RotationState a
    = Idle
    | Reversing Int [a] [a] [a] [a]
    | Appending Int [a] [a]
    | Done [a]

data HoodMelvilleQueue a = HM Int [a] (RotationState a) Int [a]

exec (Reversing ok (x:f)  f' (y:r) r') = Reversing (ok+1) f (x:f') r (y:r')
exec (Reversing ok []     f' [y]   r') = Appending ok f' (y:r')
exec (Appending 0  f'     r')          = Done r'
exec (Appending ok (x:f') r')          = Appending (ok-1) f' (x:r')
exec state = state

invalidate (Reversing ok f f' r r') = Reversing (ok-1) f f' r r'
invalidate (Appending 0 f' (x:r'))  = Done r'
invalidate (Appending ok f' r')     = Appending (ok-1) f' r'
invalidate state = state

exec2 lenf f state lenr r =
  case exec (exec state) of
    Done newf -> HM lenf newf Idle lenr r
    newstate  -> HM lenf f newstate lenr r

check lenf f state lenr r =
  if lenr < lenf
  then exec2 lenf f state lenr r
  else let newstate = Reversing 0 f [] r []
       in exec2 (lenf+lenr) f newstate 0 []

instance Queue HoodMelvilleQueue where
  empty = HM 0 [] Idle 0 []
  isEmpty (HM lenf f state lenr r) = (lenf == 0)

  snoc (HM lenf f state lenr r) x = check lenf f state (lenr+1) (x:r)

  head (HM _ []     _ _ _) = error "empty_queue"
```

```
  head (HM _ (x:f') _ _ _) = x

  tail (HM lenf []     state lenr r) = error "empty queue"
  tail (HM lenf (_:f') state lenr r) =
    check (lenf-1) f' (invalidate state) lenr r
```

```
module BootstrappedQueue (BootstrappedQueue) where

import Prelude hiding (head, tail)
import Queue

data BootstrappedQueue a
    = E
    | Q Int [a] (BootstrappedQueue [a]) Int [a]

checkQ, checkF :: Int →
                  [a] →
                  BootstrappedQueue [a] →
                  Int →
                  [a] →
                  BootstrappedQueue a

checkQ lenfm f m lenr r =
  if lenr < lenfm
  then checkF lenfm f m lenr r
  else checkF (lenfm+lenr) f (snoc m (reverse r)) 0 []

checkF lenfm [] E lenr r = E
checkF lenfm [] m lenr r = Q lenfm (head m) (tail m) lenr r
checkF lenfm  f m lenr r = Q lenfm f m lenr r

instance Queue BootstrappedQueue where
  empty = Q 0 [] E 0 []
  isEmpty E = True
  isEmpty _ = False

  snoc E x = Q 1 [x] E 0 []
  snoc (Q lenfm f m lenr r) x = checkQ lenfm f m (lenr+1) (x:r)

  head E = error "empty queue"
  head (Q lenfm (x:f') m lenr r) = x

  tail E = error "empty queue"
  tail (Q lenfm (x:f') m lenr r) = checkQ (lenfm-1) f' m lenr r
```

```
module ImplicitQueue (ImplicitQueue) where

import Prelude hiding (head, tail)
```

```
import Queue

data Digit a = Zero | One a | Two a a

data ImplicitQueue a
    = Shallow (Digit a)
    | Deep (Digit a) (ImplicitQueue (a, a)) (Digit a)

instance Queue ImplicitQueue where
  empty = Shallow Zero
  isEmpty (Shallow Zero) = True
  isEmpty _              = False

  snoc (Shallow Zero)      y = Shallow (One y)
  snoc (Shallow (One x))   y = Deep (Two x y) empty Zero
  snoc (Deep f m Zero)     y = Deep f m (One y)
  snoc (Deep f m (One x)) y = Deep f (snoc m (x, y)) Zero

  head (Shallow Zero)        = error "empty queue"
  head (Shallow (One x))     = x
  head (Deep (One x)   m r) = x
  head (Deep (Two x y) m r) = x

  tail (Shallow Zero)        = error "empty queue"
  tail (Shallow (One x))     = empty
  tail (Deep (Two x y) m r) = Deep (One y) m r
  tail (Deep (One x)   m r) =
    if isEmpty m then Shallow r
    else Deep (Two y z) (tail m) r
    where (y, z) = head m
```

A.2 Двусторонние очереди

```
module Deque (Deque(..)) where

import Prelude hiding (head, tail, last, init)

class Deque q where
  empty   :: q a
  isEmpty :: q a → Bool

  cons    :: a → q a → q a
  head    :: q a → a
  tail    :: q a → q a

  snoc    :: q a → a → q a
  last    :: q a → a
  init    :: q a → q a
```

```
module BankersDeque (BankersDeque) where

import Prelude hiding (head, tail, last, init)
import Deque

data BankersDeque a = BD Int [a] Int [a]

c = 3

check lenf f lenr r =
  if lenf > c*lenr + 1 then
    let i  = (lenf + lenr) `div` 2
        j  = lenf + lenr - i
        f' = take i f
        r' = r ++ reverse (drop i f)
    in BD i f' j r'
  else if lenr > c*lenf + 1 then
         let j  = (lenf+lenr) `div` 2
             i  = lenf + lenr - i
             r' = take j r
             f' = f ++ reverse (drop j r)
         in BD i f' j r'
       else BD lenf f lenr r

instance Deque BankersDeque where
  empty = BD 0 [] 0 []
  isEmpty (BD lenf f lenr r) = (lenf+lenr == 0)

  cons x (BD lenf f lenr r) = check (lenf+1) (x:f) lenr r

  head (BD lenf []     lenr r) = error "empty deque"
  head (BD lenf (x:f') lenr r) = x

  tail (BD lenf []     lenr r) = error "empty deque"
  tail (BD lenf (x:f') lenr r) = check (lenf-1) f' lenr r

  snoc (BD lenf f lenr r) x = check lenf f (lenr+1) (x:r)

  last (BD lenf f lenr []    ) = error "empty deque"
  last (BD lenf f lenr (x:r')) = x

  init (BD lenf f lenr []    ) = error "empty deque"
  init (BD lenf f lenr (x:r')) = check lenf f (lenr-1) r'
```

A.3 Списки с конкатенацией

```
module CatenableList (CatenableList(..)) where

import Prelude hiding (head, tail, (++))
```

```
class CatenableList c where
  empty   :: c a
  isEmpty :: c a → Bool

  cons    :: a   → c a → c a
  snoc    :: c a → a   → c a
  (++)    :: c a → c a → c a

  head    :: c a → a
  tail    :: c a → c a
```

```
module CatList (CatList) where

import Prelude hiding (head, tail, (++))
import CatenableList
import Queue (Queue)
import qualified Queue

data CatList q a
    = E
    | C a (q (CatList q a))

link (C x q) s = C x (Queue.snoc q s)

instance Queue q ⇒ CatenableList (CatList q) where
  empty = E
  isEmpty E = True
  isEmpty _ = False

  xs ++ E  = xs
  E  ++ xs = xs
  xs ++ ys = link xs ys

  cons x xs = C x Queue.empty ++ xs
  snoc xs x = xs ++ C x Queue.empty
  head E = error "empty list"

  head (C x q) = x
  tail E       = error "empty list"
  tail (C x q) = if Queue.isEmpty q then E else linkAll q
    where linkAll q = if Queue.isEmpty q' then t
                      else link t (linkAll q')
          where t  = Queue.head q
                q' = Queue.tail q
```

A.4 Двусторонние очереди с конкатенацией

```
module CatenableDeque (
  CatenableDeque(..),
  Deque(..)
  ) where

import Prelude hiding (head, tail, last, init, (++))
import Deque

class Deque d ⇒ CatenableDeque d where
  (++) :: d a → d a → d a
```

```
module SimpleCatenableDeque (SimpleCatDeque) where

import Prelude hiding (head, tail, last, init, (++))
import CatenableDeque

data SimpleCatDeque d a
    = Shallow (d a)
    | Deep (d a) (SimpleCatDeque d (d a)) (d a)

tooSmall d = isEmpty d || isEmpty (tail d)

dappendL d1 d2 = if isEmpty d1 then d2 else cons (head d1) d2
dappendR d1 d2 = if isEmpty d2 then d1 else snoc d1 (head d2)

instance Deque d ⇒ Deque (SimpleCatDeque d) where
  empty = Shallow empty
  isEmpty (Shallow d) = isEmpty d
  isEmpty _ = False

  cons x (Shallow d)  = Shallow (cons x d)
  cons x (Deep f m r) = Deep (cons x f) m r

  head (Shallow d)  = head d
  head (Deep f m r) = head f

  tail (Shallow d) = Shallow (tail d)
  tail (Deep f m r)
    | not (tooSmall f) = Deep f m r
    | isEmpty m = Shallow (dappendL f r)
    | otherwise = Deep (dappendL f (head m)) (tail m) r
    where f = tail f
— snoc, last, and init defined symmetrically...

instance Deque d ⇒ CatenableDeque (SimpleCatDeque d) where
  (Shallow d1) ++ (Shallow d2)
    | tooSmall d1 = Shallow (dappendL d1 d2)
    | tooSmall d2 = Shallow (dappendR d1 d2)
    | otherwise = Deep d1 empty d2
```

```
  (Shallow d) ++ (Deep f m r)
    | tooSmall d = Deep (dappendL d f) m r
    | otherwise = Deep d (cons f m) r
  (Deep f m r) ++ (Shallow d)
    | tooSmall d = Deep f m (dappendR r d)
    | otherwise = Deep f (snoc m r) d
  (Deep f1 m1 r1) ++ (Deep f2 m2 r2) =
    Deep f1 (snoc m1 r1 ++ cons f2 m2) r2
```

```
module ImplicitCatenableDeque (
  Sized(..),
  ImplicitCatDeque
  ) where

import Prelude hiding (head, tail, last, init, (++))
import CatenableDeque

class Sized d where
  size :: d a → Int

data ImplicitCatDeque d a
    = Shallow (d a)
    | Deep (d a) (ImplicitCatDeque d (CmpdElem d a)) (d a)
                 (ImplicitCatDeque d (CmpdElem d a)) (d a)

data CmpdElem d a
    = Simple (d a)
    | Cmpd (d a) (ImplicitCatDeque d (CmpdElem d a)) (d a)

share f r = (init f, m, tail r)
  where m = cons (last f) (cons (head r) empty)

dappendL d1 d2 =
  if isEmpty d1 then d2
  else dappendL (init d1) (cons (last d1) d2)

dappendR d1 d2 =
  if isEmpty d2 then d1
  else dappendR (snoc d1 (head d2)) (tail d2)

replaceHead x (Shallow d) = Shallow (cons x (tail d))
replaceHead x (Deep f a m b r) = Deep (cons x (tail f)) a m b r

instance (Deque d, Sized d) ⇒ Deque (ImplicitCatDeque d) where
  empty = Shallow empty
  isEmpty (Shallow d) = isEmpty d
  isEmpty _ = False

  cons x (Shallow d) = Shallow (cons x d)
```

```
  cons x (Deep f a m b r) = Deep (cons x f) a m b r

  head (Shallow d) = head d
  head (Deep f a m b r) = head f

  tail (Shallow d) = Shallow (tail d)
  tail (Deep f a m b r)
    | size f > 3 = Deep (tail f) a m b r
    | not (isEmpty a) =
      case head a of
        Simple d → Deep f' (tail a) m b r
          where f' = dappendL (tail f) d
        Cmpd f' c' r' → Deep f'' a'' m b r
          where f'' = dappendL (tail f) f'
                a'' = c' ++ replaceHead (Simple r') a
    | not (isEmpty b) =
      case head b of
        Simple d → Deep f' empty d (tail b) r
          where f' = dappendL (tail f) m
        Cmpd f' c' r' → Deep f'' a'' r' (tail b) r
          where f'' = dappendL (tail f) m
                a'' = cons (Simple f') c'
    | otherwise = Shallow (dappendL (tail f) m) ++ Shallow r

— snoc, last, and init defined symmetrically...

instance (Deque d, Sized d) ⇒ CatenableDeque (ImplicitCatDeque d) where
  (Shallow d1) ++ (Shallow d2)
    | size d1 < 4 = Shallow (dappendL d1 d2)
    | size d2 < 4 = Shallow (dappendR d1 d2)
    | otherwise = let (f, m, r) = share d1 d2 in Deep f empty m empty r
  (Shallow d) ++ (Deep f a m b r)
    | size d < 4 = Deep (dappendL d f) a m b r
    | otherwise = Deep d (cons(Simple f) a) m b r
  (Deep f a m b r) ++ (Shallow d)
    | size d < 4 = Deep f a m b (dappendR r d)
    | otherwise = Deep f a m (snoc b (Simple r)) d
  (Deep f1 a1 m1 b1 r1) ++ (Deep f2 a2 m2 b2 r2) = Deep f1 a'1 m b'2 r2
    where (r'1, m, f'2) = share r1 f2
          a'1 = snoc a1 (Cmpd m1 b1 r'1)
          b'2 = cons (Cmpd f'2 a2 m2) b2
```

A.5 Списки с произвольным доступом

```
module RandomAccessList (RandomAccessList(..)) where

import Prelude hiding (head, tail, lookup)

class RandomAccessList r where
```

```
  empty   :: r a
  isEmpty :: r a → Bool

  cons    :: a → r a → r a
  head    :: r a → a
  tail    :: r a → r a

  lookup  :: Int → r a → a
  update  :: Int → a → r a → r a
```

```
module BinaryRandomAccessList (BinaryList) where

import Prelude hiding (head, tail, lookup)
import RandomAccessList

data Tree a  = Leaf a | Node Int (Tree a) (Tree a)
data Digit a = Zero | One (Tree a)
newtype BinaryList a = BL [Digit a]

size (Leaf x) = 1
size (Node w t1 t2) = w

link t1 t2 = Node (size t1 + size t2) t1 t2

consTree t   [] = [One t]
consTree t   (Zero : ts) = One t : ts
consTree t1 (One t2 : ts) = Zero : consTree (link t1 t2) ts

unconsTree [] = error "empty list"
unconsTree [One t] = (t, [])
unconsTree (One t:ts) = (t, Zero : ts)
unconsTree (Zero:ts) = (t1, One t2 : ts')
  where (Node _ t1 t2, ts') = unconsTree ts

instance RandomAccessList BinaryList where
  empty = BL []
  isEmpty (BL ts) = null ts

  cons x (BL ts) = BL (consTree (Leaf x) ts)
  head (BL ts) = let (Leaf x, _) = unconsTree ts in x
  tail (BL ts) = let (_, ts') = unconsTree ts in BL ts'

  lookup i (BL ts) = look i ts
    where
      look i [] = error "bad subscript"
      look i (Zero : ts) = look i ts
      look i (One t : ts) =
        if i < size t then lookTree i t
        else look (i − size t) ts
```

```
      lookTree 0 (Leaf x) = x
      lookTree i (Leaf x) = error "bad subscript"
      lookTree i (Node w t1 t2) =
        if i < w `div` 2 then lookTree i t1
        else lookTree (i - w `div` 2) t2

  update i y (BL ts) = BL (upd i ts)
    where
      upd i [] = error "bad subscript"
      upd i (Zero : ts) = Zero : upd i ts
      upd i (One t : ts) =
        if i < size t then One (updTree i t) : ts
        else One t : upd (i - size t) ts

      updTree 0 (Leaf x) = Leaf y
      updTree i (Leaf x) = error "bad subscript"
      updTree i (Node w t1 t2) =
        if i < w `div` 2 then Node w (updTree i t1) t2
        else Node w t1 (updTree (i - w `div` 2) t2)
```

```
module SkewBinaryRandomAccessList (SkewList) where

import Prelude hiding (head, tail, lookup)
import RandomAccessList

data Tree a = Leaf a | Node a (Tree a) (Tree a)
newtype SkewList a = SL [(Int, Tree a)]

instance RandomAccessList SkewList where
  empty = SL []
  isEmpty (SL ts) = null ts

  cons x (SL ((w1, t1):(w2, t2):ts))
    | w1 == w2 = SL ((1+w1+w2, Node x t1 t2):ts)
  cons x (SL ts) = SL ((1, Leaf x) : ts)

  head (SL []) = error "empty list"
  head (SL ((1, Leaf x):ts)) = x
  head (SL ((w, Node x t1 t2):ts)) = x

  tail (SL []) = error "empty list"
  tail (SL ((1, Leaf x):ts)) = SL ts
  tail (SL ((w, Node x t1 t2):ts)) = SL ((w `div` 2, t1):(w `div` 2, t2):ts)

  lookup i (SL ts) = look i ts
    where
      look i [] = error "bad subscript"
      look i ((w, t):ts) =
```

```
        if i < w then lookTree w i t else look (i–w) ts

    lookTree 1 0 (Leaf x) = x
    lookTree 1 i (Leaf x) = error "bad_subscript"
    lookTree w 0 (Node x t1 t2) = x
    lookTree w i (Node x t1 t2) =
      if i ≤ w' then lookTree w' (i–1) t1
      else lookTree w' (i–1–w') t2
      where w' = w `div` 2

update i y (SL ts) = SL (upd i ts)
  where
    upd i [] = error "bad_subscript"
    upd i ((w, t):ts) =
      if i < w then (w, updTree w i t): ts
      else (w, t) : upd (i–w) ts

    updTree 1 0 (Leaf x) = Leaf y
    updTree 1 i (Leaf x) = error "bad_subscript"
    updTree w 0 (Node x t1 t2) = Node y t1 t2
    updTree w i (Node x t1 t2) =
      if i ≤ w' then Node x (updTree w' (i–1) t1) t2
      else Node x t1 (updTree w' (i–1–w') t2)
      where w' = w `div` 2
```

```
module AltBinaryRandomAccessList (BinaryList) where

import Prelude hiding (head, tail, lookup)
import RandomAccessList

data BinaryList a
    = Nil
    | Zero  (BinaryList (a, a))
    | One a (BinaryList (a, a))

uncons:: BinaryList a → (a, BinaryList a)
uncons Nil = error "empty_list"
uncons (One x Nil) = (x, Nil)
uncons (One x ps)  = (x, Zero ps)
uncons (Zero ps)   = let ((x, y), ps') = uncons ps in (x, One y ps')

fupdate :: (a → a) → Int → BinaryList a → BinaryList a
fupdate f i Nil = error "bad_subscript"
fupdate f 0 (One x ps) = One (f x) ps
fupdate f i (One x ps) = cons x (fupdate f (i–1) (Zero ps))
fupdate f i (Zero ps)  = Zero (fupdate f' (i `div` 2) ps)
  where f' (x,y) = if i `mod` 2 == 0 then (f x, y) else (x, f y)

instance RandomAccessList BinaryList where
```

```
  empty = Nil
  isEmpty Nil = True
  isEmpty _   = False

  cons x Nil = One x Nil
  cons x (Zero ps)  = One x ps
  cons x (One y ps) = Zero (cons (x, y) ps)

  head xs = fst (uncons xs)
  tail xs = snd (uncons xs)

  lookup i Nil = error "bad subscript"
  lookup 0 (One x ps) = x
  lookup i (One x ps) = lookup (i-1) (Zero ps)
  lookup i (Zero ps)  = if i `mod` 2 == 0 then x else y
    where (x, y) = lookup (i `div` 2) ps

  update i y xs = fupdate (λx → y) i xs
```

A.6 Кучи

```
module Heap (Heap(..)) where

class Heap h where
  empty     :: Ord a ⇒ h a
  isEmpty   :: Ord a ⇒ h a → Bool

  insert    :: Ord a ⇒ a → h a → h a
  merge     :: Ord a ⇒ h a → h a → h a

  findMin   :: Ord a ⇒ h a → a
  deleteMin :: Ord a ⇒ h a → h a
```

```
module LeftistHeap (LeftistHeap) where

import Heap

data LeftistHeap a = E | T Int a (LeftistHeap a) (LeftistHeap a)

rank E = 0
rank (T r _ _ _) = r

makeT x a b = if rank a ≥ rank b
              then T (rank b + 1) x a b
              else T (rank a + 1) x b a

instance Heap LeftistHeap where
  empty = E
```

```
  isEmpty E = True
  isEmpty _ = False

  insert x h = merge (T 1 x E E) h

  merge h E = h
  merge E h = h
  merge h1@(T _ x a1 b1) h2@(T _ y a2 b2) =
    if x ≤ y
    then makeT x a1 (merge b1 h2)
    else makeT y a2 (merge h1 b2)

  findMin E = error "empty heap"
  findMin (T _ x a b) = x

  deleteMin E = error "empty heap"
  deleteMin (T _ x a b) = merge a b
```

```
module BinomialHeap (BinomialHeap) where

import Heap

data Tree a = Node Int a [Tree a]
newtype BinomialHeap a = BH [Tree a]

rank (Node r x c) = r
root (Node r x c) = x

link t1@(Node r x1 c1) t2@(Node _ x2 c2) =
  if x1 ≤ x2
  then Node (r+1) x1 (t2 : c1)
  else Node (r+1) x2 (t1 : c2)

insTree t [] = [t]
insTree t ts@(t' : ts') =
  if rank t < rank t' then t:ts else insTree (link t t') ts'

mrg ts1 [] = ts1
mrg [] ts2 = ts2
mrg ts1@(t1:ts'1) ts2@(t2:ts'2)
  | rank t1 < rank t2 = t1 : mrg ts'1 ts2
  | rank t2 < rank t1 = t2 : mrg ts1 ts'2
  | otherwise = insTree (link t1 t2) (mrg ts'1 ts'2)

removeMinTree [] = error "empty heap"
removeMinTree [t] = (t, [])
removeMinTree (t:ts) =
  if root t < root t' then (t, ts) else (t', t:ts')
  where (t', ts') = removeMinTree ts
```

```
instance Heap BinomialHeap where
  empty = BH []
  isEmpty (BH ts) = null ts

  insert x (BH ts) = BH (insTree (Node 0 x []) ts)
  merge (BH ts1) (BH ts2) = BH (mrg ts1 ts2)
  findMin (BH ts) = root t
    where (t, _) = removeMinTree ts

  deleteMin (BH ts) = BH (mrg (reverse ts1) ts2)
    where (Node _ x ts1, ts2) = removeMinTree ts
```

```
module SplayHeap (SplayHeap) where

import Heap

data SplayHeap a = E | T (SplayHeap a) a (SplayHeap a)

partition pivot E = (E, E)
partition pivot t@(T a x b) =
  if x ≤ pivot then
    case b of
      E → (t, E)
      T b1 y b2 →
        if y ≤ pivot then
          let (small, big) = partition pivot b2
          in (T (T a x b) y small, big)
        else
          let (small, big) = partition pivot b1
          in (T a x small, T big y b2)
  else
    case a of
      E → (E, t)
      T a1 y a2 →
        if y ≤ pivot then
          let (small, big) = partition pivot a2
          in (T a1 y small, T big x b)
        else
          let (small, big) = partition pivot a1
          in (small, T big y (T a2 x b))

instance Heap SplayHeap where
  empty = E
  isEmpty E = True
  isEmpty _ = False

  insert x t = T a x b
    where (a, b) = partition x t
```

```
  merge E t = t
  merge (T a x b) t = T (merge ta a) x (merge tb b)
    where (ta, tb) = partition x t

  findMin E = error "empty heap"
  findMin (T E x b) = x
  findMin (T a x b) = findMin a

  deleteMin E = error "empty heap"
  deleteMin (T E x b) = b
  deleteMin (T (T E x b) y c) = T b y c
  deleteMin (T (T a x b) y c) = T (deleteMin a) x (T b y c)
```

```
module PairingHeap (PairingHeap) where

import Heap

data PairingHeap a = E | T a [PairingHeap a]

mergePairs [] = E
mergePairs [h] = h
mergePairs (h1:h2:hs) = merge (merge h1 h2) (mergePairs hs)

instance Heap PairingHeap where
  empty = E
  isEmpty E = True
  isEmpty _ = False

  insert x h = merge (T x []) h
  merge h E = h
  merge E h = h
  merge h1@(T x hs1) h2@(T y hs2) =
    if x < y
    then T x (h2 : hs1)
    else T y (h1 : hs2)

  findMin E = error "empty heap"
  findMin (T x hs) = x

  deleteMin E = error "empty heap"
  deleteMin (T x hs) = mergePairs hs
```

```
module LazyPairingHeap (PairingHeap) where

import Heap

data PairingHeap a = E | T a (PairingHeap a) (PairingHeap a)
```

```
link (T x E m) a = T x a m
link (T x b m) a = T x E (merge (merge a b) m)

instance Heap PairingHeap where
  empty = E
  isEmpty E = True
  isEmpty _ = False

  insert x a = merge (T x E E) a
  merge a E = a
  merge E b = b
  merge a@(T x _ _) b@(T y _ _) = if x ≤ y
                                  then link a b
                                  else link b a

  findMin E = error "empty heap"
  findMin (T x a m) = x

  deleteMin E = error "empty heap"
  deleteMin (T x a m) = merge a m
```

```
module SkewBinomialHeap (SkewBinomialHeap) where

import Heap

data Tree a = Node Int a [a] [Tree a]
newtype SkewBinomialHeap a = SBH [Tree a]

rank (Node r x xs c) = r
root (Node r x xs c) = x

link t1@(Node r x1 xs1 c1) t2@(Node _ x2 xs2 c2) =
  if x1 ≤ x2
  then Node (r+1) x1 xs1 (t2 : c1)
  else Node (r+1) x2 xs2 (t1 : c2)

skewLink x t1 t2 =
  let Node r y ys c = link t1 t2
  in if x ≤ y
     then Node r x (y : ys) c
     else Node r y (x : ys) c

insTree t [] = [t]
insTree t ts@(t':ts') =
  if rank t < rank t' then t:ts else insTree (link t t') ts'

mrg ts1 [] = ts1
mrg [] ts2 = ts2
```

```
mrg ts1@(t1:ts'1) ts2@(t2:ts'2)
  | rank t1 < rank t2 = t1 : mrg ts'1 ts2
  | rank t2 < rank t2 = t2 : mrg ts1 ts'2
  | otherwise = insTree (link t1 t2) (mrg ts'1 ts'2)

normalize [] = []
normalize (t:ts) = insTree t ts

removeMinTree [] = error "empty heap"
removeMinTree [t] = (t, [])
removeMinTree (t: ts) = if root t < root t' then (t, ts) else (t', t:ts')
  where (t', ts') = removeMinTree ts

instance Heap SkewBinomialHeap where
  empty = SBH []
  isEmpty (SBH ts) = null ts

  insert x (SBH (t1:t2:ts))
    | rank t1 == rank t2 = SBH (skewLink x t1 t2:ts)
  insert x (SBH ts) = SBH (Node 0 x [] [] : ts)

  merge (SBH ts1) (SBH ts2) = SBH (mrg (normalize ts1) (normalize ts2))

  findMin (SBH ts) = root t
    where (t, _) = removeMinTree ts

  deleteMin (SBH ts) = foldr insert (SBH ts') xs
    where (Node _ x xs ts1, ts2) = removeMinTree ts
          ts' = mrg (reverse ts1) (normalize ts2)
```

```
module BootstrapHeap (BootstrapHeap) where

import Heap

data BootstrapHeap h a = E | H a (h (BootstrapHeap h a))

instance Eq a ⇒ Eq (BootstrapHeap h a) where
  (H x _) == (H y _) = (x == y)

instance Ord a ⇒ Ord (BootstrapHeap h a) where
  (H x _) ≤ (H y _) = (x ≤ y)

instance Heap h ⇒ Heap (BootstrapHeap h) where
  empty = E
  isEmpty E = True
  isEmpty _ = False

  insert x h = merge (H x empty) h
```

```
merge E h = h
merge h E = h
merge h1@(H x p1) h2@(H y p2) =
  if x ≤ y
  then H x (insert h2 p1)
  else H y (insert h1 p2)


findMin E = error "empty heap"
findMin (H x p) = x

deleteMin E = error "empty heap"
deleteMin (H x p) =
  if isEmpty p then E
  else let H y p1 = findMin p
           p2 = deleteMin p
       in H y (merge p1 p2)
```

A.7 Сортируемые коллекции

```
module Sortable (Sortable(..)) where

class Sortable s where
  empty :: Ord a ⇒ s a
  add   :: Ord a ⇒ a → s a → s a
  sort  :: Ord a ⇒ s a → [a]
```

```
module BottomUpMergeSort (MergeSort) where

import Sortable

data MergeSort a = MS Int [[a]]

mrg [] ys = ys
mrg xs [] = xs
mrg xs@(x:xs') ys@(y:ys') =
  if x ≤ y
  then x : mrg xs' ys
  else y : mrg xs  ys'

instance Sortable MergeSort where
  empty = MS 0 []

  add x (MS size segs) = MS (size+1) (addSeg [x] segs size)
    where addSeg seg segs size =
            if size `mod` 2 == 0 then seg : segs
            else addSeg (mrg seg (head segs)) (tail segs) (size `div` 2)
```

```
  sort (MS size segs) = foldl mrg [] segs
```

A.8 Множества

```
{-# LANGUAGE MultiParamTypeClasses #-}
module Set (Set(..)) where

class Set s a where
  empty  :: s a
  insert :: a → s a → s a
  member :: a → s a → Bool
```

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module UnbalancedSet (UnbalancedSet) where

import Set

data UnbalancedSet a = E | T (UnbalancedSet a) a (UnbalancedSet a)

instance Ord a ⇒ Set UnbalancedSet a where
  empty = E

  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

  insert x E = T E x E
  insert x s@(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
```

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module RedBlackSet (RedBlackSet) where

import Set

data Color = R | B
data RedBlackSet a = E | T Color (RedBlackSet a) a (RedBlackSet a)

balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance color a x b = T color a x b

instance Ord a ⇒ Set RedBlackSet a where
```

```
  empty = E

  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

  insert x s = T B a y b
    where ins E = T R E x E
          ins s@(T color a y b) =
            if x < y then balance color (ins a) y b
            else if x > y then balance color a y (ins b)
                 else s
          T _ a y b = ins s -- guaranteed to be non-empty
```

A.9 Конечные отображения

```
{-# LANGUAGE MultiParamTypeClasses #-}
module FiniteMap (FiniteMap(..)) where

class FiniteMap m k where
  empty  :: m k a
  bind   :: k → a → m k a → m k a
  lookup :: k → m k a → Maybe a
```

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module Trie (Trie) where

import Prelude hiding (lookup)
import FiniteMap

data Trie mk ks a = Trie (Maybe a) (mk (Trie mk ks a))

instance FiniteMap m k ⇒ FiniteMap (Trie (m k)) [k] where
  empty = Trie Nothing empty

  lookup [] (Trie b m) = b
  lookup (k:ks) (Trie b m) = lookup k m »= λm' → lookup ks m'

  bind [] x (Trie b m) = Trie (Just x) m
  bind (k:ks) x (Trie b m) =
    let t = case lookup k m of
              Just t  → t
              Nothing → empty
        t' = bind ks x t
    in Trie b (bind k t' m)
```

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module TrieOfTrees (Tree(..), Trie) where

import Prelude hiding (lookup)
import FiniteMap

data Tree a = E | T a (Tree a) (Tree a)
data Trie mk ks a = Trie (Maybe a) (mk (Trie mk ks (Trie mk ks a)))

instance FiniteMap m k ⇒ FiniteMap (Trie (m k)) (Tree k) where
  empty = Trie Nothing empty

  lookup E (Trie v m) = v
  lookup (T k a b) (Trie v m) =
    lookup k m »= λm' →
    lookup a m' »= λm'' →
    lookup b m''

  bind E x (Trie v m) = Trie (Just x) m
  bind (T k a b) x (Trie v m) =
    let tt = case lookup k m of
               Just tt → tt
               Nothing → empty
        t = case lookup a tt of
               Just t  → t
               Nothing → empty
        t' = bind b x t
        tt' = bind a t' tt
    in Trie v (bind k tt' m)
```

Литература

[Ada93] Stephen Adams. Efficient sets — a balancing act. *Journal of Functional Programming*, 3(4):553-561, October 1993.

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages*, pages 233–246, January 1995.

[And95] Arne Andersson. A note on searching in a binary search tree. *Software—Practice and Experience*, 21(10):1125-1128, October 1991.

[AVL62] Г. М. Адельсон-Вельский и Е. М. Ландис. Один алгоритм организации информации. *Доклады Академии Наук СССР*, 146:263–266, 1962.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BAG92] Amir Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617-648, July 1992.

[BC93] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171-190, April 1993.

[Bel57] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BH89] Bjor Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Conference on Functional Programming Languages and Computer Architecture*, pages 157–165, September 1989.

[BO96] Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, November 1996.

[Bro78] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7(3):298–319, August 1978.

[Bro95] Gerth Stølting Brodal. Fast meldable priority queues. In *Workshop on Algorithms and Data Structures*, volume 995 of *LNCS*, pages 282–290. Springer-Verlag, August 1995.

[Bro96] Gerth Stølting Brodal. Worst-case priority queues. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58, January 1996.

[BST95] Adam L. Buchsbaum, Rajamani Sundar, and Robert E. Tarjan. Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues. *SIAM Journal on Computing*, 24(6):1190–1206, December 1995.

[BT95] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. *Journal of Algorithms*, 18(3):513–547, May 1995.

[Buc93] Adam L. Buchsbaum. *Data-structural bootstrapping and catenable deques*. PhD thesis, Department of Computer Science, Princeton University, June 1993.

[Bur82] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

[But83] T. W. Butler. Computer response time and user performance. In *Conference on Human Factors in Computing Systems*, pages 58–62, December 1983.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall International, 1988.

[CG93] Tung-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289-298, June 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. Русский перевод: Т. Кормен, Ч. Лейзерсон, Р. Ривест. *Алгоритмы: построение и анализ*. Москва, МЦНМО, 2001.

[CM95] Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, September 1995.

[CMP88] Svante Carlsson, Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Scandinavian Workshop on Algorithm Theory*, volume 318 of *LNCS*, pages 1–13. Springer-Verlag, July 1988.

[Cra72] Clark Allan Crane. *Linear lists and priority queues as balanced binary trees.* PhD thesis, Computer Science Department, Stanford University, February 1972. Available as STAN-CS-72-259.

[CS96] Seonghun Cho and Sartaj Sahni. Weight biased leftist trees and modified skip lists. In *International Computing and Combinatorics Conference*, pages 361–370, June 1996.

[DGST88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343-1354, November 1988.

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *ACM Symposium on Theory of Computing*, pages 122–127, May 1982.

[Die89] Paul F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *LNCS*, pages 67–74. Springer-Verlag, August 1989.

[DR91] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 78–88, January 1991.

[DR93] Paul F. Dietz and Rajeev Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications. in *Workshop on Algorithms and Data Structures*, volume 709 of *LNCS*, pages 289–301. Springer-Verlag, August 1993.

[DS87] Paul F. Dietz and Danial D. Sleator. Two algorithms for maintaining order in a list. In *ACM Symposium on Theory of Computing*, pages 365–372, May 1987.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computing and System Sciences*, 38(1):86–124, February 1989.

[DST94] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, September 1994.

[FB97] Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 75–84, June 1997.

[FMR72] Patrick C. Fischer, Albert R. Meyer and Arnold L. Rosenberg. Real-time simulation of multihead tape units. *Journal of the ACM*, 19(4):590–607, October 1972.

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. K. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[FW76] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, pages 257–281, July 1976.

[GMPR77] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *ACM Symposium on Theory of Computing*, pages 49–60, May 1977.

[Gri81] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *IEEE Symposium on Foundations of Computer Science*, pages 8–21, October 1978.

[GT86] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, April 1986.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[HJ94] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. ... An experiment in software prototyping productivity, 1994.

[HM76] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *ACM Symposium on Principles of Programming Languages*, pages 95–103, January 1976.

[HM81] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2): 50–53, November 1981.

[Hoo82] Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs.* PhD thesis, Department of Computer Science, Cornell University, August 1982. (Cornell TR 82-503).

[Hoo92] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.

[HU73] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, December 1973.

[Hug85] John Hughes. Lazy memo functions. In *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 129–146. Springer-Verlag, September 1985.

[Hug86] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, March 1986.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[Joh86] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[Jos89] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68(1):105–111, October 1989.

[KD96] Anne Kaldewaij and Victor J. Dielissen. Leaf trees. *Science of Computer Programming*, 26(1–3):149–165, May 1996.

[Kin94] David J. King. Functional binomial queues. In *Glasgow Workshop on Functional Programming*, pages 141–150, September 1994.

[KL93] Chan Meng Khoong and Hon Wai Leong. Double-ended binomial queues. In *International Symposium on Algorithms and Computation*, volume 762 of *LNCS*, pages 128–137. Springer-Verlag, December 1993.

[Knu73a] Donald E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973. Русский перевод: Дональд Э. Кнут. *Искусство программирования. Том 3: Сортировка и поиск.* Вильямс, 2012.

[Knu73b] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1973. Русский перевод: Дональд Э. Кнут. *Искусство программирования. Том 2: Получисленные алгоритмы.* Вильямс, 2011.

[KT95] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *ACM Symposium on Theory of Computing*, pages 93–102, May 1995.

[KT96a] Haim Kaplan and Robert E. Tarjan. Purely functional lists with catenation via recursive slow-down. Draft revision of [KT95], August 1996.

[KT96b] Haim Kaplan and Robert E. Tarjan. Purely functional representation of catenable sorted lists. In *ACM Symposium on Theory of Computing*, pages 202–211, May 1996.

[KTU93] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 144–154, January 1993.

[Lia92] Andrew M. Liao. Three priority queue applications revisited. *Algorithmica*, 7(4):415–427, 1992.

[LS81] Benton L. Leong and Joel I. Seiferas. New real-time simulations of multihead tape units. *Journal of the ACM*, 28(1):166–180, January 1981.

[MEP96] Alistair Moffat, Gary Eddy, and Ola Petersson. Splaysort: Fast, versatile, practical. *Software—Practice and Experience*, 26(7):781–797, July 1996.

[Mic68] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

[MS91] Bernard M. E. Moret and Henry D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Workshop on Algorithms and Data Structures*, volume 519 of *LNCS*, pages 400–411. Springer-Verlag, August 1991.

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*, pages 409–423, April 1994.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer-Verlag, April 1984.

[Mye82] Eugene W. Myers. AVL dags. Technical Report TR82-9, Department of Computer Science, University of Arizona, 1982.

[Mye83] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.

[Mye84] Eugene W. Myers. Efficient applicative data types. In *ACM Symposium on Principles of Programming Languages*, pages 66–75, January 1984.

[NPP95] Manuel Núñez, Pedro Palao, and Ricardo Peña. A second year course on data structures based on functional programming. In *Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 65–84. Springer-Verlag, December 1995.

[Oka95a] Chris Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, October 1995.

[Oka95b] Chris Okasaki. Purely functional random-access lists. In *Conference on Functional Programming Languages and Computer Architecture*, pages 86–95, June 1995.

[Oka95c] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[Oka96a] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.

[Oka96b] Chris Okasaki. The role of lazy evaluation in amortized data structures. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–72, May 1996.

[Oka97] Chris Okasaki. Catenable double-ended queues. In *ACM SIGPLAN International Conference on Functional Programming*, pages 64–74, June 1997.

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–81, January 1994.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *LNCS*. Springer-Verlag, 1983.

[Pau96] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 2nd edition, 1996.

[Pet87] Gery L. Peterson. A balanced tree scheme for meldable heaps with updates. Technical Report GIT-ICS-87-23, School of Information and Computer Science, Georgia Institute of Technology, 1987.

[Pip96] Nicholas Pippinger. Pure versus impure Lisp. In *ACM Symposium on Principles of Programming Languages*, pages 104–109, January 1996.

[PPN96] Pedro Palao Gostanza, Ricardo Peña, and Manuel Nuñez. A new look at pattern matching in abstract data types. In *ACM SIGPLAN International Conference on Functional Programming*, pages 110–121, May 1996.

[Ram92] Rajeev Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Times.* PhD thesis, Department of Computer Sciences, University of Rochester, October 1992.

[Rea92] Chris M. P. Reade. Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming*, 18(2):181–204, April 1992.

[San90] David Sands. Complexity analysis for a lazy higher-order language. In *European Symposium on Programming*, volume 432 of *LNCS*, pages 361–376. Springer-Verlag, May 1990.

[San95] David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, August 1995.

[Sar86] Neil Sarnak. *Persistent Data Structures.* PhD thesis, Department of Computer Sciences, New York university, 1986.

[Sch92] Berry Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting.* PhD thesis, Eindhoven University of Technology, September 1992.

[Sch93] Berry Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45(1):41–50, January 1993.

[Sch97] Martin Schwenke. High-level refinement of random access data structures. In *Formal Methods Pacific*, pages 317–318, July 1997.

[SS90] Jörg-Rüdiger Sack and Thomas Strothotte. A characterization of heaps and its applications. *Information and Computation*, 86(1):69–86, May 1990.

[ST85] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[ST86a] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.

[ST86b] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, February 1986.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.

[Sto70] Hans-Jörg Stoß. K-band simulation von k-Kopf-Turingmachinen. *Computing*, 6(3):309–317, 1970.

[SV87] John T. Stasko and Jeffrey S. Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM*, 30(3):234–249, March 1987.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[TvL84] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, April 1984.

[Ull94] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Vui74] Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, December 1974.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978.

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, University of Oxford, September 1971.

[Wad87] Philip Wadler. Views: A way for pattern-matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 307–313, January 1987.

[Wad88] Philip Wadler. Strictness analysis aids time analysis. In *ACM Symposium on Principles of Programming Languages*, pages 119–132, January 1988.

[WV86] Christopher van Wyk and Jeffrey Scott Vitter. The complexity of hashing with lazy deletion. *Algorithmica*, 1(1):17–29, 1986.