

Функциональное программирование

Ричард Бёрд

Жемчужины проектирования алгоритмов

Функциональный подход

С примерами на языке Haskell



Ричард Бёрд

Жемчужины проектирования
алгоритмов: функциональный подход



Москва, 2013

Pearls of Functional Algorithm Design

Richard Bird
University of Oxford



CAMBRIDGE
UNIVERSITY PRESS

Жемчужины проектирования алгоритмов: функциональный подход

С примерами на языке Haskell

Ричард Бёрд
Оксфордский университет

Перевод с английского
В. Н. Брагилевского и А. М. Пеленицына



Москва, 2013

Pearls of Functional Algorithm Design

Richard Bird
University of Oxford



CAMBRIDGE
UNIVERSITY PRESS

Жемчужины проектирования алгоритмов: функциональный подход

С примерами на языке Haskell

Ричард Бёрд
Оксфордский университет

Перевод с английского
В. Н. Брагилевского и А. М. Пеленицына



Москва, 2013

УДК 004.021+004.421
ББК 32.973-018
Б11

Б11 Ричард Бёрд
Жемчужины проектирования алгоритмов: функциональный подход / Пер. с англ. В. Н. Брагилевского и А. М. Пеленицына. – М.: ДМК Пресс, 2013. — 330 с.: ил.
ISBN 978-5-94074-867-0

В этой книге Ричард Бёрд представляет принципиально новый подход к проектированию алгоритмов, а именно проектирование посредством формального вывода. Основное содержание книги разделено на 30 коротких глав, называемых жемчужинами, в каждой из которых решается конкретная программистская задача. Эти задачи, некоторые из них абсолютно новые, происходят из таких разнообразных источников, как игры и головоломки, захватывающие комбинаторные построения и более традиционные алгоритмы сжатия данных и сопоставления строк.

Каждая жемчужина начинается с постановки задачи, формулируемой на функциональном языке программирования Haskell, чрезвычайно мощном и в то же время лаконичном, позволяющем легко и просто выражать алгоритмические идеи. Новшество книги состоит в том, что каждое решение формально вычисляется из исходной постановки задачи посредством обращения к законам функционального программирования.

Издание предназначено для программистов, увлекающихся функциональным программированием, студентов, аспирантов и преподавателей, интересующихся принципами проектирования алгоритмов, а также всех, кто желает приобрести и развить навыки рассуждений в эквациональном стиле применительно к программам и алгоритмам.

УДК 004.021+004.421
ББК 32.973-018

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок всё равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несёт ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-521-51338-8 (англ.)
ISBN 978-5-94074-867-0 (рус.)

© 2010 Cambridge University Press
© Перевод на русский язык, оформление,
ДМК Пресс, 2013

Посвящается моей жене Норме.



Оглавление

<i>Предисловие</i>	9
1 Наименьшее отсутствующее число	12
2 Превосходная задача	19
3 Улучшаем седловой поиск	24
4 Задача о выборке	35
5 Сортировка попарных сумм	42
6 Делаем сотню	49
7 Строим дерево минимальной высоты	58
8 Распутываем жадные алгоритмы	68
9 Поиск знаменитостей	75
10 Удаляем повторы	84
11 Вовсе не максимальная сумма сегмента	94
12 Ранжируем суффиксы	101
13 Преобразование Барроуза–Уилера	115
14 Последний хвост	128
15 Все общие префиксы	139

16 Алгоритм Бойера—Мура	145
17 Алгоритм Кнута—Морриса—Пратта	156
18 Планирование в «Час пик»	166
19 Простой алгоритм решения судоку	178
20 Задача «Обратного отсчёта»	189
21 Хиломорфизмы и нексусы	202
22 Три способа вычисления определителей	215
23 Внутри выпуклой оболочки	224
24 Рациональное арифметическое кодирование	236
25 Целочисленное арифметическое кодирование	247
26 Алгоритм Шора—Вейта	262
27 Упорядоченная вставка	274
28 Бесцикловые функциональные алгоритмы	287
29 Алгоритм Джонсона—Троттера	298
30 Прядение паутины для чайников	306
<i>Предметный указатель</i>	326

Предисловие

В 1990 году, когда журнал *Journal of Functional Programming* (JFP) только планировался к выходу, бывшие тогда редакторами Саймон Пейтон Джонс (Simon Peyton Jones) и Филипп Вадлер (Philip Wadler) попросили меня взяться за постоянную колонку под названием *Функциональные жемчужины* (*Functional Pearls*). Их идея заключалась в подражании чрезвычайно успешной серии эссе Джона Бентли (Jon Bentley) «Жемчужины программирования», которые публиковались в 80-е годы в журнале *Communications of the ACM*. Вот что Бентли писал о своих жемчужинах:

Как настоящие жемчужины вырастают из песчинок, раздражающих устриц, так и жемчужины программирования вырастают из реальных задач, мучающих программистов. Эти программы интересны, к тому же они учат важным программистским приемам и фундаментальным принципам проектирования.

Мне кажется, что редакторы обратились именно ко мне, потому что я интересовался следующим подходом: я брал понятную, но неэффективную функциональную программу, выступавшую в роли спецификации к решаемой задаче, а затем посредством эквациональных рассуждений приходил к более эффективной версии. Одним из факторов, стимулирующих рост интереса к функциональным языкам в 90-е годы, было именно удобство применения эквациональных рассуждений. В самом деле, акроним в названии функционального языка GOFER, изобретённого Марком Джонсом (Mark Jones), отражает как раз эту идею. GOFER стал одним из языков, повлиявших на развитие языка Haskell, на котором основывается настоящая книга. Эквациональные рассуждения доминируют здесь над всем.

За последние 20 лет в JFP и изредка на таких конференциях как *International Conference of Functional Programming* (ICFP) и *Mathematics of Program Construction Conference* (MPC) появилось около 80 жемчужин. Из них я написал примерно четверть, большая же часть написана другими. Темы этих жемчужин включают интересные выводы программ, новые структуры данных, а также маленькие, но элегантные встроенные в Haskell и ML предметно-ориентированные языки для конкретных приложений.

Мне всегда были интересны алгоритмы и их проектирование. Отсюда название этой книги — *Жемчужины проектирования алгоритмов: функциональный подход* — вместо более общего *Функциональные жемчужины*. Многие, хотя и не все, жемчужины начинаются со спецификации на языке Haskell, а затем продолжаются выводом более эффективной реализации. При написании конкретно этих жемчужин мне хотелось выяснить, до какой степени проектирование алгоритма можно представить в виде традиционного в математике вычисления результата посредством применения общезвестных принципов, теорем и законов. Хотя в математике вычисления обычно производятся для упрощения сложных выражений, в проектировании алгоритмов получается в точности наоборот: простые, но неэффективные программы преобразуются в более эффективные, но, возможно, неочевидные. Жемчужиной является не полученная в итоге программа, а скорее процесс её получения. Остальные жемчужины, в части из них совсем немного преобразований, посвящены попыткам дать простые объяснения некоторым интересным и довольно хитроумным алгоритмам. Объяснение идей, лежащих в основе алгоритма, при функциональном подходе оказывается гораздо проще, чем при императивном: составляющие алгоритм функции легче отделить, они коротки и отражают мощные вычислительные шаблоны.

Жемчужины в этой книге, появлявшиеся прежде в JFP и других местах, были неоднократно отшлифованы. Собственно, многие не слишком сильно напоминают оригиналы. Даже при этом их несложно отшлифовать ещё лучше. Золотым стандартом красоты в математике считается книга Айгнера (Aigner) и Циглера (Ziegler) *Proofs from The Book* (третье издание, Springer, 2003), содержащая совершенные доказательства математических теорем. Я всегда рассматриваю эту книгу как идеал, к которому следует стремиться.

Примерно треть жемчужин новые. С некоторыми явно обозначенными исключениями жемчужины можно читать в любом порядке, хотя главы были упорядочены до некоторой степени по темам, таким как «разделяй и властвуй», жадные алгоритмы, полный перебор и т.п. В жемчужинах присутствует небольшое повторение материала, по большей части отно-

сящегося к свойствам используемых библиотечных функций или к более общим законам, таким как законы слияния для разнообразных свёрток. При необходимости читатель может обратиться к добавленному к книге короткому предметному указателю.

Наконец, в этой книге собраны идеи многих людей. Некоторые жемчужины были изначально написаны в сотрудничестве с другими авторами. Я хотел бы поблагодарить Шэрон Кёртис (Sharon Curtis), Джереми Гиббонаса (Jeremy Gibbons), Ральфа Хинзе (Ralf Hinze), Герэйнта Джонса (Geraint Jones) и Шин-Чень Му (Shin-Cheng Mu), моих соавторов, за щедрое разрешение переработать этот материал. Джереми Гиббонс прочитал окончательную версию черновика и сделал множество полезных предложений, направленных на улучшение изложения. Некоторые жемчужины досконально обсуждались на встречах исследовательской группы Algebra of Programming в Оксфорде. Хотя многие из недостатков и ошибок были исправлены, нет никаких сомнений в том, что при этом добавились новые. Помимо упомянутых ранее, я хотел бы выразить благодарность Стивену Дрейпу (Stephen Drape), Тому Харперу (Tom Harper), Дэниэлю Джеймсу (Daniel James), Джейфри Лейку (Jeffrey Lake), Мэнг Вонг (Meng Wang) и Николасу Ву (Nicholas Wu) за предложения по улучшению текста. Я также хотел бы поблагодарить Ламберта Мертенса (Lambert Meertens) и Уга де Мура (Oege de Moor) за плодотворное многолетнее сотрудничество. Наконец, я в долгу у Дэвида Транаха (David Tranah), моего редактора в издательстве Cambridge University Press, за содействие и поддержку, и в том числе за столь необходимые технические советы по подготовке окончательной версии.

Ричард Бёрд

1

Наименьшее отсутствующее число

Введение

Рассмотрим задачу отыскания наименьшего натурального числа, отсутствующего в заданном конечном множестве натуральных чисел X . Здесь мы имеем дело с упрощённой версией более общей программистской задачи, в которой числа соответствуют некоторым объектам, а X — множество объектов, используемых в настоящий момент. Задача заключается в том, чтобы найти некоторый неиспользуемый объект, например, с наименьшим именем.

Разумеется, решение задачи зависит от способа представления множества X . Если X задано списком без повторений, где элементы упорядочены в порядке возрастания, то решение очевидно: в последовательности элементов следует искать первый пропуск. Предположим, однако, что множество X задано списком различных чисел в произвольном порядке, например:

[08, 23, 09, 00, 12, 11, 01, 10, 13, 07, 41, 04, 14, 21, 05, 17, 03, 19]

Как бы вы стали искать наименьшее число, отсутствующее в этом списке?

Не сразу становится очевидным, что имеется решение линейной сложности, ведь невозможно выполнить сортировку произвольного числового списка за линейное время. Тем не менее, такое решение существует, и целью этой жемчужины будет описание двух возможных стратегий: одна из них основана на использовании массивов языка Haskell, а вторая на методе «разделяй и властвуй».

Решение с использованием массива

Определим спецификацию задачи с помощью функции *minfree*:

$$\begin{aligned} \text{minfree} &:: [\text{Nat}] \rightarrow \text{Nat} \\ \text{minfree } xs &= \text{head} ([0..] \setminus \setminus xs) \end{aligned}$$

Выражение $us \setminus \setminus vs$ обозначает список тех элементов us , которые остаются после удаления всех элементов vs :

$$\begin{aligned} (\setminus \setminus) &:: \text{Eq } a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \\ us \setminus \setminus vs &= \text{filter } (\notin vs) us \end{aligned}$$

Хотя функция *minfree* и работает, для списка длины n она требует в худшем случае $\Theta(n^2)$ операций. К примеру, чтобы получить результат вызова $\text{minfree } [n - 1, n - 2..0]$ понадобится вычислить $i \notin [n - 1, n - 2..0]$, где $0 \leq i \leq n$, поэтому в конечном итоге получится $n(n + 1)/2$ проверок на равенство.

Ключевой факт для обеих стратегий решения, с массивом и по методу «разделяй и властвуй», заключается в том, что в списке xs содержатся не все элементы из промежутка $[0.. \text{length } xs]$. Поэтому наименьшее число, отсутствующее в xs , является одновременно наименьшим числом, отсутствующим в $\text{filter } (\leq n) xs$, где $n = \text{length } xs$. Решение, основанное на массиве, использует этот факт для построения контрольного массива чисел, находящихся в $\text{filter } (\leq n) xs$. Контрольный массив с индексами от 0 до n содержит $n + 1$ логическое значение, инициализированное *False*. Для каждого элемента x из xs , такого, что $x \leq n$, элемент контрольного массива в позиции x устанавливается равным *True*. Наименьшее отсутствующее число находится после этого по первой же позиции, равной *False*. Таким образом, $\text{minfree} = \text{search} \cdot \text{checklist}$, где

$$\begin{aligned} \text{search} &:: \text{Array Int Bool} \rightarrow \text{Int} \\ \text{search} &= \text{length} \cdot \text{takeWhile id} \cdot \text{elems} \end{aligned}$$

Функция *search* принимает на вход массив логических значений, преобразует массив в список и возвращает длину наибольшего начального сегмента, содержащего только истинные элементы. Полученная длина и будет индексом первого вхождения значения *False*.

Одним из возможных вариантов определения функции *checklist* с линейной сложностью является использование функции *accumArray* из модуля *Data.Array* стандартной библиотеки языка Haskell. Тип этой функции несколько пугающий:

$$\text{Ix } i \Rightarrow (e \rightarrow v \rightarrow e) \rightarrow e \rightarrow (i, i) \rightarrow [(i, v)] \rightarrow \text{Array } i \ e$$

Ограничение $\text{Ix } i$ требует, чтобы типовая переменная i принадлежала классу Index , например, Int или Char , она используется для обозначения индексов или позиций в массиве. Первый аргумент это «суммирующая» функция, она перевычисляет элемент массива (типа e) на основе некоторого значения (типа v). Второй аргумент определяет начальное значение для элемента массива в очередной позиции. Третий аргумент это пара из начального и конечного индексов. Наконец, четвёртый аргумент это ассоциативный список пар индекс–значение. Функция accumArray строит массив, обрабатывая ассоциативный список слева направо, при этом она меняет определяемые очередным индексом элементы массива, присваивая им результаты вызова суммирующей функции для прежнего элемента и очередного значения из ассоциативного списка. Этот процесс выполняется за линейное по длине ассоциативного списка время в предположении, что суммированию достаточно константного.

Теперь функцию checklist можно реализовать как вызов функции accumArray :

```
checklist    :: [Int] → Array Int Bool
checklist xs = accumArray (∨) False (0, n)
                  (zip (filter (≤ n) xs) (repeat True))
where n = length xs
```

Эта реализация не требует, чтобы элементы списка xs не повторялись. Единственное ограничение в том, чтобы они были натуральными числами.

Интересно, что функцией accumArray можно воспользоваться для сортировки числового списка за линейное время при условии, что все его элементы принадлежат ограниченному диапазону $(0, n)$. Заменим checklist на countlist , где

```
countlist    :: [Int] → Array Int Int
countlist xs = accumArray (+) (0, n) (zip xs (repeat 1))
```

Теперь $\text{sort xs} = \text{concat} [\text{replicate } k \ x \mid (x, k) \leftarrow \text{countlist } xs]$. В сущности, применяя countlist вместо checklist , функцию minfree можно определить как позицию первого вхождения нулевого элемента.

Приведённая выше реализация строит массив за один проход, пользуясь умной библиотечной функцией. Более прозаичный способ реализовать функцию checklist состоит в явной отметке всех вхождений шаг за шагом с

применением операции присваивания с константной сложностью. В языке Haskell это возможно, только если обработка массива выполняется в подходящей монаде, например, в монаде с состоянием. В следующем примере реализация *checklist* использует модуль *Data.Array.ST*:

```
checklist xs =
  runSTArray (do
    { a ← newArray (0, n) False;
     sequence [writeArray a x True | x ← xs, x ≤ n];
     return a })
  where n = length xs
```

Впрочем, такое решение не удовлетворит чисто функционального программиста, поскольку оно эксплуатирует традиционную императивную парадигму, хотя и в функциональных одеждах.

Решение по методу «разделяй и властвуй»

Обратимся теперь к стратегии «разделяй и властвуй». Идея в том, чтобы выразить *minfree* (*xs* ++ *ys*) в терминах *minfree xs* и *minfree ys*. Запишем некоторые свойства операции \\ $\backslash\backslash$:

$$\begin{aligned}(as + bs) \backslash\backslash cs &= (as \backslash\backslash cs) + (bs \backslash\backslash cs) \\ as \backslash\backslash (bs + cs) &= (as \backslash\backslash bs) \backslash\backslash cs \\ (as \backslash\backslash bs) \backslash\backslash cs &= (as \backslash\backslash cs) \backslash\backslash bs\end{aligned}$$

Эти свойства аналогичны соответствующим свойствам теоретико-множественных операций, в которых объединение множеств \cup заменяется на $+$, а разность множеств \setminus на $\backslash\backslash$. Предположим теперь, что *as* и *us* не пересекаются, т.е. $as \backslash\backslash us = as$, и что *bs* и *vs* также не пересекаются, т.е. $bs \backslash\backslash vs = bs$. Из указанных свойств операций $+$ и $\backslash\backslash$ следует, что

$$(as + bs) \backslash\backslash (us + vs) = (as \backslash\backslash us) + (bs \backslash\backslash vs)$$

Выберем теперь некоторое натуральное число *b* и положим *as* = $[0..b-1]$ и *bs* = $[b..]$. Пусть, далее, *us* = *filter* ($< b$) *xs* и *vs* = *filter* ($\geq b$) *xs*. Тогда *as* и *us* окажутся непересекающимися, а значит, такими же будут *bs* и *vs*. Следовательно,

$$\begin{aligned}[0..] \backslash\backslash xs &= ([0..b-1] \backslash\backslash us) + ([b..] \backslash\backslash vs) \\ \text{where } (us, vs) &= \text{partition } (< b) xs\end{aligned}$$

Haskell обеспечивает эффективную реализацию функции *partition*, которая разбивает список на те элементы, которые удовлетворяют предикату *p*, и те, которые ему не удовлетворяют. Поскольку

$$\text{head} (xs ++ ys) = \text{if } \text{null} xs \text{ then head } ys \text{ else head } xs$$

получаем, по-прежнему для любого натурального числа *b*, что

$$\begin{aligned} \text{minfree } xs &= \text{if } \text{null} ([0..b-1] \setminus\setminus us) \\ &\quad \text{then head} ([b..] \setminus\setminus vs) \\ &\quad \text{else head} ([0..] \setminus\setminus us) \\ \text{where } (us, vs) &= \text{partition} (< b) xs \end{aligned}$$

Следующий вопрос: можно ли реализовать проверку $\text{null} ([0..b-1] \setminus\setminus us)$ более эффективно, чем прямым вычислением, которое требует квадратичного времени по длине *us*? Да, так как на входе список неповторяющихся натуральных чисел, таковым же является *us*, причём каждый элемент *us* меньше *b*. Поэтому

$$\text{null} ([0..b-1] \setminus\setminus us) \equiv \text{length } us == b$$

Заметим, что предыдущее решение не зависело от предположения, что данный список не содержит дубликатов, однако оно оказывается ключевым для эффективной реализации по методу «разделяй и властвуй».

Дальнейшее изучение кода *minfree* подсказывает, что следует обобщить *minfree* до функции, скажем, *minfrom*, которая определена так:

$$\begin{aligned} \text{minfrom} &:: \text{Nat} \rightarrow [\text{Nat}] \rightarrow \text{Nat} \\ \text{minfrom } a \, xs &= \text{head} ([a..] \setminus\setminus xs) \end{aligned}$$

где предполагается, что каждый элемент *x* больше или равен *a*. Тогда, при условии, что *b* выбрано таким образом, чтобы длины *us* и *vs* были меньше длины *xs*, следующее рекурсивное определение *minfree* оказывается вполне обоснованным:

$$\begin{aligned} \text{minfree } xs &= \text{minfrom } 0 \, xs \\ \text{minfrom } a \, xs &\mid \text{null } xs &= a \\ &\mid \text{length } us == b - a &= \text{minfrom } b \, vs \\ &\mid \text{otherwise} &= \text{minfrom } a \, us \\ \text{where } (us, vs) &= \text{partition} (< b) xs \end{aligned}$$

Остается выбрать *b*. Ясно, что нам нужно $b > a$. К тому же, хотелось бы иметь *b* таким, чтобы максимум из длин *us* и *vs* был настолько малым,

насколько это возможно. Правильный выбор b , удовлетворяющего указанным требованиям, таков:

$$b = a + 1 + n \text{ div } 2$$

где $n = \text{length } xs$. Если $n \neq 0$ и $\text{length } us < b - a$, то

$$\text{length } us \leq n \text{ div } 2 < n$$

И, если $\text{length } us = b - a$, то

$$\text{length } vs = n - n \text{ div } 2 - 1 \leq n \text{ div } 2$$

При таком выборе параметра b число операций $T(n)$, необходимое для вычисления $\text{minfrom } 0 \ xs$, где $n = \text{length } xs$, удовлетворяет рекуррентному соотношению $T(n) = T(n \text{ div } 2) + \Theta(n)$, следовательно, $T(n) = \Theta(n)$.

В качестве последней оптимизации можно избежать постоянного пересчёта длины, уточнив представление данных, а именно, заменив xs на пару $(\text{length } xs, xs)$. Это приводит нас к окончательному варианту программы

$$\begin{aligned} \text{minfree } xs &= \text{minfrom } 0 (\text{length } xs, xs) \\ \text{minfrom } a (n, xs) &\mid n == 0 = a \\ &\mid m == b - a = \text{minfrom } b (n - m, us) \\ &\mid \text{otherwise} = \text{minfrom } a (m, us) \\ &\quad \text{where } (us, vs) = \text{partition } (< b) xs \\ &\quad b = a + 1 + n \text{ div } 2 \\ &\quad m = \text{length } us \end{aligned}$$

Выясняется, что вышеприведённая программа примерно в два раза быстрее инкрементной программы на основе массива и на 20% быстрее, чем программа с использованием *accumArray*.

Заключительные замечания

Это была простая задача с как минимум двумя простыми решениями. Второе решение основывалось на известном методе проектирования алгоритмов, стратегии «разделяй и властвуй». Идея разбиения списка на те элементы, которые меньше заданного значения, и все остальные возникает во многих алгоритмах, например, в быстрой сортировке Хоара (Quicksort). При поиске алгоритма со сложностью $\Theta(n)$, использующего список из n

элементов, довольно заманчиво сразу обратиться к методу, обрабатывающему каждый элемент списка за константное или хотя бы амортизированное константное время. Однако рекурсивный процесс, который делает $\Theta(n)$ операций для сведения задачи к той же задаче не более чем половинного размера, также достаточно хороши.

Одно из различий между проектировщиками чисто функциональных и императивных алгоритмов состоит в том, что первые не предполагают существования массивов с операцией изменения за константное время, по крайней мере без некоторого объёма подготовительной работы. Для чисто функционального программиста операция изменения массива требует логарифмического по размеру массива времени¹. Это объясняет, почему иногда заметен логарифмический разрыв между функциональным и императивным решениями задачи. Но иногда, как здесь, этот разрыв при ближайшем рассмотрении исчезает.

¹По правде говоря, программисты-императивщики отлично знают, что константное время индексирования и изменения возможно только для маленьких массивов.

2

Превосходная задача

Введение

В этой жемчужине мы будем выполнять маленькое упражнение по программированию от Мартина Рема (Rem, 1988a). В то время как решение Рема использует бинарный поиск, наше будет ещё одним применением метода «разделяй и властвуй». Говорят, что некоторый элемент массива превосходит данный, если он больше и расположен правее, т.е. $x[j]$ превосходит $x[i]$, если $i < j$ и $x[i] < x[j]$. Числом превосходства (surpasser count) элемента массива называют количество элементов, его превосходящих. Например, вот числа превосходства для букв слова ТЕЛЕГРАФИСТ:

Т	Е	Л	Е	Г	Р	А	Ф	И	С	Т
1	6	4	5	5	3	4	0	2	1	0

Наибольшее число превосходства равно шести. Первое вхождение буквы Е имеет шесть превосходящих её элементов: буквы Л, Р, Ф, И, С и Т. Задача Рема заключается в вычислении наибольшего числа превосходства для массива длины $n > 1$ с помощью алгоритма со сложностью $O(n \log n)$.

Спецификация

Будем предполагать, что на входе вместо массива имеется список. Функция *tsc* (сокращение от *maximum surpasser count*) может быть специфицирована следующим образом:

$$\begin{aligned}
 msc &:: Ord a \Rightarrow [a] \rightarrow Int \\
 msc\ xs &= maximum [scount\ z\ zs \mid z : zs \leftarrow tails\ xs] \\
 scount\ x\ xs &= length (filter\ (x <) xs)
 \end{aligned}$$

Значение $scount\ xs$ это число превосходства элемента x относительно списка xs , функция $tails$ возвращает непустые хвосты непустого списка в порядке убывания их длин¹:

$$\begin{aligned}
 tails\ [] &= [] \\
 tails\ (x : xs) &= (x : xs) : tails\ xs
 \end{aligned}$$

Данное выше определение функции msc работает, но требует квадратичного времени.

Разделяй и властвуй

Учитывая заданную сложность $O(n \log n)$, кажется разумным обратиться к алгоритму по методу «разделяй и властвуй». Если нам удастся найти такую функцию $join$, что

$$msc\ (xs ++ ys) = join\ (msc\ xs)\ (msc\ ys)$$

и которую можно вычислить за линейное время, то временная сложность $T(n)$ алгоритма «разделяй и властвуй» на списке длины n будет удовлетворять соотношению $T(n) = 2T(n/2) + O(n)$, а значит, $T(n) = O(n \log n)$. Однако довольно-таки очевидно, что такой функции $join$ не существует: единственное число $msc\ xs$ предоставляет слишком мало информации для любого подобного разбиения.

Наименьшим обобщением будет начать с таблицы всех чисел превосходства:

$$table\ xs = [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ xs]$$

Тогда $msc = maximum \cdot map\ snd \cdot table$. Теперь посмотрим, сможем ли мы найти линейную $join$, удовлетворяющую следующему:

$$table\ (xs ++ ys) = join\ (table\ xs)\ (table\ ys)$$

Нам понадобится следующее свойство «разделяй и властвуй» для $tails$:

¹ В отличие от стандартной функции языка Haskell с тем же именем, которая возвращает возможно пустые хвосты возможно пустого списка.

$$\text{tails} (xs \uparrow\downarrow ys) = \text{map} (\uparrow\downarrow ys) (\text{tails} xs) \uparrow\downarrow \text{tails} ys$$

Проведём вычисления:

$$\begin{aligned}
 & \text{table} (xs \uparrow\downarrow ys) \\
 &= \{ \text{определение} \} \\
 & [(z, \text{scount} z zs) \mid z : zs \leftarrow \text{tails} (xs \uparrow\downarrow ys)] \\
 &= \{ \text{свойство «разделяй и властвуй» для tails} \} \\
 & [(z, \text{scount} z zs) \mid z : zs \leftarrow \text{map} (\uparrow\downarrow ys) (\text{tails} xs) \uparrow\downarrow \text{tails} ys] \\
 &= \{ \text{дистрибутивный закон для } \leftarrow \text{ по } \uparrow\downarrow \} \\
 & [(z, \text{scount} z (zs \uparrow\downarrow ys)) \mid z : zs \leftarrow \text{tails} xs] \uparrow\downarrow \\
 & [(z, \text{scount} z zs) \mid z : zs \leftarrow \text{tails} ys] \\
 &= \{ \text{так как } \text{scount} z (zs \uparrow\downarrow ys) = \text{scount} z zs + \text{scount} z ys \} \\
 & [(z, \text{scount} z zs + \text{scount} z ys) \mid z : zs \leftarrow \text{tails} xs] \uparrow\downarrow \\
 & [(z, \text{scount} z zs) \mid z : zs \leftarrow \text{tails} ys] \\
 &= \{ \text{определение функции table и } ys = \text{map fst} (\text{table} ys) \} \\
 & [(z, c + \text{scount} z (\text{map fst} (\text{table} ys))) \mid (z, c) \leftarrow \text{table} xs] \uparrow\downarrow \text{table} ys
 \end{aligned}$$

Следовательно, функцию *join* можно определить так:

$$\begin{aligned}
 \text{join} txs tys &= [(z, c + \text{tcount} z tys) \mid (z, c) \leftarrow txs] \uparrow\downarrow tys \\
 \text{tcount} z tys &= \text{scount} z (\text{map fst} tys)
 \end{aligned}$$

Однако проблема этого определения в том, что для вычисления *join txs tys* недостаточно линейного времени по длине *txs* и *tys*. Вычисление *tcount* можно было бы ускорить, если бы список *tys* был упорядочен по возрастанию первого компонента пары. В этом случае можно рассуждать так:

$$\begin{aligned}
 & \text{tcount} z tys \\
 &= \{ \text{определение tcount и scount} \} \\
 & \text{length} (\text{filter} (z <) (\text{map fst} tys)) \\
 &= \{ \text{так как } \text{filter } p \cdot \text{map } f = \text{map } f \cdot \text{filter} (p \cdot f) \} \\
 & \text{length} (\text{map fst} (\text{filter} ((z <) \cdot \text{fst}) tys)) \\
 &= \{ \text{так как } \text{length} \cdot \text{map } f = \text{length} \} \\
 & \text{length} (\text{filter} ((z <) \cdot \text{fst}) tys) \\
 &= \{ \text{так как } tys \text{ упорядочен по первому компоненту} \} \\
 & \text{length} (\text{dropWhile} ((z \geq) \cdot \text{fst}) tys)
 \end{aligned}$$

Таким образом,

$$tcount z tys = \text{length} (\text{dropWhile} ((z \geq) \cdot \text{fst}) tys) \quad (2.1)$$

Это вычисление подсказывает, что было бы полезно поддерживать *table* в порядке возрастания первого компонента:

$$\text{table } xs = \text{sort} [(z, scount z zs) | z : zs \leftarrow \text{tails } xs]$$

Повторяя приведённое выше вычисление для упорядоченной версии *table*, получим, что

$$\text{join } txs tys = [(x, c + tcount x tys) | (x, c) \leftarrow txs] \mathbin{\text{\texttt{M}}} tys \quad (2.2)$$

где $\mathbin{\text{\texttt{M}}}$ сливает два упорядоченных списка. Пользуясь этим определением, можно получить более эффективное рекурсивное определение *join*. Один из базовых случаев, $\text{join} [] tys = tys$, очевиден. Другой, а именно, $\text{join } txs [] = txs$, следует из того, что $tcount x [] = 0$. Рекурсивную часть можно упростить, сравнивая x и y :

$$\text{join } txs @ ((x, c) : txs') tys @ ((y, d) : tys') \quad (2.3)$$

В языке Haskell символ $@$ вводит синоним, так что txs это синоним для $(x, c) : txs'$, аналогично для tys . Используя (2.2), выражение (2.3) можно свести к

$$((x, c + tcount x tys) : [(x, c + tcount x tys) | (x, c) \leftarrow txs']) \mathbin{\text{\texttt{M}}} tys$$

Чтобы узнать, какой элемент будет выдан операцией $\mathbin{\text{\texttt{M}}}$ первым, нужно сравнить x и y . Если $x < y$, то это элемент слева и, поскольку согласно (2.1) $tcount x tys = \text{length } tys$, выражение (2.3) сводится к

$$(x, c + \text{length } tys) : \text{join } txs' tys$$

Если $x = y$, необходимо сравнить $c + tcount x tys$ и d . Но $d = tcount x tys'$ по определению *table*, а $tcount x tys = tcount x tys'$ согласно (2.1). Поэтому (2.3) сводится к $(y, d) : \text{join } txs tys'$. Такой же результат будет получен и в оставшемся случае $x > y$.

Собирая всё вместе и добавляя к *join* во избежание перевычисления длины дополнительный аргумент *length tys*, приходим к следующему алгоритму вычисления *table* на основе метода «разделяй и властвуй»:

$$\begin{aligned} \text{table } [x] &= [(x, 0)] \\ \text{table } xs &= \text{join} (m - n) (\text{table } ys) (\text{table } zs) \end{aligned}$$

```

where m      = length xs
      n      = m div 2
      (ys, zs) = splitAt n xs

join 0 txs [] = txs
join n [] tys = tys
join n txs@((x, c) : txs') tys@((y, d) : tys')
| x < y = (x, c + n) : join n txs' tys
| x ≥ y = (y, d) : join (n - 1) txs tys'

```

Так как *join* выполняется за линейное время, список *table* вычисляется за $O(n \log n)$ операций, а значит такова же сложность *msc*.

Заключительные замечания

Невозможно вычислить *table* с помощью алгоритма, более быстрого, чем $O(\log n)$. Причина в том, что если *xs* это список без повторяющихся элементов, то *table xs* предоставляет достаточно информации для определения такой сортирующей перестановки списка *xs*. Более того, никакие последующие сравнения между элементами списка не требуются. Фактически *table xs* соотносится с таблицей инверсий для перестановки из *n* элементов; см. Кнута (Knuth, 1998): *table xs* это просто таблица инверсий для *reverse xs*. Так как основанная на сравнениях сортировка *n* элементов требует $\Omega(n \log n)$ операций, то столько же требует и вычисление *table*.

Как было сказано во введении, решение Рема (Rem, 1988b) отличается тем, что оно использует итеративный алгоритм с применением бинарного поиска. Программист-императивщик также мог бы обратиться к алгоритму «разделяй и властвуй», но он, вероятно, предпочёл бы алгоритм с обработкой массива просто потому, что он требует меньше памяти.

Литература

- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*, second edition. Reading, MA: Addison-Wesley. [Имеется русский перевод: Кнут Д. Искусство программирования, том 3. Сортировка и поиск. 2-е изд. М.: Вильямс. 2007.]
- Rem, M. (1988a). Small programming exercises 20. *Science of Computer Programming* **10** (1), 99–105.
- Rem, M. (1988b). Small programming exercises 21. *Science of Computer Programming* **10** (3), 319–25.

3

Улучшаем седловой поиск

Действие происходит на занятии по проектированию функциональных алгоритмов. В классе четверо студентов: Анна, Иван, Мария и Фёдор.

Учитель. Доброе утро, ребята. Сегодня я бы хотел, чтобы вы спроектировали функцию *invert* с двумя параметрами: функция f , действующая из множества пар натуральных чисел во множество натуральных чисел, и натуральное число z . Значение $\text{invert } f \ z$ должно быть списком всех пар (x, y) , удовлетворяющих равенству $f(x, y) = z$. Вы можете предполагать только то, что f строго возрастает по каждому из аргументов, и ничего иного.

Иван. Кажется, это простая задача. Так как f действует на натуральных числах и возрастает по каждому из аргументов, то мы знаем, что из равенства $f(x, y) = x$ следует, что $x \leq y$ и $y \leq z$. Поэтому можно определить *invert* простым поиском всех возможных пар значений:

$$\text{invert } f \ z = [(x, y) \mid x \leftarrow [0..z], y \leftarrow [0..z], f(x, y) == z]$$

Разве это не решение?

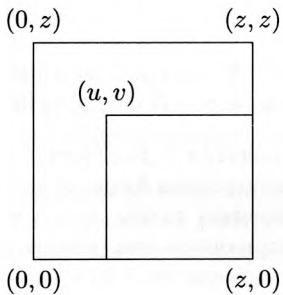
Учитель. Да, это решение, но оно требует $(z + 1)^2$ вычислений значения функции f . Поскольку вычисление значения f может оказаться очень долгим, я бы хотел увидеть решение, требующее как можно меньшее количество вызовов f .

Фёдор. Ну, нетрудно уменьшить количество вычислений вдвое. Так как при условии, что f возрастает, $f(x, y) \geq x + y$, то поиск можно проводить только по тем значениям, которые расположены на или ниже диагонали квадрата:

$$\text{invert } f z = [(x, y) \mid x \leftarrow [0..z], y \leftarrow [0..z-x], f(x, y) == z]$$

А если задуматься, то обе верхние границы можно вообще заменить на $z - f(0, 0)$ и $z - x - f(0, 0)$. Тогда, если $z < f(0, 0)$, то поиск сразу останавливается.

Анна. В предположении, что порядок найденных решений не важен, я думаю, результат можно ещё улучшить. Метод Ивана выполняет поиск от начала координат в левом нижнем углу квадрата размера $z + 1$, обходя его столбец за столбцом. Мы же можем ускориться, если начнём в левом верхнем углу квадрата в точке $(0, z)$. На каждом этапе область поиска ограничена прямоугольником с левым верхним углом в точке (u, v) и правым нижним углом в точке $(z, 0)$. Вот картинка:



Давайте определим функцию

$$\text{find } (u, v) f z = [(x, y) \mid x \leftarrow [u..z], y \leftarrow [v, v-1..0], f(x, y) == z]$$

Таким образом, $\text{invert } f z = \text{find } (0, z) f z$. Теперь нетрудно найти более эффективную реализацию функции find .

Во-первых, если $u > z$ или $v < 0$, то, очевидно, $\text{find } (u, v) f z = []$. В противном случае будем рассматривать различные возможные значения $f(u, v)$. Если $f(u, v) < z$, то остаток столбца u можно отбросить, так как при $v' < v$ имеет место неравенство $f(u, v') < f(u, v) < z$. Если $f(u, v) > z$, то аналогично отбрасываем остаток строки v . Наконец, если $f(u, v) = z$, то можно запомнить (u, v) и отбросить как столбец u , так и строку v .

Вот улучшенная версия функции *invert*:

```
invert f z = find (0, z) f z
find (u, v) f z
| u > z ∨ v < 0 = []
| z' < z = find (u + 1, v) f z
| z' == z = (u, v) : find (u + 1, v - 1) f z
| z' > z = find (u, v - 1) f z
where z' = f(u, v)
```

В худшем случае, если *find* обходит периметр квадрата от левого верхнего угла до правого нижнего, она делает $2z + 1$ вызов функции *f*. В лучшем случае, если *find* идёт напрямую либо к нижней, либо к правой границе, ей требуется лишь $z + 1$ вызов.

Фёдор. Ты можешь сократить область поиска ещё сильнее, потому что исходный квадрат с левым верхним углом $(0, z)$ и правым нижним углом $(z, 0)$ является слишком избыточной оценкой области нахождения требуемого значения. Предположим, что мы предварительно вычислили такие *m* и *n*, что

$$\begin{aligned} m &= \text{maximum} (\text{filter} (\lambda y \rightarrow f(0, y) \leq z) [0..z]) \\ n &= \text{maximum} (\text{filter} (\lambda x \rightarrow f(x, 0) \leq z) [0..z]) \end{aligned}$$

Тогда можно определить $\text{invert } f z = \text{find}(0, m) f z$, где *find* имеет в точности ту же форму, в какой её определила Анна, за тем исключением, что первое охранное выражение становится таким: $u > n \vee v < 0$. Другими словами, вместо того, чтобы анализировать весь квадрат $(z + 1) \times (z + 1)$, можно ограничиться квадратом размера $(m + 1) \times (n + 1)$.

Решающее соображение состоит в том, что *m* и *n* можно вычислить бинарным поиском. Пусть *g* — возрастающая функция на множестве натуральных чисел, предположим также, что *x*, *y* и *z* удовлетворяют условию $g z \leq z < g y$. Для отыскания единственного значения *m*, где $m = \text{bsearch } g (x, y) z$, на отрезке $x \leq m < y$ и с учётом $g m \leq z < g (m + 1)$ можно использовать инварианты $g a \leq z < g b$ и $x \leq a < b \leq y$. Таким образом, получаем функцию

```
bsearch g (a, b) z
| a + 1 == b = a
| g m \leq z = bsearch g (m, b) z
| otherwise = bsearch g (a, m) z
where m = (a + b) div 2
```

Так как $a + 1 < b \Rightarrow a < m < y$, то ни $g\ x$, ни $g\ y$ алгоритмом не вычисляются, поэтому они могут быть фиктивными значениями. В частности, имеем

$$\begin{aligned} m &= \text{bsearch } (\lambda y \rightarrow f(0, y)) (-1, z + 1) z \\ n &= \text{bsearch } (\lambda x \rightarrow f(x, 0)) (-1, z + 1) z \end{aligned}$$

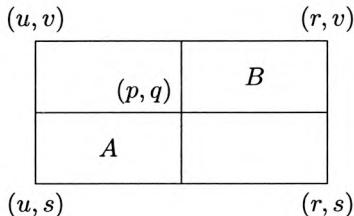
где f дополнена фиктивными значениями $f(0, -1) = 0$ и $f(-1, 0) = 0$.

Этой версии *invert* требуется примерно $2 \log z + m + n$ вызовов f в худшем случае и $2 \log z + m \min n$ в лучшем. Так как m и n могут оказаться значительно меньше z , к примеру, если $f(x, y) = 2^x + 3^y$, то мы получаем алгоритм, требующий в худшем случае только $O(\log z)$ операций.

Учитель. Мои поздравления, Анна и Фёдор, вы переоткрыли важную стратегию поиска, названную Дэвидом Грисом (David Gries) седловым поиском; см. работы Бэкхауза (Backhouse, 1986), Дейкстры (Dijkstra, 1985) и Гриса (Gries, 1981). Мне представляется, что Грис назвал его так, потому что форма трёхмерного графика f с наименьшим элементом в левом нижнем углу, наибольшим в правом верхнем и двумя крыльями напоминает седло. Основная идея, как заметила Анна, в том, чтобы начать поиск на кончике одного из крыльев, а не в точках с наименьшим или наибольшим значением. Рассматривая эту задачу, Дейкстра (Dijkstra, 1985) также указал на преимущества использования логарифмического поиска для определения подходящего начального прямоугольника.

Мария. А что если мы попробуем решение по методу «разделяй и властвуй»? В смысле, почему бы в начале не смотреть на элемент в середине прямоугольника? Уверена, что имеет смысл применить двумерный аналог бинарного поиска.

Предположим, мы ограничили поиск прямоугольником с левым верхним углом (u, v) и правым нижним (r, s) . Почему бы вместо того, чтобы смотреть на $f(u, v)$, не проанализировать $f(p, q)$, где $p = (u + r) \text{ div } 2$, а $q = (v + s) \text{ div } 2$? Вот картинка:



Если $f(p, q) < z$, то мы можем отбросить все элементы левого нижнего прямоугольника A . Аналогично, если $f(p, q) > z$, можно отбросить правый верхний прямоугольник B . Если же $f(p, q) = z$, то можно отбросить оба.

Я знаю, что эта стратегия нарушает свойство, предложенное Анной, о том, что область поиска всегда прямоугольник. Вместо этого получаем либо два прямоугольника, либо Г-образную форму. Но мы же функциональные программисты и можем не ограничивать себя простыми циклами: алгоритм «разделяй и властвуй» столь же лёгок в реализации, что и итеративный, потому что оба можно выразить рекурсией.

Иван. Тебе всё равно придётся иметь дело с Г-образной формой. Конечно, её можно разбить на два прямоугольника. Фактически это можно сделать двумя способами: либо горизонтальным разрезом, либо вертикальным. Позволь мне дать грубую оценку. Рассмотрим прямоугольник размера $m \times n$ и обозначим $T(m, n)$ количество требуемых для поиска по нему вычислений f . Если $m = 0$ или $n = 0$, то искать нечего. Если $m = 1$ или $n = 1$, имеем

$$\begin{aligned} T(1, n) &= 1 + T(1, \lceil n/2 \rceil) \\ T(m, 1) &= 1 + T(\lceil m/2 \rceil, 1) \end{aligned}$$

В противном случае, если $m \geq 2$ и $n \geq 2$, можно отбросить прямоугольник размера, как минимум, $\lfloor m/2 \rfloor \times \lfloor n/2 \rfloor$. Если мы делаем горизонтальный разрез, то остаётся два прямоугольника: один размера $\lfloor m/2 \rfloor \times \lceil n/2 \rceil$, а второй $\lceil m/2 \rceil \times n$. Таким образом,

$$T(m, n) = 1 + T(\lfloor m/2 \rfloor, \lceil n/2 \rceil) + T(\lceil m/2 \rceil, n)$$

Если же мы делаем вертикальный разрез, то имеем

$$T(m, n) = 1 + T(\lceil m/2 \rceil, \lfloor n/2 \rfloor) + T(m, \lceil n/2 \rceil)$$

Я, правда, не вижу решения этих рекуррентных соотношений.

Фёдор. Если сделать одновременно и горизонтальный, и вертикальный разрезы, то останется три прямоугольника, так что при $m \geq 2$ и $n \geq 2$ будем иметь

$$T(m, n) = 1 + T(\lceil m/2 \rceil, \lfloor n/2 \rfloor) + T(\lceil m/2 \rceil, \lceil n/2 \rceil) + T(\lfloor m/2 \rfloor, \lceil n/2 \rceil)$$

Я могу решить это соотношение. Положим $U(i, j) = T(2^i, 2^j)$, теперь

$$\begin{aligned} U(i, 0) &= i \\ U(0, j) &= j \\ U(i+1, j+1) &= 1 + 3U(i, j) \end{aligned}$$

Решением будет $U(i, j) = 3^k(|j - i| + 1/2) - 1/2$, где $k = \lfloor i \min j \rfloor$, это можно проверить индукцией. Следовательно, если $m \leq n$, то

$$T(m, n) \leq 3^{\log m} \log(2n/m) = m^{1.59} \log(2n/m)$$

Это лучше, чем $m + n$, при условии, что m значительно меньше n .

Иван. Я не думаю, что решение с тремя прямоугольниками столь же хорошее, что и с двумя. Следуя твоему подходу, Фёдор, давай положим $U(i, j) = T(2^i, 2^j)$. В предположении, что $i \leq j$, выполняя горизонтальный разрез, получаем

$$\begin{aligned} U(0, j) &= j \\ U(i+1, j+1) &= 1 + U(i, j) + U(i, j+1) \end{aligned}$$

Решением является $U(i, j) = 2^i(j - i/2 + 1) - 1$, что также можно проверить индукцией. Поэтому

$$T(m, n) \leq m \log(2n/\sqrt{m})$$

Если $i \geq j$, то следует делать вертикальный разрез вместо горизонтального. Тогда мы получим алгоритм, требующий самое большое $n \log(2m/\sqrt{n})$ вызовов f . В любом случае, если один из параметров m или n значительно меньше другого, то мы имеем алгоритм лучше, чем при седловом поиске.

Анна. Пока вы двое решали рекуррентные соотношения, я думала над проблемой нижней границы сложности для функции *invert*. Рассмотрим различные возможные результаты при поиске в прямоугольнике размера $m \times n$. Предположим, что имеется $A(m, n)$ различных решений. Каждая проверка $f(x, y)$ относительно z допускает три возможных результата, поэтому высота h тернарного дерева проверок должна удовлетворять условию $h \geq \log_3 A(m, n)$. Считая, что $A(m, n)$ достижимо, получаем нижнюю границу для числа проверок, которые необходимо выполнить. Ситуация похожа на сортировку n элементов двоичными сравнениями: имеется $n!$ возможных результатов, поэтому любой алгоритм сортировки должен в худшем случае провести как минимум $\log_2 n!$ сравнений.

$A(m, n)$ достичь нетрудно: каждый список пар (x, y) , где $0 \leq x < n$ и $0 \leq y < m$, удовлетворяющих $f(x, y) = z$, находится в соотношении один к одному со ступенчатой функцией, график которой идёт из левого верхнего угла прямоугольника размера $m \times n$ в его правый нижний угол. Значение z появляется на внутренних углах ступенек. Разумеется, этот график

необязательно задаёт именно тот путь, по которому идёт функция *find*. Количество таких путей $\binom{m+n}{n}$, а значит, таким же будет значение $A(m, n)$.

По другому этот результат можно получить, если предположить, что имеется k решений. Значение z можно разместить в m строках k раз ровно $\binom{m}{k}$ способами, и всякий раз будет $\binom{n}{k}$ возможных вариантов для столбцов. Следовательно,

$$A(m, n) = \sum_{k=0}^m \binom{m}{k} \binom{n}{k} = \binom{m+n}{n}$$

поскольку это суммирование является примером свёртки Вандермонда; см. (Graham, 1989b). Беря логарифмы, получаем нижнюю границу

$$\log A(m, n) = \Omega(m \log(1 + n/m) + n \log(1 + m/n))$$

Это приближение показывает, что если $m = n$, то не удастся сделать менее, чем $\Omega(m + n)$ шагов. Но если $m \leq n$, то $m \leq n \log(1 + m/n)$, поскольку $x \leq \log(1+x)$ при $0 \leq x \leq 1$. Таким образом, $A(m, n) = \Omega(m \log(n/m))$. Решение Ивана не слишком близко к этой границе, потому что в его алгоритме в случае $m \leq n$ число операций оценивается как $O(m \log(n/\sqrt{m}))$.

Мария. Я не думаю, что в решении Ивана действительно необходимо применять метод «разделяй и властвуй». Есть и другие способы использовать бинарный поиск. Например, можно просто выполнить m бинарных поисков, по одному для каждой строки. Так мы получим решение со сложностью $O(m \log n)$. Но мне кажется, что её можно улучшить, достигнув оптимальную асимптотическую сложность $O(m \log(n/m))$ в предположении, что $m \leq n$.

Допустим, как и прежде, что поиск ограничен прямоугольником с левым верхним углом (u, v) и правым нижним (r, s) . Таким образом, имеется $r - u$ столбцов и $s - v$ строк. Далее, предположим, что $v - s \leq r - u$, т.е. столбцов как минимум столько же, сколько и строк. Пусть мы выполняем бинарный поиск по средней строке, $q = (v + s) \text{ div } 2$, и ищем такое p , что $f(p, q) \leq z < f(p + 1, q)$. Если $f(p, q) < z$, то нужно продолжать поиск только в двух прямоугольниках $((u, v), (p, q + 1))$ и $((p + 1, q - 1), (r, s))$. Если $f(p, q) = z$, то можно вырезать столбец p и продолжить поиск в прямоугольниках $((u, v), (p - 1, q + 1))$ и $((p + 1, q - 1), (r, s))$. Рассуждения в случае, когда строк больше, чем столбцов, аналогичны. В результате, можно отбросить половину элементов массива с помощью логарифмического числа испытаний.

Вот алгоритм, который я имею в виду: реализуем *invert*

```

 $find(u, v)(r, s) f z$ 
|  $u > r \vee v < s = []$ 
|  $v - s \leq r - u = rfind(bsearch(\lambda x \rightarrow f(x, q))(u - 1, r + 1) z)$ 
| otherwise = cfind(bsearch(\lambda y \rightarrow f(p, y))(s - 1, v + 1) z)
where
   $p = (u + r) \text{ div } 2$ 
   $q = (v + s) \text{ div } 2$ 
   $rfind p = (\text{if } f(p, q) == z \text{ then } (p, q) : find(u, v)(p - 1, q + 1) f z$ 
            | else  $find(u, v)(p, q + 1) f z$ ) +
            |  $find(p + 1, q - 1)(r, s) f z$ 
   $cfind q = find(u, v)(p - 1, q + 1) f z +$ 
            |  $(\text{if } f(p, q) == z \text{ then } (p, q) : find(p + 1, q - 1)(r, s) f z$ 
            | else  $find(p + 1, q)(r, s) f z$ )

```

Рис. 3.1: пересмотренное определение функции *find*

```

 $invert f z = find(0, m)(n, 0) f z$ 
where  $m = bsearch(\lambda y \rightarrow f(0, y))(-1, z + 1) z$ 
       $n = bsearch(\lambda x \rightarrow f(x, 0))(-1, z + 1) z$ 

```

где $find(u, v)(r, s) f z$, приведённая на рис. 3.1, выполняет поиск по прямоугольнику с левым верхним углом (u, v) и правым нижним углом (r, s) .

Что же касается анализа, пусть $T(m, n)$ снова обозначает количество вызовов, необходимых для поиска по прямоугольнику $m \times n$. Предположим, что $m \leq n$. В лучшем случае, когда каждый бинарный поиск по строке возвращает самый левый или самый правый элемент, имеем $T(m, n) = \log n + T(m/2, n)$, т.е. $T(m, n) = O(\log m \times \log n)$. В худшем случае, когда бинарный поиск возвращает средний элемент, получается

$$T(m, n) = \log n + 2T(m/2, n/2)$$

Чтобы решить это соотношение, положим $U(i, j) = T(2^i, 2^j)$. Тогда имеем

$$U(i, j) = \sum_{k=0}^{i-1} 2^k(j - k) = O(2^i(j - i))$$

Следовательно, $T(m, n) = O(m \log(n/m))$, что асимптотически оптимально согласно полученной Анной нижней оценке.

Учитель. Отличная работа, все четверо! Удивительно, что примерно за 25 лет, в которые седловой поиск использовался в качестве примера формального построения программ, никто, как кажется, не заметил, что это алгоритм с не самой лучшей асимптотикой.

Послесловие

Реальная история, стоящая за этой жемчужиной, такова. Я решил использовать седловой поиск в качестве упражнения для кандидатов, проходящих собеседование на поступление в Оксфорд. Им давался двумерный числовой массив, возрастающий вдоль каждой строки и каждого столбца, и предлагалось найти систематический способ обнаружения всех вхождений заданного числа. Я хотел убедить их, что поиск из левого верхнего угла в правый нижний является хорошей стратегией. Однако те кандидаты, который изучали в школе информатику, пытались использовать бинарный поиск либо из середины каждой строки, либо из центра прямоугольника. Полагая, что седловой поиск является золотым стандартом для решения данной задачи, я намекал им, что следует поменять ход мысли. Только спустя некоторое время я заинтересовался тем, могли ли они быть правы.

Помимо описания нового алгоритма для старой задачи, как мне кажется, есть два других методологических аспекта, достойных упоминания. Во-первых, формальное построение программ испытывает сильное влияние со стороны доступных в целевом языке программирования методов вычисления. Последнее решение, приведённое Марией, вряд ли кто-то назовёт элегантным, тем не менее, оно достаточно элементарно в ситуации, когда рекурсия и конкатенация списков являются базовыми операциями, однако его реализация в языке с массивами и циклами может оказаться довольно затруднительной. Во-вторых, как совершенно уверены проектировщики алгоритмов, формальное построение программ должно сопровождаться идеями относительно возможных путей улучшения эффективности. Такие идеи могут возникнуть частично благодаря решению рекуррентных соотношений, а частично из определения нижних границ.

Впервые эта жемчужина появилась в (Bird, 2006). Один из рецензентов исходной работы указал следующее:

Сложная реализация помимо собственно эффективности приносит и некоторые накладные расходы, которые так часто упускают из виду при анализе, подобном представленному в этой работе. Если автор действительно хочет убедить нас, что

Таблица 3.1. Количество вызовов f_i

Алгоритм	f_0	f_1	f_2	f_3	f_4
Анна	7501	5011	6668	5068	9989
Фёдор	2537	38	1749	157	5025
Мария	121	42	445	181	134

Таблица 3.2. Абсолютное время выполнения

Алгоритм	f_0	f_1	f_2	f_3	f_4
Анна	0.42	0.40	0.17	0.15	0.54
Фёдор	0.06	0.01	0.05	0.01	0.15
Мария	0.01	0.01	0.02	0.02	0.01

его алгоритм лучше, чем алгоритм Гриса, то ему следует предъявить нам очевидные доказательства. Пусть он запустит алгоритм для конкретной функции на конкретных значениях и сравнит результаты.

В таблицах 3.1 и 3.2 приводятся требуемые доказательства. Почти случайно были выбраны пять функций:

$$\begin{aligned}f_0(x, y) &= 2^y(2x + 1) - 1 \\f_1(x, y) &= x2^x + y2^y + 2x + y \\f_2(x, y) &= 3x + 27y + y^2 \\f_3(x, y) &= x^2 + y^2 + x + y \\f_4(x, y) &= x + 2^y + y - 1\end{aligned}$$

В таблице 3.1 указаны точные количества вызовов f_i , необходимые для вычисления $\text{invert } f_i$ 5000, с использованием исходной реализации седлового поиска, принадлежащей Анне, версии Фёдора (с бинарным поиском для вычисления границ) и окончательной версии Марии. В таблице 3.2 сравнивается время выполнения в секундах для интерпретатора GHCi. Близкое соответствие таблиц показывает, что количество вызовов хорошо отражает абсолютное время выполнения.

Литература

- Backhouse, R. (1986). *Program Construction and Verification*. International Series in Computer Science. Prentice Hall.
- Improving saddleback search: a lesson in algorithm design. *Mathematics of Program Construction*, LNCS 4014, pp. 82–9.
- Dijkstra, E. W. (1985). The saddleback search. EWD-934. <http://www.cs.utexas.edu/users/EWD/index09xx.html>.
- Gries, D. (1981). *The Science of Programming*. Springer-Verlag. [Имеется русский перевод: Грис Д. Наука программирования. М.: Мир. 1984.]
- Graham, R. L., Knuth, D. E. and Patashnik, O. (1989). *Concrete Mathematics*. Reading, MA: Addison-Wesley. [Имеется русский перевод: Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики. М.: Мир, Бином. Лаборатория знаний. 2006.]

4

Задача о выборке

Введение

Пусть X и Y — два конечных непересекающихся множества с элементами некоторого упорядоченного типа, общее количество которых превосходит k . Рассмотрим задачу отыскания k -го наименьшего элемента объединения $X \cup Y$. По определению, k -й наименьший элемент множества это элемент, для которого существуют в точности k элементов, меньших его, так что нулевой наименьший элемент это просто наименьший элемент множества. Сколько времени потребуется на выполнение такого вычисления?

Разумеется, ответ зависит от способа представления X и Y . Если оба эти множества заданы отсортированными списками, то достаточно $O(|X| + |Y|)$ операций. Два списка можно слить в один за линейное время, а k -й наименьший можно найти в k -й позиции объединённого списка за последующие $O(k)$ операций. На самом деле общее время можно считать равным $O(k)$, поскольку в объединённом списке достаточно вычислить только первые $k + 1$ элементов. Однако, если два исходных множества заданы отсортированными массивами, то, как мы покажем ниже, время можно сократить до $O(\log |X| + \log |Y|)$ операций. Эта граница зависит от наличия функции доступа к элементам за константное время. Та же граница достижима, если оба множества X и Y заданы сбалансированными бинарными деревьями поиска, и это несмотря на то, что два таких дерева невозможно слить менее чем за линейное время.

Быстрый алгоритм даёт ещё один пример применения метода «разделяй и властвуй», а доказательство его корректности основывается на

некоторой специальной связи между слиянием и выборкой. Наша цель в рамках этой жемчужины в том, чтобы разобраться в этой связи, получить основанный на списках алгоритм по методу «разделяй и властвуй», а затем реализовать его для списков, представленных массивами.

Формализация и первые шаги

В терминах двух упорядоченных непересекающихся списков xs и ys задача заключается в вычислении

$$\begin{aligned} \text{smallest} &:: \text{Ord } a \Rightarrow \text{Int} \rightarrow ([a], [a]) \rightarrow a \\ \text{smallest } k \ (xs, ys) &= \text{union} \ (xs, ys) !! k \end{aligned}$$

Значением операции $xs !! k$ является элемент списка xs , который находится в k -й позиции, считая от нуля. Функция $\text{union} :: \text{Ord } a \Rightarrow ([a], [a]) \rightarrow [a]$, сливающая два непересекающихся списка, каждый из которых возрастает, определяется так:

$$\begin{aligned} \text{union} \ (xs, []) &= xs \\ \text{union} \ ([], ys) &= ys \\ \text{union} \ (x : xs, y : ys) &\mid x < y = x : \text{union} \ (xs, y : ys) \\ &\mid x > y = y : \text{union} \ (x : xs, ys) \end{aligned}$$

Наша цель состоит в получении алгоритма для smallest по методу «разделяй и властвуй», поэтому для $!!$ и union нам нужны некоторые правила разбиения. Для первой операции, обозначая $\text{length } xs$ через $|xs|$, имеем

$$(xs ++ ys) !! k = \text{if } k < |xs| \text{ then } xs !! k \text{ else } ys !! (k - |xs|) \quad (4.1)$$

Очевидное доказательство опущено. Для union имеется следующее свойство. Предположим, что $xs ++ ys$ и $us ++ vs$ это такие отсортированные непересекающиеся списки, что

$$\text{union} \ (xs, vs) = xs ++ vs \text{ и } \text{union} \ (us, ys) = us ++ ys$$

Другими словами, в xs не существует элемента, превосходящего или равного какому-либо элементу vs ; аналогично для us и ys . Тогда

$$\text{union} \ (xs ++ ys, us ++ vs) = \text{union} \ (xs, ys) ++ \text{union} \ (ys, vs) \quad (4.2)$$

Полезно переписать (4.2), используя инфиксный символ \cup вместо union :

$$(xs ++ ys) \cup (us ++ vs) = (xs \cup us) ++ (ys \cup vs)$$

Сравните это свойство с похожим тождеством¹ для разности списков \\ $:$

$$(xs \text{ ++ } ys) \setminus\setminus (us \text{ ++ } vs) = (xs \setminus\setminus us) \text{ ++ } (ys \setminus\setminus vs)$$

которое имеет место, если $xs \setminus\setminus vs = xs$ и $ys \setminus\setminus us = ys$.

Свойство (4.2) для ++ и \cup , надеемся, достаточно очевидно для того, чтобы ссылку на него можно было опускать в формальном доказательстве.

В дальнейшем, условие $\text{union}(xs, ys) = xs \text{ ++ } ys$ будет сокращённо обозначаться как $xs \triangleleft ys$. Таким образом, $xs \triangleleft ys$, если для любого элемента x из xs и y из ys выполняется $x < y$. Заметим, что если ограничиться только непустыми списками, то отношение \triangleleft оказывается транзитивным.

Разделяй и властвуй

В этом разделе мы выполним разбиение выражения

$$\text{smallest } k (xs \text{ ++ } [a] \text{ ++ } ys, us \text{ ++ } [b] \text{ ++ } vs)$$

Мы будем иметь дело со случаем $a < b$, поскольку случай $a > b$ полностью аналогичен. Ключевой момент состоит в том, что в силу возрастания элементов всех списков, если $a < b$, то $(xs \text{ ++ } [a]) \triangleleft ([b] \text{ ++ } vs)$.

Предположим сначала, что $k < |xs \text{ ++ } [a] \text{ ++ } us|$, или, эквивалентно, $k \leq |xs \text{ ++ } us|$. Рассуждаем:

$$\begin{aligned} & \text{smallest } k (xs \text{ ++ } [a] \text{ ++ } ys, us \text{ ++ } [b] \text{ ++ } vs) \\ &= \{ \text{определение} \} \\ &= \{ \text{выбираем такие } ys_1 \text{ и } ys_2, \text{ что } ys = ys_1 \text{ ++ } ys_2, \\ &\quad (xs \text{ ++ } [a] \text{ ++ } ys_1) \triangleleft ([b] \text{ ++ } vs) \text{ и } us \triangleleft ys_2 \} \\ &= \{ \text{union}(xs \text{ ++ } [a] \text{ ++ } ys_1 \text{ ++ } ys_2, us \text{ ++ } [b] \text{ ++ } vs) !! k \} \\ &= \{ \text{свойство (4.2), выбор } ys_1 \text{ и } ys_2 \} \\ &= \{ \text{union}(xs \text{ ++ } [a] \text{ ++ } ys_1, us) \text{ ++ } \text{union}(ys_2, [b] \text{ ++ } vs) !! k \} \\ &= \{ \text{используем (4.1) и предположение, что } k < |xs \text{ ++ } [a] \text{ ++ } us| \} \\ &= \{ \text{union}(xs \text{ ++ } [a] \text{ ++ } ys_1, us) !! k \} \\ &= \{ \text{снова используем (4.1)} \} \\ &= \{ \text{union}(xs \text{ ++ } [a] \text{ ++ } ys_1, us) \text{ ++ } \text{union}(ys_2, []) !! k \} \\ &= \{ \text{снова свойство (4.2), поскольку } xs \text{ ++ } [a] \text{ ++ } ys_1 \triangleleft [] \} \end{aligned}$$

¹Использовано в главе 1: «Наименьшее отсутствующее число».

$$\begin{aligned}
 & \text{union} (xs ++ [a] ++ ys_1 ++ ys_2, us ++ []) !! k \\
 = & \quad \{ \text{определение } ys \text{ и } \text{smallest} \} \\
 & \text{smallest } k (xs ++ [a] ++ ys, us)
 \end{aligned}$$

Теперь предположим, что $k \geq |xs ++ [a] ++ us|$. Симметричная аргументация даёт нам следующее:

$$\begin{aligned}
 & \text{smallest } k (xs ++ [a] ++ ys, us ++ [b] ++ vs) \\
 = & \quad \{ \text{определение } \} \\
 & \text{union} (xs ++ [a] ++ ys, us ++ [b] ++ vs) !! k \\
 = & \quad \{ \text{выбираем такие } us_1 \text{ и } us_2, \text{ что } us = us_1 ++ us_2, \\
 & \quad us_1 \triangleleft ys \text{ и } (xs ++ [a]) \triangleleft (us_2 ++ [b] ++ vs) \} \\
 & \text{union} (xs ++ [a] ++ ys, us_1 ++ us_2 ++ [b] ++ vs) !! k \\
 = & \quad \{ \text{свойство (4.2), выбор } us_1 \text{ и } us_2 \} \\
 & (\text{union} (xs ++ [a], us_1) ++ \text{union} (ys, us_2 ++ [b] ++ vs)) !! k \\
 = & \quad \{ \text{используем (4.1) и предположение, что } k \geq |xs ++ [a] ++ us| \} \\
 & \text{union} (ys, us_2 ++ [b] ++ vs) !! (k - |xs ++ [a] ++ us|) \\
 = & \quad \{ \text{снова используем (4.1)} \} \\
 & (\text{union} ([], us_1) ++ \text{union} (us, us_2 ++ [b] ++ vs)) !! (k - |xs ++ [a]|) \\
 = & \quad \{ \text{как прежде} \} \\
 & \text{smallest} (k - |xs ++ [a]|) (ys, us ++ [b] ++ vs)
 \end{aligned}$$

Подводя итог, при $a < b$ имеем

$$\begin{aligned}
 & \text{smallest } k (xs ++ [a] ++ ys, us ++ [b] ++ vs) \\
 | k \leq p + q & = \text{smallest } k (xs ++ [a] ++ ys, us) \\
 | k > p + q & = \text{smallest} (k - p - 1) (ys, us ++ [b] ++ vs) \\
 & \text{where } (p, q) = (\text{length } xs, \text{length } us)
 \end{aligned}$$

Совершенно аналогичные рассуждения в случае $a > b$ дают

$$\begin{aligned}
 & \text{smallest } k (xs ++ [a] ++ ys, us ++ [b] ++ vs) \\
 | k \leq p + q & = \text{smallest } k (xs, us ++ [b] ++ vs) \\
 | k > p + q & = \text{smallest} (k - q - 1) (xs ++ [a] ++ ys, vs) \\
 & \text{where } (p, q) = (\text{length } xs, \text{length } us)
 \end{aligned}$$

Для завершения алгоритма для *smallest* по методу «разделяй и властвуй» нам остаётся рассмотреть базовый случай, когда один из данных списков

является пустым. Это несложно, поэтому в результате получаем следующую программу:

```

smallest k ([] , ws) = ws !! k
smallest k (zs , []) = zs !! k
smallest k (zs , ws) =
  case (a < b , k ≤ p + q) of
    (True , True) → smallest k (zs , us)
    (True , False) → smallest (k - p - 1) (ys , ws)
    (False , True) → smallest k (xs , ws)
    (False , False) → smallest (k - q - 1) (zs , vs)
  where p = (length zs) div 2
        q = (length ws) div 2
        (xs , a : ys) = splitAt p zs
        (us , b : vs) = splitAt q ws

```

Время выполнения $\text{smallest } k (xs, ys)$ линейно по длинам списков xs и ys , поэтому алгоритм «разделяй и властвуй» не быстрее спецификации. Выгода возникает, когда xs и ys даются как отсортированные массивы, а не списки. В этом случае программу можно модифицировать так, чтобы чтобы она выполнялась за логарифмическое по размеру массивов время. Вместо того, чтобы постоянно разбивать два списка, всё можно будет реализовать с помощью индексов в массиве. Скажем точнее, список xs представлен массивом xa и двумя индексами (lx, rx) в соответствии с условием $xs = map (x!) [lx .. rx - 1]$, где (!) — это операция индексирования массива в модуле языка Haskell *Data.Array*. Этот модуль предоставляет эффективные операции над неизменяемыми массивами, массивами, которые создаются за один проход. В частности, (!) выполняется за константное время. Список xs можно преобразовать в массив xa , индексируемый с нуля, так:

$$xa = listArray (0, length xs - 1) xs$$

Теперь можно определить

```

smallest          :: Int → (Array Int a, Array Int a) → a
smallest k (xa, ya) = search k (0, m + 1) (0, n + 1)
  where (0, m) = bounds xa
        (0, n) = bounds ya

```

Функция bounds возвращает верхнюю и нижнюю границы массива, проиндексированного здесь с нуля. Наконец, функция search , локальная по

```

search k (lx, rx) (ly, ry)
| lx == rx    = ya ! k
| ly == ry    = xa ! k
| otherwise   = case (xa ! mx < ya ! my, k ≤ mx + my) of
                  (True, True) → search k (lx, rx) (ly, my)
                  (True, False) → search (k - mx - 1) (mx, rx) (ly, ry)
                  (False, True) → search k (lx, mx) (ly, ry)
                  (False, False) → search (k - my - 1) (lx, rx) (my, ry)
                  where mx = (lx + rx) div 2; my = (ly + ry) div 2

```

Рис. 4.1: определение функции *search*

отношению к *smallest*, поскольку она обращается к массивам *xa* и *ya*, приводится на рис. 4.1.

На каждом рекурсивном вызове имеется константный объём работы, кроме того, каждый вызов сокращает вдвое один или другой из двух промежутков, поэтому время выполнения *search* логарифмическое.

Заключительные замечания

Хотя мы сформулировали эту задачу в терминах непересекающихся множеств, представленных списками, элементы которых возрастают, существуют и другие вариации, в которых списки не обязательно непересекающиеся и должны лишь не убывать. Такие списки представляют мульти-множества (*multiset* или *bag*). Рассмотрим вычисление *merge*(*xs*, *ys*)!!*k*, где *merge* сливает два списка в порядке возрастания, т.е. *merge* = *uncurry* λ :

<i>merge</i> ([], <i>ys</i>)	=	<i>ys</i>
<i>merge</i> (<i>xs</i> , [])	=	<i>xs</i>
<i>merge</i> (<i>x</i> : <i>xs</i> , <i>y</i> : <i>ys</i>)		<i>x</i> ≤ <i>y</i> = <i>x</i> : <i>merge</i> (<i>xs</i> , <i>y</i> : <i>ys</i>)
		<i>x</i> ≥ <i>y</i> = <i>y</i> : <i>merge</i> (<i>x</i> : <i>xs</i> , <i>ys</i>)

Таким образом, *merge* определяется так же, как и *union*, за тем исключением, что $<$ и $>$ заменены на \leq и \geq . Разумеется, результат теперь не обязательно *k*-й наименьший элемент объединённого списка. Более того, при условии, что мы заменим $<$ на \trianglelefteq , где *xs* \trianglelefteq *ys*, если *merge*(*xs*, *ys*) = *xs* ++ *ys*, или, эквивалентно, если *x* ≤ *y* для всех *x* из *xs* и *y* из *ys*, то показанный вы-

ше вывод останется действительным для случаев, когда неравенства $a < b$ и $a > b$ ослаблены до $a \leq b$ и $a \geq b$.

И последнее замечание: эта жемчужина изначально появилась под другим названием в (Bird, 1997). Но не смотрите туда, потому что там я переусердствовал со связью между слиянием и выборкой. Позднее Джереми Гиббонс в (Gibbons, 1997) указал на гораздо более простой способ действий, и именно его вывод приведён выше.

Литература

Bird, R. S. (1997). On merging and selection. *Journal of Functional Programming* 7 (3), 349–54.

Gibbons, J. (1997). More on merging and selection. Technical Report CMS-TR-97-08, Oxford Brookes University, UK.

5

Сортировка попарных сумм

Введение

Пусть A это некоторое линейно упорядоченное множество, а операция $(\oplus) :: A \rightarrow A \rightarrow A$ это такая монотонная бинарная операция на A , что $x \leqslant x' \wedge y \leqslant y' \Rightarrow x \oplus y \leqslant x' \oplus y'$. Рассмотрим задачу вычисления

$$\begin{aligned}sortsums &:: [A] \rightarrow [A] \rightarrow [A] \\sortsums\ xs\ ys &= sort [x \oplus y \mid x \leftarrow xs, y \leftarrow ys]\end{aligned}$$

Сколько операций потребует вычисление $sortsums\ xs\ ys$, если считать только сравнения и предполагать, что длина как xs , так и ys равна n ?

Разумеется, $O(n^2 \log n)$ сравнений достаточно. Всего имеется n^2 сумм, а сортировка списка длины n^2 может быть выполнена с использованием $O(n^2 \log n)$ сравнений. Эта верхняя оценка не зависит от факта монотонности \oplus . На самом деле без дополнительной информации относительно \oplus и A эта граница является одновременно и нижней. Допущение о монотонности \oplus уменьшает лишь константный множитель, не изменяя при этом асимптотической сложности.

Но предположим теперь, что об операции \oplus и множестве A мы знаем больше: а именно, что (\oplus, A) является абелевой группой. Таким образом, операция \oplus ассоциативна и коммутативна, имеется единица e и такая операция $negate :: A \rightarrow A$, что $x \oplus negate\ x = e$. Вооружившись этой дополнительной информацией, Жан-Люк Ламбер (Lambert, 1992) доказал, что $sortsums$ можно вычислить за $O(n^2)$ сравнений. Однако его алгоритм также требует $Cn^2 \log n$ дополнительных операций, где C довольно велико.

Поставленный впервые около 35 лет назад в (Harper *et al.*, 1975) вопрос о том, можно ли уменьшить общее количество операций до $O(n^2)$ сравнений и $O(n^2)$ остальных действий, остаётся открытым.

Алгоритм Ламбера это ещё один искусственный пример применения метода «разделяй и властвуй». Наша цель в этой жемчужине — представить его основные идеи и привести реализацию на языке Haskell.

Алгоритм Ламбера

Давайте для начала докажем, что если единственным допущением относительно (\oplus) является её монотонность, то нижней границей будет $\Omega(n^2 \log n)$. Предположим, что списки xs и ys упорядочены по возрастанию, и рассмотрим матрицу размера $n \times n$:

$$[[x \oplus y \mid y \leftarrow ys] \mid x \leftarrow xs]$$

Каждые строка и столбец матрицы также возрастают. Эта матрица является примером стандартной диаграммы Юнга, и, как следует из Теоремы Н раздела 5.1.4 (Knuth, 1998) имеется в точности

$$E(n) = (n^2)! / \left(\frac{(2n-1)!}{(n-1)!} \frac{(2n-2)!}{(n-2)!} \cdots \frac{n!}{0!} \right)$$

способов присвоить значения от 1 до n^2 элементам матрицы, а значит, ровно $E(n)$ возможных перестановок, которые упорядочивают входные данные. Используя тот факт, что $\log E(n) = \Omega(n^2 \log n)$, заключаем, что требуется как минимум такое количество сравнений.

Теперь о сути упражнения. Алгоритм Ламбера основывается на двух простых фактах. Определим операцию вычитания $(\ominus) :: A \rightarrow A \rightarrow A$ соотношением $x \ominus y = x \oplus \text{negate } y$. Тогда

$$x \oplus y = x \ominus \text{negate } y \tag{5.1}$$

$$x \ominus y \leqslant x' \ominus y' \equiv x \ominus x' \leqslant y \ominus y' \tag{5.2}$$

Проверить (5.1) несложно, а вот для проверки (5.2), что мы оставляем в качестве упражнения, понадобится применить все свойства абелевой группы. Как результат, равенство (5.1) утверждает, что задачу упорядоченных сумм можно свести к задаче упорядоченных вычитаний, а эквивалентность (5.2) говорит о том, что последняя задача в свою очередь может быть сведена к задаче упорядоченных вычитаний над одним списком.

Вот как используются свойства (5.1) и (5.2). Рассмотрим список $\text{subs } xs \ ys$ помеченных вычитаний, определённый так:

```
subs      :: [A] → [A] → [Label A]
subs xs ys = [(x ⊖ y, (i, j)) | (x, i) ← zip xs [1..], (y, j) ← zip ys [1..]]
```

где *Label a* это синоним для $(a, (\text{Int}, \text{Int}))$. Таким образом, каждый терм вида $x \ominus y$ помечен позициями x в *xs* и y в *ys*. Информация из меток понадобится позднее. Свойство (5.1) даёт первый факт:

```
sortsums xs ys = map fst (sortsubs xs (map negate ys))
sortsubs xs ys = sort (subs xs ys)
```

Суммы сортируются по связанным с ними помеченым вычитаниям, после чего метки отбрасываются.

Следующий шаг заключается в использовании (5.2) для того, чтобы показать, как вычислить *sortsubs xs ys* за квадратичное число сравнений. Построим список *table*:

```
table          :: [A] → [A] → [(Int, Int, Int)]
table xs ys    = map snd (map (tag 1) xxs  $\text{M}$  map (tag 2) yys)
                  where xxs = sortsubs xs xs
                        yys = sortsubs ys ys
tag i (x, (j, k)) = (x, (i, j, k))
```

Операция M сливает два упорядоченных списка. Словом, *table* строится слиянием двух упорядоченных списков *xxs* и *yys*, элементы которых перед тем помечаются номером исходного списка, что позволяет определить, из какого из списков пришёл конкретный элемент результирующего списка. Согласно (5.2) *table* содержит достаточно информации, чтобы *sortsubs xs ys* можно было вычислить без сравнений элементов из *A*. Предположим, к примеру, что разность $x \ominus y$ имеет метку (i, j) , а $x' \ominus y'$ имеет метку (k, ℓ) . В этом случае $x \ominus y \leqslant x' \ominus y'$ тогда и только тогда, когда элемент $(1, i, k)$ списка *table* предшествует элементу $(2, j, \ell)$. Поэтому никакие дополнительные сравнения элементов *A* помимо тех, что уже использованы при построении *table*, не требуются.

Для реализации этой идеи нам нужно уметь быстро вычислять информацию о предшествовании. Проще всего это делать, если сконвертировать *table* в массив языка Haskell:

```
mkArray xs ys = array b (zip (table xs ys) [1..])
                  where b = ((1, 1, 1), (2, p, p))
                        p = max (length xs) (length ys)
```

Функция *mkArray* использует модуль *Data.Array* языка Haskell. Первый аргумент *b* функции *array* это пара границ, наименьший и наибольший

```

sortsums xs ys = map fst (sortsubs xs (map negate ys))
sortsubs xs ys = sortBy (cmp (mkArray xs ys)) (subs xs ys)
subs xs ys = [(x ⊖ y, (i, j)) | (x, i) ← zip xs [1..], (y, j) ← zip ys [1..]]
cmp a (x, (i, j)) (y, (k, ℓ)) = compare (a ! (1, i, k)) (a ! (2, j, ℓ))
mkArray xs ys = array b (zip (table xs ys) [1..])
  where b = ((1, 1, 1), (2, p, p))
        p = max (length xs) (length ys)
table xs ys = map snd (map (tag 1) xxs ⋀ map (tag 2) yys)
  where xxs = sortsubs' xs
        yys = sortsubs' ys
tag i (x, (j, k)) = (x, (i, j, k))

```

Рис. 5.1: полный код функции *sortsums* за исключением *sortsubs'*

индексы массива. Второй аргумент — это ассоциативный список пар индекс–значение. В этом представлении $(1, i, k)$ предпстествует $(2, j, \ell)$ в списке *table*, если $a ! (1, i, k) < a ! (2, j, \ell)$, где $a = mkArray xs ys$. Операция индексирования массива (!) выполняется за константное время, поэтому проверка предшествования также выполняется за константное время. Теперь *sortsubs xs ys* можно вычислить, воспользовавшись вспомогательной функцией *sortBy*:

```

sortsubs xs ys = sortBy (cmp (mkArray xs ys)) (subs xs ys)
cmp a (x, (i, j)) (y, (k, ℓ))
  = compare (a ! (1, i, k)) (a ! (2, j, ℓ))

```

Функция *compare* принадлежит классу типов *Ord*. В частности, $sort = sortBy compare$, а $(\text{M}) = mergeBy compare$. Определение *sortBy* в терминах *mergeBy* по методу «разделяй и властвуй» мы опустим.

Полученная к данному моменту программа приведена на рис. 5.1. Она не содержит пока только определения *sortsubs'*, где $sortsubs' xs = sortsubs xs xs$. Однако это определение нельзя использовать непосредственно в *sortsums*, потому что в этом случае рекурсия не была бы вполне обоснованной (well founded). Хотя вычисление *sortsubs xs ys* и требует $O(mn \log mn)$ операций, никакие сравнения элементов *A* кроме тех, что нужны для построения *table*, не выполняются. А для *table* необходимо только $O(m^2 + n^2)$ сравнений плюс те сравнения, что нужны для построения

sortsubs' xs и *sortsubs' ys*. Остаётся только показать, как вычислить *sortsubs'* за квадратичное число сравнений.

Разделяй и властвуй

Забывая на мгновение о метках и записывая *xs* \ominus *ys* вместо [*x* \ominus *y* | *x* \leftarrow *xs*, *y* \leftarrow *ys*], укажем на ключевой для алгоритма по методу «разделяй и властвуй» факт — тождество:

$$\begin{aligned} & (xs \dagger ys) \ominus (xs \dagger ys) \\ &= (xs \ominus xs) \dagger (xs \ominus ys) \dagger (ys \ominus xs) \dagger (ys \ominus ys) \end{aligned}$$

Следовательно, чтобы отсортировать список слева, достаточно отсортировать и слить четыре списка справа. Наличие меток слегка затрудняет применение алгоритма «разделяй и властвуй», потому что метки необходимо соответствующим образом корректировать. В помеченной версии это тождество выглядит следующим образом:

$$\begin{aligned} & subs (xs \dagger ys) (xs \dagger ys) \\ &= subs xs xs \dagger map (incr m) (subs xs ys) \dagger \\ & \quad map (incl m) (subs ys xs) \dagger map (incb m) (subs ys ys) \end{aligned}$$

где *m* = *length xs* и

$$\begin{aligned} incl m (x, (i, j)) &= (x, (m + i, j)) \\ incr m (x, (i, j)) &= (x, (i, m + j)) \\ incb m (x, (i, j)) &= (x, (m + i, m + j)) \end{aligned}$$

Чтобы вычислить *sortsubs' ws*, мы разбиваем *ws* на две равные половины *xs* и *ys*. Списки *sortsubs' xs* и *sortsubs' ys* вычисляются рекурсивно. Список *sortsubs' xs ys* вычисляется согласно алгоритму из предыдущего раздела. Таким же образом можно было бы вычислить и *sortsubs' ys xs*, но вместо этого мы просто обратим *sortsubs' xs ys*, заменив элементы на противоположные:

$$\begin{aligned} sortsubs' ys xs &= map switch (reverse (sortsubs' xs ys)) \\ switch (x, (i, j)) &= (negate x, (j, i)) \end{aligned}$$

Программа для *sortsubs'* приведена на рис. 5.2. Число сравнений *C(n)*, необходимое для вычисления *sortsubs'* для списка длины *n*, удовлетворяет рекуррентному соотношению *C(n)* = *2C(n/2)* + *O(n²)*, решение которого *C(n)* = *O(n²)*. Значит, *sortsums* можно вычислить за *O(n²)* сравнений. Однако общее время *T(n)* удовлетворяет соотношению

```

 $sortsubs' [] = []$ 
 $sortsubs' [w] = [(w \ominus w, (1, 1))]$ 
 $sortsubs' ws = foldr1 (\wedge) [xss, map (incr m) xys,$ 
 $map (incl m) yxs, map (incb m) yys]$ 
where  $xss = sortsubs' xs$ 
 $yxs = sortBy (cmp (mkArray xs ys)) (subs xs ys)$ 
 $yys = map switch (reverse xys)$ 
 $ys = sortsubs' ys$ 
 $(xs, ys) = splitAt m ws$ 
 $m = length ws \text{ div } 2$ 
 $incl m (x, (i, j)) = (x, (m + i, j))$ 
 $incr m (x, (i, j)) = (x, (i, m + j))$ 
 $incb m (x, (i, j)) = (x, (m + i, m + j))$ 
 $switch (x, (i, j)) = (negate x, (j, i))$ 

```

Рис. 5.2: код функции *sortsubs'*

$T(n) = 2T(n/2) + O(n^2 \log n)$, а его решение $T(n) = O(n^2 \log n)$. Логарифмический множитель можно было бы из $T(n)$ удалить, если бы удалось вычислить *sortBy* стр *стр* за квадратичное время, но этот результат остаётся недостижимым. В любом случае, дополнительная сложность, возникающая из замены сравнений на другие операции, делает алгоритм очень неэффективным на практике.

Заключительные замечания

Задача сортировки попарных сумм участвует как Задача №41 в Проекте открытых задач (Demaine *et al.*, 2009), веб-ресурсе, предназначенном для регистрации нерешённых проблем, которые представляют интерес для исследователей в рамках вычислительной геометрии и смежных областей. Первая известная ссылка на эту задачу встречается в (Fedman, 1976), где она приписывается Эльвину Берлекэмпу (Elwyn Berlekamp). Все указанные упоминания рассматривают задачу в терминах чисел, нежели абелевых групп, но идея остаётся прежней.

Литература

- Demaine, E. D., Mitchell, J. S. B. and O'Rourke, J. (2009). The Open Problems Project. <http://mave.smith.edu/~orourke/TOPP/>.
- Fedman, M. L. (1976). How good is the information theory lower bound in sorting? *Theoretical Computer Science* **1**, 355–61.
- Harper, L. H., Payne, T. H., Savage, J. E. and Straus, E. (1975). Sorting $X + Y$. *Communications of the ACM* **18** (6), 347–9.
- Knuth, D. E. (1998). *The Art of Computer Programming: Volume 3, Sorting and Searching*, second edition. Reading, MA: Addison-Wesley. [Имеется русский перевод: Кнут Д. Искусство программирования, том 3. Сортировка и поиск. 2-е изд. М.: Вильямс. 2007.]
- Lambert, J.-L. (1992). Sorting the sums $(x_i + y_j)$ in $O(n^2)$ comparisons. *Theoretical Computer Science* **103**, 137–41.

6

Делаем сотню

Введение

Задача этой жемчужины заключается в перечислении всех способов размещения знаков операций $+$ и \times среди элементов списка цифр $[1..9]$ так, чтобы в результате вычисления получилось число 100. Вот два таких способа:

$$100 = 12 + 34 + 5 \times 6 + 7 + 8 + 9$$

$$100 = 1 + 2 \times 3 + 4 + 5 + 67 + 8 + 9$$

Заметьте, что скобки в выражении не допускаются, а операция \times имеет больший приоритет нежели $+$. Несколько известно, единственный способ решить эту задачу — организовать поиск среди всех возможных выражений, иными словами, выполнить полный перебор. Первейшая цель этой жемчужины в небольшом исследовании теории полного перебора и выявлении тех его свойств, которые могут способствовать улучшению быстродействия. После этого теория применяется к задаче построения сотни.

Немного теории

Начнём с трёх типов *Data*, *Candidate* и *Value* и трёх функций:

```
candidates :: Data → [Candidate]
value      :: Candidate → Value
good       :: Value → Bool
```

Эти три функции предназначены для определения функции *solutions*:

$$\begin{aligned} \text{solutions} &:: \text{Data} \rightarrow [\text{Candidate}] \\ \text{solutions} &= \text{filter} (\text{good} \cdot \text{value}) \cdot \text{candidates} \end{aligned}$$

Функция *solutions* выполняет полный перебор списка кандидатов, выбирая те, которые имеют требуемое значение. Даже если необходим только один ответ, нет нужды предпринимать какие бы то ни было специальные действия, поскольку ленивость гарантирует, что будет выполнена только та часть работы, которая необходима для получения первого решения. За исключением этого общего замечания относительно пользы ленивости, о функции *solutions* ничего большего сказать нельзя, по крайней мере, пока мы не сделаем некоторые предположения относительно её ингредиентов.

Первое предположение в том, что *Data* это список значений, скажем, *[Datum]*, и что *candidates* :: *[Datum] → [Candidate]* принимает форму:

$$\text{candidates} = \text{foldr extend} [] \quad (6.1)$$

где *extend* :: *Datum → [Candidate] → [Candidate]* это некоторая функция, строящая список расширений кандидатов на основе единицы данных (*datum*) и списка исходных кандидатов.

Второе предположение состоит из двух частей. Во-первых, имеется такой предикат *ok*, что любое искомое значение (удовлетворяющее предикату *good*) удовлетворяло бы и предикату *ok*, т.е. $\text{good } v \Rightarrow \text{ok } v$ для всех *v*. Следовательно,

$$\text{filter} (\text{good} \cdot \text{value}) = \text{filter} (\text{good} \cdot \text{value}) \cdot \text{filter} (\text{ok} \cdot \text{value}) \quad (6.2)$$

Во-вторых, кандидаты, удовлетворяющие предикату *ok*, являются расширениями кандидатов, удовлетворяющих этому же предикату:

$$\begin{aligned} \text{filter} (\text{ok} \cdot \text{value}) \cdot \text{extend } x \\ = \text{filter} (\text{ok} \cdot \text{value}) \cdot \text{extend } x \cdot \text{filter} (\text{ok} \cdot \text{value}) \end{aligned} \quad (6.3)$$

Пользуясь этими предположениями, проведём рассуждения:

$$\begin{aligned} \text{solutions} \\ = & \{ \text{определение solutions} \} \\ & \text{filter} (\text{good} \cdot \text{value}) \cdot \text{candidates} \\ = & \{ \text{равенство (6.1)} \} \\ & \text{filter} (\text{good} \cdot \text{value}) \cdot \text{foldr extend} [] \end{aligned}$$

$$\begin{aligned}
 &= \{ \text{равенство (6.2)} \} \\
 &\quad \text{filter}(\text{good} \cdot \text{value}) \cdot \text{filter}(\text{ok} \cdot \text{value}) \cdot \text{foldr extend} [] \\
 &= \{ \text{пользуясь } \text{extend}' x = \text{filter}(\text{ok} \cdot \text{value}) \cdot \text{extend } x; \text{ см. ниже} \} \\
 &\quad \text{filter}(\text{good} \cdot \text{value}) \cdot \text{foldr extend}' []
 \end{aligned}$$

Последний шаг здесь основывается на законе слияния для функции *foldr*. Напомним, этот закон утверждает, что $f \cdot \text{foldr } g a = \text{foldr } h b$ при соблюдении следующих трёх условий: (а) f является строгой функцией; (б) $f a = b$; (в) $f(g x y) = h x (f y)$ для всех x и y . В частности, выбрав $f = \text{filter}(\text{ok} \cdot \text{value})$ и $g = \text{extend}$, видим, что (а) выполнено, (б) сохраняется при $a = b = []$, а (в) это просто (6.3), где $h = \text{extend}'$.

Мы показали, что

$$\text{solutions} = \text{filter}(\text{good} \cdot \text{value}) \cdot \text{foldr extend}' []$$

Новая версия функции *solutions* лучше предыдущей, поскольку на каждом этапе строится потенциально гораздо меньший список кандидатов, а именно лишь те, что удовлетворяют предикату *ok*. С другой стороны, при каждом обращении к *extend'* значение функции *value* пересчитывается.

Указанное повторное вычисление можно избежать, если воспользоваться третьим предположением:

$$\text{map value} \cdot \text{extend } x = \text{modify } x \cdot \text{map value} \tag{6.4}$$

Предположение (6.4) утверждает, что значения из множества расширений кандидатов можно получить изменением значений тех кандидатов, расширения которых строятся в данный момент.

Предположим, что функция *candidates* переопределена следующим образом:

$$\text{candidates} = \text{map}(\text{fork}(id, \text{value})) \cdot \text{foldr extend}' []$$

где $\text{fork}(f, g) x = (f x, g x)$. Новая версия функции *candidates* возвращает список пар из кандидатов и их значений. Форма нового определения подсказывает, что вновь можно обратиться к закону слияния для *foldr*. Для главного условия слияния следует найти функцию, скажем *expand*, которая удовлетворяет равенству:

$$\text{map}(\text{fork}(id, \text{value})) \cdot \text{extend}' x = \text{expand } x \cdot \text{map}(\text{fork}(id, \text{value}))$$

В этом случае получаем $\text{candidates} = \text{foldr expand} []$.

Для отыскания функции *expand* мы собираемся применить простые эквивалентные рассуждения (equational reasoning). Для этого нам необходимо некоторое количество комбинаторных законов относительно *fork*, законов, которые используются во многих примерах формального построения программ. Первый закон таков:

$$fst \cdot fork(f, g) = f \text{ и } snd \cdot fork(f, g) = g \quad (6.5)$$

Второй закон это простой закон слияния:

$$fork(f, g) \cdot h = fork(f \cdot h, g \cdot h) \quad (6.6)$$

Что же касается третьего закона, определим функцию *cross* равенством $cross(f, g)(x, y) = (f x, g y)$. Тогда имеем:

$$fork(f \cdot h, g \cdot k) = cross(f, g) \cdot fork(h, k) \quad (6.7)$$

Следующие два закона соотносят *fork* с двумя функциями, *zip* и *unzip*. Вот определение функции *unzip*:

$$\begin{aligned} unzip &:: [(a, b)] \rightarrow ([a], [b]) \\ &unzip = fork(\text{map } fst, \text{map } snd) \end{aligned}$$

Функция *zip* :: $[[a], [b]] \rightarrow [(a, b)]$ может быть определена спецификацией $zip \cdot unzip = id^1$. В частности, можно рассуждать так:

$$\begin{aligned} &unzip \cdot map(fork(f, g)) \\ &= \{ \text{определение } unzip \} \\ &\quad fork(\text{map } fst, \text{map } snd) \cdot map(fork(f, g)) \\ &= \{ \text{свойство (6.6) и } map(f \cdot g) = map f \cdot map g \} \\ &\quad fork(\text{map}(fst \cdot fork(f, g)), \text{map}(snd \cdot fork(f, g))) \\ &= \{ (6.5) \} \\ &\quad fork(\text{map } f, \text{map } g) \end{aligned}$$

Следовательно,

$$fork(\text{map } f, \text{map } g) = unzip \cdot map(fork(f, g)) \quad (6.8)$$

С использованием $zip \cdot unzip = id$ из (6.8) следует, что

$$map(fork(f, g)) = zip \cdot fork(\text{map } f, \text{map } g) \quad (6.9)$$

¹Функция $zip :: [a] \rightarrow [b] \rightarrow [(a, b)]$ языка Haskell определена как каррированная функция.

Последний закон связывает *fork* и *filter*:

$$\begin{aligned} & \text{map}(\text{fork}(f, g)) \cdot \text{filter}(p \cdot g) \\ &= \text{filter}(p \cdot \text{snd}) \cdot \text{map}(\text{fork}(f, g)) \end{aligned} \tag{6.10}$$

Вычисление выражения в правой части равенства можно организовать более эффективно, чем выражения слева, потому что *g* необходимо вычислить только один раз для каждого элемента переданного списка.

Выявив разнообразные связующие комбинаторы и правила, которые их соотносят, мы готовы к последнему рассуждению:

$$\begin{aligned} & \text{map}(\text{fork}(id, value)) \cdot \text{extend}' x \\ &= \{ \text{определение } \text{extend}' \} \\ & \text{map}(\text{fork}(id, value)) \cdot \text{filter}(ok \cdot value) \cdot \text{extend } x \\ &= \{ (6.10) \} \\ & \text{filter}(ok \cdot \text{snd}) \cdot \text{map}(\text{fork}(id, value)) \cdot \text{extend } x \end{aligned}$$

Сфокусируемся теперь на первых двух термах и продолжим:

$$\begin{aligned} & \text{map}(\text{fork}(id, value)) \cdot \text{extend } x \\ &= \{ (6.9) \text{ и } \text{map } id = id \} \\ & \text{zip} \cdot \text{fork}(id, \text{map value}) \cdot \text{extend } x \\ &= \{ (6.6) \} \\ & \text{zip} \cdot \text{fork}(\text{extend } x, \text{map value} \cdot \text{extend } x) \\ &= \{ (6.4) \} \\ & \text{zip} \cdot \text{fork}(\text{extend } x, \text{modify } x \cdot \text{map value}) \\ &= \{ (6.7) \} \\ & \text{zip} \cdot \text{cross}(\text{extend } x, \text{modify } x) \cdot \text{fork}(id, \text{map value}) \\ &= \{ (6.8) \} \\ & \text{zip} \cdot \text{cross}(\text{extend } x, \text{modify } x) \cdot \text{unzip} \cdot \text{map}(\text{fork}(id, value)) \end{aligned}$$

Объединяя результаты рассуждений, окончательно получаем:

$$\begin{aligned} \text{solutions} &= \text{map } \text{fst} \cdot \text{filter}(\text{good} \cdot \text{snd}) \cdot \text{foldr } \text{expand} [] \\ \text{expand } x &= \text{filter}(ok \cdot \text{snd}) \cdot \text{zip} \cdot \text{cross}(\text{extend } x, \text{modify } x) \cdot \text{unzip} \end{aligned}$$

Это последняя версия функции *solutions*. Она зависит только от определений *good*, *ok*, *extend* и *modify*. Выражение *foldr expand []* строит список кандидатов вместе с их значениями, а *solutions* выбирает тех кандидатов,

значения которых удовлетворяют предикату *good*. Функция *expand x* строит список расширений кандидатов, сохраняя при этом то свойство, что значения всех расширенных кандидатов удовлетворяют предикату *ok*.

Делаем сотню

Давайте вернёмся к нашей задаче, которая заключается в перечислении всех способов размещения знаков операций + и × среди элементов списка цифр [1..9] так, чтобы в результате вычисления получилось число 100.

Претенденты на решение являются выражениями, построенными из + и ×. Каждое выражение это сумма списка термов, каждый терм это произведение списка сомножителей, каждый сомножитель это список цифр. Это означает, что мы можем определить выражения, термы и сомножители посредством подходящих синонимов типов:

```
type Expression = [Term]
type Term      = [Factor]
type Factor    = [Digit]
type Digit     = Int
```

Таким образом, тип *Expression* соответствует типу [[[*Int*]]].

Значение выражения вычисляется функцией *valExpr*, определённой следующим образом:

```
valExpr :: Expression → Int
valExpr = sum · map valTerm
valTerm :: Term → Int
valTerm = product · map valFact
valFact :: Factor → Int
valFact = foldl1 (λn d → 10 * n + d)
```

Подходящим выражением является то, значение которого равно 100:

```
good   :: Int → Bool
good v = (v == 100)
```

Постановка задачи завершается определением функции *expressions*, которая генерирует список всех возможных выражений, допускающих построение из заданного списка цифр. Это можно сделать двумя способами. В одном из них можно вызвать стандартную функцию *partitions*, тип которой

$[a] \rightarrow [[[a]]]$. Эта функция разбивает список всеми возможными способами на один или более подсписков. Если применить *partitions* к списку цифр *xs*, то мы получим список всех возможных способов разбиения *xs* на списки сомножителей. Затем, применяя *partitions* снова уже к каждому из списков сомножителей, получим список всевозможных разбиений списков сомножителей на списки термов. Следовательно,

```
expressions :: [Digit] → [Expression]
expressions = concatMap partitions · partition
```

Иначе, можно определить *expressions* как $expressions = foldr extend []$, где

<i>extend</i>	$:: Digit \rightarrow [Expression] \rightarrow [Expression]$
<i>extend x []</i>	$= [[[x]]]$
<i>extend x es</i>	$= concatMap (glue x) es$
<i>glue</i>	$:: Digit \rightarrow Expression \rightarrow [Expression]$
<i>glue x ((xs : xss) : xsss)</i>	$= (((x : xs) : xss) : xsss,$ $\quad ([x] : xs : xss) : xsss,$ $\quad [[x]] : (xs : xss) : xsss)$

Для объяснения этих определений заметим, что из одной цифры *x* можно построить только одно выражение, а именно $[[[x]]]$. Это подтверждает первую строчку определения *extend*. Выражение, построенное из более чем одной цифры, можно разбить на первый сомножитель (список цифр, скажем, *xs*), первый терм (список сомножителей, скажем, *xss*) и остаток выражения (список термов, скажем, *xsss*). Новую цифру *x* можно вставить в выражение в точности тремя различными способами: расширяя текущий сомножитель новой цифрой слева, начиная новый сомножитель или начиная новый терм. Это рассуждение подтверждает вторую строчку определения *extend* и определение *glue*. Второе определение полезно тем, что из него видно, что из списка цифр $[1..9]$ можно построить $6561 = 3^8$ выражений; в самом деле, 3^{n-1} выражений для списка *n* цифр.

Вычисление $filter(good \cdot valExpr) \cdot expressions$ и отображение результатов в подходящем виде даёт следующие семь возможных ответов:

$$\begin{aligned} 100 &= 1 \times 2 \times 3 + 4 + 5 + 6 + 7 + 8 \times 9 \\ 100 &= 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 \times 9 \\ 100 &= 1 \times 2 \times 3 \times 4 + 5 + 6 + 7 \times 8 + 9 \\ 100 &= 12 + 3 \times 4 + 5 + 6 + 7 \times 8 + 9 \\ 100 &= 1 + 2 \times 3 + 4 + 5 + 67 + 8 + 9 \end{aligned}$$

$$100 = 1 \times 2 + 34 + 5 + 6 \times 7 + 8 + 9$$

$$100 = 12 + 34 + 5 \times 6 + 7 + 8 + 9$$

Вычисления выполняются не слишком долго, потому что достаточно проверить всего 6561 возможное выражение. Но что если в какой-то момент понадобится поработать с другими входными данными и гораздо большим числом знаков? Поэтому полезно потратить некоторое время на попытку ускорить поиск.

Согласно теории полного перебора из предыдущего раздела нам нужно найти определение такого предиката *ok*, что все подходящие выражения ему удовлетворяют, и любое выражение, ему удовлетворяющее, обязательно строится из выражения, также ему удовлетворяющего. Поскольку $good v = (v == c)$, где c — заданное значение, единственным разумным определением *ok* будет $ok v = (v \leq c)$. Так как разрешаются только операции $+$ и \times , каждое выражение со значением c должно строиться из подвыражений, значение которых не превосходит c .

Нам также нужно найти такое определение *modify*, что

$$\text{map } valExpr \cdot \text{extend } x = \text{modify } x \cdot \text{map } valExpr$$

Здесь возникает небольшая сложность, потому что не все значения выражений в *glue* $x e$ можно определить только из значения e : нам также нужны значения первого сомножителя и первого терма. Поэтому мы определим *value* не как *valExpr*, а как

$$\begin{aligned} \text{value } ((xs : xss) : xsss) &= (10^n, \text{valFact } xs, \text{valTerm } xss, \text{valExpr } xsss) \\ \text{where } n &= \text{length } xs \end{aligned}$$

Первый компонент 10^n включён для того, чтобы сделать вычисление *valFact* ($x : xs$) более эффективным. Теперь получаем:

$$\begin{aligned} \text{modify } x (k, f, t, e) \\ &= [(10 * k, k * x + f, t, e), (10, x, f * t, e), (10, x, 1, f * t + e)] \end{aligned}$$

Соответственно изменяем определения *good* и *ok*:

$$\begin{aligned} \text{good } c (k, f, t, e) &= (f * t + e == c) \\ \text{ok } c (k, f, t, e) &= (f * t + e \leq c) \end{aligned}$$

Подстановка этих определений в функцию *expand* позволяет ускорить полный перебор:

```
solutions c = map fst · filter (good c · snd) · foldr (expand c) []
expand c x = filter (ok c · snd) · zip · cross (extend x, modify x) · unzip
```

Определение *expand* можно упростить:

```
expand c x [] = [[[x]], (10, x, 1, 0))]
expand c x evs = concat (map (filter (ok c · snd) · glue x) evs)
glue x ((xs : xss) : xsss, (k, f, t, e)) =
  [(((x : xs) : xss) : xsss, (10 * k, k * x + f, t, e)),
   ([[x] : xs : xss) : xsss, (10, x, f * t, e)),
   ([[x]] : (xs : xss) : xsss, (10, x, 1, f * t + e))]
```

Результатом является гораздо более быстрая, чем первая версия, программа для *solutions*. Только на одном тесте, при $c = 1000$ и заданных первых 14 цифрах числа π , вторая версия оказалась в 200 раз быстрее.

Заключительные замечания

Задача построения сотни обсуждается в упражнении 122 (Knuth, 2006), в котором рассматриваются также и другие её вариации, например, с разрешением скобок и других арифметических операций; смотрите также жемчужину 20 об обратном отчёте далее в этой книге. Мораль упражнения в том, что при поиске претендентов, значение которых удовлетворяет некоторому критерию, полезно генерировать претендентов одновременно с их значениями. Благодаря этому можно избежать перевычисления значений. Обычно достаточно очевидно, как это можно делать напрямую, без использования формальных рассуждений по типу описанных выше, однако приятно знать, что так тоже можно. Полезно также замечать, возможно ли ослабить условие на подходящие значения, удовлетворяющие предикату *good*, до другого предиката, выполняющегося для всех претендентов. С помощью этого подхода можно сократить множество претендентов, которые следует рассматривать.

Литература

Knuth, D. E. (2006). *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees*. Reading, MA: Addison-Wesley. [Имеется русский перевод: Кнут Д. Искусство программирования, том 4, выпуск 4. Генерация всех деревьев. М.: Вильямс. 2007.]

7

Строим дерево минимальной высоты

Введение

Рассмотрим задачу построения дерева минимальной высоты, листья которого помечены числами из заданного списка целых, образующего, таким образом, крону дерева. Соответствующие деревья определяются так:

```
data Tree = Leaf Int | Fork Tree Tree
```

Кrona дерева это список меток листовых узлов в порядке слева направо. Известно два алгоритма решения этой задачи, каждый из которых можно выполнить за линейное время. Один рекурсивный, идущий сверху вниз, он разбивает список на две половины, рекурсивно строит деревья для каждой половины и затем соединяет результаты конструктором *Fork*. Второй алгоритм итеративный, идущий снизу вверх, в нём крону сначала преобразуется в список листьев, а затем последовательно соединяются соседние пары, пока в списке не останется одно дерево. Оба алгоритма приводят к различным деревьям, но в каждом из случаев это деревья наименьшей высоты.

Вид алгоритма, идущего снизу вверх, подсказывает заманчивое обобщение: существует ли линейный алгоритм, строящий единое дерево минимальной длины из произвольного списка деревьев вместе с их высотами? Естественно, подразумевается следующее ограничение: все исходные деревья должны оказаться поддеревьями результирующего дерева в том же порядке, в котором они шли в списке. В частном случае, если на входе оказывается список листьев, задача сводится к той, что обсуждалась выше,

однако нет особых причин быть уверенным, что и в более общем случае задача должна решаться за линейное время. Тем не менее, в этой жемчужине мы выведем именно такой алгоритм.

Первые шаги

Эта задача имеет альтернативную, но эквивалентную постановку: дана последовательность натуральных чисел $xs = [x_1, x_2, \dots, x_n]$ (числа представляют высоты заданных деревьев), можно ли найти линейный алгоритм построения дерева с кроной xs , в котором минимизировано значение $cost$:

$$\begin{aligned} cost(Leaf\ x) &= x \\ cost(Fork\ u\ v) &= 1 + (cost\ u\ \max\ cost\ v) \end{aligned}$$

Таким образом, $cost$ определена так же, как и $height$, только «высота» узла $Leaf\ x$ равна x , а не нулю.

Задача в такой формулировке заключается в вычислении

$$mincostTree = minBy\ cost \cdot trees$$

где функция $trees$ строит все возможные деревья с заданной кроной, а $minBy\ cost$ выбирает одно с наименьшим значением $cost$. Конструктивное определение $trees$ можно дать разными способами, следуя либо схеме «сверху-вниз», либо схеме «снизу-вверх». Мы, однако, выберем третий путь, а именно, индуктивный алгоритм:

$$\begin{aligned} trees &\quad :: [Int] \rightarrow [Tree] \\ trees[x] &= [Leaf\ x] \\ trees(x : xs) &= concatMap\ (prefixes\ x)\ (trees\ xs) \end{aligned}$$

Функция языка Haskell $concatMap\ f$ это просто $concat \cdot map\ f$. Значение $prefixes\ x\ t$ это список всех способов, которыми x можно вставить в дерево t как самый левый лист:

$$\begin{aligned} prefixes &\quad :: Int \rightarrow Tree \rightarrow [Tree] \\ prefixes\ x\ t@(Leaf\ y) &= [Fork\ (Leaf\ x)\ t] \\ prefixes\ x\ t@(Fork\ u\ v) &= [Fork\ (Leaf\ x)\ t] ++ \\ &\quad [Fork\ u'\ v\mid u' \leftarrow prefixes\ x\ u] \end{aligned}$$

Можно было бы взять $trees[] = []$ и определить $trees$ как экземпляр свёртки $foldr$. Но $minBy\ cost$ не определена на пустом множестве деревьев, поэтому лучше ограничить входные данные непустыми списками. В языке

Haskell отсутствует обобщённая свёртка для непустых списков (функция *foldr1* недостаточно общая), но если мы определим *foldrn*:

$$\begin{aligned} \text{foldrn} &:: (a \rightarrow b \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow b \\ \text{foldrn } f \ g [x] &= g x \\ \text{foldrn } f \ g (x : xs) &= f x (\text{foldrn } f \ g xs) \end{aligned}$$

то *trees* можно выразить как экземпляр *foldrn*:

$$\begin{aligned} \text{trees} &= \text{foldrn} (\text{concatMap} \cdot \text{prefixes}) (\text{wrap} \cdot \text{Leaf}) \\ \text{wrap } x &= [x] \end{aligned}$$

Там, где есть деревья, есть и леса, и многие определения для первых можно перефразировать, зачастую более чётко, в терминах последних. Это же имеет место и для функции *trees*. Более точное определение можно получить, строя сначала список лесов (где лес в свою очередь является списком деревьев), а затем сворачивая каждый лес в дерево:

$$\begin{aligned} \text{trees} &= \text{map rollup} \cdot \text{forests} \\ \text{forests} &= [\text{Int}] \rightarrow [\text{Forest}] \\ \text{forests} &= \text{foldrn} (\text{concatMap} \cdot \text{prefixes}) (\text{wrap} \cdot \text{wrap} \cdot \text{Leaf}) \\ \text{prefixes} &:: \text{Int} \rightarrow \text{Forest} \rightarrow [\text{Forest}] \\ \text{prefixes } x \ ts &= [\text{Leaf } x : \text{rollup} (\text{take } k \ ts) : \text{drop } k \ ts \\ &\quad | \ k \leftarrow [1 \dots \text{length } ts]] \\ \text{rollup} &:: \text{Forest} \rightarrow \text{Tree} \\ \text{rollup} &= \text{foldl1 Fork} \end{aligned}$$

В этой версии *trees* каждый лес представляет собой левую ось (left spine) дерева, т.е. последовательность правых поддеревьев на пути от самого левого листа к корню дерева. Первый элемент оси и есть самый левый лист. Сворачивание оси даёт дерево. Второе определение функции *prefixes* предпочтительнее первого, поскольку оно явно показывает, что происходит при построении результирующих деревьев. К этому определению *trees* мы ещё вернёмся позднее.

Остается определить функцию *minBy cost* :: [Tree] → Tree. От неё требуется, чтобы она возвращала некоторое дерево минимальной стоимости:

$$\text{minBy cost } ts \in ts \wedge (\forall t \in ts : \text{cost} (\text{minBy cost } ts) \leq \text{cost } t)$$

Эта спецификация не определяет результат единственным образом, поэтому функция *minBy cost* является недетерминированной. Её можно определить, к примеру, так:

$$\begin{aligned} \text{minBy } f &= \text{foldl1 } (\text{cmp } f) \\ \text{cmp } f \ u \ v &= \text{if } f \ u \leqslant f \ v \text{ then } u \text{ else } v \end{aligned}$$

Эта реализация, однако, выбирает первое дерево в ts , стоимость которого минимальна, а значит, её результат зависит от порядка деревьев в ts . Менее предвзятую, но детерминированную реализацию можно получить, если ввести полный линейный порядок \preccurlyeq , который сохраняет $cost$, т.е. $u \preccurlyeq v \Rightarrow cost \ u \leqslant cost \ v$, и заменить $\text{cmp } f$ на cmp , где

$$\text{cmp } u \ v = \text{if } u \preccurlyeq v \text{ then } u \text{ else } v$$

Это определение, правда, зависит от выбранного отношения \preccurlyeq , а значит, снова слишком конкретно. Поэтому оставим функцию minBy cost недетерминированной.

Слияние

Будучи реализованной напрямую, $\text{minBy cost} \cdot \text{trees}$ выполняется за экспоненциальное время. Очевидный способ ускориться — обратиться к закону слияния для foldrn . В его простейшей форме этот закон утверждает, что

$$h(\text{foldrn } f \ g \ xs) = \text{foldrn } f' \ g' \ xs$$

для всех конечных непустых списков xs при условии, что $h(g \ x) = g' \ x$ и $h(f \ x \ y) = f' \ x \ (h \ y)$ для всех x и y . Однако, требовать выполнения равенства термов в ситуации, когда h недетерминированная функция, бессмысленно: нам нужно лишь, чтобы терм справа был уточнением терма слева. Предположим, мы определили отношение $f \ x \rightsquigarrow g \ x$ так, что множество возможных результатов $f \ x$ включает (непустое) множество возможных результатов $g \ x$. В частности, если g обычная недетерминированная функция, то $f \ x \rightsquigarrow g \ x$ означает, что $g \ x$ это один из возможных результатов $f \ x$. Более слабое утверждение для слияния выглядит так:

$$h(\text{foldrn } f \ g \ xs) \rightsquigarrow \text{foldrn } f' \ g' \ xs$$

для всех конечных непустых списков xs при условии, что $h(g \ x) \rightsquigarrow g' \ x$ для всех x , и из $h \ y \rightsquigarrow y'$ следует, что $h(f \ x \ y) \rightsquigarrow f' \ x \ y'$ для всех x, y и y' .

Так как $\text{minBy cost} \cdot \text{wrap} = id$, обратившись к слиянию, получаем:

$$\begin{aligned} \text{minBy cost}(\text{foldrn}(\text{concatMap} \cdot \text{prefixes})(\text{wrap} \cdot \text{Leaf}) \ xs) \\ \rightsquigarrow \text{foldrn insert Leaf} \ xs \end{aligned}$$

при условии, что функцию *insert* можно определить так, чтобы она удовлетворяла предусловиям слияния:

$$\begin{aligned} \text{minBy cost } ts &\rightsquigarrow t \\ \Rightarrow \text{minBy cost } (\text{concatMap } (\text{prefixes } x) ts) &\rightsquigarrow \text{insert } x t \end{aligned} \tag{7.1}$$

Предположим, что *insert* специфицирована условием:

$$\text{minBy cost } \cdot \text{prefixes } x \rightsquigarrow \text{insert } x$$

Словом, *insert x t* возвращает некоторое дерево минимальной стоимости в *prefixes x t*. Тогда можно утверждать, что (7.1) выполняется, если выполняется следующее:

$$\begin{aligned} \text{minBy cost } ts &\rightsquigarrow t \\ \Rightarrow \text{minBy cost } (\text{map } (\text{insert } x) ts) &\rightsquigarrow \text{insert } x t \end{aligned} \tag{7.2}$$

Чтобы это доказать, нужен такой факт:

$$\text{minBy cost } \cdot \text{concat} = \text{minBy cost } \cdot \text{map minBy cost}$$

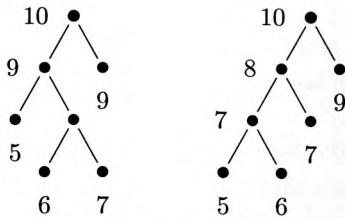
для всех непустых списков непустых списков. Словом, каждый возможный результат левой части это возможный результат правой части, и наоборот. Теперь рассуждаем:

$$\begin{aligned} &\text{minBy cost } (\text{concatMap } (\text{prefixes } x) ts) \\ &= \{ \text{разворачиваем concatMap} \} \\ &\quad \text{minBy cost } (\text{concat } (\text{map } (\text{prefixes } x) ts)) \\ &= \{ \text{указанный выше факт} \} \\ &\quad \text{minBy cost } (\text{map } (\text{minBy cost } \cdot \text{prefixes } x) ts) \\ &\rightsquigarrow \{ \text{так как из } f \rightsquigarrow f' \text{ следует, что } \text{map } f \rightsquigarrow \text{map } f' \text{ и} \\ &\quad g \cdot f \rightsquigarrow g \cdot f' \} \\ &\quad \text{minBy cost } (\text{map } (\text{insert } x) ts) \end{aligned}$$

Условие слияния (7.2) выполняется при условии, что

$$\text{cost } u \leq \text{cost } v \Rightarrow \text{cost } (\text{insert } x u) \leq \text{cost } (\text{insert } x v) \tag{7.3}$$

для всех деревьев *u* и *v* с одинаковой кроной. Правда, (7.3) не имеет места. Рассмотрим следующие два дерева *u* и *v*:



В каждом дереве левые оси помечены информацией о стоимости, так что стоимость обоих деревьев равна 10. Вставка элемента 8 в левое дерево и даст дерево минимальной стоимости 11, а вставка того же элемента в правое дерево v даст дерево стоимости 10. Поэтому (7.3) не работает.

Заметьте, однако, что стоимости $[10, 8, 7, 5]$, если читать их, спускаясь вдоль левой оси дерева v , лексикографически меньше стоимостей $[10, 9, 5]$ на спуске вдоль левой оси u . Что мы можем установить, так это условие монотонности:

$$\text{cost}' u \leq \text{cost}' v \Rightarrow \text{cost}' (\text{insert } x u) \leq \text{cost}' (\text{insert } x v) \quad (7.4)$$

где

$$\text{cost}' = \text{map cost} \cdot \text{reverse} \cdot \text{spine}$$

Функция spine является обратной к функции rollup , т.е. $\text{spine} \cdot \text{rollup} = \text{id}$. Минимизация cost' одновременно приводит к минимизации cost , поскольку $\text{xs} \leq \text{ys} \Rightarrow \text{head xs} \leq \text{head ys}$. Как мы покажем через мгновение, из (7.4) следует, что

$$\text{minBy cost}' \cdot \text{map} (\text{insert } x) \rightsquigarrow \text{insert } x \cdot \text{minBy cost}'$$

Так как на сцене возникли оси, имеет смысл воспользоваться определением trees в терминах осей и подытожить проведённые к данному моменту рассуждения:

$$\begin{aligned}
 & \text{minBy cost} \cdot \text{trees} \\
 & \rightsquigarrow \{ \text{уточнение} \} \\
 & \text{minBy cost}' \cdot \text{trees} \\
 & = \{ \text{определение trees в терминах лесов} \} \\
 & \text{minBy cost}' \cdot \text{map rollup} \cdot \\
 & \text{foldr}(\text{concatMap} \cdot \text{prefixes})(\text{wrap} \cdot \text{wrap} \cdot \text{Leaf}) \\
 & = \{ \text{определим costs} = \text{map cost} \cdot \text{reverse} \}
 \end{aligned}$$

$$\begin{aligned}
 & \minBy (\text{costs} \cdot \text{spine}) \cdot \text{map rollup} \cdot \\
 & \quad \text{foldrn} (\text{concatMap} \cdot \text{prefixes}) (\text{wrap} \cdot \text{wrap} \cdot \text{Leaf}) \\
 = & \quad \{ \text{утверждение: см. ниже} \} \\
 & \quad \text{rollup} \cdot \minBy \text{ costs} \cdot \\
 & \quad \text{foldrn} (\text{concatMap} \cdot \text{prefixes}) (\text{wrap} \cdot \text{wrap} \cdot \text{Leaf}) \\
 \rightsquigarrow & \quad \{ \text{слияние, } \minBy \text{ costs} \cdot \text{prefixes } x \rightsquigarrow \text{insert } x \} \\
 & \quad \text{rollup} \cdot \text{foldrn insert} (\text{wrap} \cdot \text{Leaf})
 \end{aligned}$$

Утверждение заключается в том, что

$$\minBy (\text{costs} \cdot \text{spine}) \cdot \text{map rollup} = \text{rollup} \cdot \minBy \text{ costs}$$

Оно следует из определения \minBy и того факта, что spine и rollup обратны друг к другу.

Чтобы показать, что выполняется (7.4), и что это ведёт к программе, линейной по времени, остаётся реализовать функцию insert .

Оптимальная вставка

Рассмотрим два дерева на рис. 7.1. Дерево слева имеет ось ts , где $ts = [t_1, t_2, \dots, t_n]$. Дерево справа имеет ось $\text{Leaf } x : \text{rollup} (\text{take } j \text{ ts}) : \text{drop } j \text{ ts}$. Каждая ось также помечена информацией о стоимости: c_k определено для $2 \leq k \leq n$ с помощью равенства

$$c_k = 1 + (\max_{k-1} \text{cost } t_k)$$

а c_1 это значение листа t_1 . Похожее равенство определяет c'_k для k в промежутке $j+1 \leq k \leq n$. Заметьте, что в этом промежутке $c'_j > c_j$ и $c'_k \geq c_k$. Имея в виду (7.4), мы хотим выбрать некоторое j из промежутка $1 \leq j \leq n$, которое минимизирует

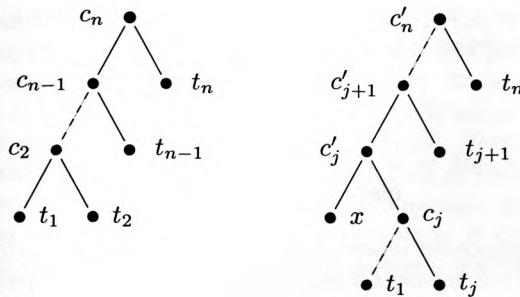
$$[c'_n, c'_{n-1}, \dots, c'_{j+1}, c'_j, x]$$

Утверждаем, что минимум достигается при выборе такого наименьшего значения из промежутка $1 \leq j < n$, для которого

$$1 + (x \max c_j) < c_{j+1} \tag{7.5}$$

если оно, конечно, существует. В противном случае выберем $j = n$. Для доказательства этого утверждения заметим, что если выполняется (7.5), то

$$c'_j = 1 + (x \max c_j) < c_{j+1}$$

Рис. 7.1: вставка x в дерево

и значит, $c'_k = c_k$ для $j + 1 \leq k \leq n$. Более того, если (7.5) выполняется также и для $i < j$, то

$$\begin{aligned}
 & [c'_n, c'_{n-1}, \dots, c'_{i+1}, c'_i, x] \\
 &= \{ \text{поскольку } c'_k = c_k \text{ при } i + 1 \leq k \leq n \text{ и } i < j \} \\
 & [c_n, c_{n-1}, \dots, c_{j+1}, c_j, \dots, c_{i+1}, c'_i, x] \\
 &< \{ \text{поскольку } c_j < c'_j \} \\
 & [c_n, c_{n-1}, \dots, c_{j+1}, c'_j, x] \\
 &= \{ \text{поскольку } c'_k = c_k \text{ при } j + 1 \leq k \leq n \} \\
 & [c'_n, c'_{n-1}, \dots, c'_{j+1}, c'_j, x]
 \end{aligned}$$

Итак, чем меньше j , удовлетворяющее (7.5), тем лучше. Если (7.5) не выполняется для j , то при $j + 2 \leq k \leq n$ имеют место оценки $c'_{j+1} > c_{j+1}$ и $c'_k \geq c_k$, так что стоимость оказывается хуже.

Теперь докажем условие монотонности (7.4). Пусть u и v — деревья со стоимостями $\text{cost}' u$ и $\text{cost}' v$ соответственно. Ясно, что если стоимости равны, то равны и стоимости вставки произвольного x в любое из них. В противном случае, предположим

$$\text{cost}' u = [c_n, c_{n-1}, \dots] < [d_m, d_{m-1}, \dots, d_1] = \text{cost}' v$$

Удаляя общий префикс этих стоимостей, скажем, префикс длины k , мы остаёмся с $[c_{n-k}, \dots, c_1]$ и $[d_{m-k}, \dots, d_1]$, где $c_{n-k} < d_{m-k}$. Но ведь вставка x в соответствующее поддерево u даёт дерево с не большими стоимостями, чем вставка x в соответствующее поддерево дерева v .

Наконец, так как (7.5) эквивалентно условию $x \max c_j < cost = t_{j+1}$, мы можем реализовать *insert* следующим образом:

$$\begin{aligned} insert\ x\ ts &= Leaf\ x : split\ x\ ts \\ split\ x\ [u] &= [u] \\ split\ x\ (u : v : ts) &= \text{if } x \max cost\ u < cost\ v \text{ then } u : v : ts \\ &\quad \text{else } split\ x\ (Fork\ u\ v : ts) \end{aligned}$$

Вычисления стоимостей можно избежать, если немного уточнить данные, представив каждое дерево t в виде пары $(cost\ t, t)$, что приведёт к окончательной версии алгоритма:

$$\begin{aligned} mincostTree &= foldl1\ Fork \cdot map\ snd \cdot foldrn\ insert\ (wrap \cdot leaf) \\ insert\ x\ ts &= leaf\ x : split\ x\ ts \\ split\ x\ [u] &= [u] \\ split\ x\ (u : v : ts) &= \text{if } x \max fst\ u < fst\ v \text{ then } u : v : ts \\ &\quad \text{else } split\ x\ (fork\ u\ v : ts) \\ leaf\ x &= (x, Leaf\ x) \\ fork\ (a, u)\ (b, v) &= (1 + a \max b, Fork\ u\ v) \end{aligned}$$

Умные конструкторы *leaf* и *fork* создают пары, содержащие стоимость и дерево.

Остается только замерить время выполнения программы, что можно сделать, подсчитав количество вызовов функции *split*. Пользуясь индукцией, докажем, что функция *foldrn insert (wrap · leaf)*, будучи применённой к списку длины n и возвратившая в результате лес длины m , выполняет самое большое $2n - m$ вызовов функции *split*. Базовый случай, $n = 1$ и $m = 1$, очевиден. Для обоснования индуктивного перехода заметим, что *split*, применённая к списку длины m' и возвратившая список длины m , вызывается $m' - m$ раз. А поскольку $2(n - 1) - m' + m' - m + 1 \leq 2n - m$, индукция обоснована. Следовательно, алгоритм выполняется за линейное время.

Заключительные замечания

Задача построения дерева с минимальной стоимостью и её решение являются упражнением на проектирование жадного алгоритма. Жадные алгоритмы сложны, но не столько из-за того, что их реализация далеко не очевидна, сколько из-за необходимости привлечения довольно тонких рассуждений, доказывающих их корректность. Во-первых, разработчику

обычно нужно придумывать усиление целевой функции и на каждом шаге её минимизировать. На этом этапе решающим для успеха всего предприятия оказывается условие монотонности (7.4). Во-вторых, работая с большинством оптимизационных задач, необходимо вводить в рассмотрение отношения или недетерминированные функции. Результат рассуждений неэквивалентен изначальной спецификации, он её уточняет. Мы лишь слегка коснулись работы с отношениями, положившись для продолжения рассуждений на аналогичную языку Haskell нотацию. В книге (Bird and de Moor, 1997) использование отношений в формальном построении программ организовано гораздо более систематично.

Наконец, стоит заметить, что есть и другой способ решения задачи построения дерева минимальной стоимости, а именно алгоритм Ху–Таккера (Hu–Tucker) или его более современная версия, алгоритм Гарсиа–Уочса (Garsia–Wachs); см. (Hu, 1982) или (Knuth, 1998). Алгоритм Ху–Таккера оказывается применим благодаря тому, что функция *cost* является регулярной целевой функцией в смысле (Hu, 1982). Однако наилучшая реализация этого алгоритма выполняется за время $\Theta(n \log n)$.

Литература

- Bird, R. S. and de Moor, O. (1997). *Algebra of Programming*. Hemel Hempstead: Prentice Hall.
- Hu, T. C. (1982). *Combinatorial Algorithms*. Reading, MA: Addison-Wesley. [Второе издание этой книги переведено на русский язык: Ху Т.Ч., Шинг М.Т. Комбинаторные алгоритмы. Нижний Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского. 2004.]
- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Searching and Sorting*, second edition. Reading, MA: Addison-Wesley. [Имеется русский перевод: Кнут Д. Искусство программирования, том 3. Сортировка и поиск. 2-е изд. М.: Вильямс. 2007.]

8

Распутываем жадные алгоритмы

Ему распутайте всё это дело.
Гамлет, Акт 3, Сцена 4.¹

Введение

Как уже говорилось в предыдущей жемчужине, жадные алгоритмы — сложная штука. Поэтому этот предмет заслуживает второго примера. Здесь рассматривается ещё одна задача, которую можно решить жадным алгоритмом, а путь к её решению имеет много общего с представленным выше.

Назовём *распуткой* (an unravel) последовательности *xs* мультимножество (bag) непустых подпоследовательностей *xs*, элементы которых при спутывании снова составляют *xs*. Например, буквы слова «проследовать» можно распутать на четыре списка: «ад», «весь», «лор» и «опт». Порядок этих списков несущественен, но повторения важны; например, слово «скок» можно распутать в две копии «сок». Таким образом, распутка этого действительно мультимножество списков, а не просто их множество.

Распутка называется возрастающей (upravel), если все компоненты, её составляющие, не убывают. Так как последовательности «ад», «весь», «лор» и «опт» возрастают, то они образуют возрастающую распутку слова «проследовать», такой же будет распутка «авдоо», «лпрт», «есь». Каждая непустая последовательность имеет по крайней мере одну возрастающую¹⁰

¹ В переводе М. Лозинского.

распутку, а именно распутку, состоящую только из одноэлементных подпоследовательностей. Однако из всех возрастающих распуток мы хотим найти одну, с наименьшим количеством элементов.

Спецификация

Вот спецификация функции *supravel* (наименьшая возрастающая распутка — shortest upravel):

$$\begin{aligned} \textit{supravel} &:: \textit{Ord } a \Rightarrow [a] \rightarrow [[a]] \\ \textit{supravel} &= \textit{minBy length} \cdot \textit{filter (all up)} \cdot \textit{unravels} \end{aligned}$$

Мульти множество последовательностей в возрастающей распутке представлено списком. Функция *minBy f* это недетерминированная функция, введённая в предыдущей жемчужине. Она специфицирована условием:

$$\textit{minBy } f \textit{ xs} \in \textit{xs} \wedge (\forall x \in \textit{xs} : f(\textit{minBy } f \textit{ xs}) \leqslant f x)$$

Предикат *up*, чьё определение опущено, выясняет, возрастает ли его аргумент. Функция *unravels* возвращает все распутки последовательности, её можно определить индуктивным образом так:

$$\begin{aligned} \textit{unravels} &:: [a] \rightarrow [[[a]]] \\ \textit{unravels} &= \textit{foldr} (\textit{concatMap} \cdot \textit{prefixes}) [] \\ \textit{prefixes } x [] &= [[[x]]] \\ \textit{prefixes } x (xs : xss) &= [(x : xs) : xss] \uparrow\!\!\! \uparrow \textit{map} (xs:) (\textit{prefixes } x xss) \end{aligned}$$

Функция *prefixes x* добавляет к распутке новый элемент *x*, присоединяя его всеми возможными способами к началу каждой последовательности распутки.

Вывод

Воспользуемся для начала законом слияния для *foldr*, соединив *filter (all up)* и *unravels*. Определим *upravels*:

$$\textit{upravels} = \textit{filter (all up)} \cdot \textit{unravels}$$

Применение закона слияния для *foldr* приводит к следующему определению *upravels*:

$$\begin{aligned}
 upravels &:: Ord a \Rightarrow [a] \rightarrow [[[a]]] \\
 upravels &= foldr (concatMap \cdot uprefixes) [[]] \\
 uprefixes x [] &= [[[x]]] \\
 uprefixes x (xs : xss) &= \text{if } x \leq head xs \text{ then} \\
 &\quad [(x : xs) : xss] ++ map (xs:) (uprefixes x xss) \\
 &\text{else map (xs:) (uprefixes x xss)}
 \end{aligned}$$

Функция *uprefixes x* добавляет к распутьке новый элемент *x*. При этом она пытается присоединить его к началу тех последовательностей распутьки, чей первый элемент не меньше *x*.

Теперь мы подходим к сути вопроса: как реализовать слияние функции *minBy length* с *upravels*. Вспомните смысл отношения уточнения \rightsquigarrow из предыдущей жемчужины: для недетерминированной функции *f* и обычной детерминированной функции *g* имеем, что $f \rightsquigarrow g$, если для всех *x* значение *g x* является одним из возможных результатов *f x*. Предположим, что функция *insert* специфицирована так:

$$\text{minBy length} \cdot \text{uprefixes } x \rightsquigarrow \text{insert } x$$

Таким образом, *insert x ur* возвращает некоторый кратчайший способ вставки *x* в возрастающую распутьку *ur*. Обращаясь к более слабому закону слияния для *foldr* в терминах уточнения вместо равенства, мы получим:

$$\text{minBy length} (\text{upravels } xs) \rightsquigarrow \text{foldr insert } [] xs$$

для всех конечных списков *xs* при выполнении условия слияния

$$\begin{aligned}
 \text{minBy length } urs &\rightsquigarrow ur \\
 \Rightarrow \text{minBy length} (\text{map } (\text{insert } x) urs) &\rightsquigarrow \text{insert } x ur
 \end{aligned}$$

для всех возрастающих распутьок *urs* каждого из этих списков. Условие слияния, в свою очередь, выполняется, если

$$\begin{aligned}
 \text{length } ur &\leq \text{length } vr \\
 \Rightarrow \text{length } (\text{insert } x ur) &\leq \text{length } (\text{insert } x vr)
 \end{aligned} \tag{8.1}$$

для любых двух распутьок *ur* и *vr* одного списка.

К сожалению, (8.1) не имеет места. Возьмите две возрастающие распутьки одинаковой длины для строки "ada": ["ad", "a"] и ["aa", "d"]. Вставка "c" в первую из них даёт наилучшую возможную возрастающую распутьку ["ad", "a", "c"], тогда как вставка "c" во вторую даст более короткую распутьку ["aa", "cd"].

Выход, как и в задаче построения дерева минимальной стоимости, в том, что нужно усилить минимизируемую целевую функцию. Совершенно ясно, что длина $\text{insert } x \text{ ur}$ зависит только от x и первых элементов каждой последовательности из ur . Предположим, мы определили heads так:

$$\begin{aligned}\text{heads} &:: \text{Ord } a \Rightarrow [[a]] \rightarrow [a] \\ \text{heads} &= \text{sort} \cdot \text{map head}\end{aligned}$$

Неформально, чем больше (лексикографически) heads ur , тем более вероятно, что x можно вставить в начало некоторой последовательности в ur , что, таким образом, гарантирует длину $\text{insert } s \text{ ur}$ не большую, чем ur . Сложность замены minBy length на maxBy heads в том, что

$$\text{heads ur} \geq \text{heads vr} \Rightarrow \text{length ur} \leq \text{length vr}$$

даже для возрастающих распуток ur и vr одной и той же последовательности. Поэтому нужно искать что-то ещё.

Можно, к примеру, забыть про лексикографическое упорядочивание и рассмотреть вместо него частичный предпорядок \preccurlyeq , определённый следующим образом:

$$\text{ur} \preccurlyeq \text{vr} = \text{heads ur} \trianglelefteq \text{heads vr}$$

где $[x_1, x_2, \dots, x_m] \trianglelefteq [y_1, y_2, \dots, y_n]$, если $m \leq n$ и $x_j \geq y_j$ для всех j из промежутка $1 \leq j \leq n$. Таким образом, $\text{ur} \preccurlyeq \text{vr}$, если $\text{length ur} \leq \text{length vr}$, а элементы heads ur покомпонентно не меньше элементов heads vr . Так как \preccurlyeq сохраняет length , можно заменить minBy length на $\text{minWith}(\preccurlyeq)$, где

$$\text{minWith } (\preccurlyeq) \text{ urs} \in \text{urs} \text{ и } (\forall \text{ur} \in \text{urs} : \text{minWith } (\preccurlyeq) \text{ urs} \preccurlyeq \text{ur})$$

и установить условие монотонности

$$\text{ur} \preccurlyeq \text{vr} \Rightarrow \text{insert } x \text{ ur} \preccurlyeq \text{insert } x \text{ vr} \tag{8.2}$$

где $\text{insert } x$ это некоторое уточнение $\text{minWith } (\preccurlyeq) \cdot \text{uprefixes } x$. Условие (8.2) теперь даёт

$$\text{minWith } (\preccurlyeq) \cdot \text{uprvales} \rightsquigarrow \text{foldr insert } []$$

Однако с частичными порядками имеется небольшая техническая сложность: вообще говоря, нет никакой гарантии, что существует наименьший, не путать с минимальным, элемент. Элемент является наименьшим, если он меньше всех других претендентов, и минимальным, если не существует

претендентов, меньших его. Например, во множестве $\{\{a, b\}, \{a, c\}, \{a, b, c\}\}$ нет наименьшего элемента по отношению \subseteq , однако имеется два минимальных элемента, а именно $\{a, b\}$ и $\{a, c\}$. Поэтому придётся проверить, имеются ли вообще элементы в $\minWith(\preccurlyeq)(upravels xs)$. Это, правда, нетрудно доказать индукцией по xs , пользуясь (8.2) и тем, что

$$\minWith(\preccurlyeq) \cdot uprefixes x \rightsquigarrow insert x$$

Давайте теперь посмотрим, как сконструировать $insert x$. Пусть

$$heads ur = [x_1, x_2, \dots, x_m]$$

т.е. $x_i \leqslant x_j$ при $1 \leqslant i \leqslant j \leqslant m$. Определим k условием $x_k < x \leqslant x_{k+1}$, где можно положить $x_{m+1} = \infty$. Тогда

$$heads(uprefixes x ur) = map([x_1, x_2, \dots, x_k, x]++) xss$$

где xss это списки

$$\begin{aligned} &[x_{k+2}, x_{k+3}, \dots, x_m] \\ &[x_{k+1}, x_{k+3}, \dots, x_m] \\ &\dots \\ &[x_{k+1}, x_{k+2}, \dots, x_{m-1}] \\ &[x_{k+1}, x_{k+2}, \dots, x_{m-1}, x_m] \end{aligned}$$

Первый из этих списков является покомпонентно наибольшим (хотя не обязательно единственным), следовательно, он является наименьшим относительно отношения \trianglelefteq . Словом, отношение \preccurlyeq можно минимизировать, если добавить x в начало последовательности, чей первый элемент больше или равен x . Фактически $insert$ можно определить так:

$$\begin{aligned} insert x [] &= [[x]] \\ insert x (xs : xss) &= \text{if } x \leqslant head xs \text{ then } (x : xs) : xss \\ &\quad \text{else } xs : insert x xss \end{aligned}$$

Инвариантом для $insert x ur$ является тот факт, что $map head ur$ строго возрастает. Следовательно, лучший способ вставить x это добавить его к началу первой последовательности в ur , элемент в голове которой не меньше x . Функция $insert s ur$ выполняется за линейное время по длине ur , однако сложность можно уменьшить до логарифмической, либо представив возрастающие распутки как массивы списков и применив бинарный поиск,

либо воспользовавшись сбалансированными деревьями. Мы опустим дальнейшие детали и просто заявим, что *foldr insert []* можно реализовать так, чтобы требовалось $O(n \log n)$ шагов.

Давайте теперь обратимся к доказательству утверждения (8.2). Пусть $\text{heads } ur = [x_1, x_2, \dots, x_m]$, а $\text{heads } vr = [y_1, y_2, \dots, y_n]$, так что $m \leq n$ и $y_i \leq x_i$ при $1 \leq i \leq m$. Как мы видели выше,

$$\begin{aligned}\text{heads}(\text{insert } x \text{ } ur) &= [x_1, x_2, \dots, x_k, x, x_{k+2}, x_{k+3}, \dots, x_m] \\ \text{heads}(\text{insert } x \text{ } vr) &= [y_1, y_2, \dots, y_\ell, x, x_{\ell+2}, x_{\ell+3}, \dots, y_n]\end{aligned}$$

где $x_k < x \leq x_{k+1}$ и $y_\ell < x \leq y_{\ell+1}$. Но $y_k \leq x_k$, поэтому $k \leq \ell$. Выравнивая два списка примерно так:

$$\begin{aligned}[x_1, x_2, \dots, x_k, x, &\quad x_{k+2}, \dots, x_\ell, x_{\ell+1}, x_{\ell+2}, \dots, x_m] \\ [y_1, y_2, \dots, y_k, y_{k+1}, y_{k+2}, \dots, y_\ell, x, &\quad y_{\ell+2}, \dots, y_m]\end{aligned}$$

можно заметить, что первый покомпонентно больше второго, поскольку $y_{k+1} \leq x$ и $x \leq x_{\ell+1}$.

Подводя итог, задача вычисления кратчайшей возрастающей распутки заданного списка может быть решена жадным алгоритмом, которому необходимо $O(n \log n)$ операций по длине списка.

Заключительные замечания

Задача отыскания кратчайшей возрастающей распутки была предложена и решена Ламбертом Мертенсом (Lambert Mertens) в сентябре 1984 года на встрече IFIP Working Group 2.1 в Понт-а-Муссоне во Франции (Meertens, 1984). Позднее в (Kaldewaij, 1985) было опубликовано совершенно другое решение. Одностороннее решение Калдевайжа основывалось на конструктивном доказательстве специальной версии теоремы Дилвортса (Dilworth): размер кратчайшей возрастающей распутки списка *xs* равен длине наибольшей убывающей его подпоследовательности. Этот факт можно совместить с известным алгоритмом поиска длины наибольшей убывающей подпоследовательности со сложностью $O(n \log n)$, получив алгоритм той же сложности для кратчайшей возрастающей распутки. Эта жемчужина основана на (Bird, 1992), где также рассматривается другой жадный алгоритм, начинающийся со следующего определения *unravels*:

$$\begin{aligned}\text{unravels} [] &= [[]] \\ \text{unravels } xs &= [ys : yss \mid ys \leftarrow \text{subseqs } xs, \text{not } (\text{null } ys), \\ &\quad yss \leftarrow \text{unravels } (xs \setminus ys)]\end{aligned}$$

Кратчайшую возрастающую распутьку можно затем получить, извлекая на каждой стадии самую правую максимальную возрастающую последовательность, вычисляемую функцией $rmi = foldr op []$, где

$$\begin{aligned} op\ x\ [] &= [x] \\ op\ x\ (y : ys) &= \text{if } x \leq y \text{ then } x : y : ys \text{ else } y : ys \end{aligned}$$

Вывод этой версии алгоритма оставляется в качестве упражнения.

Литература

- Bird, R. S. (1992). The smallest upravel. *Science of Computer Programming* **18**, 281–92.
- Kaldewaij, A. (1985). On the decomposition of sequences into ascending subsequences. *Information Processing Letters* **21**, 69.
- Meertens, L. G. L. T. (1984). Some more examples of algorithmic developments. *IFIP WG2.1 Working Paper*, Pont-à-Mousson, France. См. также *An Abstract Reader prepared for IFIP WG 2.1*. Technical Report CWI Note CS-N8702, Centrum voor Wiskunde en Informatica, April 1987.

9

Поиск знаменитостей

Действие происходит на занятии по проектированию функциональных алгоритмов. В классе четверо студентов: Анна, Иван, Мария и Фёдор.

Учитель. Доброе утро, ребята. Сегодня я бы хотел, чтобы вы решили вот такую задачу. Представьте множество участников вечеринки P . Будем говорить, что подмножество C множества P составляет клику знаменитостей, если любой участник вечеринки знает каждого члена C , тогда как члены C знают только друг друга. Ваша задача заключается в написании функциональной программы для отыскания клики знаменитостей на вечеринке в предположении, что она существует. В качестве исходных данных вы получаете бинарный предикат `knows` и множество P в виде списка без дубликатов `ps`.

Иван. Уточняю для ясности, действительно ли каждый член клики знаменитостей знает любого другого члена клики? Знает ли каждый самого себя?

Учитель. Что касается первого вопроса, ответ «да», это следует из определения: каждый в клике известен любому участнику вечеринки. Что же до второго вопроса, то ответ вряд ли имеет отношение к задаче, поэтому лучше спросите у знакомого философа. Если это что-то упростит, то можете считать для любого x на вечеринке x знает x .

Фёдор. Предполагается, что это трудная задача? В смысле, задача определения, имеется ли клика размера k на вечеринке с n участниками, по-

требует $\Omega(n^k)$ операций, поэтому нам нужен экспоненциальный по времени алгоритм.

Анна. Необязательно, потому что свойство быть кликой знаменитостей гораздо сильнее, нежели быть просто кликой. Клика в направленном графе — это множество вершин, для каждой пары которых имеются дуги в обоих направлениях. А вот клика знаменитостей требует дополнительно, чтобы существовала дуга из каждой вершины графа в каждую вершину этой клики и не существовало дуг, выходящих из вершин клики за её пределы.

Мария. Да, тогда как в графе может быть сколько угодно клик, более одной клики знаменитостей быть не может. Пусть C_1 и C_2 — две клики знаменитостей. Выберем произвольные c_1 в C_1 и c_2 в C_2 . Так как любой член клики C_2 известен любому участнику вечеринки, то c_1 знает c_2 . Но поскольку члены клики знаменитостей знают только других членов клики, то $c_2 \in C_1$. В силу произвольности c_2 получаем, что $C_2 \subseteq C_1$ и, симметрично, $C_1 \subseteq C_2$.

Фёдор. Согласен, это разные задачи. Давайте формализуем нашу. Для простоты я собираюсь предположить, что предикат x знает x является истинным для любого x . По определению, C является кликой знаменитостей вечеринки P , если $\emptyset \subset C \subseteq P$ и

$$(\forall x \in P, y \in C : x \text{ knows } y \wedge (y \text{ knows } x \Rightarrow x \in C))$$

Обозначим последнее условие как $C \triangleleft P$. При заданных списках ps и cs , представляющих соответственно P и C , это условие можно записать в виде генератора списка:

$$cs \triangleleft ps = \text{and} [x \text{ knows } y \wedge (y \text{ knows } x \Rightarrow x \in cs) \mid x \leftarrow ps, y \leftarrow cs]$$

Теперь определим $cclique ps = \text{head} (\text{filter} (\triangleleft ps) (\text{subseqs} ps))$, где $\text{subseqs} ps$ — это список всех подпоследовательностей ps :

$$\begin{aligned} \text{subseqs} [] &= [[]] \\ \text{subseqs} (x : xs) &= \text{map} (x:) (\text{subseqs} xs) \uplus \text{subseqs} xs \end{aligned}$$

Так как $\text{subseqs} ps$ генерирует подпоследовательности в порядке убывания длины, значением $cclique ps$ будет либо пустой список, если клика знаменитостей не существует, либо найденная единственная клика знаменитостей.

Иван. Решение с полным перебором, которое привёл Фёдор, кажется подходящим вариантом для начала рассуждений. Ясно, что улучшить эффективность можно, объединив фильтрацию с генерацией подпоследовательностей. В базовом случае, когда на вечеринке никого нет, получаем $\text{filter}(\triangleleft[])(\text{subseqs}[]) = [[],$ поскольку $cs \triangleleft [] = \text{True}$. На шаге индукции можно рассуждать так:

$$\begin{aligned} & \text{filter}(\triangleleft(p : ps))(\text{subseqs}(p : ps)) \\ &= \{ \text{определение } \text{subseqs} \} \\ & \text{filter}(\triangleleft(p : ps))(\text{map}(p:)(\text{subseqs } ps) + \text{subseqs } ps) \\ &= \{ \text{так как для filter и } + \text{ действует дистрибутивный закон} \} \\ & \text{filter}(\triangleleft(p : ps))(\text{map}(p:)(\text{subseqs } ps)) + \\ & \quad \text{filter}(\triangleleft(p : ps))(\text{subseqs } ps) \end{aligned}$$

Что дальше?

Анна. Нужно упростить $(p : cs) \triangleleft (p : ps)$ и $cs \triangleleft (p : ps)$, где cs это подпоследовательность ps , а p не содержится в cs . Давайте сначала поработаем со вторым случаем. Из определения \triangleleft следует, что $cs \triangleleft (p : ps)$ только тогда, когда $cs \triangleleft ps$, т.е. никакая знаменитость из cs не знает p , а p знает всех знаменитостей в cs . Формально:

$$cs \triangleleft (p : ps) = cs \triangleleft ps \wedge \text{nonmember } p \text{ } cs$$

где

$$\text{nonmember } p \text{ } cs = \text{and } [p \text{ knows } c \wedge \text{not } (c \text{ knows } p) \mid c \leftarrow cs]$$

Теперь можно рассуждать:

$$\begin{aligned} & \text{filter}(\triangleleft(p : ps))(\text{subseqs } ps) \\ &= \{ \text{приведённое выше упрощение } cs \triangleleft (p : ps) \} \\ & \text{filter}(\lambda cs \rightarrow cs \triangleleft ps \wedge \text{nonmember } p \text{ } cs)(\text{subseqs } ps) \\ &= \{ \text{так как } \text{filter}(\lambda x \rightarrow p \text{ } x \wedge q \text{ } x) = \text{filter } q \cdot \text{filter } p \} \\ & \text{filter}(\text{nonmember } p)(\text{filter}(\triangleleft ps)(\text{subseqs } ps)) \end{aligned}$$

Теперь разберёмся с первым случаем. Условие $(p : cs) \triangleleft (p : ps)$ выполняется только в том случае, когда $cs \triangleleft ps$ и p — только что обнаруженная знаменитость, т.е. все знают p , а p знакомо только со всеми членами cs . Формально:

$$(p : cs) \triangleleft (p : ps) = cs \triangleleft ps \wedge \text{member } p \text{ } ps \text{ } cs$$

где

$$\text{member } p \text{ } ps \text{ } cs = \text{and} [x \text{ knows } p \wedge (p \text{ knows } x \Leftrightarrow x \in cs) \mid x \leftarrow ps]$$

Похожие на предыдущие вычисления дают нам:

$$\begin{aligned} \text{filter} (\triangleleft (p : ps)) (\text{map} (p:) (\text{subseqs} \text{ } ps)) \\ = \text{map} (p:) (\text{filter} (\text{member} \text{ } p \text{ } ps) (\text{filter} (\triangleleft ps) (\text{subseqs} \text{ } ps))) \end{aligned}$$

Собирая вместе два этих результата, получаем $ccliques = \text{head} \cdot ccliques_8$, где

$$\begin{aligned} ccliques [] &= [[]] \\ ccliques (p : ps) &= \text{map} (p:) (\text{filter} (\text{member} \text{ } p \text{ } ps) \text{ } css) \uplus \\ &\quad \text{filter} (\text{nonmember} \text{ } p) \text{ } css \\ &\text{where } css = ccliques \text{ } ps \end{aligned}$$

Предикаты *member* и *nonmember* можно вычислить за линейное время, а поскольку *ccliques* возвращает самое большее два списка, искомую клику знаменитостей и пустой список, то экспоненциальный алгоритм свёлся к квадратичному.

Фёдор. Ну, сделать это быстрее, чем за квадратичное время, невозможно, по крайней мере в худшем случае. Предположим, что существует менее чем квадратичное решение. Тогда как минимум одно значение предиката *knows* не было бы проверено. Предположим далее, что все значения истинны, так что все знают всех, и клика знаменитостей состоит из всех участников вечеринки. Изменим теперь непроверенное значение предиката, скажем $\text{knows } x \text{ } y$, на ложное. Тогда y уже не знаменитость. Но все за исключением x по-прежнему знают y , поэтому они не могут быть знаменитостями. Получается, что единственной знаменитостью может оказаться только x , но если в вечеринке участвуют не только x и y , то существует не-знаменитость, которую знает x , поэтому и x не может быть знаменитостью. Следовательно, на подправленной таким образом вечеринке нет клики знаменитостей, и выдуманное нами менее чем квадратичное решение дало неверный ответ. Поэтому в худшем случае для получения правильного ответа следует проверить каждое значение предиката *knows*.

Учитель. Это всё замечательно, Фёдор, и совершенно верно, но задача заключается не в том, чтобы определить, имеется ли на вечеринке клика

знаменитостей или нет. Всё что я просил, это найти клику знаменитостей в предположении, что она существует. В твоём сценарии ответа ps достаточ- но в каждом из двух случаев: в первом случае это правильный ответ, а во втором клики знаменитостей вообще нет, поэтому подойдёт любой ответ.

Пауза, в течение которой учащиеся обдумывают полученную информацию.

Мария. Есть идея! Рассуждение Анны показало, что для всех x

$$xs \triangleleft (p : ps) \Rightarrow (xs \setminus\setminus [p]) \triangleleft ps$$

где $\setminus\setminus$ обозначает разность списков. Другими словами, если $cs \triangleleft ps$ и p — некто, только что пришедший на вечеринку ps , то возможными значениями для xs , удовлетворяющими условию $xs \triangleleft (p : ps)$, будут $[]$, $[p]$, cs или $p : cs$. Думаю, это приведёт нас к другому способу решения задачи. Предположим сначала, что список cs пуст, так что единственной возможной кликой знаменитостей на вечеринке $p : ps$ является $[p]$. Формально:

$$\text{null}(\text{cclique } ps) \Rightarrow \text{cclique}(p : ps) \in \{[], [p]\}$$

С другой стороны, предположим, что cs не пуст и содержит некоторую знаменитость c . Каковы возможные результаты присоединения к вечеринке нового участника p ? Ну, положим, p не знает c . Тогда ни c более не знаменитость в расширенной вечеринке, ни все другие элементы cs , поскольку они все знают c . Следовательно,

$$c \in \text{cclique } ps \wedge \text{not}(p \text{ knows } c) \Rightarrow \text{cclique}(p : ps) \in \{[], [p]\}$$

Положим теперь, что p знает c , но c не знает p . В этом случае cs остаётся кликой знаменитостей расширенной вечеринки:

$$\begin{aligned} c \in \text{cclique } ps \wedge p \text{ knows } c \wedge \text{not}(c \text{ knows } p) \\ \Rightarrow \text{cclique}(p : ps) = \text{cclique } ps \end{aligned}$$

Наконец, если p и c знают друг друга, то единственной кликой знамени- тостей вечеринки $p : ps$ становится $p : cs$:

$$\begin{aligned} c \in \text{cclique } ps \wedge p \text{ knows } c \wedge c \text{ knows } p \\ \Rightarrow \text{cclique}(p : ps) \in \{[], p : cs\} \end{aligned}$$

Фёдор. Хотя я и согласен, что твои рассуждения правильны, Мария, мне не понятно, как они помогут решить задачу. Ты показала только то, что если мы знаем значение $cclique\ ps$ и если на вечеринке $p : ps$ есть клика знаменитостей, то мы можем быстро её найти. Но как нам искать значение $cclique\ ps$? Кажется, ты предполагаешь, что если мы определим $cclique'$ так:

$$\begin{aligned} cclique' &= foldr\ op\ [] \\ op\ p\ cs &\mid null\ cs = [p] \\ &\mid not\ (p\ knows\ c) = cs \\ &\mid not\ (c\ knows\ p) = p : cs \\ &\mid \text{otherwise} = p : cs \\ &\text{where } c = head\ cs \end{aligned}$$

то

$$not\ (null\ (cclique\ ps)) \Rightarrow cclique\ ps = cclique'\ ps \quad (9.1)$$

Но я не понимаю, почему твои рассуждения доказывают (9.1).

Мария. Позволь попробовать ещё раз. Я буду доказывать (9.1) индукцией по ps . Имеется два случая. В случае пустого списка $cclique[] = []$, поэтому (9.1) выполняется по умолчанию. В случае списка $p : ps$ предположим, что $cclique\ (p : ps)$ непусто. Возникает два варианта в зависимости от того, пусто ли $cclique\ ps$ или нет. Если нет, то $cclique\ ps = cclique'\ ps$ согласно предположению индукции. В этом случае имеем:

$$\begin{aligned} &cclique\ (p : ps) \\ &= \{ \text{предыдущие рассуждения и определение } op \} \\ &\quad op\ p\ (cclique\ ps) \\ &= \{ \text{так как } cclique\ ps = cclique'\ ps \} \\ &\quad op\ p\ (cclique'\ ps) \\ &= \{ \text{определение } cclique' \} \\ &\quad cclique'\ (p : ps) \end{aligned}$$

Если же $cclique\ ps$ пусто, то

$$cclique\ (p : ps)$$

$$\begin{aligned}
 &= \{ \text{предположение о непустоте } cclique(p : ps) \\
 &\quad \text{и первый случай предыдущих рассуждений} \} \\
 [p] \\
 &= \{ p \text{ является единственной знаменитостью, поэтому} \\
 &\quad \text{not}(p \text{ knows } c) \text{ для всех } c \in cclique' ps \} \\
 op\ p\ (cclique'\ ps) \\
 &= \{ \text{как ранее} \} \\
 cclique'\ (p : ps)
 \end{aligned}$$

Таким образом, доказан этот случай и индукционный переход в целом.

Анна. Удивительно, простой линейный алгоритм! Но мы пришли к решению только благодаря тому, что Мария так умна. Я всё же предпочту формальный вывод $cclique'$ из какого-нибудь подходящего закона слияния.

Учитель. Спасибо, Анна, хорошо, что ты здесь есть.

Мария. Мы можем записать $subseqs$, пользуясь $foldr$:

$$\begin{aligned}
 subseqs &= foldr add [[]] \\
 add\ x\ xss &= map\ (x:) xss + xss
 \end{aligned}$$

Поэтому выходит так, что мы обращаемся к некоторому закону слияния для $foldr$. Известное утверждение закона слияния для $foldr$ гласит, что $f \cdot foldr g a = foldr h b$ при условии, что f строгая функция, $f a = b$, а $f(g x y) = h x (f y)$ для всех x и y . Условие строгости не требуется, если нам достаточно утверждения, что $f(foldr g a xs) = foldr h b xs$ для всех конечных списков xs . Это правило слияния нельзя напрямую применить к задаче о клике знаменитостей, а именно $filter(\triangleleft ps)(subseqs ps)$, во-первых, потому, что $filter(\triangleleft ps)$ принимает в качестве параметра ps , и во-вторых, потому, что нам требуется нечто более общее, чем равенство двух сторон.

Фёдор. Первое ограничение не проблема. Всегда можно определить вариант $subseqs$, в котором возвращались бы и подпоследовательности списка, и сам список. Допустим, мы определили:

$$\begin{aligned}
 subseqs' &= foldr step ([], [[]]) \\
 step\ x\ (xs, xss) &= (x : xs, map\ (x:) xss + xss)
 \end{aligned}$$

Тогда $cclique = f \cdot subseqs'$, где $f(ps, css) = head(filter(\triangleleft ps) css)$. Благодаря этому дополнительного параметра больше нет.

Мария. Приведённое выше утверждение закона слияния не достаточно общо для нашей задачи. Пусть имеется некоторое отношение \rightsquigarrow на значениях, неважно какое. Тогда индукцией легко показать, что

$$f(foldr g a xs) \rightsquigarrow foldr h b xs$$

для всех конечных списков xs при условии, что $f a \rightsquigarrow b$ и $f y \rightsquigarrow z \Rightarrow f(g x y) \rightsquigarrow h x z$ для всех x, y и z .

Иван. Да, точно. Мы хотим определить $xs \rightsquigarrow ys$ так:

$$xs \rightsquigarrow ys = not(null xs) \Rightarrow xs = ys$$

Тогда нам следует установить, во-первых, что

$$head(filter(\triangleleft [])([[]])) \rightsquigarrow []$$

и, во-вторых, что

$$\begin{aligned} head(filter(\triangleleft ps) css) &\rightsquigarrow cs \\ \Rightarrow head(filter(\triangleleft (p : ps)) (map(p:) css ++ css)) &\rightsquigarrow op p cs \end{aligned}$$

Рассуждения, которые приводила Мария, доказывают именно это.

Учитель. Да. Более общее утверждение относительно слияния обеспечивается параметричностью из статьи (Wadler, 1989) «Бесплатные теоремы» (“Theorems for free”). Приятно встретить пример, где требуется более общее утверждение. Что в этой задаче интересно, так это то, что она доставляет первый известный мне пример, в котором найти решение в предположении, что оно существует, асимптотически быстрее, чем проверить, что оно действительно является решением. Похожая ситуация возникает в задаче о голосующем большинстве — см., к примеру, главу 18 в (Morgan, 1994) — там дан список xs , для которого требуется определить, содержит ли он значение, встречающееся в нём более $\lfloor length xs / 2 \rfloor$ раз. Проще сначала вычислить предполагаемое большинство, а затем проверить, является ли оно реальным. Правда, для проверки на большинство достаточно линейного, а не квадратичного, времени, поэтому асимптотический зазор там не возникает.

Послесловие

За задачей о клике знаменитостей стоит реальная история. Я читал курс лекций по формальному конструированию программ с императивным подходом вместо функционального и думал над обобщением одной из задач в книге (Kaldewaij, 1990). Однако после целого дня борьбы с инвариантами циклов я так и не смог предложить решение, достаточно простое для того, чтобы представить его на лекции. Поэтому я оставил эту задачу на самостоятельное решение. О ней же я рассказал на исследовательской встрече в следующую пятницу. После выходных Шарон Кёртис (Sharon Curtis) предложил простой линейный алгоритм, а Джюлиан Тибл (Julian Tibble), студент-второкурсник, предоставил хороший способ рассуждений.

Будучи уверенным, что всё, что можно сделать циклами и инвариантами, можно, как минимум, столь же просто реализовать с использованием законов порождения функциональных программ, я переформулировал задачу в функциональном окружении и сочинил этот разговор. Впоследствии, в 2004 году, задача была представлена на встрече WG2.1 в Ноттингеме. Приятно, что обсуждение довольно близко следовало началу описанного выше разговора. После настойчивых рекомендаций подумать получше Andres Löh (Andres Löh) и Йохан Джеринг (Johan Jeuring) вернулись через день с линейными решениями.

Литература

- Kaldewaij, A. (1990). *Programming the Derivation of Algorithms*. Hemel Hempstead: Prentice Hall.
- Morgan, C. (1994). *Programming from Specifications*, 2nd edition. Hemel Hempstead: Prentice Hall.
- Wadler, P. (1989). Theorems for free! *Fourth International Symposium on Functional Programming Languages and Computer Architecture*. ACM Press, pp. 347–59.

10

Удаляем повторы

Введение

Библиотечная функция языка Haskell *nub* удаляет из списка повторяющиеся элементы:

$$\begin{aligned} nub &:: Eq a \Rightarrow [a] \rightarrow [a] \\ nub [] &= [] \\ nub (x : xs) &= x : nub (xs \setminus\setminus [x]) \end{aligned}$$

Значение $xs \setminus\setminus ys$ это то, что остаётся от списка xs после удаления всех элементов ys . Например, $nub "calculus" = "calus"$. Вычисление значения функции *nub* на списке длины n требует $\Theta(n^2)$ операций. Это лучшее, на что можно надеяться, потому что любой алгоритм требует в худшем случае $\Omega(n^2)$ сравнений. При данном выше определении *nub xs* возвращает список неповторяющихся элементов списка xs в том порядке, в котором они входили в xs . Другими словами, позиция списка *nub xs* как подпоследовательности xs является лексикографически наименьшей среди всех возможных решений.

Изменим задачу, потребовав, чтобы $nub :: Ord a \Rightarrow [a] \rightarrow [a]$ возвращала просто лексикографически наименьшее решение. Заметьте тонкое различие: ранее лексикографически наименьшей должна была быть позиция, теперь же сама подпоследовательность. Например, согласно второму определению $nub "calculus" = "aclus"$. Изменение типа *nub* необходимо, чтобы сделать новую задачу осмыслинной. Изменение типа *nub* уменьшает

нижнюю границу сложности: теперь в худшем случае необходимо только $\Omega(n \log n)$ операций. Замечательное доказательство этого утверждения приводится в (Bloom and Wright, 2003). Можно ли написать программу для новой версии *nub* со сложностью $\Theta(n \log n)$? Оказывается, ответ положительный, но алгоритм не так уж очевиден, и для его вывода потребуется поработать. Будьте готовы.

Первый вариант

Начнём со спецификации:

$$\text{nub} = \text{minimum} \cdot \text{longest} \cdot \text{filter nodups} \cdot \text{subseqs}$$

Словом, вычисляем все возможные подпоследовательности заданного списка (*subseqs*), фильтруем список подпоследовательностей так, чтобы в нём не осталось подпоследовательностей с повторениями (*filter nodups*), вычисляем самые длинные подпоследовательности (*longest*) и, наконец, выбираем наименьшую (*minimum*).

Из этой спецификации не слишком трудно получить следующее рекурсивное определение *nub*:

$$\begin{aligned}\text{nub} [] &= [] \\ \text{nub} (x : xs) &= \text{if } x \notin xs \text{ then } x : \text{nub} xs \text{ else} \\ &\quad (x : \text{nub} (xs \setminus [x])) \min (\text{nub} xs)\end{aligned}$$

Мы опустим детали, оставив их в качестве упражнения для заинтересованного читателя. В любом случае, рекурсивный вариант довольно интуитивный: в случае $x : xs$ либо x не встречается в xs , поэтому выбора нет, либо встречается, и тогда результатом будет наименьшая из двух альтернатив: выбирать ли x сейчас или позднее.

Проблема рекурсивного определения *nub* в том, что оно требует экспоненциального времени, поскольку число рекурсивных вызовов на каждом шаге удваивается. Поэтому для достижения оценки $\Theta(n \log n)$ нам предстоит потрудиться.

Обобщение

Первой мыслью, с учётом целевой сложности, является алгоритм по методу «разделяй и властвуй» с использованием функции *join*, для которой

$$\text{nub} (xs ++ ys) = \text{join} (\text{nub} xs) (\text{nub} ys) \tag{10.1}$$

Но такой функции не существует. Например, (10.1) требует, чтобы

$$\text{join "bca" "c" } = \text{join} (\text{nub "bca"}) (\text{nub "c"}) = \text{nub "bcac" } = \text{"bac"}$$

Одновременно с этим (10.1) требует, чтобы

$$\text{join "bca" "c" } = \text{join} (\text{nub "bcab"}) (\text{nub "c"}) = \text{nub "bcabc" } = \text{"abc"}$$

Этот пример также показывает, что *nub* невозможно выразить как экземпляр свёртки *foldl*. Нетрудно сконструировать пример, показывающий, что *nub* нельзя выразить и правой свёрткой *foldr*.

Таким образом, нам необходимо некоторое обобщение *nub*. Чтобы понять, что это может быть, рассмотрим список в виде $x : y : xs$, где $x \in xs$, $y \in xs$ и $x \neq y$. Разворачивая определение *nub* ($x : y : xs$) и используя ассоциативность *min* вместе с тем фактом, что для $(x:)$ имеет место дистрибутивный закон относительно *min*, находим:

$$\begin{aligned} \text{nub} (x : y : xs) &= x : y : \text{nub} (xs \setminus [x, y]) \text{ min} \\ &\quad x : \text{nub} (xs \setminus [x]) \text{ min} \\ &\quad y : \text{nub} (xs \setminus [y]) \text{ min} \\ &\quad \text{nub} xs \end{aligned}$$

Предположим теперь, что $x < y$, т.е. второй терм лексикографически меньше третьего. Это означает, что третий терм можно отбросить:

$$\begin{aligned} \text{nub} (x : y : xs) &= x : y : \text{nub} (xs \setminus [x, y]) \text{ min} \\ &\quad x : \text{nub} (xs \setminus [x]) \text{ min} \\ &\quad \text{nub} xs \end{aligned}$$

Если, с другой стороны, $x > y$, то можно отбросить первые два терма:

$$\begin{aligned} \text{nub} (x : y : xs) &= y : \text{nub} (xs \setminus [y]) \text{ min} \\ &\quad \text{nub} xs \end{aligned}$$

Вид двух этих выражений подсказывает идею обобщения, которое мы назовём *hub*. Далее, чтобы сохранить выражения достаточно краткими, будем сокращать *minim* как *min*. Вот определение *hub*:

$$\text{hub ws xs } = \text{min} [\text{is} + \text{nub} (xs \setminus \text{is}) \mid \text{is} \leftarrow \text{inits ws}] \tag{10.2}$$

где *ws* это список, элементы которого строго возрастают. Стандартная функция *inits* возвращает список всех начальных сегментов, или префиксов, заданного списка. Приведённый выше пример теперь можно записать так:

$$\text{nub } (x : y : xs) = \text{if } x < y \text{ then } hub [x, y] xs \text{ else } hub [y] xs$$

Функция hub обобщает nub , поскольку $\text{inits} [] = [[]]$ и $xs \setminus\setminus [] = xs$, а значит, $\text{nub } xs = hub [] xs$. Два непосредственных факта относительно hub состоят в том, что $hub ws xs$ начинается с префикса ws и что $hub ws xs \leq nub xs$, поскольку пустой список является префиксом любого списка.

Наша цель теперь в том, чтобы вывести индуктивное определение hub . В базовом случае рассуждаем так:

$$\begin{aligned} hub ws [] &= \{ \text{определение} \} \\ &= \min [\text{is} \doteqdot \text{nub} ([] \setminus\setminus \text{is}) \mid \text{is} \leftarrow \text{inits } ws] \\ &= \{ \text{так как } [] \setminus\setminus \text{is} = [] \text{ и } \text{nub } [] = [] \} \\ &= \min [\text{is} \mid \text{is} \leftarrow \text{inits } ws] \\ &= \{ \text{так как } [] \text{ является лексикографически наименьшим списком} \\ &\quad \text{в } \text{inits } ws \} \\ &= [] \end{aligned}$$

Следовательно, $hub ws [] = []$. На индукционном переходе (10.2) даёт следующее:

$$hub ws (x : xs) = \min [\text{is} \doteqdot \text{nub} ((x : xs) \setminus\setminus \text{is}) \mid \text{is} \leftarrow \text{inits } ws] \quad (10.3)$$

Чтобы упростить правую часть, нам нужно знать, принадлежит ли x списку ws или нет, поэтому начинаем с разбиения ws на два списка us и vs , определяемые так:

$$(us, vs) = (\text{takeWhile } (< x) ws, \text{dropWhile } (< x) ws)$$

Короче говоря, $(us, vs) = \text{span } (< x) ws$, где span это стандартная функция языка Haskell. Поскольку $ws = us \doteqdot vs$ и ws возрастает, то и us , и vs также возрастают. Более того, если $x \in ws$, то $x = \text{head } vs$; если же нет, то либо vs пусто, либо $x < \text{head } vs$.

Для упрощения (10.3) ключевым является следующее свойство inits :

$$\text{inits } (us \doteqdot vs) = \text{inits } us \doteqdot \text{map } (us \doteqdot) (\text{inits}^+ vs) \quad (10.4)$$

где $\text{inits}^+ vs$ возвращает список непустых префиксов vs . Используя это выражение для inits в (10.3) и разбивая генератор на две части, получаем, что $hub ws (x : xs) = A \min B$, где

$$A = \min [\text{is} \doteqdot \text{nub} ((x : xs) \setminus\setminus \text{is}) \mid \text{is} \leftarrow \text{inits } us] \quad (10.5)$$

$$B = \min [us \doteqdot \text{is} \doteqdot \text{nub} ((x : xs) \setminus\setminus (us \doteqdot \text{is})) \mid \text{is} \leftarrow \text{inits}^+ vs] \quad (10.6)$$

Разберёмся, для начала, с A , рассмотрев по отдельности два случая $x \notin xs$ и $x \in xs$. В первом случае, $x \notin xs$, получаем:

$$\begin{aligned}
 A &= \{ \text{определение (10.5)} \} \\
 &= \min [is ++ nub ((x : xs) \setminus\setminus is) \mid is \leftarrow \text{inits } us] \\
 &= \{ \text{рекурсивное определение } nub \text{ с учётом того,} \\
 &\quad \text{что } x \notin xs \text{ и } x \notin us \} \\
 &= \min [is ++ [x] ++ nub (xs \setminus\setminus is) \mid is \leftarrow \text{inits } us] \\
 &= \{ \text{так как } us < is ++ [x], \text{ если } is \in \text{inits } us \} \\
 &= us ++ [x] ++ nub (xs \setminus\setminus us) \\
 &= \{ \text{так как } nub xs = hub [] xs \} \\
 &= us ++ [x] ++ hub [] (xs \setminus\setminus us)
 \end{aligned}$$

Во втором случае, $x \in xs$, имеем:

$$\begin{aligned}
 A &= \{ \text{рекурсивное определение } nub \text{ с учётом того,} \\
 &\quad \text{что } x \in xs \text{ и } x \notin us \} \\
 &= \min [is ++ ([x] ++ nub (xs \setminus\setminus (is ++ [x]))) \min nub (xs \setminus\setminus is)] \\
 &\quad \mid is \leftarrow \text{inits } us \\
 &= \{ \text{вынося } \min \text{ из генератора} \} \\
 &= \min [is ++ [x] ++ nub (xs \setminus\setminus (is ++ [x])) \mid is \leftarrow \text{inits } us] \min \\
 &\min [is ++ nub (xs \setminus\setminus is) \mid is \leftarrow \text{inits } us] \\
 &= \{ \text{так как } us < is ++ [x], \text{ если } is \in \text{inits } us \} \\
 &= (us ++ [x] ++ nub (xs \setminus\setminus (us ++ [x]))) \min \\
 &\min [is ++ nub (xs \setminus\setminus is) \mid is \leftarrow \text{inits } us] \\
 &= \{ \text{так как } \text{inits}(us ++ [x]) = \text{inits } us ++ [us ++ [x]] \} \\
 &= \min [is ++ nub (xs \setminus\setminus is) \mid is \leftarrow \text{inits}(us ++ [x])] \\
 &= \{ \text{определение (10.2)} \} \\
 &= hub (us ++ [x]) xs
 \end{aligned}$$

Собирая всё вместе, получаем, что A равно

$$\text{if } x \in xs \text{ then } hub (us ++ [x]) xs \text{ else } us ++ [x] ++ hub [] (xs \setminus\setminus us)$$

Теперь обратимся к B . Согласно (10.6), если vs пусто (т.е. пусто $\text{inits}^+ vs$), то B является фиктивным значением $\min []$. В противном случае рассуждаем:

$$\begin{aligned}
 & B \\
 & = \{ \text{определение (10.6)} \} \\
 & min [us ++ is ++ nub ((x : xs) \setminus\setminus (us ++ is)) \mid is \leftarrow inits^+(v : vs')] \\
 & = \{ \text{так как } inits^+(v : vs') = map(v) (inits vs') \} \\
 & min [us ++ [v] ++ is ++ nub ((x : xs) \setminus\setminus (us ++ [v] ++ is)) \mid is \leftarrow inits vs'] \\
 & = \{ \text{так как } min \cdot map(ys++) = (ys++) \cdot min \} \\
 & us ++ [v] ++ min [is ++ nub ((x : xs) \setminus\setminus (us ++ [v] ++ is)) \mid is \leftarrow inits vs']
 \end{aligned}$$

В частности, B начинается с $us ++ [v]$. В случае $x \notin ws$ делать больше ничего не нужно, информации для отыскания $hub ws (x : xs)$ у нас достаточно. Действительно, либо vs пусто, т.е. $A < B$, либо vs непусто и начинается с v , где $x < v$. В последней ситуации снова получается, что $A < B$, поскольку A начинается с префикса $us ++ [x]$ и $us ++ [x] < us ++ [v]$. Следовательно,

$$x \notin ws \Rightarrow hub ws (x : xs) = A$$

Остаётся решить вопрос со случаем $x \in ws$, т.е. $x = v$. В этом случае B упрощается до

$$us ++ [x] ++ min [is ++ nub (xs \setminus\setminus (us ++ [x] ++ is)) \mid is \leftarrow inits vs']$$

Теперь нужно провести окончательный разбор вариантов. Предположим сначала, что $x \notin xs$. Вычисляем:

$$\begin{aligned}
 & B \\
 & = \{ \text{см. выше} \} \\
 & us ++ [x] ++ min [is ++ nub (xs \setminus\setminus (us ++ [x] ++ is)) \mid is \leftarrow inits vs'] \\
 & = \{ \text{в силу } xs \setminus\setminus (us ++ [x] ++ is) = xs \setminus\setminus (us ++ is) = (xs \setminus\setminus us) \setminus\setminus is \} \\
 & us ++ [x] ++ min [is ++ nub ((xs \setminus\setminus us) \setminus\setminus is) \mid is \leftarrow inits vs'] \\
 & = \{ \text{определение (10.2)} \} \\
 & us ++ [x] ++ hub vs' (xs \setminus\setminus us)
 \end{aligned}$$

Следовательно:

$$\begin{aligned}
 & hub ws (x : xs) \\
 & = \{ \text{выражения для } A \text{ и } B \text{ в предположении } x \notin xs \} \\
 & (us ++ [x] ++ nub (xs \setminus\setminus us)) min (us ++ [x] ++ hub vs' (xs \setminus\setminus us)) \\
 & = \{ \text{так как } hub vs' (xs \setminus\setminus us) \leqslant nub (xs \setminus\setminus us) \}
 \end{aligned}$$

$$\begin{aligned}
 nub &= hub []
 \\
 hub ws [] &= []
 \\
 hub ws (x : xs) &= \text{case } (x \in xs, x \in ws) \text{ of} \\
 &\quad (\text{False}, \text{False}) \rightarrow us ++ [x] ++ hub [] (xs \setminus\setminus us) \\
 &\quad (\text{False}, \text{True}) \rightarrow us ++ [x] ++ hub (tail ws) (xs \setminus\setminus us) \\
 &\quad (\text{True}, \text{False}) \rightarrow hub (us ++ [x]) xs \\
 &\quad (\text{True}, \text{True}) \rightarrow hub ws xs \\
 \text{where } (us, vs) &= span (<x) ws
 \end{aligned}$$
Рис. 10.1: второе определение *nub*

$$us ++ [x] ++ hub ws' (xs \setminus\setminus us)$$

В последнем случае, $x \in xs$, можно рассуждать так:

$$\begin{aligned}
 hub ws (x : xs) &= \{ \text{выражения для } A \text{ и } B \text{ в предположении } x \in xs \} \\
 hub (us ++ [x]) xs &\min \\
 us ++ [x] ++ min [is ++ nub (xs \setminus\setminus (us ++ [x] ++ is)) | is \leftarrow inits vs'] \\
 &= \{ \text{так как } ws = us ++ [x] ++ vs' \text{ и (10.4)} \} \\
 min [is ++ nub (xs \setminus\setminus is) | is \leftarrow inits ws] \\
 &= \{ \text{определение } hub \} \\
 hub ws xs
 \end{aligned}$$

Результат этих вычислений приведён на рис. 10.1. Каждая проверка на вхождение, вычисление разности списков и функция *span* выполняются за линейное время, поэтому вычисление *hub* требует линейного времени при каждом рекурсивном вызове и квадратичного времени в целом.

Вводим множества

Последний шаг заключается в использовании эффективного представления для множеств. Это позволит уменьшить сложность вспомогательных операций с линейной до логарифмической. Вместо того, чтобы реализовывать операции с множествами вручную, мы воспользуемся библиотекой *Data.Set*. Эта библиотека предоставляет тип *Set a* и среди прочих следующие операции:

$$\begin{aligned}
 empty &:: Set a \\
 member :: Ord a \Rightarrow a \rightarrow Set a \rightarrow Bool
 \end{aligned}$$

```

insert  :: Ord a ⇒ a → Set a → Set a
split    :: Ord a ⇒ a → Set a → (Set a, Set a)
elems   :: Ord a ⇒ Set a → [a]

```

Значение *empty* обозначает пустое множество, *member* это проверка на принадлежность, *insert* *x* *xs* вставляет новый элемент *x* во множество *xs*, тогда как *split* *x* разбивает множество на элементы, большие и меньшие *x*. Функция *elems* возвращает элементы множества в порядке возрастания. Что касается сложности этих операций, то *empty* выполняется за константное время, *member*, *insert* и *split* требуют $O(n \log n)$ операций на множестве размера *n*, а *elems* выполняется за $O(n)$ операций.

Чтобы ввести множества в определение *nub*, необходима предварительная обработка, которая ассоциирует каждый элемент *x* списка *xs* с множеством элементов, следующих за ним. Таким образом, нужно вычислить:

$$(x_1, \{x_2, x_3, \dots, x_n\}), (x_2, \{x_3, \dots, x_n\}), \dots, (x_n, \{\})$$

Это можно сделать с помощью функции *scanr*:

```

preprocess  :: Ord a ⇒ [a] → [(a, Set a)]
preprocess xs = zip xs (tail (scanr insert empty xs))

```

Выражение *scanr insert empty* $[x_1, x_2, \dots, x_n]$ возвращает список

$$[\{x_1, x_2, \dots, x_n\}, \{x_2, \dots, x_n\}, \dots, \{x_n\}, \{\}]$$

и требует для этого $O(n \log n)$ операций.

Результат после добавления множеств в решение на рис. 10.1 приведён на рис. 10.2. К сожалению, время пока не $O(n \log n)$. Чтобы в этом убедиться, давайте посчитаем затраты на различные операции. Каждая проверка на принадлежность добавляется к стоимости каждого рекурсивного вызова $O(\log n)$ операций. То же самое делает *split*. Пусть *m* — размер *us*. Так как *elems* выполняется за $O(m)$ шагов, и столько же шагов требуется для присоединения результата к остальной части списка, а в итоговом списке *n* элементов, то суммарный вклад *elems* и ++ в общую стоимость равен $O(n)$ операций. Однако затраты на вычисление *yss* равны $\Omega(n \log m)$, а суммирование этих затрат даёт $\Omega(n^2)$ операций. В качестве конкретного примера рассмотрим список

$$[1..n] \text{ ++ } [j \mid j \leftarrow [1..n], j \bmod 3 \neq 0]$$

```

nub          = hub empty · preprocess
preprocess xs = zip xs (tail (scanr insert empty xs))
hub ws []    = []
hub ws ((x, xs) : xss) =
  case (member x xs, member x ws) of
    (False, False) → eus ++ [x] ++ hub empty yss
    (False, True)  → eus ++ [x] ++ hub vs yss
    (True, False)  → hub (insert x us) xss
    (True, True)   → hub ws xss
  where (us, vs) = split x ws
        eus      = elems us
        yss      = [(x, xs) | (x, xs) ← xss, not (member x us)]
```

Рис. 10.2: использование множеств

Результат равен $[1..n]$. Каждое кратное числа 3 заставляет программу удалять из множества ws по два элемента, а именно $[1, 2], [4, 5], [7, 8]$ и т.д., значит, общие затраты на вычисление yss квадратичны по n .

Один из способов решить задачу заключается во введении в функцию hub дополнительного параметра ps , определим таким образом функцию hub' :

```

hub'           :: Set a → Set a → [(a, Set a)] → [a]
hub' ps ws xss = hub ws [(x, xs) | (x, xs) ← xss, x ∉ ps]
```

Тогда мы получим программу на рис. 10.3. Стоимость вычисления qs равна $O(m \log n)$, где m это размер us , а не $O(n \log m)$ для вычисления yss , как это было в предыдущем варианте. Так как суммарный размер множеств us , для которых выполняется эта операция, не превосходит n , то общее время выполнения равно $O(n \log n)$ операций.

Заключительные замечания

Для достижения этого результата потребовалось проделать много работы, а алгоритм не получился ни красивым, ни интуитивным. Остаются мучительные сомнения, кажется, что у задачи с такой простой постановкой должно быть гораздо более простое решение. Однако до сих пор найти его мне не удалось. Основное вычисление оказывается довольно замысловатым, к тому же его ещё больше запутывает разбор вариантов. Тем не менее, план сражения достаточно традиционен: получаем рекурсивную

```

nub      = hub' empty empty · preprocess
preprocess xs = zip xs (tail (scanr insert empty xs))
hub' ps ws [] = []
hub' ps ws ((x, xs) : xss) =
  if member x ps then hub' ps ws xss else
    case (member x xs, member x ws) of
      (False, False) → eus ++ [x] ++ hub' qs empty xss
      (False, True) → eus ++ [x] ++ hub' qs vs xss
      (True, False) → hub' ps (insert x us) xss
      (True, True) → hub' ps ws xss
  where (us, vs) = split x ws
    eus     = elems us
    qs      = foldr insert ps eus

```

Рис. 10.3: окончательное решение

постановку задачи, а затем ищем обобщённую версию, которая допускает эффективную реализацию. Тот же план возникает при выводе многих эффективных алгоритмов.

Литература

Bloom, S. L. and Wright, R. S. (2003). Some lower bounds on comparison-based algorithms. Unpublished research paper. Department of Computer Science, Steven's Institute of Technology, Hoboken, NJ, USA.

11

Вовсе не максимальная сумма сегмента

Введение

Задача отыскания сегмента с максимальной суммой была очень популярна в конце восьмидесятых, когда программисты любили использовать её как пример, иллюстрирующий их любимый стиль разработки программ или систему доказательства теорем. Задача заключается в вычислении максимальной из сумм всевозможных сегментов списка целых чисел, как положительных, так и отрицательных. Однако эта жемчужина не о максимальной сумме сегмента. Она о максимальной сумме *не-сегмента*. Сегментом списка называют непрерывную подпоследовательность, тогда как не-сегмент это подпоследовательность, не являющаяся сегментом. Например, для списка

$$[-4, -3, -7, +2, +1, -2, -1, -4]$$

максимальная сумма сегмента равна 3 (это сегмент $[+2, +1]$), а максимальная сумма не-сегмента равна 2 (не-сегмент $[+2, +1, -1]$). В списке из двух или менее элементов не-сегментов не бывает. Хотя в списке длины n имеется только $\Theta(n^2)$ сегментов, всего подпоследовательностей $\Theta(2^n)$, поэтому не-сегментов гораздо больше, чем сегментов. Можно ли вычислить максимальную сумму не-сегмента за линейное время? Да, существует простой линейный по времени алгоритм, и цель этой жемчужины в его построении.

Спецификация

Вот спецификация функции *mnss* (максимальная сумма не-сегмента – maximum non-segment sum):

$$\begin{aligned} mnss &:: [Int] \rightarrow Int \\ mnss &= \text{maximum} \cdot \text{map sum} \cdot \text{nonsegs} \end{aligned}$$

Функция *nonsegs* возвращает список всех не-сегментов списка. Чтобы эту функцию определить, можно помечать каждый элемент списка булевым значением: *True* означает, что соответствующий элемент включается в не-сегмент, а *False* — что не включается. Ставим метки всеми возможными способами, оставляем только те наборы меток, которые соответствуют не-сегментам, и затем извлекаем те не-сегменты, чьи элементы помечены *True*. Функция *markings* возвращает все возможные разметки:

$$\begin{aligned} markings &:: [a] \rightarrow [[(a, Bool)]] \\ markings xs &= [\text{zip } xs \text{ bs} \mid bs \leftarrow \text{booleans}(\text{length } xs)] \\ \text{booleans } 0 &= [[]] \\ \text{booleans } (n + 1) &= [b : bs \mid b \leftarrow [\text{True}, \text{False}], bs \leftarrow \text{booleans } n] \end{aligned}$$

Разметки находятся в соответствии один-к-одному с подпоследовательностями. Теперь можно определить:

$$\begin{aligned} \text{nonsegs} &:: [a] \rightarrow [[a]] \\ \text{nonsegs} &= \text{extract} \cdot \text{filter nonseg} \cdot \text{markings} \\ \text{extract} &:: [[(a, Bool)]] \rightarrow [[a]] \\ \text{extract} &= \text{map} (\text{map fst} \cdot \text{filter snd}) \end{aligned}$$

Функция *nonseg* :: $[(a, Bool)] \rightarrow Bool$ возвращает *True* для списка *xms* в том и только в том случае, когда *map snd xms* представляет собой разметку не-сегмента. Булев список *ms* является разметкой не-сегмента тогда и только тогда, когда он является элементом множества, представленного регулярным выражением

$$F^* T^+ F^+ T (T + F)^*$$

в котором *T* обозначает *True*, а *F* — *False*. Регулярное выражение находит самый левый промежуток $T^+ F^+ T$, который и делает подпоследовательность не-сегментом.

Конечный автомат, распознающий элементы соответствующего регулярного множества, должен иметь четыре состояния:

data State = $E \mid S \mid M \mid N$

Состояние E (Empty — пусто) это начальное состояние. Когда автомат находится в состоянии E , распознаются только разметки из множества F^* . В состоянии S (Suffix — суффикс) автомат обработал одно или несколько значений T , что означает принадлежность разметок множеству F^*T^+ , разметок с непустым суффиксом из T . Состояние M (Middle — средний) указывает на разметки из множества $F^*T^+F^+$, средний сегмент, и состояние N (Non-segment) обозначает разметку не-сегмента. Теперь можно определить

$nonseg = (== N) \cdot foldl step E \cdot map snd$

где терм в середине $foldl step E$ выполняет шаги конечного автомата:

$step E False = E$	$step M False = M$
$step E True = S$	$step M True = N$
$step S False = M$	$step N False = N$
$step S True = S$	$step N True = N$

Конечный автомат обрабатывает вход слева направо, что объясняет применение $foldl$. Точно так же можно было бы обрабатывать списки справа налево, отыскивая самый правый промежуток, но зачем без необходимости нарушать принятые соглашения? Заметьте также, что в свойстве $nonseg$ нет ничего особенного: любое свойство разметки, которое можно распознать конечным автоматом, привело бы к в точности тому же способу решения.

Вывод

Вот снова определение $msss$:

$$\begin{aligned} msss &= maximum \cdot map sum \cdot extract \cdot filter nonseg \cdot markings \\ extract &= map (map fst \cdot filter snd) \\ nonseg &= (== N) \cdot foldl step E \cdot map snd \end{aligned}$$

План действий такой: выразим $extract \cdot filter nonseg \cdot markings$ как экземпляр свёртки $foldl$, а затем применим закон слияния для $foldl$, что и приведёт к более удачному алгоритму. С этой целью определим $pick$:

$pick :: State \rightarrow [a] \rightarrow [[a]]$

$pick q = extract \cdot filter ((== q) \cdot foldl step E \cdot map snd) \cdot markings$

В частности, $\text{nonsegs} = \text{pick } N$. Утверждаем, что имеют место следующие семь равенств:

$$\begin{aligned}
 \text{pick } E \text{ xs} &= [[]] \\
 \text{pick } S [] &= [] \\
 \text{pick } S (\text{xs} ++ [x]) &= \text{map} (+[x]) (\text{pick } S \text{ xs} ++ \text{pick } E \text{ xs}) \\
 \text{pick } M [] &= [] \\
 \text{pick } M (\text{xs} ++ [x]) &= \text{pick } M \text{ xs} ++ \text{pick } S \text{ xs} \\
 \text{pick } N [] &= [] \\
 \text{pick } N (\text{xs} ++ [x]) &= \text{pick } N \text{ xs} ++ \\
 &\quad \text{map} (+[x]) (\text{pick } N \text{ xs} ++ \text{pick } M \text{ xs})
 \end{aligned}$$

Есть отличный способ вывода этих равенств, исходящий из определения $\text{pick } q$, но шаги вывода довольно утомительны, и мы этим заниматься не будем. Вместо этого, каждое равенство можно подтвердить, обратившись к step . Например, равенство для $\text{pick } E$ подтверждается, поскольку step возвращает E только для пустых подпоследовательностей. Похожим образом можно показать $\text{pick } S$, потому что step возвращает S , только если x помечен *True*, а предшествующий ему элемент либо $\text{pick } E$, либо $\text{pick } S$. Остальные утверждения можно обосновать таким же методом. Повторяем: в не-сегментах нет ничего особенного, любой конечный автомат с k состояниями, распознающий правильные разметки, можно систематически преобразовать в k функций, работающих непосредственно на заданном входе.

Следующий шаг заключается в приведении определения pick к экземпляру свёртки foldl . Рассмотрим функцию pickall со следующей спецификацией:

$$\text{pickall } \text{xs} = (\text{pick } E \text{ xs}, \text{pick } S \text{ xs}, \text{pick } M \text{ xs}, \text{pick } N \text{ xs})$$

Определение pickall в виде экземпляра foldl вытекает из данных выше определений:

$$\begin{aligned}
 \text{pickall} &= \text{foldl } \text{step} ([[[],[],[],[]]]) \\
 \text{step } (\text{ess}, \text{nss}, \text{mss}, \text{sss}) \text{ x} &= (\text{ess}, \\
 &\quad \text{map} (+[x]) (\text{sss} ++ \text{ess}), \\
 &\quad \text{mss} ++ \text{sss}, \\
 &\quad \text{nss} ++ \text{map} (+[x]) (\text{nss} ++ \text{mss}))
 \end{aligned}$$

Теперь задача принимает форму

$$\text{mnss} = \text{maximum} \cdot \text{map sum} \cdot \text{fourth} \cdot \text{pickall}$$

где fourth возвращает последний элемент четвёрки. Введя функцию

$$\text{tuple } f(w, x, y, z) = (f w, f x, f y, f z)$$

можно переместить *fourth* в начало выражения справа. Тогда получаем:

$$\text{maximum} \cdot \text{map sum} \cdot \text{fourth} = \text{fourth} \cdot \text{tuple}(\text{maximum} \cdot \text{map sum})$$

Поэтому $msss = \text{fourth} \cdot \text{tuple}(\text{maximum} \cdot \text{map sum}) \cdot \text{pickall}$.

Как и планировалось, теперь можно применить закон слияния для *foldl*. Этот закон утверждает, что $f(\text{foldl } g a xs) = \text{foldl } h b xs$ для всех конечных списков *xs* при условии, что $f a = b$, а $f(g x y) = h(f x) y$ для всех *x* и *y*. Соответствующие функции в нашем случае имеют вид:

$$f = \text{tuple}(\text{maximum} \cdot \text{map sum})$$

$$g = \text{step}$$

$$a = ([[], [], [], []])$$

Остается найти *h* и *b*, удовлетворяющие условиям слияния. Во-первых,

$$\text{tuple}(\text{maximum} \cdot \text{map sum})([], [], [], []) = (0, -\infty, -\infty, -\infty)$$

потому что максимум пустого числового списка равен $-\infty$. Это даёт нам определение *b*. Для *h* необходимо удовлетворить уравнению:

$$\begin{aligned} & \text{tuple}(\text{maximum} \cdot \text{map sum})(\text{step}(ess, sss, mss, nss) x) \\ &= h(\text{tuple}(\text{maximum} \cdot \text{map sum})(ess, sss, mss, nss)) x \end{aligned}$$

Для вывода *h* будем по очереди смотреть на каждый компонент. В целях сокращения записи будем писать *max* вместо *maximum*. Относительно четвёртого компонента рассуждаем так:

$$\begin{aligned} & \text{max}(\text{map sum}(nss + map(+(+x])(nss + mss))) \\ &= \{ \text{определение map} \} \\ & \text{max}(\text{map sum} nss + \text{map}(\text{sum} \cdot (+(+x]))(nss + mss)) \\ &= \{ \text{так как } \text{sum} \cdot (+(+x)) = (+x) \cdot \text{sum} \} \\ & \text{max}(\text{map sum} nss + \text{map}((+x) \cdot \text{sum})(nss + mss)) \\ &= \{ \text{так как } \text{max}(xs + ys) = (\text{max } xs) \text{ max } (\text{max } ys) \} \\ & \text{max}(\text{map sum} nss) \text{ max } \text{max}(\text{map}((+x) \cdot \text{sum})(nss + mss)) \\ &= \{ \text{так как } \text{max} \cdot \text{map}(+x) = (+x) \cdot \text{max} \} \\ & \text{max}(\text{map sum} nss) \text{ max } (\text{max}(\text{map sum}(nss + mss)) + x) \\ &= \{ \text{вводим } n = \text{max}(\text{map sum} nss) \text{ и} \\ & \quad m = \text{max}(\text{map sum} mss) \} \end{aligned}$$

$$n \max ((n \max m) + x)$$

Рассматривая похожим образом три оставшихся компонента, получаем:

$$\begin{aligned} h(e, s, m, n) x \\ = (e, (s \max e) + x, m \max s, n \max ((n \max m) + x)) \end{aligned}$$

и $mnss = fourth \cdot foldl h (0, -\infty, -\infty, -\infty)$.

По существу, это всё. Нам, правда, ещё нужно разобраться с фиктивными значениями $-\infty$. Вероятно, самый лучший способ полностью от них избавиться заключается в отдельном рассмотрении первых трёх элементов списка:

$$\begin{aligned} mnss xs &= fourth (foldl h (start (take 3 xs)) (drop 3 xs)) \\ start [x, y, z] &= (0, \max [x + y + z, y + z, z], \max [x, x + y, y], x + z) \end{aligned}$$

Не так красиво, но эффективнее.

Заключительные замечания

Задача о максимальной сумме сегмента восходит к примерно 1975 году, её история описана в одной из жемчужин программирования в (Bentley, 1987). Вывод с использованием инвариантов описан в (Gries, 1990); алгебраический подход дан в (Bird, 1989). Задача отказывается уходить, её вариации по-прежнему активно изучаются проектировщиками алгоритмов, причина в наличии потенциальных применений в интеллектуальном анализе данных (data mining) и биоинформатике. Недавние результаты имеются в (Mu, 2008).

Самое интересное в задаче о не-сегментах в том, что она учит решению любой задачи о максимальной разметке, в которой критерий размечивания можно сформулировать в виде регулярного выражения. Например, сразу очевидно, что существует алгоритм вычисления максимальной суммы сегмента длины не менее k со сложностью $O(nk)$, так как выражение $F^* T^n F^*$ ($n \geq k$) можно распознать автоматом с k состояниями. Даже случай нерегулярных условий типа $F^* T^n F^* T^n F^*$ ($n \geq 0$), чей распознаватель требует наличия неограниченного количества состояний, поддаётся решению тем же методом. Более того, необязательно даже говорить о списках, похожим образом решается задача о максимальной разметке многих других типов данных.

Литература

- Bentley, J. R. (1987). *Programming Pearls*. Reading, MA: Addison-Wesley. [*Имеется русский перевод*: Бентли Дж. Жемчужины программирования. СПб.: Питер, 2002.]
- Bird, R. S. (1989). Algebraic identities for program calculation. *Computer Journal* **32** (2), 122–6.
- Gries, D. (1990). The maximum segment sum problem. In *Formal Development of Programs and Proofs*, ed. E. W. Dijkstra *et al.* University of Texas at Austin Year of Programming Series. Menlo Park. Addison-Wesley, pp. 43–5.
- Mu, S.-C. (2008). The maximum segment sum is back. *Partial Evaluation and Program Manipulation (PEPM '08)*, pp. 31–9.

12

Ранжируем суффиксы

Введение

Идея отранжировать элементы списка возникает довольно часто. Элементу x присваивается ранг r , если в списке имеется в точности r элементов, меньших x . Например, $\text{rank} [51, 38, 29, 51, 63, 38] = [3, 1, 0, 3, 5, 1]$. Эта схема ранжирует элементы с нуля от наименьшего к наибольшему, но можно и по другому: с единицы от наибольшего к наименьшему, как упорядочиваются абитуриенты согласно их оценкам на экзамене. Все ранги различны тогда и только тогда, когда список не содержит повторов, и в этом случае $\text{rank } xs$ является перестановкой списка $[0 \dots \text{length } xs - 1]$.

В этой жемчужине вместо ранжирования самого списка мы рассмотрим задачу ранжирования его суффиксов. Для ранжирования списка длины n в предположении, что проверка $x < y$ выполняется за константное время, требуется $\Theta(n \log n)$ операций. Поскольку в худшем случае лексикографическое сравнение двух суффиксов длины n требует $\Theta(n)$ таких проверок, можно ожидать, что ранжирование суффиксов потребует $\Theta(n^2 \log n)$ сравнений элементов. В этой жемчужине мы покажем, что на самом деле достаточно только $\Theta(n \log n)$ операций. Получается, что ранжирование суффиксов списка требует асимптотически не большие времени, чем ранжирование самого списка. Удивительно, но это так.

Спецификация

В языке Haskell суффиксы списка называются его хвостами (*tails*), поэтому далее и мы будем называть их хвостами. Функция *tails* возвращает непустые хвосты списка в порядке убывания их длин:

$$\begin{aligned} \text{tails} &:: [a] \rightarrow [[a]] \\ \text{tails} [] &= [] \\ \text{tails} xs &= xs : \text{tails} (\text{tail} xs) \end{aligned}$$

Определение функции *tails* отличается от стандартной функции языка Haskell с тем же именем, которая возвращает все хвосты, в том числе и пустой. Функцию *rank* можно специфицировать так:

$$\begin{aligned} \text{rank} &:: \text{Ord } a \Rightarrow [a] \rightarrow [\text{Int}] \\ \text{rank} xs &= \text{map} (\lambda x \rightarrow \text{length} (\text{filter} (<x) xs)) xs \end{aligned}$$

Это определение требует $\Theta(n^2)$ операций на списке длины n , но *rank* можно улучшить до $\Theta(n \log n)$ операций, к этому мы вернёмся позднее.

Искомая функция, скажем, *ranktails*, теперь определяется так:

$$\begin{aligned} \text{ranktails} &:: \text{Ord } a \Rightarrow [a] \rightarrow [\text{Int}] \\ \text{ranktails} &= \text{rank} \cdot \text{tails} \end{aligned}$$

Наша задача заключается в построении такой реализации *ranktails*, которая требовала бы $\Theta(n \log n)$ операций.

Свойства ранга

Нам нужны различные свойства функции *rank*, самое важное из которых в том, что *rank* сохраняет порядок: если мы знаем только $\text{rank} xs$, то мы знаем всё об относительном расположении элементов списка *xs*, хотя и ничего не знаем относительно природы самих элементов. Предположим, мы определили $xs \approx ys$ так, что это означает $\text{rank} xs = \text{rank} ys$. Тогда $xs \approx ys$, если элементы списка *xs* имеют тот же относительный порядок, что и элементы *ys*. Вот только два примера:

$$xs \approx \text{zip} xs xs \quad \text{и} \quad \text{zip} (\text{zip} xs ys) zs \approx \text{zip} xs (\text{zip} ys zs)$$

Нам также необходимо следующее свойство *rank*. Пусть $\text{select} :: [a] \rightarrow [a]$ является произвольной функцией, такой что:

- a) каждый элемент из $\text{select} xs$ есть также и в *xs*;

6) $\text{select} \cdot \text{map } f = \text{map } f \cdot \text{select}$ для любой f .

Тогда

$$\text{rank} \cdot \text{select} \cdot \text{rank} = \text{rank} \cdot \text{select} \quad (12.1)$$

В частности, взяв $\text{select} = \text{id}$ получаем $\text{rank} \cdot \text{rank} = \text{rank}$, а выбрав $\text{select} = \text{tail}$, имеем $\text{rank} \cdot \text{tail} \cdot \text{rank} = \text{rank} \cdot \text{tail}$. Доказательство (12.1) остается как полезное упражнение для заинтересованного читателя.

Наконец, свяжем с ранжированием полезную идею уточнения одного ранжирования другим. Предположим, мы определили операцию \ll (произносится как «уточняется»):

$$xs \ll ys = \text{rank}(\text{zip } xs \; ys) \quad (12.2)$$

Например, $[3, 1, 3, 0, 1] \ll [2, 0, 3, 4, 0] = [2, 1, 3, 0, 1]$. Таким образом, одинаковые ранги в xs можно уточнить до различных рангов в $xs \ll ys$. Операция \ll является ассоциативной. Вот доказательство:

$$\begin{aligned} & (xs \ll ys) \ll zs \\ &= \{ (12.2) \} \\ &= \text{rank}(\text{zip}(\text{rank}(\text{zip} xs \; ys)) \; zs) \\ &= \{ \text{так как } \text{zip } us \; vs \approx \text{zip}(\text{rank } us) \; vs \} \\ &= \text{rank}(\text{zip}(\text{zip} xs \; ys) \; zs) \\ &= \{ \text{так как } \text{zip}(\text{zip} xs \; ys) \; zs \approx \text{zip } xs \; (\text{zip } ys \; zs) \} \\ &= \text{rank}(\text{zip } xs \; (\text{zip } ys \; zs)) \\ &= \{ \text{как ранее} \} \\ &= xs \ll (ys \ll zs) \end{aligned}$$

Заметьте также, что если ранжирование xs состоит из различных элементов, а значит, является перестановкой $[0..n-1]$, где $n = \text{length } xs$, то $xs \ll ys = xs$ для любого списка ys одинаковой с xs длины. Словом, как только ранжирование оказалось перестановкой, уточнить его далее нельзя.

Улучшенный алгоритм

Очевидный подход к улучшению производительности ranktails , с учётом его желаемой сложности, заключается в применении метода «разделяй и властвуй» и декомпозиции следующего вида:

$$\text{tails}(xs \mathbin{+} ys) = \text{map}(\mathbin{+} ys)(\text{tails } xs) \mathbin{+} \text{tails } ys$$

Вряд ли, правда, это куда-нибудь приведёт. Вместо этого мы применим иной подход и для начала обобщим функцию *ranktails* до функции *rats*, заменив лексикографическое сравнение ($<$) на сравнение ($<_k$), определённое условием $xs <_k ys = take\ k\ xs < take\ k\ ys$. Другими словами, мы будем ранжировать хвосты списка, глядя только на первые k элементов каждого хвоста. Определим *rats* так:

$$rats\ k = rank \cdot map\ (take\ k) \cdot tails \quad (12.3)$$

Получаем $ranktails\ xs = rats\ (length\ xs)\ xs$, поэтому *rats* действительно является обобщением *ranktails*. Мы назвали функцию *rats* так, потому что это имя соединяет *rank* и *tails*, к тому же его легко произносить, и оно достаточно короткое, что позволит избежать длинных выражений в наших вычислениях.

Ключ к выводу более быстрого алгоритма для *ranktails* в следующем свойстве *rats*:

$$rats\ (2 * k)\ xs = rats\ k\ xs \ll shiftBy\ k\ (rats\ k\ xs) \quad (12.4)$$

Мы это докажем, а также дадим определение *shiftBy*, но позднее. Поскольку $xs \approx map\ (take\ 1)\ (tails\ xs)$, имеем $rats\ 1 = rank$. Идея в использовании (12.4) для последовательного ранжирования *map\ (take\ 2)* (*tails\ xs*), *map\ (take\ 4)* (*tails\ xs*) и т.д., пока мы не достигнем перестановки. Таким образом, для *ranktails* предлагается следующий алгоритм:

$$\begin{aligned} ranktails &= applyUntil\ isperm\ rerankings \cdot rank \\ rerankings &= map\ rerank\ (iterate\ (*2)\ 1) \\ rerank\ k\ rs &= rs \ll shiftBy\ k\ rs \end{aligned}$$

Функция *applyUntil* является вариацией стандартной функции *until* и определяется так:

$$\begin{aligned} applyUntil &\quad :: (a \rightarrow Bool) \rightarrow [a \rightarrow a] \rightarrow a \rightarrow a \\ applyUntil\ p\ (f : fs)\ x &= \text{if } p\ x \text{ then } x \text{ else } applyUntil\ p\ fs\ (f\ x) \end{aligned}$$

Функция *isperm* проверяет, является ли ранжирование перестановкой, её можно определить с помощью массивов языка Haskell¹:

$$\begin{aligned} isperm &\quad :: [Int] \rightarrow Bool \\ isperm\ is &= and\ (elems \end{aligned}$$

¹Почти идентичная функция использовалась в первой жемчужине «Наименьшее отсутствующее число».

$$(accumArray (\vee) False (0, n - 1) (zip is (repeat True)))) \\ \text{where } n = length is$$

Такое определение *isperm* требует линейного времени.

Словом, *ranktails* сначала ранжирует элементы исходного списка, а затем последовательно применяет *rerank 1*, *rerank 2*, *rerank 4* и т.д., пока результатом не окажется перестановка. Заметьте, что только самое первое ранжирование проверяет входной список, все остальные имеют дело исключительно с рангами, т.е. со списками целых чисел. В худшем случае *ranktails xs* требует $\log n$ повторных ранжирований, где $n = length xs$. В предположении, что *rank* выполняется за $\Theta(n \log n)$ операций, а *shiftBy* за $\Theta(n)$, новая версия *ranktails* требует $\Theta(n \log^2 n)$ операций. Лучше, чем $\Theta(n^2 \log n)$, но пока не требующаяся нам сложность $\Theta(n \log n)$.

Доказательство

Докажем теперь (12.4), выяснив одновременно определение *shiftBy*. Нам потребуется несколько дополнительных свойств, касающихся списков, хвостов и ранжирований, первое из которых таково:

$$\text{all} (\text{not} \cdot \text{null}) xss \Rightarrow xss = \text{zipWith} (\:) (\text{map head} xss) (\text{map tail} xss)$$

В целях уменьшения количества скобок в выражениях перепишем следствие в иной форме:

$$id = \text{zipWith} (\:) \cdot \text{fork} (\text{map head}, \text{map tail}) \quad (12.5)$$

где $\text{fork}(f, g)x = (f x, g x)$, а *zipWith* (*:*) предполагается некаррированной. Доказательство этого и других свойств опускается. Начнём с вычисления:

$$\begin{aligned} & rats(k + 1) \\ &= \{ (12.3) \} \\ & rank \cdot \text{map} (\text{take}(k + 1)) \cdot tails \\ &= \{ (12.5), \text{ так как all} (\text{not} \cdot \text{null}) \cdot \text{map} (\text{take}(k + 1)) \cdot tails \} \\ & rank \cdot \text{zipWith} (\:) \cdot \text{fork} (\text{map head}, \text{map tail}) \cdot \\ & \quad \text{map} (\text{take}(k + 1)) \cdot tails \end{aligned}$$

Теперь $\text{fork}(f, g) \cdot h = \text{fork}(f \cdot h, g \cdot h)$ и

$$\begin{aligned} \text{map head} \cdot \text{map} (\text{take}(k + 1)) \cdot tails &= id \\ \text{map tail} \cdot \text{map} (\text{take}(k + 1)) \cdot tails &= snoc [] \cdot tail \cdot \text{map} (\text{take} k) \cdot tails \end{aligned}$$

где $snoc\ x\ xs = xs \# [x]$. Следовательно, можно продолжить:

$$\begin{aligned} & rank \cdot zipWith(:) \cdot fork (map head, map tail) \cdot \\ & map (take (k + 1)) \cdot tails \\ = & \{ \text{выше} \} \\ & rank \cdot zipWith(:) \cdot fork (id, snoc [] \cdot tail \cdot map (take k) \cdot tails) \end{aligned}$$

Следующим шагом заметим, что $zipWith(:) xs\ xss \approx zip\ xs\ xss$ или, короче, $zipWith(:) \approx zip$, где функция zip также предполагается некаррированной. Таким образом, получаем:

$$rats(k + 1) = rank \cdot zip \cdot fork (id, snoc [] \cdot tail \cdot map (take k) \cdot tails)$$

Теперь согласно (12.2) $rank \cdot zip = (\ll)$. А поскольку имеет место равенство $xs \ll ys = xs \ll rank\ ys$, то приходим к

$$rats(k + 1) = (\ll) \cdot fork (id, rank \cdot snoc [] \cdot tail \cdot map (take k) \cdot tails)$$

Сейчас $rank \cdot snoc [] = lift \cdot rank$, где $lift = snoc 0 \cdot map (+1)$, поэтому

$$rats(k + 1) = (\ll) \cdot fork (id, lift \cdot rank \cdot tail \cdot map (take k) \cdot tails)$$

Сделаем утверждение: $lift \cdot rank \cdot tail \approx lift \cdot tail \cdot rank$. Вот доказательство:

$$\begin{aligned} & lift (rank (tail xs)) \approx lift (tail (rank xs)) \\ \Leftarrow & \{ \text{так как из } xs \approx ys \text{ следует, что } lift\ xs \approx lift\ ys \} \\ & rank (tail xs) \approx tail (rank xs) \\ \equiv & \{ \text{определение } \approx \} \\ & rank (rank (tail xs)) = rank (tail (rank xs)) \\ \Leftarrow & \{ \text{используя дважды (12.1), первый раз для } subseq = id, \\ & \text{а второй для } subseq = tail \} \\ & true \end{aligned}$$

Следовательно, $rats(k + 1) = (\ll) \cdot fork (id, lift \cdot tail \cdot rats\ k)$. Эквивалентно:

$$rats(k + 1)\ xs = rank\ xs \ll shift (rats\ k\ xs) \tag{12.6}$$

где $shift = lift \cdot tail$, а $lift\ is = map (+1)\ is \# [0]$.

На следующем шаге воспользуемся тем фактом, что

$$shift (is \ll js) = shift\ is \ll shift\ js$$

и ассоциативностью \ll . Тогда из (12.6) получим, что

$$\text{rats } k \text{ xs} = \text{rs} \ll \text{shift } \text{rs} \ll \text{shift}^2 \text{rs} \ll \dots \ll \text{shift}^{k-1} \text{rs}$$

где $\text{rs} = \text{rank } \text{xs}$, а shift^k является k -кратной композицией функции shift с самой собой. В этом разложении rats в ряд нетрудно сгруппировать термы так, чтобы получилось

$$\text{rats } (2 * k) \text{ xs} = \text{rats } k \text{ xs} \ll \text{shift}^k(\text{rats } k \text{ xs})$$

Наконец, для достижения (12.4) можно задать $\text{shiftBy } k = \text{shift}^k$. Фактически получаем:

$$\text{shiftBy } k \text{ rs} = \text{map } (+k) (\text{drop } k \text{ rs}) \text{++} [k - 1, k - 2 .. 0]$$

Вычисление $\text{shiftBy } k \text{ rs}$ требует $\Theta(n)$ операций, где $n = \text{length } \text{rs}$.

Улучшенное ранжирование

Теперь перейдём к улучшенному методу для вычисления rank . Как известно любому экзаменатору, чтобы отранжировать список пар студент–оценка, заданный в порядке следования студентов, нужно сначала разбить студентов на группы, упорядочив их по оценкам и поместив студентов с одинаковыми оценками в одну группу. Тогда каждый студент из первой группы получает ранг 0, каждый студент из второй группы — ранг g_0 , где g_0 это размер первой группы, каждый студент из третьей группы — ранг $g_0 + g_1$, где g_1 это размер второй группы, и т.д. Затем результат сортируется в соответствии с исходным упорядочиванием по студентам. Этот метод можно formalизовать следующим образом:

$$\text{rank} = \text{resort} \cdot \text{concat} \cdot \text{label} \cdot \text{psort} \cdot \text{zip} [0 ..]$$

Функция psort (сокращение от partition sort — сортировка разбиениями) разбивает список пар студент–оценка, упорядочивая их по оценкам и группируя студентов с равными оценками по сериям. Оценки, выполнившие свою задачу, отбрасываются. Один из способов реализации psort представлен на рис. 12.1. Используемый метод является вариацией троичной быстрой сортировки, в которой в качестве опорного элемента на шаге разбиения применяется голова списка. Как следствие, psort выполняется в худшем случае за $\Theta(n^2)$ операций. Выбор в качестве опорного медианного элемента снижает сложность до $\Theta(n \log n)$ операций.

Функция label определена так:

<i>psort</i>	$:: Ord b \Rightarrow [(a, b)] \rightarrow [[a]]$
<i>psort xys</i>	$= pass\ xys\ []$
<i>pass [] xss</i>	$= xss$
<i>pass (e@(x, y) : xys) xss</i>	$= step\ xys\ []\ [x]\ []\ xss$
where	
<i>step [] as bs cs xss</i>	$= pass\ as\ (bs : pass\ cs\ xss)$
<i>step (e@(x, y') : xys) as bs cs xss</i>	$ y' < y = step\ xs\ (e : as)\ bs\ cs\ xss$ $ y' == y = step\ xs\ as\ (x : bs)\ cs\ xss$ $ y' > y = step\ xs\ as\ bs\ (e : cs)\ xss$

Рис. 12.1: сортировка разбиениями

```

label      :: [[a]] → [[(a, Int)]]
label xss = zipWith tag xss (scanl (+) 0 (map length xss))
tag xs k  = [(x, k) | x ← xs]
  
```

Наконец, *resort* можно реализовать, применяя массив языка Haskell:

```

resort    :: [(Int, Int)] → [Int]
resort ijs = elems (array (0, length ijs - 1) ijs)
  
```

Вызов *array* ($0, n - 1$) *ijs* строит проиндексированный от 0 до $n - 1$ массив, чьи пары индекс–значение заданы ассоциативным списком *ijs*. Для того чтобы результат был корректным, первые компоненты *ijs* должны быть перестановкой $[0..n-1]$. И *array*, и *elems* выполняются за линейное время, поэтому *resort* также выполняется за линейное время.

Окончательный алгоритм

Пересмотренная реализация *rank* ведёт к новой реализации *ranktails*, в которой ранжирование подчиняется разбиению. Предположим, мы добавили функцию:

```

partition :: Ord a ⇒ [a] → [[Int]]
partition = psort · zip [0..]
  
```

Тогда $rank = resort \cdot concat \cdot label \cdot partition$. Более того, операцию \ll можно выразить в терминах *partition*, потому что она выражается через *rank*. Это означает, что можно перегруппировать термы и выразить *ranktails* через

partition вместо *rank*. У таких манипуляций есть одно преимущество: *rank* возвращает перестановку в том и только в том случае, когда *partition* возвращает список одноэлементных списков, поэтому *isperm* можно заменить на *all single*, где *single* определяет, является ли список одноэлементным. Применив указанные соображения, получаем:

$$\begin{aligned} \text{ranktails} &= \text{resort} \cdot \text{concat} \cdot \text{label} \cdot \\ &\quad \text{applyUntil}(\text{all single}) \text{ repartitions} \cdot \text{partition} \\ \text{repartitions} &= \text{map} \text{ repartition} (\text{iterate} (*2) 1) \\ \text{repartition } k \text{ iss} &= \text{partition} (\text{zip} \text{ rs} (\text{shiftBy} \text{ k} \text{ rs})) \\ &\quad \text{where } \text{rs} = \text{resort} (\text{concat} (\text{label} \text{ iss})) \end{aligned}$$

Эта версия имеет ту же временную сложность, что и предыдущая, но открывает путь к дальнейшей оптимизации. Фактически мы сейчас находимся всего в двух шагах от нашей цели — алгоритма для *ranktails* со сложностью $\Theta(n \log n)$.

Первый из них состоит в использовании тождества:

$$\begin{aligned} \text{partition} (\text{zip} \text{ xs} \text{ ys}) & \\ = \text{concatMap} (\text{psort} \cdot \text{map} (\text{install} \text{ ys})) (\text{partition} \text{ xs}) & \end{aligned} \tag{12.7}$$

где

$$\text{install} \text{ ys} \text{ i} = (i, \text{ys} \text{ !!} i) \tag{12.8}$$

Словом, список пар можно разбить, если сначала разбить его относительно первых компонентов, а затем уточнить результат с использованием вторых компонентов. Так как каждая серия в разбиении является списком позиций, то для неё всегда можно задать корректные вторые компоненты пар. После их задания каждая серия сортируется разбиениями, а результаты соединяются.

Строго говоря, (12.7) выполняется, только если *psort* является устойчивой сортировкой, тогда как реализация на рис. 12.1 неустойчива. Если сортировка *psort* неустойчива, то элементы из каждой серии будут идти слева и справа в разных порядках. Однако, если мы будем интерпретировать равенство двух разбиений как равенство с точностью до некоторой перестановки элементов каждой серии, то (12.7) будет иметь место. Так как вычисление *ranktails* не зависит от точного порядка элементов в серии, то нам этого достаточно.

Теперь можно переписать функцию *repartitions*:

$$\text{repartition } k \text{ iss}$$

$$\begin{aligned}
 &= \{ \text{определение, полагаем } rs = resort(concat(label iss)) \} \\
 &\quad partition(zip rs (shiftBy k rs)) \\
 &= \{ (12.7) \} \\
 &\quad concatMap(psorth · map(install(shiftBy k rs))) (partition rs) \\
 &= \{ \text{так как из } iss = partition xs \text{ следует } rs = rank xs \text{ и} \\
 &\quad partition · rank = partition \} \\
 &\quad concatMap(psorth · map(install(shiftBy k rs))) iss
 \end{aligned}$$

Следовательно,

$$\begin{aligned}
 repartition k iss &= concatMap(psorth · map(install rs)) iss \\
 \text{where } rs &= shiftBy k (resort(concat(label iss)))
 \end{aligned}$$

Мы уже почти на месте, но (12.8) даёт слишком неэффективный способ для вычисления *install*, потому что в нём используется индексирование списков (!!), время выполнения которого не является константным. Лучше использовать индексирование массивов (!), которое выполняется за константное время. Последний шаг состоит в переписывании *install*:

$$\begin{aligned}
 &(shiftBy k (resort(concat(label iss)))) !! i \\
 &= \{ \text{определение } shiftBy \} \\
 &\quad (map(+k) (drop k (resort(concat(label iss)))) + \\
 &\quad [k - 1, k - 2 .. 0]) !! i \\
 &= \{ \text{арифметика, } n = length(concat(label iss)) \text{ и } j = i + k \} \\
 &\quad \text{if } j < n \text{ then } k + (resort(concat(label iss))) !! j \text{ else } n - i - 1 \\
 &= \{ \text{определение } resort \text{ и } elems a !! i = a ! i \} \\
 &\quad \text{if } j < n \text{ then } k + array(0, n - 1) (concat(label iss))) ! j \\
 &\quad \text{else } n - i - 1
 \end{aligned}$$

Окончательный вид *ranktails* приведён на рис. 12.2. Длина входного списка вычисляется только однажды, а затем она передаётся в различные вспомогательные функции, которые в ней нуждаются.

Анализ

Остается посчитать время выполнения *ranktails*. Сделаем это, оценив общее число сравнений $T(n, k)$, необходимых для выполнения k переразбиений на списке длины n . Ключ к анализу в понимании того факта, что

```

ranktails           :: Ord a => [a] -> [Int]
ranktails xs       := (resort n · concat · label ·
                        applyUntil (all single) (repartitions n) ·
                        psort · zip [0..]) xs
                     where n = length xs
resort n            = elems · array (0, n - 1)
label iss            = zipWith tag iss (scaml (+) 0 (map length iss))
tag is j             = [(i, j) | i <- is]
repartitions n      = map (repartition n) (iterate (*2) 1)
repartition n k iss = concatMap (psort · map install) iss
                     where install i = (i, if j < n then k + a ! j else n - i - 1)
a                   = array (0, n - 1) (concat (label iss))

```

Рис. 12.2: окончательный вид функции *ranktails*

вычисление *ranktails* это в сущности то же самое, что и сортировка разбиениями списка n k -мерных векторов, при которой каждая размерность упорядочивается по очереди. Первые компоненты этих векторов состоят из элементов входа, а каждый последующий компонент это целое число, определяемое динамически на основе результатов сортировки относительно предыдущих компонентов. В отличие от данной на рис. 12.1 реализации *psort*, мы будем предполагать, что для каждого разбиения в качестве опорного элемента берётся медиана всех возможных опорных элементов. Например, при сортировке по первому компоненту опорный элемент p выбирается равным некоторому положительному числу элементов списка (обозначим это число через x), первые компоненты которых равны p , причём количества тех элементов, первые компоненты которых меньше p , и тех, вторые компоненты которых больше p , равны и составляют $(n - x)/2$. Серия из x элементов сортируется по первому компоненту, поэтому остаётся упорядочить их по остальным компонентам. Однако оставшиеся два списка нужно по-прежнему сортировать по всем компонентам. Так как разбиение n элементов требует $n - 1$ сравнений, мы можем определить $T(n, k)$ следующим рекуррентным соотношением:

$$T(0, k) = 0$$

$$T(n, 0) = 0$$

$$T(n, k) = (\max x : 1 \leqslant x \leqslant n : n - 1 + T(x, k - 1) + 2T((n - x)/2, k))$$

Покажем теперь, что $T(n, k) \leq n(\log n + k)$. Это можно доказать индукцией, основываясь на следующем неравенстве:

$$n - 1 + x(\log x + k - 1) + (n - x)[\log(n - x) + k - 1] \leq n(\log n + k)$$

для $1 \leq x \leq n$. Его можно доказать с помощью фактов из школьной программы. Наконец, поскольку $k = \log n$, получаем, что число сравнений, необходимых для вычисления *ranktails* от списка длины n не превосходит $2n \log n$, а общее время выполнения составляет $\Theta(n \log n)$ операций.

Экспериментальные данные

Для получения окончательной версии алгоритма потребовалось пройти значительный объём вычислений, но действительно ли он лучше наивного на практике? Мы протестировали три версии алгоритма:

- (А) спецификация *ranktails*, *rank* для которой определено на основе сортировки разбиениями;
- (Б) улучшенный алгоритм с повторными ранжированием;
- (В) окончательная версия алгоритма на рис. 12.2.

Все алгоритмы запускались сначала с реализацией *psort*, приведённой на рис. 12.1 (АлгA1, АлгB1 и АлгB1), а затем с алгоритмом, основанном на сортировке слияниями, гарантированное время выполнения которого равно $\Theta(n \log n)$ (АлгA2, АлгB2 и АлгB2). Программы компилировались GHC. В качестве входа использовались данные пяти разных видов: (dna) файл с ДНК; (ps) файл POSTSCRIPT; (txt) файл alice29.txt из кентерберийского корпуса (Canterbury corpus), содержащий текст сказки «Алиса в стране чудес»; (ptt5) файл ptt5 с изображением из того же корпуса; и (alla) текст, содержащий 1000 вхождений буквы «а». Время выполнения в секундах приведено в табл. 12.1. Во втором столбце указано общее количество символов в файле, а в третьем — количество различных символов. Ячейка таблицы в столбце АлгA1 и строке ptt5 пуста, потому что вычисления были досрочно прерваны спустя 12 часов. Выделенные полужирным ячейки показывают наилучшие результаты по каждой строке.

Эти данные вскрывают довольно запутанное положение. Во-первых, как и ожидалось, алгоритмы АлгB1 и АлгB2 не имеют преимуществ ни перед наивным, ни перед окончательным алгоритмами. Однако, за исключением двух последних строк, наивный алгоритм ведёт себя в точности так же, как и окончательный. Более того, алгоритмы, основанные на троичной быстрой сортировке, оказались почти в два раза быстрее, чем алгоритмы,

Таблица 12.1. Время выполнения трёх версий алгоритма для разных файлов

Файл	Размер	Символы	АлгА1	АлгБ1	АлгВ1	АлгА2	АлгБ2	АлгВ2
dna	10424	5	0.08	0.10	0.08	0.04	0.08	0.16
ps	367639	87	2.76	10.96	2.32	3.18	22.50	3.54
txt	148480	72	0.48	3.08	0.72	0.86	6.06	1.28
ptt5	513216	159	—	136.38	8.70	611.3	37.78	6.32
alla	1000	1	2.96	0.10	0.02	0.18	0.06	0.02

основанные на сортировке слияниями. Правда, картина резко меняется для файла *ptt5*, состоящего почти сплошь из нулевых символов, и *alla*, содержащего только символы «а». Здесь окончательный алгоритм оказался гораздо лучше. В среднем, алгоритм АлгВ1 кажется наилучшим, хотя он и не лучше остальных в каждом из отдельных случаев.

Заключительные замечания

Наш окончательный алгоритм для *ranktails* довольно близок к алгоритму сортировки суффиксов списка, предложенному в (Larsson and Sadakane, 1999). И в самом деле, вся эта жемчужина была вдохновлена их работой. Операции ранжирования списка и его сортировки тесно связаны, и каждую из них можно быстро вычислить на основе другой. В частности, функцию *sorttails*, возвращающую единственную перестановку, которая сортирует хвосты списка, можно получить из окончательной версии *ranktails* заменой *resort · concat · label* из первой строки на *concat*. Функция *sorttails* нужна как предварительный шаг в алгоритме сжатия данных Барроуза—Уилера (Burrows–Wheeler), который мы будем рассматривать в следующей жемчужине. Задача сортировки суффиксов строки изучалась в литературе очень интенсивно, потому что она имеет приложения в сопоставлении строк и биоинформатике; хорошим источником по данной теме является книга (Gusfield, 1997).

Эта жемчужина переписывалась несколько раз. Изначально мы стартали с идеи вычисления *perm*, перестановки, сортирующей список. Но *perm* является слишком специфичной из-за того способа, которым она обрашается с повторяющимися элементами: список с повторами можно отсортировать более чем одной перестановкой. Если не обобщать *rank* или *partition*, то на одной только *perm* далеко не уедешь. Задача была переформулирована в терминах *rank*, однако закончилось всё не ранжированием, а идеей разбиения списка. Однако в самом конце *rank* снова появляется на сцене, она используется на последнем шаге оптимизации.

Литература

- Larsson, N. J. and Sadakane, K. (1999). Faster suffix sorting. Research Report LU-CS-TR-99-214, Department of Computer Science, Lund University, Sweden.
- Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge, UK: Cambridge University Press. [Имеется русский перевод: Гасфилд Д. Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, БХВ-Петербург. 2003.]

13

Преобразование Барроуза–Уилера

Введение

Преобразованием Барроуза–Уилера (ПБУ) называется метод перестановки элементов списка таким образом, чтобы повторяющиеся элементы оказались рядом друг с другом. Оно преимущественно применяется в качестве предварительной стадии при сжатии данных. Списки с большим количеством повторяющихся соседних элементов допускают компактное кодирование с использованием таких простых схем, как длины серий из одинаковых элементов или кодирование движением к началу (*move-to-front encoding*). Результат можно затем передать на вход более совершенным алгоритмам сжатия, например, методу Хаффмана или арифметическому кодированию¹, получив тем самым ещё более хорошее сжатие исходных данных.

Ясно, что наилучшим способом размещения повторяющихся элементов рядом друг с другом является просто сортировка списка. Однако использование этой идеи для подготовки данных перед сжатием имеет следующий изъян: если в процессе сортировки не формировать одновременно сортирующую перестановку, то восстановить исходный список будет невозможно. Сжатие данных без возможности получения исходных данных бесполезно, а если нужно создавать перестановку, то сжатие становится неэффективным. ПБУ позволяет получить менее амбициозную перестановку: оно перемещает некоторые, хотя и не все повторяющиеся элементы

¹ Арифметическое кодирование рассматривается в двух подряд идущих жемчужинах, 24 и 25.

0	л о г о в о	0	в о л о г о
1	о г о в о л	1	г о в о л о
2	г о в о л о	2	л о г о в о
3	о в о л о г	3	о в о л о г
4	в о л о г о	4	о г о в о л
5	о л о г о в	5	о л о г о в

Рис. 13.1: циклические сдвиги и результат их сортировки

в соседние позиции. Главное достоинство ПБУ состоит в том, что для обращения преобразования достаточно предоставить только одну дополнительную единицу данных, а именно целое число k из промежутка $0 \leq k < n$, где n это длина (непустого) входного списка. В этой жемчужине мы опишем ПБУ, выявив фундаментальную причину, по которой оказывается возможным обращение, и воспользовавшись ею для вывода обратного преобразования из спецификации.

Определяем ПБУ

Будучи применённым к списку xs , ПБУ сортирует его циклические сдвиги, формируя из них матрицу, а затем возвращает последний столбец этой матрицы вместе с позицией в ней списка xs . Возьмём, для примера, слово `лого`. Циклические сдвиги и результат их сортировки показаны на рис. 13.1. Результатом преобразования является текст `оооглав`, последний столбец второй матрицы, и число 2, поскольку исходное слово `лого` расположено в строке 2 второй матрицы.

Специфицировать ПБУ нетрудно:

```
transform    :: Ord a ⇒ [a] → ([a], Int)
transform xs = (map last xss, position xs xss)
  where xss = sort (rots xs)
```

Положение строки xs в матрице xss определяется так:

```
position xs xss = length (takeWhile (≠ xs) xss)
```

а циклические сдвиги непустого списка так:

```
rots      :: [a] → [[a]]
rots xs = take (length xs) (iterate lrot xs)
  where lrot (x : xs) = xs ++ [x]
```

Вспомогательная функция *lrot* выполняет один циклический сдвиг влево. Код функции *transform* неэффективен, но мы эту проблему пока отложим².

Функция *transform* полезна для сжатия, потому что если её применить к тексту, то она соберёт вместе символы с общим контекстом. Например, в русскоязычных текстах очень часто встречаются сочетания букв «то» и «но». Следовательно, многие из циклических сдвигов, начинающихся с «о» будут заканчиваться на «т» или «н». Приведём один хитрый пример: преобразование текста «вот тот торт хороши» даст

$$("тттшхттвроооп\text{ }о", 3)$$

Здесь все буквы «о», кроме одной, а также все пробелы собраны вместе.

Обратное преобразование, *untransform* :: $Ord a \Rightarrow ([a], Int) \rightarrow [a]$, что достаточно естественно, специфицируется (для непустых списков) так:

$$untransform \cdot transform = id$$

Задача заключается в вычислении *untransform*. Очевидный метод, если удастся заставить его работать, в том, чтобы сначала восстановить упорядоченную матрицу сдвигов, воспользовавшись первым компонентом результата преобразования *transform*, а затем взять ту её строку, номер которой содержится во втором компоненте. Конкретизируем эту мысль: предположим, что можно указать функцию *recreate* :: $Ord a \Rightarrow [a] \rightarrow [[a]]$, которая удовлетворяет равенству:

$$recreate \cdot map last \cdot sort \cdot rrots = sort \cdot rrots$$

Тогда можно определить $untransform (ys, k) = (recreate ys) !! k$, где (!!)) это операция индексирования списка. Но возможно ли воссоздать целую матрицу упорядоченных сдвигов, зная лишь её последний столбец? Да, это возможно, и вывод соответствующей функции *recreate* является восхитительным примером формального построения программ.

Вычисляем, воссоздавая

Будем всюду далее считать, что на вход функции *recreate* подаётся список длины *n*, т.е. она должна восстановить матрицу размера $n \times n$. Идея в том, чтобы строить её столбец за столбцом. Определим *takeCols*:

²Мы также не будем учитывать тот факт, что $transform [] = ([], 0)$, считая, что допустимыми параметрами *transform* являются только непустые списки.

$$\begin{aligned}takeCols &:: Int \rightarrow [[a]] \rightarrow [[a]] \\takeCols j &= map (take j)\end{aligned}$$

Таким образом, $takeCols j$ берёт первые j столбцов матрицы $n \times n$. В частности, $takeCols n$ является тождественной функцией. Теперь заменим функцию $recreate$ на $recreate n$, специфицировав последнюю свойством

$$recreate j \cdot map last \cdot sort \cdot rrots = takeCols j \cdot sort \cdot rrots \quad (13.1)$$

Функция $recreate j$ восстанавливает первые j столбцов упорядоченной матрицы сдвигов на основе её последнего столбца.

План действий заключается в построении индуктивного определения $recreate$. В базовом случае всё легко:

$$recreate 0 = map (const [])$$

Результатом вызова функции $recreate 0$, применённой к списку ys длины n , является столбец из n пустых списков. Так как $takeCols 0$, применённая к любой матрице размера $n \times n$, также даёт n пустых списков, (13.1) для случая $j = 0$ действительно выполняется.

Веселье начинается на индуктивном переходе. Для приготовления коктейля нам понадобится три дополнительных ингредиента. Первый ингредиент это функция $rrot$, выполняющая один циклический сдвиг вправо:

$$\begin{aligned}rrot &:: [a] \rightarrow [a] \\rrot xs &= [last xs] ++ init xs\end{aligned}$$

Или, что то же самое, $rrot (xs ++ [x]) = [x] ++ xs$. Решающее свойство $rrot$ таково:

$$map rrot \cdot rrots = rrot \cdot rrots \quad (13.2)$$

Докажем:

$$\begin{aligned}&map rrot (rots xs) \\&= \{ \text{определение } rrots \} \\&map rrot [xs, lrot xs, lrot^2 xs, \dots, lrot^{n-1} xs] \\&= \{ \text{определение } map, \text{ применяя } rrot \cdot lrot = id \} \\&[rrot xs, xs, lrot xs, \dots, lrot^{n-2} xs] \\&= \{ \text{определение } rrot \} \\&rrot [xs, lrot xs, \dots, lrot^{n-2} xs, rrot xs]\end{aligned}$$

$$\begin{aligned}
 &= \{ \text{так как } rrot xs = lrot^{n-1}xs \} \\
 rrot [xs, lrot xs, \dots, lrot^{n-2}xs, lrot^{n-1}xs] \\
 &= \{ \text{определение } rots \} \\
 rrot (rots xs)
 \end{aligned}$$

Второй ингредиент это функция *hdsort*, которая упорядочивает матрицу по её первому столбцу. Мы можем определить *hdsort*, воспользовавшись стандартной функцией *sortBy*:

$$\begin{aligned}
 hdsort :: Ord a \Rightarrow [[a]] \rightarrow [[a]] \\
 hdsort = sortBy cmp \text{ where } cmp (x : xs) (y : ys) = compare x y
 \end{aligned}$$

Третий ингредиент это функция *consCol*, добавляющая к матрице новый столбец:

$$\begin{aligned}
 consCol :: ([a], [[a]]) \rightarrow [[a]] \\
 consCol (xs, xss) = zipWith (:) xs xss
 \end{aligned}$$

Приведённые выше ингредиенты удовлетворяют разнообразным тождествам, которые потребуются нам в процессе приготовления конструктивного определения функции *recreate*, удовлетворяющего свойству (13.1). Во-первых, при $j < n$ имеем:

$$takeCols (j + 1) \cdot map rrot = consCol \cdot fork (map last, takeCols j) \quad (13.3)$$

где $fork (f, g) x = (f x, g x)$. Правая часть равенства описывает операцию помещения последнего столбца матрицы перед первыми j столбцами, а левая его часть — операцию выполнения для каждой строки матрицы циклического сдвига на один элемент вправо и последующего взятия первых $j + 1$ столбца. Это тождество выражает тот факт, что эти две операции дают одинаковый результат.

Вот второе тождество:

$$takeCols (j + 1) \cdot hdsort = hdsort \cdot takeCols (j + 1) \quad (13.4)$$

Словом, сортируя матрицу размера $n \times n$ по её первому столбцу и выбирая затем положительное число столбцов, получаем в точности тот же результат, если сначала выбрать то же количество столбцов, а затем отсортировать их по первому.

Третье тождество, на этот раз ключевое, не столь очевидно:

$$hdsort \cdot map rrot \cdot sort \cdot rots = sort \cdot rots \quad (13.5)$$

Словом, следующая операция на матрице упорядоченных циклических сдвигов является тождественной: передвинуть последний столбец в начало, а затем пересортировать строки по новому первому столбцу. Вообще говоря, (13.5) истинно, только если алгоритм сортировки *hdsort* является устойчивым, т.е. столбцы с одним и тем же первым элементом сохранят на выходе тот же порядок, который они имели на входе. В рамках этого предположения, применяя это свойство к матрице $n \times n$, получаем:

$$\text{sort} = (\text{hdsort} \cdot \text{map rrot})^n$$

Это тождество утверждает, что матрицу $n \times n$ (а на самом деле и любой список списков, длина каждого из которых n) можно отсортировать, n раз применив операцию циклического сдвига последнего столбца в первую позицию, а затем отсортировав её только по первому столбцу алгоритмом устойчивой сортировки. Поскольку это тождество в точности соответствует операции поразрядной сортировки, будем называть его *свойством поразрядной сортировки*. Его можно доказать индукцией по n , но мы детали опустим. С небольшой помощью оно приведёт нас к доказательству (13.5):

$$\begin{aligned} & \text{hdsort} \cdot \text{map rrot} \cdot \text{sort} \cdot \text{rots} \\ = & \quad \{ \text{свойство поразрядной сортировки} \} \\ & (\text{hdsort} \cdot \text{map rrot})^{n+1} \cdot \text{rots} \\ = & \quad \{ \text{композиция} \} \\ & (\text{hdsort} \cdot \text{map rrot})^n \cdot \text{hdsort} \cdot \text{map rrot} \cdot \text{rots} \\ = & \quad \{ \text{свойство поразрядной сортировки} \} \\ & \text{sort} \cdot \text{hdsort} \cdot \text{map rrot} \cdot \text{rots} \\ = & \quad \{ \text{тождество (13.2)} \} \\ & \text{sort} \cdot \text{hdsort} \cdot \text{rrot} \cdot \text{rots} \\ = & \quad \{ \text{так как } \text{sort} \cdot \text{hdsort} = \text{sort} \} \\ & \text{sort} \cdot \text{rrot} \cdot \text{rots} \\ = & \quad \{ \text{так как } \text{sort} \cdot \text{rrot} = \text{sort} \} \\ & \text{sort} \cdot \text{rots} \end{aligned}$$

Два тождества, использованных на последних двух шагах, можно обобщить до $\text{sort} \cdot \text{perm} = \text{sort}$ для любой перестановки входных данных *perm*.

Теперь мы готовы к работе с *recreate*. Записывая для краткости $sr = \text{sort} \cdot \text{rots}$, выводим:

$$\begin{aligned}
 & recreate(j+1) \cdot map\ last \cdot sr \\
 = & \quad \{ \text{спецификация (13.1)} \} \\
 & takeCols(j+1) \cdot sr \\
 = & \quad \{ \text{свойство (13.5)} \} \\
 & takeCols(j+1) \cdot hdsort \cdot map\ rrot \cdot sr \\
 = & \quad \{ \text{свойство (13.4)} \} \\
 & hdsort \cdot takeCols(j+1) \cdot map\ rrot \cdot sr \\
 = & \quad \{ \text{свойство (13.3)} \} \\
 & hdsort \cdot consCol \cdot fork(map\ last, takeCols\ j) \cdot sr \\
 = & \quad \{ \text{спецификация (13.1)} \} \\
 & hdsort \cdot consCol \cdot fork(map\ last, recreate\ j \cdot map\ last) \cdot sr \\
 = & \quad \{ \text{так как } fork(f \cdot h, g \cdot h) = fork(f, g) \cdot h \} \\
 & hdsort \cdot consCol \cdot fork(id, recreate\ j) \cdot map\ last \cdot sr
 \end{aligned}$$

Следовательно, $recreate(j+1) = hdsort \cdot consCol \cdot fork(id, recreate\ j)$.

Более быстрый алгоритм

К сожалению, и это довольно существенное сожаление, идея воссоздания полной матрицы упорядоченных циклических сдвигов, предшествующего выбору одной конкретной строки, ведёт к неприемлемо неэффективному методу вычисления *untransform*. Воссоздание матрицы размера $n \times n$ выполняется за $\Omega(n^2)$ операций, а квадратичное время для *untransform* нам просто не подходит. На самом деле приведённое выше определение *recreate n* из-за n -кратного вычисления *hdsort* требует $\Omega(n^2 \log n)$ операций. В этом разделе будет показано, как вычислить *untransform* за $\Theta(n \log n)$ операций.

Для получения более хорошего алгоритма вычисления *untransform* нам потребуется ещё больше ингредиентов. Во-первых, заметим, что сортировка списка одноэлементных списков по первому столбцу (*hdsort*) соответствует просто сортировке этого списка:

$$hdsort \cdot map\ wrap = map\ wrap \cdot sort$$

Здесь $wrap\ x = [x]$. Во-вторых, совершенно необязательно несколько раз применять *hdsort*, ведь каждая сортировка применяет одну и ту же перестановку. Точнее говоря, предположим:

$$sort\ ys = apply\ p\ ys$$

где p , зависящее от ys , это перестановка списка $[0..n-1]$, а $apply$ применяет перестановку к списку. Перестановка p определяется так:

$$p = map\ snd\ (sort\ (zip\ ys\ [0..n-1]))$$

а $apply$ так:

$$apply\ p\ xs = [xs\ !!\ (p\ !!\ i) \mid i \leftarrow [0..n-1]]$$

Первое утверждение относительно $apply$ заключается в том, что для любой перестановки p

$$apply\ p\cdot consCol = consCol\cdot pair\ (apply\ p) \quad (13.6)$$

где $pair f(x, y) = (f x, f y)$. Это утверждение можно сформулировать в более общем виде, если обозначить через $beside(M, N)$ операцию размещения матрицы M рядом с матрицей N . Действительно:

$$apply\ p\cdot beside = beside\cdot pair\ (apply\ p)$$

Свойство (13.6) является частным случаем этого утверждения, когда M является матрицей размера $1 \times n$. Пользуясь (13.6), рассуждаем:

$$\begin{aligned} & recreate(j+1) \\ &= \{ \text{определение} \} \\ & hdsort\cdot consCol\cdot fork(id, recreate j) \\ &= \{ \text{берём } hdsort = apply\ p, \text{ где } p \text{ определено выше} \} \\ & apply\ p\cdot consCol\cdot fork(id, recreate j) \\ &= \{ (13.6) \} \\ & consCol\cdot pair\ (apply\ p)\cdot fork(id, recreate j) \\ &= \{ \text{так как } pair f\cdot fork(g, h) = fork(f\cdot g, f\cdot h) \} \\ & consCol\cdot fork(apply\ p, apply\ p\cdot recreate j) \end{aligned}$$

Следовательно, $recreate$ удовлетворяет следующему:

$$\begin{aligned} recreate\ 0 &= map\ (const\ []) \\ recreate\ (j+1) &= consCol\cdot fork\ (apply\ p, apply\ p\cdot recreate\ j) \end{aligned}$$

В этом варианте $recreate$ повторные применения $hdsort$ заменены на повторные применения $apply\ p$.

Следующий шаг состоит в использовании полученного рекурсивного определения *recreate* для отыскания альтернативного определения. Словом, мы разрешаем рекурсию. Покажем, что определение

$$\text{recreate } j = tp \cdot \text{take } j \cdot \text{tail} \cdot \text{iterate } (\text{apply } p) \quad (13.7)$$

является таким решением. Новый ингредиент это функция *tp* (сокращение от *transpose* — транспонировать), стандартная функция языка Haskell, транспонирующая матрицу. Нам потребуются три свойства *tp*:

$$tp \cdot \text{take } 0 = \text{map } (\text{const } []) \quad (13.8)$$

$$tp \cdot \text{take } (j + 1) = \text{consCol} \cdot \text{fork } (\text{head}, tp \cdot \text{take } j \cdot \text{tail}) \quad (13.9)$$

$$\text{apply } p \cdot tp = tp \cdot \text{map } (\text{apply } p) \quad (13.10)$$

Свойство (13.8) утверждает, что результатом транспонирования матрицы $0 \times n$ является матрица $n \times 0$. Транспозицией пустого списка, т.е. списка, не содержащего ни одного списка длины n , является список из n пустых списков. Свойство (13.9) утверждает, что для транспонирования матрицы размера $(j + 1) \times n$ можно добавить первую строку к первому столбцу результата транспозиции матрицы $j \times n$, сформированной из оставшихся строк. Наконец, свойство (13.10), которое можно также сформулировать в эквивалентной форме:

$$\text{apply } p = tp \cdot \text{map } (\text{apply } p) \cdot tp$$

утверждает, что применить перестановку к строкам матрицы можно, сначала её транспонировав, затем применив эту перестановку к каждому столбцу, а затем снова её транспонировав.

Свойство (13.8) немедленно подтверждает (13.7) в случае $j = 0$. Для проверки индуктивного перехода начнём с

$$\text{recreate } (j + 1) = \text{consCol} \cdot \text{fork } (\text{apply } p, \text{apply } p \cdot \text{recreate } j)$$

и будем рассуждать так:

$$\text{fork } (\text{apply } p, \text{apply } p \cdot \text{recreate } j)$$

$$= \{ \text{предполагая (13.7)} \}$$

$$\text{fork } (\text{apply } p, \text{apply } p \cdot tp \cdot \text{map } (\text{apply } p) \cdot \text{take } j \cdot \text{tail} \cdot \text{iterate } (\text{apply } p))$$

$$= \{ (13.10) \}$$

$$\text{fork } (\text{apply } p, tp \cdot \text{map } (\text{apply } p) \cdot \text{take } j \cdot \text{tail} \cdot \text{iterate } (\text{apply } p))$$

$$\begin{aligned}
 &= \{ \text{так как } map f \cdot take j = take j \cdot map f \text{ и} \\
 &\quad map f \cdot tail = tail \cdot map f \} \\
 &fork (apply p, tp \cdot take j \cdot tail \cdot map (apply p) \cdot iterate (apply p)) \\
 &= \{ \text{так как } map f \cdot iterate f = tail \cdot iterate f \} \\
 &fork (apply p, tp \cdot take j \cdot tail \cdot tail \cdot iterate (apply p)) \\
 &= \{ \text{так как } f = head \cdot tail \cdot iterate f \text{ и} \\
 &\quad fork (f \cdot h, g \cdot h) = fork (f, g) \cdot h \} \\
 &fork (head, tp \cdot take j \cdot tail) \cdot tail \cdot iterate (apply p)
 \end{aligned}$$

Теперь, используя (13.9), получим:

$$recreate (j + 1) = tp \cdot take (j + 1) \cdot tail \cdot iterate (apply p)$$

Этот результат завершает доказательство (13.7).

Теперь мы готовы к последнему рассуждению. Вспомним, что если $n = length ys$, то

$$untransform (ys, k) = (recreate n ys) !! k$$

Рассуждаем:

$$\begin{aligned}
 &(!!k) \cdot recreate n \\
 &= \{ (13.7) \} \\
 &(!!k) \cdot tp \cdot take n \cdot tail \cdot iterate (apply p) \\
 &= \{ \text{так как } (!!k) \cdot tp = map (!!k) \} \\
 &map (!!k) \cdot take n \cdot tail \cdot iterate (apply p) \\
 &= \{ \text{так как } map f \cdot take n = take n \cdot map f \text{ и} \\
 &\quad map f \cdot tail = tail \cdot map f \} \\
 &take n \cdot tail \cdot map (!!k) \cdot iterate (apply p)
 \end{aligned}$$

На последнем шаге необходим закон *iterate*: если $f x \oplus y = x \oplus g y$, то имеет место равенство

$$map (\oplus y) (iterate f x) = map (x \oplus) (iterate g y)$$

В доказательстве следует индукцией по n показать, что $f^n x \oplus y = x \oplus g^n y$. Право восстановить детали предоставляемся читателю.

В силу $(apply p ys) !! k = ys !! (p !! k)$ приведённый выше закон даёт:

$$map (!!k) (iterate (apply p ys)) = map (ys !!) (iterate (p !!) k)$$

В целом, *untransform* (*ys*, *k*) вычисляется так:

$$\text{take} (\text{length } ys) (\text{tail} (\text{map} (\text{ys}!!) (\text{iterate} (\text{p}!!) k)))$$

Если бы только операция (!!) выполнялась за константное время, то это вычисление выполнялось бы за линейное. Но это не так, поэтому блюдо пока ещё не совсем готово покидать кухню. На самом-самом последнем шаге привлекаем массивы языка Haskell:

$$\begin{aligned} \text{untransform} (ys, k) &= \text{take } n (\text{tail} (\text{map} (\text{ya}!) (\text{iterate} (\text{pa}!) k))) \\ \text{where } n &= \text{length } ys \\ \text{ya} &= \text{listArray} (0, n - 1) ys \\ \text{pa} &= \text{listArray} (0, n - 1) (\text{map} \text{ snd} (\text{sort} (\text{zip} ys [0 ..]))) \end{aligned}$$

Пересмотренное определение *untransform* использует модуль языка Haskell для работы с массивами *Data.Array*. Выражение *listArray* (*0*, *n* – 1) *ys* строит массив, границы которого 0 и *n* – 1, а значения являются элементами списка *ys*. Эта функция выполняется за линейное время. В отличие от операции индексирования списков (!!), операция индексирования массивов (!) требует константного времени. Вычисление *pa* происходит за $\Theta(n \log n)$ операций, тогда как все остальные вычисления выполняются за $\Theta(n)$ операций.

Возвращаемся к преобразованию

Наконец, давайте вернёмся к функции *transform* и посмотрим, можно ли улучшить её быстродействие. Ключевая идея заключается в том, чтобы выразить сортировку циклических сдвигов списка через сортировку суффиксов некоторого связанного с ним списка. Положим *xs* = *xs* ++ [*eof*], где *eof* (end of file — конец файла) это некоторый элемент, гарантированно отсутствующий в списке *xs*. Например, если *xs* это строка, то в качестве *eof* можно взять символ с кодом 0, а если *xs* это список неотрицательных целых чисел, то можно взять *x* = –1. Поскольку *eof* отличен от всех элементов списка *xs*, перестановка, сортирующая первые *n* суффиксов *tag xs* это та же самая перестановка, которая сортирует *n* циклических сдвигов списка *xs*. Следовательно,

$$\text{sort} (\text{rots } xs) = \text{apply } p (\text{rots } xs)$$

где

$$p = \text{map} \text{ snd} (\text{sort} (\text{zip} (\text{tails} (\text{tag } xs)) [0 .. n - 1]))$$

Стандартная функция *tails* возвращает суффиксы или хвосты списка в порядке уменьшения их длины. Теперь рассуждаем:

$$\begin{aligned}
 & \text{map } \text{last} \cdot \text{sort} \cdot \text{rots} \\
 = & \quad \{ \text{с учётом данного выше определения } p \} \\
 & \text{map } \text{last} \cdot \text{apply } p \cdot \text{rots} \\
 = & \quad \{ \text{так как } \text{map } f \cdot \text{apply } p = \text{apply } p \cdot \text{map } f \} \\
 & \text{apply } p \cdot \text{map last} \cdot \text{rots} \\
 = & \quad \{ \text{так как } \text{map last} \cdot \text{rots} = \text{rrrot} \} \\
 & \text{apply } p \cdot \text{rrrot}
 \end{aligned}$$

Таким образом, $\text{map last} \cdot \text{sort} \cdot \text{rots} = \text{apply } p \cdot \text{rrrot}$. Более того,

$$\begin{aligned}
 & \text{position } xs (\text{sort} (\text{rots } xs)) \\
 = & \quad \{ \text{с учётом данного выше определения } p \} \\
 & \text{position } xs (\text{apply } p (\text{rots } xs)) \\
 = & \quad \{ \text{так как } \text{position } xs (\text{rots } xs) = 0 \} \\
 & \text{position } 0 \ p
 \end{aligned}$$

Следовательно,

$$\begin{aligned}
 & \text{transform } xs = ([xa ! (pa ! i) \mid i \leftarrow [0..n-1]], k) \\
 \text{where } & n = \text{length } ys \\
 & k = \text{length} (\text{takeWhile } (\neq 0) \ ps) \\
 & xa = \text{listArray} (0, n-1) (\text{rrrot } xs) \\
 & pa = \text{listArray} (0, n-1) \ ps \\
 & ps = \text{map } \text{snd} (\text{sort} (\text{zip} (\text{tails} (\text{tag } xs)) [0..n-1]))
 \end{aligned}$$

В этом алгоритме имеется узкое горлышко — вычисление *ps*. Все остальные шаги выполняются за $\Theta(n)$ операций. Как вы видели в предыдущей жемчужине, перестановку, сортирующую хвосты списка длины *n*, можно найти за $\Theta(n \log n)$ операций. Собственно говоря, если наш список это список элементов над некоторым конечным алфавитом, то его суффиксы можно упорядочить за линейное время, построив суффиксное дерево, см. (Gusfield, 1997).

Заключительные замечания

ПБУ было впервые описано в (Burrows and Wheeler, 1994), хотя на самом деле алгоритм был открыт Уилером ещё в 1983 году. В статье

(Nelson, 1996), приковавшей к ПБУ внимание всего мира, было показано, что получающийся в результате алгоритм сжатия может превосходить многие имевшиеся в то время коммерческие приложения. Сейчас ПБУ интегрировано в высокопроизводительную утилиту `bzip2`, доступную на сайте www.bzip.org. Поразрядная сортировка рассматривается в (Gibbons, 1999), где она выводится из сортировки деревом посредством бесточечных преобразований.

Литература

- Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Research report 124, Digital Systems Research Center, Palo Alto, USA.
- Gibbons, J. (1999). A pointless derivation of radix sort. *Journal of Functional Programming* 9 (3) 339–46.
- Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK. [Имеется русский перевод: Гасфильд Д. Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, БХВ-Петербург. 2003.]
- Nelson, M. (1996). Data compression with the Burrows–Wheeler transform. *Dr. Dobb's Journal, September*.

14

Последний хвост

Введение

Предположим, что хвосты списка упорядочены в словарном порядке. Какой из хвостов будет последним? К примеру, последний хвост слова «введение» это «ние», а для слова «томат» это сам «томат», поскольку «т» идёт в словаре раньше «томат». Из одной из предыдущих жемчужин о ранжировании и сортировке суффиксов (см. жемчужину 12) следует, что эту задачу для списка длины n можно решить за $\Theta(n \log n)$ операций. Но, быть может, последний хвост удастся вычислить за $\Theta(n)$ операций? Оказывается, ответ на этот вопрос положителен, но алгоритм для столь просто формулируемой задачи на удивление сложный, а его вывод требует существенных усилий. Итак, мы вас предупредили.

Индуктивное определение

Наша задача заключается в вычислении

$$\begin{aligned} \textit{maxtail} &:: \textit{Ord } a \Rightarrow [a] \rightarrow [a] \\ \textit{maxtail} &= \textit{maximum} \cdot \textit{tails} \end{aligned}$$

Функция *tails* здесь это стандартная функция языка Haskell, которая возвращает все хвосты списка, в том числе и возможно пустой хвост пустого списка, поэтому *maxtail* [] = []. Функция *maximum* возвращает наибольший в лексикографическом порядке список. Непосредственное выполнение

maxtail требует в худшем случае квадратичного времени, это случится, например, если список содержит n повторов одного и того же значения.

Наша стратегия улучшений заключается в работе с индуктивным определением *maxtail*. Это означает, что мы хотим выразить *maxtail* от списка длины $n + 1$ в терминах *maxtail* от списка длины n . Базовый случай следует непосредственно из того, что $\text{maxtail} [] = []$. При обсуждении индуктивного перехода есть две возможности: можно либо выразить *maxtail* ($x : xs$) через x и *maxtail* xs , либо через них же выразить *maxtail* ($xs \uparrow [x]$). Однако, к примеру, наибольший суффикс слова «хорёк», собственно «хорёк», нельзя выразить через «х» и максимальный суффикс «орёк», т.е. «рёк». Поэтому попытаемся найти такую операцию *op*, что

$$\text{maxtail} (xs \uparrow [x]) = op (\text{maxtail} xs) x \quad (14.1)$$

Иными словами, мы ищем *op*, для которой $\text{maxtail} = foldl op []$.

Положим с этой целью $\text{maxtail} xs = ys$ и $\text{maxtail} (xs \uparrow [x]) = zs \uparrow [x]$. Чтобы удовлетворить (14.1), следует показать, что $zs \in tails ys$. Так как и ys , и zs являются хвостами xs , по определению *maxtail* имеем, что $zs \leq ys$ и $ys \uparrow [x] \leq zs \uparrow [x]$. Пусть $zs \leq ys$, но zs не является префиксом ys . Тогда $us \uparrow [z] \sqsubseteq zs$, а $us \uparrow [y] \sqsubseteq ys$, где $z < y$. В этом случае $zs \uparrow [x] < ys \uparrow [x]$.

Таким образом, zs является префиксом ys . Но они оба являются хвостами xs , поэтому zs одновременно префикс и суффикс ys . Следовательно, *op* можно определить двумя способами:

$$\begin{aligned} op ys x &= \text{maximum} [zs \uparrow [x] \mid zs \leftarrow tails ys] \\ op ys x &= \text{maximum} [zs \uparrow [x] \mid zs \leftarrow tails ys, zs \sqsubseteq ys] \end{aligned} \quad (14.2)$$

Первый шаг сделан, хотя ни одно из определений и не даёт нам линейный по времени алгоритм.

Границы

Определение (14.2) имеет более богатую, нежели его компаньон, структуру, поэтому оно достойно дальнейшего изучения. Введём в рассмотрение функцию *borders* со следующим определением:

$$\text{borders} xs = [ys \mid ys \leftarrow tails xs, ys \sqsubseteq xs] \quad (14.3)$$

В «строковедении» (Crochemore and Rytter, 2003) границами (*borders*) списка называются суффиксы, одновременно являющиеся префиксами. Вот два примера:

$$\begin{aligned} \text{borders } "7412741274" &= ["7412741274", "741274", "74", ""] \\ \text{borders } "mammam" &= ["mammam", "mam", "m", ""] \end{aligned}$$

Определение (14.3) можно заменить на

$$\begin{aligned} \text{borders } [] &= [[]] \\ \text{borders } xs &= xs : \text{borders } (\text{border } xs) \end{aligned}$$

где $\text{border } xs$ это самый длинный собственный общий префикс и суффикс списка xs . Форма второго определения совпадает с формой определения tails :

$$\begin{aligned} \text{tails } [] &= [[]] \\ \text{tails } xs &= xs : \text{tails } (\text{tail } ys) \end{aligned}$$

Если переформулировать (14.2) в терминах borders , получим:

$$\text{op } ys \ x = \text{maximum } [zs \uparrow [x] \mid zs \leftarrow \text{borders } ys]$$

Предположим, что $\text{borders } ys = [zs_0, zs_1, \dots, zs_n]$, т.е. $zs_0 = ys$, а $zs_{i+1} = \text{border } zs_i$ при $0 \leq i < n$ и $zs_n = []$. Посмотрим, какую информацию о значении $\text{op } ys \ x$ можно извлечь из этих ингредиентов. Во-первых, при $0 \leq i < j \leq n$ имеем

$$zs_i \uparrow [x] \geq zs_j \uparrow [x] \equiv \text{head } (zs_i \downarrow zs_j) \geq x \quad (14.4)$$

где $us \downarrow vs$ (произносится как « us после vs ») специфицировано равенством $(xs \uparrow ys) \downarrow xs = ys$. Вот доказательство (14.4):

$$\begin{aligned} &zs_i \uparrow [x] \geq zs_j \uparrow [x] \\ &\equiv \{ \text{определение лексикографического порядка, так как } zs_j \sqsubseteq zs_i \} \\ &\quad (zs_i \downarrow zs_j) \uparrow [x] \geq [x] \\ &\equiv \{ \text{так как список } zs_i \downarrow zs_j \text{ непуст при } i < j \} \\ &\quad \text{head } (zs_i \downarrow zs_j) \geq x \end{aligned}$$

Во-вторых, в случае $ys = \text{maxtail } xs$ для некоторого xs получаем при $0 < i < j \leq n$, что

$$\text{head } (zs_{i-1} \downarrow zs_i) \leq \text{head } (zs_{j-1} \downarrow zs_j) \quad (14.5)$$

Доказательство:

$$\text{head } (zs_{i-1} \downarrow zs_i) \leq \text{head } (zs_{j-1} \downarrow zs_j)$$

$$\begin{aligned}
 &= \{ \text{так как из } 0 < k \text{ следует } \text{head}(zs_{k-1} \downarrow zsk) = \text{head}(ys \downarrow zsk) \} \\
 &\quad \text{head}(ys \downarrow zsi) \leqslant \text{head}(ys \downarrow zsj) \\
 &\Leftarrow \{ \text{лексикографическое упорядочивание} \} \\
 &\quad zsj \mathbin{\text{++}} (ys \downarrow zsi) \leqslant zsk \mathbin{\text{++}} (ys \downarrow zsk) \\
 &\equiv \{ \text{так как } ys = zsk \mathbin{\text{++}} (ys \downarrow zsk) \text{ для любых } k \} \\
 &\quad zsj \mathbin{\text{++}} (ys \downarrow zsi) \leqslant ys \\
 &\Leftarrow \{ \text{так как } ys \text{ является максимальным суффиксом} \} \\
 &\quad zsj \mathbin{\text{++}} (ys \downarrow zsi) \in \text{tails } ys \\
 &\equiv \{ \text{так как } ys = zsi \mathbin{\text{++}} (ys \downarrow zsi) \text{ и } zsj \in \text{tails } zsi \} \\
 &\quad \text{true}
 \end{aligned}$$

Из (14.4) и (14.5) следует, что $\text{op } ys \ x = zsi \mathbin{\text{++}} [x]$, где i это наименьшее i из диапазона $0 \leqslant i < n$, удовлетворяющее условию $\text{head}(zsi \downarrow zsi+1) \geqslant x$, если оно, конечно, существует. Если такого i нет, то $\text{op } ys \ x = [x]$. Непосредственная реализация поиска приводит к следующему определению:

$$\begin{array}{lll}
 \text{op } ys \ x \mid \text{null } ys & = [x] \\
 \mid \text{head}(ys \downarrow zs) \geqslant x & = ys \mathbin{\text{++}} [x] \\
 \mid \text{otherwise} & = \text{op } zs \ x \\
 & \text{where } zs = \text{border } ys
 \end{array}$$

Для завершения этого определения op нам необходимо получить определение функции border .

Граница

Напомним, что $\text{border } ys$ это самый длинный собственный общий префикс и суффикс ys . Значение $\text{border } ys$ определено только для непустого ys . Если обратиться к индуктивному определению, то сразу ясно, что $\text{border } [x] = []$. На индуктивном переходе для $\text{border } (ys \mathbin{\text{++}} [x])$ нам необходимо следующее свойство упорядочивания префиксов:

$$zs \mathbin{\text{++}} [x] \sqsubseteq ys \mathbin{\text{++}} [x] \equiv zs \sqsubseteq ys \wedge (zs \neq ys \Rightarrow x = \text{head}(ys \downarrow zs))$$

Теперь для непустого ys рассуждаем так:

$$\begin{aligned}
 &\text{borders}(ys \mathbin{\text{++}} [x]) \\
 &= \{ (14.3) \} \\
 &\quad [zs \mid zs \leftarrow \text{tails}(ys \mathbin{\text{++}} [x]), zs \sqsubseteq ys \mathbin{\text{++}} [x]]
 \end{aligned}$$

$$\begin{aligned}
 &= \{ \text{так как } tails(ys \uplus [x]) = map(\uplus[x])(tails ys) \uplus [[]] \} \\
 &[zs \uplus [x] \mid zs \leftarrow tails ys, zs \uplus [x] \sqsubseteq ys \uplus [x]] \uplus [[]] \\
 &= \{ \text{свойство, приведённое выше} \} \\
 &[zs \uplus [x] \mid zs \leftarrow tails ys, zs \sqsubseteq ys, zs \neq ys \Rightarrow x = head(ys \downarrow zs)] \uplus [[]] \\
 &= \{ (14.3) \} \\
 &[zs \uplus [x] \mid zs \leftarrow borders ys, zs \neq ys \Rightarrow x = head(ys \downarrow zs)] \uplus []
 \end{aligned}$$

Следовательно, поскольку $\text{border} = \text{head} \cdot \text{tail} \cdot \text{borders}$ и $\text{head} \cdot \text{borders} = id$, получаем:

$$\begin{aligned}
 &\text{border}(ys \uplus [x]) \\
 &= \text{head}([zs \uplus [x] \mid zs \leftarrow tail(borders ys), x = head(ys \downarrow zs)]) \uplus []
 \end{aligned}$$

Этот генератор списков можно заменить на явный поиск:

$$\begin{aligned}
 &\text{border}(ys \uplus [x]) \mid \text{head}(ys \downarrow zs) == x = zs \uplus [x] \\
 &\quad | \text{otherwise} = \text{border}(zs \uplus [x]) \\
 &\quad \text{where } zs = \text{border } ys
 \end{aligned}$$

Предполагая, что для некоторых xs выполняется $ys = maxtail xs$, мы можем воспользоваться (14.5) и оптимизировать поиск:

$$\begin{aligned}
 &\text{border}(ys \uplus [x]) \mid \text{head}(ys \downarrow zs) < x = \text{border}(zs \uplus [x]) \\
 &\quad | \text{head}(ys \downarrow zs) == x = zs \uplus [x] \\
 &\quad | \text{head}(ys \downarrow zs) > x = [] \\
 &\quad \text{where } zs = \text{border } ys
 \end{aligned}$$

Рекурсивные определения $maxtail$ и border оказались похожими, поэтому было бы логично на следующем шаге объединить их в одну функцию.

Коктейль

Рассмотрим функцию $cocktail$ (комбинация двух ингредиентов!), определяемую так:

$$\text{cocktail } xs = \text{if } null \, xs \text{ then } ([] , []) \text{ else } (\text{border}(\text{maxtail } xs), \text{maxtail } xs \downarrow \text{border}(\text{maxtail } xs))$$

В частности, $\text{maxtail} = uncurry(\uplus) \cdot cocktail$.

Теперь, пользуясь рекурсивными определениями $maxtail$ и border , выведем рекурсивное определение $cocktail$. Положив $cocktail \, xs = (zs, ws)$, вычислим $cocktail(xs \uplus [x])$, разбирая отдельные случаи.

Случай $ws = []$. В этом случае $xs = ys = zs = []$, поэтому

$$\text{cocktail}(xs ++ [x]) = ([], [x])$$

В остальных случаях, в которых $ws \neq []$, имеем $ys = \text{maxtail } xs$, где $ys = zs ++ ws$, и $zs = \text{border } ys$, поэтому $ws = ys \downarrow zs$.

Случай $\text{head } ws < x$. В этом случае определение maxtail даёт:

$$\text{maxtail}(xs ++ [x]) = \text{maxtail}(zs ++ [x])$$

поэтому $\text{cocktail}(xs ++ [x]) = \text{cocktail}(zs ++ [x])$.

Случай $\text{head } ws = x$. Получаем:

$$\begin{aligned} \text{maxtail}(xs ++ [x]) &= ys ++ [x] \\ \text{border}(ys ++ [x]) &= zs ++ [x] \end{aligned}$$

Так как $(ys ++ [x]) \downarrow (zs ++ [x]) = \text{tail } ws ++ [x]$, то

$$\text{cocktail}(xs ++ [x]) = (zs ++ [x], \text{tail } ws ++ [x])$$

Случай $\text{head } ws > x$. В этом случае

$$\begin{aligned} \text{maxtail}(xs ++ [x]) &= ys ++ [x] \\ \text{border}(ys ++ [x]) &= [] \end{aligned}$$

поэтому $\text{cocktail}(xs ++ [x]) = ([], ys ++ [x])$.

В целом, мы показали, что

$$\begin{aligned} \text{maxtail} &= \text{uncurry } (+) \cdot \text{cocktail} \\ \text{cocktail} &= \text{foldl } op ([] , []) \\ op(zs, ws) \ x \mid \text{null } ws &= ([], [x]) \\ | \ w < x &= \text{cocktail}(zs ++ [x]) \\ | \ w == x &= (zs ++ [x], \text{tail } ws ++ [x]) \\ | \ w > x &= ([], zs ++ ws ++ [x]) \\ \text{where } w &= \text{head } ws \end{aligned}$$

Ну что ж, неплохо, правда новая версия maxtail всё ещё требует квадратичного времени. Одна из причин в том, что операция $(+)$ выполняется не с константным временем. Но даже если бы она оказалась таковой, общее время вычисления всё равно было бы квадратичным. Рассмотрим входные данные в виде $1^n 2$, где 1^n обозначает n -кратное повторение единицы. Спустя n операций вычисление $\text{cocktail} 1^n 2$ сведётся к вычислению $op(1^{n-1}, 1) 2$.

Так как $1 < 2$, на следующем шаге нужно будет вычислить $\text{cocktail } 1^{n-1}2$. Следовательно, время вычислений квадратичное.

Сложность лежит в вызове $\text{cocktail}(\text{zs} + [x])$. Если бы мы смогли как-нибудь ограничить длину zs не более чем половиной длины текущего наибольшего хвоста $\text{ys} = \text{zs} \text{++ ws}$, то вычисление cocktail потребовало бы линейного времени (пренебрегая затратами на операцию ++). Если удастся потратить линейное время на сведение задачи к не более чем половинному размеру, то и вся задача решится алгоритмом с линейным временем. К счастью, как мы сейчас покажем, размер zs можно ограничить именно так.

Уменьшение размера задачи

Если положить $\text{cocktail xs} = (\text{zs}, \text{ws})$, то $\text{ys} = \text{zs} \text{++ ws}$ окажется наибольшим хвостом xs , а $\text{zs} = \text{border ys}$. Предположим, что $|\text{zs}| \geq |\text{ws}|$, где $|\text{xs}|$ обозначает длину списка xs . В этом случае ws является хвостом zs . Например, если $\text{ys} = "7412741274"$, то $\text{zs} = "741274"$, а $\text{ws} = "1274"$.

Определим zs' как $\text{zs} = \text{zs}' \text{++ ws}$. Тогда zs' это одновременно префикс и хвост zs , а значит, префикс и хвост ys . В приведённом выше примере $\text{zs}' = "74"$. Это рассуждение можно повторить, если $|\text{zs}'| \geq |\text{ws}|$. Отсюда следует, что если мы определим q и r так:

$$(q, r) = (|\text{zs}| \text{ div } |\text{ws}|, |\text{zs}| \text{ mod } |\text{ws}|)$$

и положим $\text{zs}' = \text{take } r \text{ zs}$, то $\text{zs}' \in \text{borders ys}$, а $\text{zs} = \text{zs}' \text{++ ws}^q$, где ws^q это конкатенация q копий ws . Далее, поскольку $|\text{zs}'| < |\text{ws}|$, получаем, что zs' является хвостом ws (значит, $\text{zs}' = \text{drop}(|\text{ws}| - r) \text{ ws}$, этим фактом мы воспользуемся позднее), и каждый из списков $\text{zs}' \text{++ ws}^p$ при $1 \leq p < q$ является также границей zs . Но $(\text{zs}' \text{++ ws}^p) \downarrow (\text{zs}' \text{++ ws}^{p-1}) = \text{ws}$, поэтому в случае $\text{head ws} < x$ при вычислении $\text{op}(\text{zs}, \text{ws})$ x нет необходимости проверять ни одну из этих границ. Следовательно, определение op можно заменить на новую версию:

$$\begin{aligned} \text{op}(\text{zs}, \text{ws}) x \mid \text{null ws} &= ([], [x]) \\ \mid w < x &= \text{cocktail}(\text{take } r \text{ zs} + [x]) \\ \mid w == x &= (\text{zs} \text{++} [x], \text{tail ws} \text{++} [x]) \\ \mid w > x &= ([], \text{zs} \text{++ ws} \text{++} [x]) \\ \text{where } w &= \text{head ws} \\ r &= (\text{length zs}) \text{ mod } (\text{length ws}) \end{aligned}$$

Более того, $2r < |\text{zs} \text{++ ws}|$, так как $r \leq |\text{zs}|$ и $r < |\text{ws}|$. Вооружившись этим

фактом, покажем, что вычисление *cocktail* использует в целом не более чем $2n - m$ вызовов *op*, где $n = |xs|$, а $m = |\maxtail xs|$.

Доказательство проведём индукцией. В случае $n = 0$ имеем, что $m = 0$, и вызовов *op* нет. При $n = 1$ имеем, что $m = 1$, и есть один вызов *op*. Эти рассуждения подтверждают базис индукции.

На индуктивном переходе рассмотрим вычисление *cocktail* ($xs \mathbin{+} [x]$), в котором сначала находится значение *cocktail* xs , а затем вычисляется *op* (*cocktail* xs) x . Согласно предположению индукции для вычисления *cocktail* xs необходимо $2n - m$ вызовов *op*, при этом возвращается пара (zs, ws) , где $|zs \mathbin{+} ws| = m$, а *head ws* = w . Если $w \geq x$, то больше вызовов *op* нет, поэтому общее их количество равно $2n - m + 1$. Но в этом случае полученный наибольший хвост имеет длину $m + 1$, и так как $2n - m + 1 = 2(n + 1) - (m + 1)$, то индуктивный переход в этом случае обоснован. Если же $w < x$, то следует считать вызовы *op* для *cocktail* (*taker* $zs \mathbin{+} [x]$). По индукции их будет $2(r + 1) - m'$, где m' это длина полученного наибольшего хвоста. Сумма, следовательно, равна $2n - m + 1 + 2(r + 1) - m'$, что в силу $2r + 1 \leq m$ не превосходит $2(n + 1) - m'$.

Таким образом, если не учитывать затраты на вычисление *length* и $\mathbin{+}$, то *cocktail* и *maxtail* вычисляются за линейное время. Остаётся избавиться от операций *length* и $\mathbin{+}$.

Окончательная оптимизация

Сначала мы удалим вычисление длин из определения *op*, захватив за одно первый вызов $\mathbin{+}$ в последнем условии. Этот результат можно достичь, уточнив данные следующим образом: заменим состояние (zs, ws) на четвёрку (p, q, ys, ws) , в которой $ys = zs \mathbin{+} ws$, $p = \text{length } zs$, а $q = \text{length } ws$. Причина, по которой можно отбросить аргумент zs , заключается в том, что *take r* $zs = \text{drop}(q - r) ws$. Длина zs нам, однако, нужна. Применить эти изменения несложно, поэтому детали мы опустим. В результате получим программу на рис. 14.1, в которой функция *cocktail* уже уточнена, а *thd* извлекает третий компонент четвёрки.

Теперь остались только операции $(\mathbin{+}[x])$. Одним из способов гарантировать константное время для таких операций было бы преобразование всех списков в очереди, эффективные реализации которых обеспечивают вставку в конец очереди и удаление из её начала за константное время. Другой способ подсказан следующим наблюдением: оба списка ys и ws , возникающие при вычислении *cocktail* xs , являются хвостами xs . Операция *op* формирует эти списки шаг за шагом, и было бы эффективнее создавать их разом. Это можно сделать, преобразовав полученный нами алгоритм в

```

maxtail = thd · cocktail
cocktail = foldl op (0, 0, [], [])
op (p, q, ys, ws) x
| q == 0      = (0, 1, [x], [x])
| w < x      = cocktail (drop (q - r) ws ++ [x])
| w == x      = (p + 1, q, ys ++ [x], tail ws ++ [x])
| otherwise    = (0, p + q + 1, ys ++ [x], ys ++ [x])
  where w = head ws
        r = p mod q

```

Рис. 14.1: результат уточнения данных

итеративный.

Предположим, что функция *step* определена так:

$$\text{step} (p, q, ys', ws', xs) = \text{thd} (\text{foldl} \text{ op} (p, q, ys' \uparrow xs, ws' \uparrow xs) xs)$$

где $us \uparrow vs$ (произносится «*us* перед *vs*») это то, что остаётся, когда хвост *vs* списка *us* удаляется из *us*. Таким образом, $us = (us \uparrow vs) ++ vs$. В частности,

$$\begin{aligned} \text{maxtail} (x : xs) &= \text{thd} (\text{foldl} \text{ op} (0, 1, [x], [x]) xs) \\ &= \text{thd} (\text{foldl} \text{ op} (0, 1, (x : xs) \uparrow xs, (x : xs) \uparrow xs) xs) \\ &= \text{step} (0, 1, x : xs, x : xs, xs) \end{aligned}$$

Теперь получим рекурсивное определение для $\text{step} (p, q, ys', ws', xs)$. Так как $us \uparrow [] = us$, получаем:

$$\text{step} (p, q, ys', ws', []) = ys'$$

В случае, когда *xs* непуст (значит, непусты и *ys'*, и *ws'*), имеем:

$$\begin{aligned} \text{step} (p, q, ys', ws', x : xs) &= \text{thd} (\text{foldl} \text{ op} (p, q, ys' \uparrow (x : xs), ws' \uparrow (x : xs)) (x : xs)) \\ &= \text{thd} (\text{foldl} \text{ op} (\text{op} (p, q, ys' \uparrow (x : xs), ws' \uparrow (x : xs)) x) xs) \end{aligned}$$

Требуется разбор случаев.

Случай $\text{head} (ws' \uparrow (x : xs)) < x$. В этом случае

$$\begin{aligned} \text{op} (p, q, ys' \uparrow (x : xs), ws' \uparrow (x : xs)) x &= \\ \text{cocktail}' (\text{drop} (q - r) (ws' \uparrow (x : xs)) ++ [x]) \end{aligned}$$

где $r = p \bmod q$. Обозначив $\text{drop}(q - r)(ws' \uparrow (x : xs))$ через vs , будем рассуждать:

$$\begin{aligned}
 & \text{step}(p, q, ys', ws', x : xs) \\
 &= \{ \text{определение и разбираемый случай} \} \\
 & \text{thd}(\text{foldl } op(\text{cocktail}'(vs + [x])) xs) \\
 &= \{ \text{определение cocktail}' \} \\
 & \text{thd}(\text{foldl } op(\text{foldl } op(0, 0, [], [], [])(vs + [x])) xs) \\
 &= \{ \text{так как } \text{foldl } f(\text{foldl } f e xs) ys = \text{foldl } f e (xs + ys) \} \\
 & \text{thd}(\text{foldl } op([], [], [])(vs + x : xs)) \\
 &= \{ \text{так как } \text{drop}(q - r)(ws' \uparrow (x : xs)) + x : xs = \text{drop}(q - r) ws' \} \\
 & \text{thd}(\text{foldl } op([], [], [])(\text{drop}(q - r) ws')) \\
 &= \{ \text{определение maxtail} \} \\
 & \text{maxtail}(\text{drop}(q - r) ws')
 \end{aligned}$$

Случай $\text{head}(ws' \uparrow (x : xs)) = x$. Здесь

$$\begin{aligned}
 & op(p, q, ys' \uparrow (x : xs), ws' \uparrow (x : xs)) x = \\
 & (p + 1, q, ys' \uparrow (x : xs) + [x], \text{tail}(ws' \uparrow (x : xs)) + [x]) \\
 &= (p + 1, q, ys' \uparrow xs, \text{tail}(ws' \uparrow xs))
 \end{aligned}$$

Следовательно, $\text{step}(p, q, ys', ws', x : xs) = (p + 1, q, ys', \text{tail } ws', xs)$.

Случай $\text{head}(ws' \uparrow (x : xs)) > x$. Теперь

$$\begin{aligned}
 & op(p, q, ys' \uparrow (x : xs), ws' \uparrow (x : xs)) x = \\
 & (0, p + q + 1, ys' \uparrow (x : xs) + [x], ys' \uparrow (x : xs) + [x]) \\
 &= (0, p + q + 1, ys' \uparrow xs, ys' \uparrow xs)
 \end{aligned}$$

Следовательно, $\text{step}(p, q, ys', ws', x : xs) = (0, p + q + 1, ys', ys', xs)$.

Собрав всё вышеприведённое, получаем окончательный вид нашей программы:

$$\begin{aligned}
 & \text{maxtail}[] = [] \\
 & \text{maxtail}(x : xs) = \text{step}(0, 1, x : xs, x : xs, xs) \\
 & \text{step}(p, q, ys, ws, []) = ys \\
 & \text{step}(p, q, ys, w : ws, x : xs) \\
 & \quad | w < x = \text{maxtail}(\text{drop}(q - r)(w : ws)) \\
 & \quad | w == x = \text{step}(p + 1, q, ys, ws, xs) \\
 & \quad | w > x = \text{step}(0, p + q + 1, ys, ys, xs) \\
 & \text{where } r = p \bmod q
 \end{aligned}$$

Заключительные замечания

Окончательную версию функции *maxtail* нетрудно преобразовать в простой цикл while. Это, вероятно, неудивительно, поскольку в наши намерения входило получение индуктивного определения, а форма, к которой мы пришли, а именно экземпляр свёртки *foldl*, представляет собой не что иное как цикл while в функциональных одеждах. И тем не менее, полученный в итоге алгоритм имеет совершенно императивный дух, поэтому было бы небезынтересно посмотреть на его вывод в императивном стиле с инвариантами циклов. Наш вывод оказался довольно долгим, и в нём потребовались достаточно тонкие рассуждения, это произошло, в основном, из-за присущей задаче внутренней структуры. Возможно, правда, что для самой короткой спецификации в этой книге имеется и более простое решение.

Литература

Crochemore, M. and Rytter, W. (2003). *Jewels of Stringology*. Hong Kong: World Scientific.

15

Все общие префиксы

Введение

Пусть $llcp\ xs\ ys$ обозначает длину самого длинного общего префикса двух списков xs и ys . Например, $llcp\ "компот"\ "компьютер"$ = 4. Рассмотрим теперь функцию $allcp$ (длины всех общих префикс), определённую так:

$$allcp\ xs\ =\ map\ (llcp\ xs)\ (tails\ xs)$$

где $tails\ xs$ возвращает непустые хвосты xs . Например,

xs	a	b	a	c	a	b	a	c	a	b
$allcp\ xs$	10	0	1	0	6	0	1	0	2	0

Первый элемент $allcp\ xs$ это, конечно, $length\ xs$. При непосредственном выполнении, это определение $allcp$ требует квадратичного времени. Но можно ли это сделать за линейное время? Можно, и цель этой жемчужины в том, чтобы это показать. Функция $allcp$ является важным компонентом алгоритма сопоставления строк Бойера—Мура, который мы изучим в следующей жемчужине, поэтому линейное решение представляет как практический, так и теоретический интерес.

Ключевое свойство

Ключевое свойство функции $llcp$, на котором и основывается быстрый алгоритм, заключается в следующем. Пусть us , vs и ws это три произволь-

ных списка. Тогда, если $llcp\ us\ vs = m$ и $llcp\ vs\ ws = n$, то

$$llcp\ us\ ws = \begin{cases} m \min n, & \text{если } m \neq n \\ m + llcp\ (drop\ m\ us)\ (drop\ m\ ws), & \text{если } m = n \end{cases} \quad (15.1)$$

Чтобы это доказать, заметим, что первые $m \min n$ элементов совпадают во всех трёх списках. Если $m < n$, то следующий элемент us (если он есть) отличается от следующего элемента ws . Следовательно, $llcp\ us\ ws = m$. Рассуждения в случае $m > n$ аналогичны. Наконец, если $m = n$, то сопоставление необходимо продолжать со строками $drop\ m\ us$ и $drop\ m\ ws$.

Чтобы воспользоваться (15.1), возьмём i и j из диапазона $1 \leq i, j < n$, где $n = length\ xs$, и положим

$$\begin{aligned} p &= llcp\ xs\ (drop\ i\ xs) \\ q &= llcp\ xs\ (drop\ j\ xs) \end{aligned}$$

Другими словами, элементы в позициях i и j в $allcp\ xs$ это p и q соответственно. Далее, предположим, что $j \leq p$. Тогда по определению $llcp$ имеем:

$$p = j + llcp\ (drop\ j\ xs)\ (drop\ (i + j)\ xs)$$

Положим также $us = xs$, $vs = drop\ j\ xs$ и $ws = drop\ k\ xs$, где $k = i + j$, (15.1) теперь даёт:

$$llcp\ xs\ (drop\ k\ xs) = \begin{cases} (p - j) \ min q, & \text{если } q \neq p - j \\ q + llcp\ (drop\ q\ xs)\ (drop\ (q + k)\ xs), & \text{если } q = p - j \end{cases}$$

Иными словами, мы можем найти k -е вхождение в $allcp\ xs$ по i -му и j -му без всякого труда (верхняя строка) или с небольшой дополнительной работой (нижняя строка). Разумеется, работу можно избежать, только если $1 < i < k$ и $j = k - i < p$, поскольку вторая строка не приводит к сокращению объёма вычислений при $j = p$. В частности, случаи $k = 0$ и $k = 1$ придётся вычислять напрямую.

Опишем, как можно пользоваться этой информацией для вычисления k -го вхождения в $allcp$ в порядке $k = 1, 2, \dots, n$. Предположим, что на каждом шаге мы выбираем i согласно условию, что $i + p$ велико настолько, насколько это возможно с учётом $1 \leq i < k$. Если $k < i + p$, то описанное выше сокращение применимо с $j = k - i$. Если $k \geq i + p$, то не остаётся ничего, кроме как вычислять $llcp\ xs\ (drop\ k\ xs)$ напрямую. Мы можем начать с $(i, p) = (0, 0)$, что обеспечит прямое вычисление случая $k = 1$, а

```

 $\text{allcp } xs = \text{fst4 } (\text{until } (\text{done } n) (\text{step } xs) ([n], 0, 0, 1))$ 
  where  $n = \text{length } xs$ 
 $\text{done } n (as, i, p, k) = k == n$ 
 $\text{step } xs (as, i, p, k)$ 
  |  $k \geq i + p = (\text{snoc } as\ a, k, a, k + 1)$ 
  |  $q \neq r = (\text{snoc } as\ (q \min r), i, p, k + 1)$ 
  |  $q == r = (\text{snoc } as\ b, k, b, k + 1)$ 
  where  $q = as\ !!\ (k - i)$ 
         $r = p - (k - i)$ 
         $a = llcp\ xs\ (\text{drop } k\ xs)$ 
         $b = q + llcp\ (\text{drop } q\ xs)\ (\text{drop } (q + k)\ xs)$ 
 $\text{fst4 } (a, b, c, d) = a$ 
 $\text{snoc } xs\ x = xs \text{++} [x]$ 
 $\text{llcp } xs\ [] = 0$ 
 $\text{llcp } []\ ys = 0$ 
 $\text{llcp } (x : xs)\ (y : ys) = \text{if } x == y \text{ then } 1 + \text{llcp } xs\ ys \text{ else } 0$ 

```

Рис. 15.1: исходная программа

затем будем изменять значение (i, p) , как только находится более удачный вариант.

Всё это приводит к программе на рис. 15.1, которая принимает вид простого цикла. Чтобы убедиться в корректности изменения значений i и p , заметим, что в первом варианте определения step из $k \geq i + p$ следует, что $k + a \geq i + p$, а в третьем варианте $k + b \geq k + q = k + r = i + p$.

Мы утверждаем, что эта программа выполняется за линейное время в предположении, что все операции snoc , $(!!)$ и drop требуют константного времени. Для доказательства этого утверждения достаточно показать, что общее число проверок на равенство в llcp линейно по n . Эти проверки возвращают либо True (совпадение), либо False (несовпадение). Каждый вызов step заканчивается не более чем одним несовпадением, поэтому всего может быть не более чем $n - 1$ несовпадение. Ограничить число совпадений можно, заметив, что на любом шаге, на котором случилось m совпадений, либо $a = m$, либо $b = m$, поэтому значение $i + p$ увеличивается как минимум на m . Так как $i + p \leq n$, то общее число совпадений не превосходит n .

```


$$\begin{aligned}
allcp \ xs &= extract \ (until \ done \ step \ (as, empty, 0, 1)) \\
\text{where} \\
extract \ (as, qs, h, k) &= elems \ as \\
done \ (as, qs, h, k) &= (k == n) \\
n &= length \ xs \\
as &= insert \ empty \ n \\
xa &= listArray \ (0, n - 1) \ xs \\
step \ (as, qs, h, k) &| k \geq h = (insert \ as \ a, insert \ as' \ a, k + a, k + 1) \\
&| q \neq r = (insert \ as \ m, insert \ qs' \ m, h, k + 1) \\
&| q == r = (insert \ as \ b, insert \ as' \ b, k + b, k + 1) \\
&\quad \text{where } as' = snd \ (remove \ as) \\
&\quad (q, qs') = remove \ qs \\
&\quad r = h - k \\
&\quad m = q \min \ r \\
&\quad a = llcp' \ 0 \ k \\
&\quad b = q + llcp' \ q \ (q + k) \\
llcp' \ j \ k &| j == n \vee k == n = 0 \\
&| xa ! j == xa ! k = 1 + llcp' \ (j + 1) \ (k + 1) \\
&| \text{otherwise} = 0
\end{aligned}$$


```

Рис. 15.2: окончательный вид программы

Уточнение данных

К сожалению, операции *snoc*, (!!) и *drop* выполняются не за константное время. Остаток этого обсуждения это просто уточнение данных, которое должно привести к тому, что эти операции станут операциями с константным временем.

Давайте сначала поработаем с *drop*. Идея в том, чтобы воспользоваться библиотекой *Data.Array* и заменить *llcp* другой версией, в которой индексируется глобальный массив $xa = listArray \ (0, n - 1) \ xs$, где $n = length \ xs$:

$$\begin{aligned}
llcp' \ j \ k \mid j == n \vee k == n &= 0 \\
\mid xa ! j == xa ! k &= 1 + llcp' \ (j + 1) \ (k + 1) \\
\mid \text{otherwise} &= 0
\end{aligned}$$

Это означает, что определения a и b из функции *step* можно заменить на:

$$\begin{aligned}
a &= llcp' \ 0 \ k \\
b &= q + llcp' \ q \ (q + k)
\end{aligned}$$

Остается решить вопрос с операциями *spos* и $(!!)$. Очевидным кажется решение снова воспользоваться массивом. Однако добавление элемента в конец массива является операцией с константным временем, только если всё вычисление выполняется в соответствующей монаде, а этого нам хотелось бы избежать. Другой вариант — библиотека *Data.Sequence*. Она предоставляет *spos* с константным временем, но лишь логарифмическую операцию индексирования. На практике этого достаточно, но мы обещали решение с линейным временем, поэтому нужно ещё немного поработать.

В своём решении мы воспользуемся очередью, точнее, двумя очередями. Предложенная Крисом Окасаки (Okasaki, 1995) реализация очередей предоставляет тип *Queue a* со следующими четырьмя операциями:

```
insert :: Queue a → a → Queue a
remove :: Queue a → (a, Queue a)
empty :: Queue a
elems :: Queue a → [a]
```

Функция *insert* вставляет новый элемент в конец очереди, *remove* возвращает её первый элемент и оставшиеся элементы непустой очереди, *empty* создаёт пустую очередь, а *elems* возвращает список элементов очереди.

Мы заменяем компонент *as* аргумента *step* на очередь, назвав её также *as*, и добавляем вторую очередь *qs*, которая будет представлять суффикс $drop(k - i) as$. Тогда $q = as !! (k - i)$ это первый элемент *qs*. Параметр *i* теперь не нужен, поэтому его можно убрать, заменив *p* на *h = i + p*. Применение этих преобразований выполняется непосредственно и ведёт к окончательному виду программы на рис. 15.2.

Заключительные замечания

Задача вычисления *allcp* отмечена в (Gusfield, 1997) как важный этап предварительной обработки в задаче сопоставления строк. В указанной работе она называется «Z-алгоритмом». Та же задача под именем «таблица префиксов» рассматривается в (Crochemore and Rytter, 2003). Наш подход достаточно близко соответствует подходу Гасфильда (Gusfield) за исключением свойства (15.1), выявленного нами в качестве ключевого для *llcp* свойства, запускающего всю работу. Мы также добавили очереди для эффективного вычисления операций *spos* и $(+)$.

Литература

- Crochemore, M. and Rytter, W. (2003). *Jewels of Stringology*. Hong Kong: World Scientific.
- Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge, UK: Cambridge University Press. [Имеется русский перевод: Гасфилд Д. Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, БХВ-Петербург. 2003.]
- Okasaki, C. (1995). Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5 (4), 583–92.

16

Алгоритм Бойера—Мура

Введение

Задача поиска подстроки состоит в отыскании всех вхождений одной непустой строки, называемой *образцом*, в другой строке, именуемой *текстом*. Спецификация задачи выглядит так:

```
matches    :: Eq a => [a] -> [a] -> [Int]
matches ws = map length . filter (endswith ws) . inits
```

Функция *inits* возвращает список префиксов текста в порядке увеличения их длины. Выражение *endswith ws xs* проверяет, является ли *ws* суффиксом *xs*. Значением *matches ws xs* становится список целых чисел *p*, таких что *ws* это суффикс *take p xs*. Например,

```
matches "abcab" "ababcabcab" = [7, 10]
```

Другими словами, *matches ws xs* вернёт список целых *p*, таких что *ws* расположено в строке *xs* и заканчивается в позиции с номером *p* (считая позиции с единицы).

Функция *matches* полиморфна, поэтому любой алгоритм решения нашей задачи для получения информации об элементах обоих списков может использовать лишь проверку на равенство ($==$) :: *a* → *a* → *Bool*. Полиморфный поиск подстроки исключает любые алгоритмы, зависящие от конечности *a*. Считая проверку на равенство выполнимой за константное время, сложность по времени *matches ws xs* составляет $\Theta(mn)$ операций в худшем случае, где *m* = *length ws* и *n* = *length xs*. Наша

цель в данной жемчужине вывести знаменитый алгоритм поиска подстроки Бойера–Мура (БМ), который снижает сложность до $\Theta(m + n)$ операций. В следующей жемчужине мы получим не менее известный алгоритм Кнута–Морриса–Пратта (КМП) для решения той же задачи с такой же сложностью. Секрет в том, чтобы просто применить подходящие законы, повышающие производительность, которые диктуются формой рассматриваемого выражения.

Лемма о просмотре

Для поиска подстроки, а на самом деле — для любой задачи, привлекающей функцию *inits*, наиболее важный закон известен как лемма о просмотре и выглядит так:

$$\text{map} (\text{foldl } op \ e) \cdot \text{inits} = \text{scanl } op \ e$$

Выражение слева от знака равенства выполняется на списке длины n за $\Theta(n^2)$ вычислений *op*, в то время как эквивалентное выражение, записанное в терминах стандартной функции Haskell под названием *scanl*, требует всего $\Theta(n)$ этих операций.

Хотя в определении *matches* присутствует *map*, там есть также и *filter*, так что первым делом в преобразовании *matches* используем закон:

$$\text{map } f \cdot \text{filter } p = \text{map } \text{fst} \cdot \text{filter } \text{snd} \cdot \text{map} (\text{fork} (f, p)) \quad (16.1)$$

где $\text{fork} (f, p) x = (f x, p x)$. Это правило используется просто для того, чтобы перенести *map* в позицию непосредственно следом за *inits*, в качестве подготовки к применению леммы о просмотре. Использование (16.1) приводит к такому определению *matches*:

$$\begin{aligned} & \text{matches } ws \\ &= \text{map } \text{fst} \cdot \text{filter } \text{snd} \cdot \text{map} (\text{fork} (\text{length}, \text{endswith } ws)) \cdot \text{inits} \end{aligned}$$

На очереди следующий вопрос: можно ли $\text{fork} (\text{length}, \text{endswith } ws)$ превратить в экземпляр свёртки *foldl*? Разумеется, $\text{length} = \text{foldl } \text{count } 0$, где $\text{count } n \ x = n + 1$. Предположим на секунду, что можно найти *e* и *op*, каждое из которых зависит от *ws*, такие что:

$$\text{endswith } ws = \text{foldl } op \ e \quad (16.2)$$

Тогда мы получим возможность применить другой стандартный закон: правило кортежей для *foldl*. Это правило гласит:

$$\text{fork}(\text{foldl } op1 \ e1, \text{foldl } op2 \ e2) = \text{foldl } op(e1, e2)$$

где $op(a, b) x = (op1 a x, op2 b x)$. Использование правила кортежей приводит к следующему:

$$\begin{aligned}\text{fork}(\text{length}, \text{endswith } ws) &= \text{foldl step}(0, e) \\ \text{step}(n, x) y &= (n + 1, op x y)\end{aligned}$$

Наконец, можно применить и лемму о просмотре, чтобы получить:

$$\text{matches } ws = \text{map } fst \cdot \text{filter } snd \cdot \text{scanl step}(0, e)$$

Если op требует константного или хотя бы амортизированного константного времени, то столько же понадобится и для выполнения $step$, в результате мы получим линейную по времени программу. По сути, отсюда и происходят все эффективные полиморфные алгоритмы поиска подстроки.

Проблема в том, что никакие op и e не удовлетворяют (16.2). Функция $\text{endswith } ws$ возвращает одно булевское значение, что не даёт достаточно сведений для записи его через $foldl$. Следующей хорошей мыслью было бы выразить $\text{endswith } ws$ композицией:

$$\text{endswith } ws = p \cdot \text{foldl } op \ e \tag{16.3}$$

Форма (16.3) подсказана желанием применить лемму о просмотре и ничем больше. Вместо (16.1) мы можем использовать небольшое обобщение:

$$\text{map } f \cdot \text{filter } (p \cdot g) = \text{map } fst \cdot \text{filter } (p \cdot snd) \cdot \text{map } (\text{fork } (f, g)) \tag{16.4}$$

Тогда получится:

$$\text{matches } ws = \text{map } fst \cdot \text{filter } (p \cdot snd) \cdot \text{scanl step}(0, e)$$

Если p и op требуют амортизированного константного времени, matches займёт линейное время.

Остаётся найти p , op и e , которые удовлетворяют (16.3). Однако мы до сих пор не определили endswith формально. Вот два подходящих определения:

$$\begin{aligned}\text{endswith } ws \ xs &= \text{reverse } ws \sqsubseteq \text{reverse } xs \\ \text{endswith } ws \ xs &= ws \in \text{tails } xs\end{aligned}$$

В первом определении $ws \sqsubseteq vs$ означает, что ws это префикс vs . Очевидно, что ws это суффикс xs тогда и только тогда, когда обращённая задом

наперёд ws является префиксом обращённой xs . Отношение префикса реализовать проще, чем случай суффикса:

$$\begin{array}{lll} [] \sqsubseteq vs & = & \text{True} \\ (u : us) \sqsubseteq [] & = & \text{False} \\ (u : us) \sqsubseteq (v : vs) & = & (u == v \wedge us \sqsubseteq vs) \end{array}$$

Хотя оба выражения для *endswith* определяют одну и ту же функцию, они имеют различную форму. И, поскольку именно форма, а не функция, диктует направление рассуждений, мы стоим на распутьи. Как станет видно позднее, первый путь приведёт к (БМ), а второй к (КМП). До конца этой жемчужины мы пойдём по первому пути. В следующей исследуем второй.

Алгоритм Бойера—Мура

Первое определение *endswith* может быть переписано в виде композиции:

$$\text{endswith } ws = (\text{reverse } ws \sqsubseteq) \cdot \text{reverse}$$

Таким образом, обращение к (16.4) даёт:

$$\begin{aligned} \text{matches } ws \\ = \text{map } \text{fst} \cdot \text{filter } ((sw \sqsubseteq) \cdot \text{snd}) \cdot \text{map } (\text{fork } (\text{length}, \text{reverse})) \cdot \text{inits} \\ \text{where } sw = \text{reverse } ws \end{aligned}$$

Но $\text{reverse} = \text{foldl } (\text{flip } (:)) []$, и мы снова можем воспользоваться правилом кортежей для *foldl*, а вслед за ним леммой о просмотре, получив:

$$\begin{aligned} \text{matches } ws \\ = \text{map } \text{fst} \cdot \text{filter } ((sw \sqsubseteq) \cdot \text{snd}) \cdot \text{scanl } \text{step } (0, []) \\ \text{where } sw = \text{reverse } ws \\ \text{step } (n, sx) x = (n + 1, x : sx) \end{aligned}$$

Это основная форма БМ-алгоритма. Применение *scanl* порождает последовательно расположенные «окна» в тексте с указанием их позиций. Каждое окно содержит некоторый начальный отрезок текста, обращённый задом наперёд, а последующие окна отличаются на одну позицию — говорят, что имеет место «сдвиг» на один уровень. Термины «окно» и «сдвиг» взяты из работы (Lecroq, 2003), которая содержит очень хорошо написанное введение в алгоритмы поиска подстрок. Каждое из упомянутых окон обрабатывается путём сравнения с образцом ws справа налево.

Выполнение сдвигов

Как он был сформулирован выше, БМ-алгоритм всё ещё требует в худшем случае $\Omega(mn)$ операций, потому что проверка ($sw \sqsubseteq$) может занять $\Omega(m)$ (всё далее мы обозначаем $m = \text{length } ws$ и предполагаем, что $m \neq 0$). Один пример худшего случая таков: шаблон представляет из себя m повторений одного значения, а текст это список из n экземпляров того же самого значения. Это положение можно улучшить, если понять, нельзя ли сдвинуться сразу на несколько окон из-за того, что ни одно из них не может давать вхождение искомой подстроки. Оказывается, что такой сдвиг зависит от размера участка подстроки, который встретился в текущем окне.

Пусть $llcp sw sx$ обозначает длину самого длинного общего префикса sw и sx . Мы встречали эту функцию в предыдущей жемчужине. Очевидно, $sw \sqsubseteq sx$, если и только если $m = llcpsw sx$. Полагая $i = llcpsw sx$ для текущего окна (n, sx) , можно ли получить нижнюю границу для позиции $n + k$ следующего окна, которое могло бы дать совпадение с искомой строкой? Конечно, $0 < k \leq m$, иначе можно пропустить одно вхождение подстроки целиком. Предположим, что следующее окно имеет вид $(n + k, ys \sqplus sx)$, где $k = \text{length } ys$. Если в этом окне имеется совпадение, то есть $sw \sqsubseteq ys \sqplus sx$, тогда $\text{take } k sw = ys$ и $\text{drop } k sw \sqsubseteq sx$.

Используя эти данные и полагая $i = llcp sw sx$, можно показать:

$$\text{llcp } sw (\text{drop } k sw) = i \min(m - k) \quad (16.5)$$

Во-первых, предположим, что $i < m - k$. Тогда

$$\begin{aligned} & \text{take } i (\text{drop } k sw) \\ &= \{ \text{поскольку } \text{drop } k sw \sqsubseteq sx \text{ влечёт } \text{drop } k sw = \text{take } (m - k) sx \} \\ & \text{take } i (\text{take } (m - k) sx) \\ &= \{ \text{поскольку } i \leq m - k \} \\ & \text{take } i sx \\ &= \{ \text{поскольку } i = llcp sw sx \} \\ & \text{take } i sw \end{aligned}$$

Аналогичное рассуждение даёт:

$$\text{take } (i + 1) (\text{drop } k sw) \neq \text{take } (i + 1) sw$$

Другими словами, если $i < m - k$, то $\text{llcp } sw (\text{drop } k sw) = i$. В противном случае, $i \geq m - k$, мы рассуждаем так:

$$\begin{aligned}
 & drop k sw \\
 = & \{ \text{поскольку } length (drop k sw) = m - k \leq i \} \\
 & take i (drop k sw) \\
 \sqsubseteq & \{ \text{поскольку } drop k sw \sqsubseteq sx \} \\
 & take i sx \\
 = & \{ \text{поскольку } i = llcp sw sx \} \\
 & take i sw \\
 \sqsubseteq & \{ \text{определение } \sqsubseteq \} \\
 & sw
 \end{aligned}$$

Но $drop k sw \sqsubseteq sw \equiv llcp sw (drop k sw) = m - k$, что и даёт искомое (16.5).

Теперь, для некоторого i в промежутке $0 \leq i \leq m$, положим k равным наименьшему положительному значению в интервале $1 \leq k \leq m$, которое удовлетворяет (16.5). В случае $m \neq 0$ вариант $k = m$ подходит для (16.5), если не подошло ни одно меньшее значение. Следовательно, мы можем отбросить следующие $k - 1$ окно, не упустив ни одного вхождения подстроки. Значение k специфицируется так: $k = shift sw i$, где

$$shift sw i = head [k \mid k \leftarrow [1..m], llcp sw (drop k sw) == i \min (m - k)]$$

Это не лучший способ посчитать $shift sw i$, так как вычисление может потребовать $\Omega(m^2)$ операций в худшем случае. В следующем разделе мы покажем, как вычислить $map (shift sw) [0..m]$ за $O(m)$ шагов.

Подытоживая, после совпадения i символов в текущем окне (n, sx) , следующие $shift sw i$ окон могут быть пропущены без ущерба для результата. Это означает, что *matches* можно переопределить так:

$$\begin{aligned}
 matches ws &= test \cdot scanl step (0, []) \\
 \text{where} \\
 test [] &= [] \\
 test ((n, sx) : nxs) &= \text{if } i == m \\
 &\quad \text{then } n : test (drop (k - 1) nxs) \\
 &\quad \text{else } test (drop (k - 1) nxs) \\
 &\quad \text{where } i = llcp sw sx \\
 &\quad k = shift sw i \\
 (sw, m) &= (reverse ws, length ws)
 \end{aligned}$$

Отметим, что две версии *matches* эквивалентны только в случае $m \neq 0$.

Последний штрих

Можно улучшить полученный алгоритм ещё немного. Как и прежде, пусть $i = llcp sw sx$ и $k = shift sw i$. Предположим также, что $m - k \leq i$, тогда $llcp sw (drop k sw) = m - k$. Это означает, что $drop k sw$ является префиксом sw . Так как $m - k \leq i$, то $llcp (drop k sw) sx = m - k$.

Следующее окно для проверки это $(n + k, ys + sx)$, где $length ys = k$. Проведём следующие рассуждения:

$$\begin{aligned} & llcp sw (ys + sx) \\ &= \{ \text{полагая } i' = llcp sw ys, \text{ так что } i' \leq k \} \\ &\quad \text{if } i' == k \text{ then } k + llcp (drop k sw) sx \text{ else } i' \\ &= \{ \text{как отмечено выше, поскольку } llcp (drop k sw) sx = m - k, \\ &\quad \text{если } m - k \leq i \} \\ &\quad \text{if } i' == k \text{ then } m \text{ else } i' \end{aligned}$$

Отсюда с учётом $m - k \leq i$ размер наиболее длинного общего префикса sw и текста в следующем окне может быть посчитан сравнением только первых k элементов. Если $m - k > i$, то экономия исчезает, и следующее окно может потребовать до m сравнений.

Реализовать эту идею можно, снабдив *test* дополнительным параметром j , который укажет длину участка следующего окна для проверки. Сделав это последнее уточнение, получим программу на рис. 16.1, которой не хватает лишь определений *llcp* и *shift*. Эта программа воспроизводит версию БМ-алгоритма из (Galil, 1979). Если пренебречь временем вычисления *shift*, временная сложность *matches* составит $O(m + n)$ для текста длины n . За доказательством этого (достаточно нетривиального) факта можно обратиться к теореме 3.2.3 (Gusfield, 1997).

Вычисление сдвигов

Определение *shift sw*, данное в предыдущем разделе, приводит к кубическому по времени алгоритму для вычисления

$$shifts sw = map (shift sw) [0..m]$$

так как выполнение *shift sw i* может занять квадратичное время, а в качестве i выступают $m + 1$ различных значений. Если бы удалось вычислить *shifts sw* за линейное время и сохранить результат в массив a , то замена *shift sw i* на $a[i]$ дала бы линейный по времени алгоритм *matches*. Цель

```


$$\begin{aligned}
\text{matches } ws &= \text{test } m \cdot \text{scanl step} (0, []) \\
\text{where} \\
\text{test } j [] &= [] \\
\text{test } j ((n, sx) : nxs) &= \begin{cases} i == m &= n : \text{test } k (\text{drop } (k - 1) nxs) \\ m - k \leq i &= \text{test } k (\text{drop } (k - 1) nxs) \\ \text{otherwise} &= \text{test } m (\text{drop } (k - 1) nxs) \\ \text{where } i' = llcp sw (\text{take } j sx) & \\ i &= \text{if } i' == j \text{ then } m \text{ else } i' \\ k &= \text{shift } sw i \end{cases} \\
(sw, m) &= (\text{reverse } ws, \text{length } ws)
\end{aligned}$$


```

Рис. 16.1: итоговая программа

данного раздела — продемонстрировать вычисление $shifts sw$ за линейное время. Возможно, это самый тонкий момент БМ-алгоритма.

Прежде всего, обозначим для краткости $f(k) = llcp sw (\text{drop } k sw)$. Заметим, что $f(m) = 0$ и $f(k) \leq m - k$. Проведём для начала рассуждения в предположении $0 \leq i \leq m$:

$$\begin{aligned}
&\text{shift } sw i \\
&= \{ \text{определение} \} \\
&\text{minimum } [k \mid k \leftarrow [1..m], f(k) == i \min (m - k)] \\
&= \{ \text{разбор вариантов для min} \} \\
&\text{minimum } ([k \mid k \leftarrow [1..m - i], f(k) == i] + \\
&\quad [k \mid k \leftarrow [m - i + 1..m], f(k) + k == m]) \\
&= \{ \text{поскольку } f(k) = i \Rightarrow k \leq m - i \} \\
&\text{minimum } ([k \mid k \leftarrow [1..m], f(k) == i] + \\
&\quad [k \mid k \leftarrow [m - i + 1..m], f(k) + k == m])
\end{aligned}$$

Далее мы обратимся к библиотеке *Data.Array* языка Haskell и, в частности, к функции *accumArray*. Впервые эта функция появилась в жемчужине 1. Следующий факт об *accumArray* получается сразу из определения:

$$(accumArray op e (0, m) vks)! i = foldl op e [k \mid (v, k) \leftarrow vks, v == i]$$

Это справедливо для всех i в промежутке $0 \leq i \leq m$ при условии, что $\text{map fst vks} \subseteq [0..m]$. Последняя оговорка необходима, так как *accumArray* не определена, если индекс выходит за пределы допустимого интервала. В частности, если

$$\begin{aligned} a &= \text{accumArray min } m (0, m) \text{ vks} \\ \text{vks} &= [(f(k), k) \mid k \leftarrow [1..m]] \end{aligned}$$

имеем

$$a ! i = \text{minimum} ([k \mid k \leftarrow [1..m], f(k) == i] \uplus [m])$$

для $0 \leq i \leq m$. Это относится к первой части определения $\text{shift sw } i$. Теперь надо разобраться со второй. Идея в том, чтобы заменить a на

$$a = \text{accumArray min } m (0, m) (vks \uplus vks')$$

где vks' это любая подходящая перестановка элементов списка

$$[(i, \text{minimum} [k \mid k \leftarrow [m - i + 1..m], f(k) + k == m]) \mid i \leftarrow [1..m]]$$

Тогда $\text{shift sw } i = a ! i$.

Утверждается, что следующее определение vks' , которое вычисляет список, упомянутый выше, в обратном порядке, подходит для наших целей:

$$\begin{aligned} vks' &= \text{zip} [m, m - 1..1] (\text{foldr } op [] vks) \\ vks &= [(f(k), k) \mid k \leftarrow [1..m]] \\ op (v, k) ks &= \text{if } v + k == m \text{ then } k : ks \text{ else head } ks : ks \end{aligned}$$

Отметим, что $op (f(m), m) [] = [m]$, потому как $f(m) = 0$, и, значит, $f(m) + m = m$. К примеру, если $xs = \text{foldr } op [] vks$, имеем

$$\begin{array}{cccccccccc} f & 2 & 4 & 0 & 5 & 2 & 3 & 0 & 2 & 0 \\ k & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ xs & 4 & 4 & 4 & 4 & 6 & 6 & 9 & 9 & 9 \end{array}$$

Элемент в i -ой позиции xs (считая с нуля) это наименьшее k , такое что $k > i$ и $f(k) + k = m$. В vks' индекс $m - i$ стоит в паре с $xs !! i$; равносильное утверждение заключается в том, что i составляет пару с $xs !! (m - i)$, а именно это и требовалось.

Последним шагом вернёмся к определению $allcp$ из прошлой жемчужины:

$$allcp xs = [llcp xs (drop k xs) \mid k \leftarrow [0..length xs - 1]]$$

Там было показано, как вычислить $allcp$ за линейное время. Для нынешних целей нам нужен вариант $allcp xs$, в котором исключён первый элемент, а в конце добавлен элемент $llcp xs []$. Это дополнительное значение равно нулю, так что

$$\text{allcp}' \ xs = \text{tail} (\text{allcp} \ xs) ++ [0]$$

Наконец, проведём следующие рассуждения:

$$\begin{aligned} & [(f(k), k) \mid k \leftarrow [1..m]] \\ = & \{ \text{определение } f \} \\ & [(\text{llcp} \ sw (\text{drop} \ k \ sw), k) \mid k \leftarrow [1..m]] \\ = & \{ \text{определение } \text{zip} \} \\ & \text{zip} [\text{llcp} \ sw (\text{drop} \ k \ sw) \mid k \leftarrow [1..m]] [1..m] \\ = & \{ \text{определение } \text{allcp}' \} \\ & \text{zip} (\text{allcp}' \ sw) [1..m] \end{aligned}$$

Собирая всё воедино, получаем:

$$\begin{aligned} a &= \text{accumArray min m} (0, m) (vks ++ vks') \\ \text{where} \\ m &= \text{length} \ sw \\ vks &= \text{zip} (\text{allcp}' \ sw) [1..m] \\ vks' &= \text{zip} [m, m - 1..1] (\text{foldr} \ op [] vks) \\ op(v, k) \ ks &= \text{if } v + k == m \text{ then } k : ks \text{ else head } ks : ks \end{aligned}$$

Заменив $\text{shift} \ sw \ i$ на $a!i$ в коде на рис. 16.1, получим линейный по времени алгоритм для *matches*.

Заключительные замечания

БМ-алгоритм впервые был описан в (Boyer and Moore, 1977), обратившись за дальнейшими разъяснениями и обсуждением метода можно к (Cormen *et al.*, 2001), (Crochemore and Rytter, 2003) и (Gusfield, 1997). Чаще всего алгоритм объясняется в терминах двух правил, правила столбца и правила безопасного суффикса, ни одно из которых не появилось в явном виде в рассуждениях выше. Наш вывод БМ-алгоритма, по меньшей мере, в его основной форме, был простым упражнением в символьных преобразованиях с применением подходящих законов, повышающих производительность и диктуемых только формой возникавших выражений. Важнейшими из последних стали лемма о просмотре и правило кортежей для *foldl*. Более того, ключевая идея БМ-алгоритма, а именно, сопоставление образца и текста в порядке справа налево, появилась просто как следствие одного удобного определения *endswith*. Последующие оптимизации зависели в большей степени от составных частей выражения,

нежели от его формы, но этого следует ждать в любом алгоритме, содержащем достаточно не очевидные идеи.

Литература

- Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM* **20**, 762–72.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001). *Introduction to Algorithms*, second edition. Cambridge, MA: The MIT Press. [Имеется русский перевод: Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: Построение и анализ. 2-е изд. М.: Вильямс. 2005.]
- Crochemore, M. and Rytter, W. (2003). *Jewels of Stringology*. Hong Kong: World Scientific.
- Galil, Z. (1979). On improving the worst cast of the Boyer–Moore string matching algorithm. *Communications of the ACM* **22** (9), 505–8.
- Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge, UK: Cambridge University Press. [Имеется русский перевод: Гасфилд Д. Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, БХВ-Петербург. 2003.]
- Lecroq, T. (2003). Experimental results on string matching algorithms. *Software – Practice and Experience* **25** (7), 727–65.

17

Алгоритм Кнута—Морриса—Пратта

Введение

В этой жемчужине мы продолжим разбирать задачу поиска подстроки и на распутьи двинемся по дороге, обозначенной таким определением *endswith*:

endswith ws xs = ws ∈ tails xs

Этот путь приведёт нас к КМП-алгоритму. Напомним, что цель в том, чтобы найти функции *p* и *op* и значение *e*, с которыми имеет место равенство *endswith ws = p · foldl op e*. В этом случае получим:

```
matches      :: Eq a ⇒ [a] → [a] → [Int]
matches ws   = map fst · filter (p · snd) · scanl step (0, e)
step (n, x) y = (n + 1, op x y)
```

Значением *matches ws xs* является список целых чисел *n*, таких что вхождение шаблона *ws* в строку *xs* заканчивается в позиции *n*. Если *p* и *op* требуют константного или хотя бы амортизированного константного времени, вычисление *matches* займёт $\Theta(m + n)$ операций для шаблона длины *m* и текста длины *n*.

Первые шаги

Вот один способ записи *endswith ws* в виде композиции функций:

$$\text{endswith } ws = \text{not} \cdot \text{null} \cdot \text{filter } (== ws) \cdot \text{tails}$$

Но $\text{filter } (== ws) \cdot \text{tails}$ не может быть представлен как экземпляр свёртки foldl , потому что он возвращает пустой список или $[ws]$, а этого недостаточно для индуктивного определения функции. Более перспективным выглядит $\text{filter } (\sqsubseteq ws) \cdot \text{tails}$. Если применить его к xs , получится список всех суффиксов xs , которые одновременно составляют префикс xs и упорядочены по убыванию длины. Первым элементом этого списка будет ws в том и только том случае, когда верно $\text{endswith } ws xs$. Таким образом,

$$\text{endswith } ws = (== ws) \cdot \text{head} \cdot \text{filter } (\sqsubseteq ws) \cdot \text{tails}$$

Конечно, проверка $(== ws)$ не выполнима за константное время. Эта трудность решается обобщением $\text{filter } (\sqsubseteq ws) \cdot \text{tails}$ до функции split , которую определим так:

$$\text{split } ws xs = \text{head } [(us, ws \downarrow us) \mid us \leftarrow \text{tails } xs, us \sqsubseteq ws]$$

Операция \downarrow это по определению $(us + vs) \downarrow us = vs$. Следовательно, $\text{split } ws xs$ разбивает ws на две части us и vs так, что $us + vs = ws$. Значение us представляет собой длиннейший суффикс xs , который одновременно является префиксом ws . Например,

$$\text{split "endnote" "append" } = ("end", "note")$$

Теперь $\text{endswith } ws = \text{null} \cdot \text{snd} \cdot \text{split } ws$. Остаётся найти op и e , такие что $\text{split } ws = \text{foldl } op e$. Иначе, e и op должны удовлетворять определению:

$$\begin{aligned} \text{split } ws [] &= e \\ \text{split } ws (xs + [x]) &= op (\text{split } ws xs) x \end{aligned}$$

Имеем $\text{split } ws [] = ([], ws)$, откуда получается значение e , поэтому остаётся отыскать op .

Решающим оказывается следующее наблюдение:

$$\text{split } ws xs = (us, vs) \Rightarrow \text{split } ws (xs + [x]) = \text{split } ws (us + [x])$$

Иными словами, длиннейший суффикс $xs + [x]$, который является одновременно префиксом ws , это в точности суффикс $us + [x]$. Он не может быть более длинным суффиксом, иначе это значило бы, что существует более длинный суффикс xs , чем us , который является префиксом ws , что противоречит определению us как длиннейшего такого суффикса.

Чтобы отыскать op , выразим для начала split рекурсивно:

$$\text{split ws xs} = \text{if } xs \sqsubseteq ws \text{ then } (xs, ws \downarrow xs) \text{ else } \text{split ws}(\text{tail } xs)$$

Теперь, полагая $\text{split ws xs} = (us, vs)$, так что $ws = us ++ vs$, проведём следующие рассуждения:

$$\begin{aligned} & \text{split ws}(xs ++ [x]) \\ &= \{ \text{наблюдение выше} \} \\ & \text{split ws}(us ++ [x]) \\ &= \{ \text{рекурсивное определение split} \} \\ & \text{if } us ++ [x] \sqsubseteq ws \text{ then } (us ++ [x], ws \downarrow (us ++ [x])) \\ & \quad \text{else } \text{split ws}(\text{tail } (us ++ [x])) \\ &= \{ \text{используя } ws = us ++ vs \text{ и определения } \sqsubseteq \text{ и } \downarrow \} \\ & \text{if } [x] \sqsubseteq vs \text{ then } (us ++ [x], \text{tail } vs) \\ & \quad \text{else } \text{split ws}(\text{tail } (us ++ [x])) \\ &= \{ \text{разбор вариантов для } us \} \\ & \text{if } [x] \sqsubseteq vs \text{ then } (us ++ [x], \text{tail } vs) \\ & \text{else if } \text{null } us \text{ then } ([], ws) \\ & \quad \text{else } \text{split ws}(\text{tail } us ++ [x]) \end{aligned}$$

Приведённое вычисление даёт искомое определение op :

$$\begin{aligned} op(us, vs) x \mid [x] \sqsubseteq vs &= (us ++ [x], \text{tail } vs) \\ \mid \text{null } us &= ([], ws) \\ \mid \text{otherwise} &= op(\text{split ws}(\text{tail } us)) x \end{aligned}$$

Подводя промежуточные итоги, запишем:

$$\begin{aligned} matches ws &= map fst \cdot filter (\text{null} \cdot snd \cdot snd) \cdot \\ &\quad scanl step (0, ([], ws)) \\ step(n, (us, vs)) x &= (n + 1, op(us, vs) x) \end{aligned}$$

Это основная форма КМП-алгоритма: на каждом шаге поддерживается актуальное разбиение (us, vs) шаблона ws , в котором us составляет длиннейший префикс ws , совпадающий с некоторым суффиксом текущей части текста. Позиции, где $vs = []$, содержат целое вхождение шаблона и записываются в результирующий список.

Проблема с op в том, что она неэффективна: третья часть требует вычисления $\text{split ws}(\text{tail } us)$, которое в свою очередь может повлечь неоднократный пересчёт $\text{split ws} zs$ для произвольной подстроки zs шаблона ws . Очевидно, op делает лишнюю работу, и нам следует найти что-то лучшее.

Уточнение данных

Один способ увеличения эффективности состоит в попытке изменить представление первого аргумента op , а именно текущего разбиения (us , vs) шаблона ws . Более точно, предположим, что функции abs и rep имеют типы:

$$\begin{aligned} abs &:: Rep([a], [a]) \rightarrow ([a], [a]) \\ rep &:: ([a], [a]) \rightarrow Rep([a], [a]) \end{aligned}$$

для некоторого типа данных Rep . Функция rep это представляющая функция, а abs — абстрагирующая. Это стандартная терминология при уточнении данных. Требуется также $abs \cdot rep = id$, то есть abs является левой обратной к rep . Это условие говорит, что абстрагированное значение может быть восстановлено по любому его представлению. Обратное равенство $rep \cdot abs = id$ выполнено только в том случае, если изменение представления является биекцией, что, как правило, неверно при уточнении данных.

Если удастся найти необходимые составляющие для

$$foldl op ([] , ws) = abs \cdot foldl op' (rep ([] , ws)) \quad (17.1)$$

а abs и op' будут занимать константное время, то получится переопределить $matches$:

$$\begin{aligned} matches\ ws &= map\ fst \cdot filter\ (null \cdot snd \cdot abs \cdot snd) \cdot \\ &\quad scanl\ step\ (0, rep\ ([] , ws)) \\ step\ (n, r)\ x &= (n + 1, op'\ r\ x) \end{aligned}$$

Чтобы найти abs , op' и rep , удовлетворяющие (17.1), обратимся к закону слияния для $foldl$. Он гласит, что $f \cdot foldl g a = foldl h b$, если выполнены три условия: (а) f это строгая функция; (б) $f\ a = b$; (в) $f\ (g\ x\ y) = h\ (f\ y)\ x$ для всех x и y . Первое условие не требуется, если мы применяем слияние лишь для конечных списков. Хитрость в том, что в данном случае нужно применить правило в противоположном направлении, выполнить расщепление свёртки на две части.

Второе условие получается сразу: $abs\ (rep\ ([] , ws)) = ([] , ws)$. Кроме того, имеется очевидное определение op' , которое удовлетворяет третьему условию, а именно:

$$op'\ r = rep \cdot op\ (abs\ r) \quad (17.2)$$

В этом случае имеем:

$$\text{abs} (\text{op}' r x) = \text{abs} (\text{rep} (\text{op} (\text{abs} r) x)) = \text{op} (\text{abs} r) x$$

Подставляя определение op в (17.2), получаем:

$$\begin{aligned} \text{op}' r x \mid [x] \sqsubseteq vs &= \text{rep} (\text{us} ++ [x], \text{tail } vs) \\ \mid \text{null } us &= \text{rep} ([], ws) \\ \mid \text{otherwise} &= \text{op}' (\text{rep} (\text{split } ws (\text{tail } us))) x \\ \quad \text{where } (\text{us}, vs) &= \text{abs } r \end{aligned}$$

Остаётся выбрать Rep и две функции, abs и rep .

Деревья

В функциональном программировании практически любое эффективное представление так или иначе связано с деревьями некоторого вида, и наш случай не исключение. Определим

$$\text{data } \text{Rep } a = \text{Null} \mid \text{Node } a (\text{Rep } a) (\text{Rep } a)$$

То есть Rep это бинарное дерево. Функция abs определяется так:

$$\text{abs} (\text{Node} (us, vs) \ell r) = (us, vs) \tag{17.3}$$

и, очевидно, требует константного времени. Определение rep выглядит так:

$$\text{rep} (us, vs) = \text{Node} (us, vs) (\text{left } us \text{ vs}) (\text{right } us \text{ vs}) \tag{17.4}$$

где

$$\begin{aligned} \text{left } [] \text{ vs} &= \text{Null} \\ \text{left } (u : us) \text{ vs} &= \text{rep} (\text{split } ws us) \\ \text{right } us [] &= \text{Null} \\ \text{right } us (v : vs) &= \text{rep} (us ++ [v], vs) \end{aligned}$$

Причина именно такого выбора rep в том, что op' в этом случае принимает достаточно простую форму:

$$\begin{aligned} \text{op}' (\text{Node} (us, vs) \ell r) x \mid [x] \sqsubseteq vs &= r \\ \mid \text{null } us &= \text{root} \\ \mid \text{otherwise} &= \text{op}' \ell x \end{aligned}$$

где $\text{root} = \text{rep} ([], ws)$. Например, первое равенство подтверждается следующими рассуждениями:

$$\begin{aligned}
 & op' (\text{Node}(us, vs) \ell r) x \\
 = & \quad \{ \text{определение } op' \text{ в случае } [x] \sqsubseteq vs \} \\
 & rep (us ++ [x], tail vs) \\
 = & \quad \{ \text{определение } right \text{ и } x = head vs \} \\
 & right us vs \\
 = & \quad \{ \text{определение } rep \} \\
 & r
 \end{aligned}$$

Остальные равенства доказываются аналогично. Если мы, кроме этого, положим $op' Null x = root$, то op' примет ещё более простую форму:

$$\begin{array}{lll}
 op' Null x & = & root \\
 op' (\text{Node}(us, vs) \ell r) x & | & [x] \sqsubseteq vs = r \\
 & | & \text{otherwise} = op' \ell x
 \end{array}$$

Видно, что op' требует не константного, но амортизированного константного времени. Дерево $root$ имеет высоту m , равную длине шаблона; переход в правую ветвь дерева уменьшает высоту ровно на единицу, в то время как выбор левой ветви даёт прирост высоты, самое меньшее, на один. Стандартный аргумент при доказательстве амортизированного времени подтверждает, что выполнение $\text{foldl } op' root$ на списке длины n требует не более $2m + n$ вызовов op' .

Остаётся только показать, как эффективно вычислить rep . И здесь на сцене появляется последняя стандартная техника преобразования программ: использование аккумулирующего параметра. Идея заключается в том, чтобы определить более общую версию rep , назовём её $grep$, такую что:

$$rep (us, vs) = grep (left us vs) (us, vs)$$

а затем вывести явное определение $grep$. Из (17.4) следует:

$$grep \ell (us, vs) = Node (us, vs) \ell (right us vs)$$

Теперь, по определению $right$, имеем: $right us [] = Null$ и

$$\begin{aligned}
 right us (v : vs) &= rep (us ++ [v], vs) \\
 &= grep (left (us ++ [v]) vs) (us ++ [v], vs)
 \end{aligned}$$

Чтобы упростить $left$, надо провести разбор вариантов для us . В случае $us = []$ справедливы рассуждения:

$$\begin{aligned}
 & left ([] ++ [v]) vs \\
 = & \quad \{ \text{определение } left \} \\
 & rep (split ws []) \\
 = & \quad \{ \text{определение } split \} \\
 & rep ([], ws) \\
 = & \quad \{ \text{определение } root \} \\
 & root
 \end{aligned}$$

В случае шага индукции $u : us$ рассуждения таковы:

$$\begin{aligned}
 & left (u : us ++ [v]) vs \\
 = & \quad \{ \text{определение } left \} \\
 & rep (split ws (us ++ [v])) \\
 = & \quad \{ \text{определение } split \} \\
 & rep (op (split ws us) v) \\
 = & \quad \{ \text{определение (17.2) для } op' \} \\
 & op' (rep (split ws us)) v \\
 = & \quad \{ \text{определение } left \} \\
 & op' (left (u : us) vs) v
 \end{aligned}$$

Подытоживая это вычисление, имеем:

$$left (us ++ [v]) vs = \text{if } null us \text{ then } root \text{ else } op' (left us vs) v$$

Следовательно, *grep* может быть определена так:

$$\begin{aligned}
 grep \ell (us, []) &= Node (us, []) \ell Null \\
 grep \ell (us, v : vs) &= Node (us, v : vs) \ell \\
 &\quad (grep (op' \ell v) (us ++ [v], vs))
 \end{aligned}$$

Соберём, наконец, все части воедино. Имеем:

$$matches ws = map fst \cdot filter (ok \cdot snd) \cdot scanl step (0, root)$$

где

$$\begin{aligned}
 ok (Node (us, vs) \ell r) &= null vs \\
 step (n, t) x &= (n + 1, op t x) \\
 root &= grep Null ([], ws)
 \end{aligned}$$

<i>matches ws</i>	$= \text{map } \text{fst} \cdot \text{filter } (\text{ok} \cdot \text{snd}) \cdot \text{scaml step } (0, \text{root})$
where	
<i>ok (Node vs ℓ r)</i>	$= \text{null vs}$
<i>step (n, t) x</i>	$= (n + 1, \text{op } t x)$
<i>op Null x</i>	$= \text{root}$
<i>op (Node [] ℓ r) x</i>	$= \text{op } \ell x$
<i>op (Node (v : vs) ℓ r) x</i>	$= \text{if } v == x \text{ then } r \text{ else } \text{op } \ell x$
<i>root</i>	$= \text{grep Null ws}$
<i>grep ℓ []</i>	$= \text{Node [] } \ell \text{ Null}$
<i>grep ℓ (v : vs)</i>	$= \text{Node } (v : vs) \ell (\text{grep } (\text{op } \ell v) \text{ vs})$

Рис. 17.1: итоговое определение *matches*

Функция *op* (переименованная *op'*) определяется так:

$$\begin{aligned} \text{op Null } x &= \text{root} \\ \text{op } (\text{Node } (us, []) \ell r) x &= \text{op } \ell x \\ \text{op } (\text{Node } (us, v : vs) \ell r) x &= \text{if } v == x \text{ then } r \text{ else } \text{op } \ell x \end{aligned}$$

а функция *grep* так:

$$\begin{aligned} \text{grep } \ell (us, []) &= \text{Node } (us, []) \ell \text{ Null} \\ \text{grep } \ell (us, v : vs) &= \text{Node } (us, v : vs) \ell (\text{grep } (\text{op } \ell v) (us ++ [v], vs)) \end{aligned}$$

Пристальный взгляд на правые части равенств показывает, что первая компонента *us* пары *(us, vs)* не играет никакой роли, так как её значение нигде не используется. Отбрасывая *us*, получаем итоговую программу, представленную на рис. 17.1

Дерево *root* имеет цикл: левые поддеревья ссылаются «назад», на более ранние узлы в дереве либо на *Null*. Это дерево инкапсулирует функцию обработки ошибок КМП-алгоритма, представляющую собой граф с циклами. Выполнение *op* занимает амортизированное константное время в предположении, что проверка на равенство стоит константу. Время вычисления *root* составляет $\Theta(m)$ шагов, где $m = \text{length ws}$. Значит, *matches* требует $\Theta(m)$ операций, чтобы построить *root*, а затем $\Theta(n)$ операций, где *n* это длина текста, чтобы вычислить все вхождения шаблона в текст.

Представленная программа это не совсем полная версия КМП-алгоритма и соответствует тому, что принято называть алгоритмом Морриса—Пратта. Полный КМП-алгоритм содержит ещё один приём. Определим функцию *next* так:

$$\begin{aligned}
 next \ Null \ x &= Null \\
 next \ (Node \ [\] \ell \ r) \ x &= Node \ [\] \ell \ r \\
 next \ (Node \ (v : vs) \ \ell \ r) \ x &= \text{if } v == x \text{ then } next \ \ell \ x \\
 &\quad \text{else } Node \ (v : vs) \ \ell \ r
 \end{aligned}$$

По сути $next \ t \ x$ заменяет дерево t первым деревом в списке левых поддеревьев t , у которого метка начинается с любого символа, кроме x . Важное наблюдение относительно $next$ состоит в том, что, как можно видеть из определения op , верно:

$$op \ (Node \ (v : vs) \ \ell \ r) \ x = op \ (Node \ (v : vs) \ (next \ \ell \ v) \ r) \ x$$

Отсюда следует, что выполнение op станет более эффективным, если заменить каждый узел $Node \ (v : vs) \ \ell \ r$ в дереве другим узлом, а именно, $Node \ (v : vs) \ (next \ \ell \ v) \ r$. Однако мы не будем вдаваться в эти детали.

Заключительные замечания

КМП-алгоритм впервые был описан в (Knuth *et al.*, 1977). Однако имеется множество других его изложений (например, (Gusfield, 1997); (Cormen *et al.*, 2001); (Crochemore and Rytter, 2002)). Фактически более сотни публикаций посвящено проблеме поиска подстроки в целом и КМП- и БМ-алгоритмам в частности. Мы написали две статьи о КМП: (Bird, 1977), (Bird *et al.*, 1989), одну из которых — за тридцать лет до того, как принципы функционального программирования были прочно утверждены. Приведённое выше описание КМП-алгоритма это вычищенная и пересмотренная версия того, что описано в (Bird *et al.*, 1989). Не так давно Оливер Денви (Olivier Danvy) с коллегами из BRICS сделал несколько работ на тему получения алгоритмов КМП и БМ с помощью частичного исполнения. Например, (Ager *et al.*, 2003) использует схожие с содержащимися в (Bird, 1977) идеи, чтобы решить давнюю открытую проблему в области частичного исполнения, а именно, как получить КМП из наивного алгоритма с помощью процесса частичного исполнения, который занимает линейное время. А в (Danvy and Rohde, 2005) приводится порождение фазы поиска БМ-алгоритма с использованием частичного исполнения, представляя правило стоп-символа как улучшение времени связывания.

Литература

- Ager, M. S., Danvy, O. and Rohde, H. K. (2003). Fast partial evaluation of pattern matching in strings. BRICS Report Series, RS-03-11, University of Aarhus, Denmark.

- Bird, R. S. (1977). Improving programs by the introduction of recursion. *Communications of the ACM* **20** (11), 856–63.
- Bird, R. S., Gibbons, J. and Jones, G. (1989). Formal derivation of a pattern matching algorithm. *Science of Computer Programming* **12**, 93–104.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001). *Introduction to Algorithms*, second edition. Cambridge, MA: MIT Press. [Имеется русский перевод: Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: Построение и анализ. 2-е изд. М.: Вильямс. 2005.]
- Crochemore, M. and Rytter, W. (2002). *Jewels of Stringology*. Hong Kong: World Scientific.
- Danvy, O. and Rohde, H. K. (2005). On obtaining the Boyer–Moore string-matching algorithm by partial evaluation. BRICS Research Report RS-05-14, University of Aarhus, Denmark.
- Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge, UK: Cambridge University Press. [Имеется русский перевод: Гасфилд Д. Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, БХВ-Петербург. 2003.]
- Knuth, D. E., Morris, J. H. and Pratt, V. B. (1977). Fast pattern matching in strings. *SIAM Journal on Computing* **6**, 323–50.

18

Планирование в «Час пик»

Введение

«Час пик» это захватывающая головоломка из разряда игр с передвижением по полю. Она была придумана знаменитым создателем головоломок Нобом Йошигахарой (Nob Yoshigahara) и запущена в продажу компанией Think Fun¹. Игра происходит на поле 6×6 и может быть решена за разумное время полным перебором поиском в ширину. Более общая версия с полем $n \times n$ является PSPACE-полной задачей, так что экспоненциальный алгоритм решения это, видимо, лучшее, на что можно надеяться. Однако с помощью подходящего алгоритма планирования можно существенно ускорить полный перебор, и цель данной жемчужины — показать как. Более точное описание правил игры оставим на потом, а сначала обратимся к более абстрактным определениям головоломок, поиска в ширину и планирования.

Головоломки

Рассмотрим абстрактную головоломку, определённую двумя конечными множествами, множеством состояний и множеством ходов. Заданы три функции:

$$\begin{aligned} moves &:: State \rightarrow [Move] \\ move &:: State \rightarrow Move \rightarrow State \end{aligned}$$

¹Официальный сайт «Часа пик»: <http://www.puzzles.com/products/rushhour.htm>.

solved :: *State* → *Bool*

Функция *moves* определяет возможные ходы из текущего состояния, *move* возвращает новое состояние после заданного хода. Функция *solved* показывает, какие состояния считаются выигрышными. Определённая таким образом головоломка представляет собой конечный автомат. Решить головоломку значит найти последовательность переходов, желательно более короткую, которая бы приводила из начального состояния в допускающее:

solve :: *State* → *Maybe* [*Move*]

Значение *solve q* равно *Nothing*, если последовательности шагов из *q* в некоторое допускающее состояние не существует, и *Just ms* иначе, в этом случае для *ms* выполнено условие: *solved (foldl move q ms)*.

Возможно реализовать *solve* с помощью поиска в ширину или поиска в глубину. В обоих случаях ключевое соображение заключается в синонимах:

```
type Path      = ([Move], State)
type Frontier = [Path]
```

Путь состоит из последовательности ходов из некоторого начального состояния, объединённой с результирующим состоянием. Фронт (*Frontier*) это список путей, которые предстоит продолжить. Поиск в ширину можно определить так:

```
bfsearch      :: [State] → Frontier → Maybe [Move]
bfsearch qs [] = Nothing
bfsearch qs (p@(ms, q) : ps)
| solved q    = Just ms
| q ∈ qs      = bfsearch qs ps
| otherwise    = bfsearch (q : qs) (ps ++ succs p)
```

где

```
succs        :: Path → [Path]
succs (ms, q) = [(ms ++ [m], move q m) | m ← moves q]
```

Первый параметр *qs* функции *bfsearch* представляет множество просмотренных состояний. При поиске в ширину фронт устроен по принципу очереди, так что все пути одинаковой длины, считая от начального состояния, просматриваются раньше, чем их продолжения. При просмотре пути он может быть принят, если последнее состояние в нём допускающее, может

быть отвергнут, если последнее состояние уже встречалось, иначе к фронту для дальнейшей обработки добавляются все «последователи» данного пути (полученные всеми возможными ходами из последнего состояния). Поиск в ширину найдёт кратчайшее решение, если оно существует.

Всего одно изменение отличает поиск в глубину от поиска в ширину. Оно состоит в замене выражения $ps \uparrow\downarrow succs p$ на $succs p \uparrow\downarrow ps$. При поиске в глубину фронт устроен по принципу стека, так что все следующие за данным путем исследуются ранее любого другого пути той же длины, что и данный. Поиск в глубину найдёт решение, если оно существует, однако это решение может не быть кратчайшим.

При поиске в ширину фронт может вырасти экспоненциально по сравнению с поиском в глубину. Следовательно, по определению выше, $bfsearch$ отнимет намного больше времени, чем $dfsearch$. Причина в том, что вычисление $ps \uparrow\downarrow succs p$ требует пропорционального длине фронта ps времени. Возможный способ ускорить приведённый код, что, однако, не снижает сложность по памяти, связан с введением аккумулирующего параметра:

$$bfsearch' qs pss ps = bfsearch qs (ps \uparrow\downarrow concat (reverse pss))$$

Тогда после нескольких простых выкладок, которые мы опустим, получится:

$$\begin{aligned} bfsearch' &:: [State] \rightarrow [Frontier] \rightarrow Frontier \rightarrow Maybe [Move] \\ bfsearch' qs [] [] &= Nothing \\ bfsearch' qs pss [] &= bfsearch' qs [] (concat (reverse pss)) \\ bfsearch' qs pss (p@(ms, q) : ps) \\ | solved q &= Just ms \\ | q \in qs &= bfsearch' qs pss ps \\ | otherwise &= bfsearch' (q : qs) (succs p : pss) ps \end{aligned}$$

На самом деле есть более простая версия $bfsearch'$, в которой аккумулятор имеет тип $Frontier$, а не $[Frontier]$:

$$\begin{aligned} bfsearch' &:: [State] \rightarrow Frontier \rightarrow Frontier \rightarrow Maybe [Move] \\ bfsearch' qs [] [] &= Nothing \\ bfsearch' qs rs [] &= bfsearch' qs [] rs \\ bfsearch' qs rs (p@(ms, q) : ps) \\ | solved q &= Just ms \\ | q \in qs &= bfsearch' qs rs ps \\ | otherwise &= bfsearch' (q : qs) (succs p \uparrow\downarrow rs) ps \end{aligned}$$

Поведение этого варианта $bfsearch'$ отличается от предыдущего тем, что последовательные фронты просматриваются то слева направо, то справа

налево, хотя кратчайшее решение по-прежнему находится, если оно существует.

Теперь можно определить:

$$\text{bfsolve } q = \text{bfsearch}' [] [] [([], q)]$$

Функция *bfsolve* реализует *solve* посредством поиска в ширину.

Планирование

К этому моменту мы попросту разработали стратегию перебора каждой возможной последовательности ходов, который завершается, когда находится подходящая. Люди решают головоломки не так. Вместо этого они строят планы. В нашем случае план это последовательность ходов, которые, если могут быть выполнены, приводят к выигрышу. То есть:

$$\text{type Plan} = [\text{Move}]$$

План должен состоять из неповторяющихся ходов, иначе он не может быть выполнен. Если, чтобы сделать ход m , нужно сперва сделать ход t , то из этого ничего не получится. Пустой план означает успех. В ином случае, предположим, что первый ход по плану это m . Если в текущем состоянии это возможно, то m делается. Иначе привлечём функцию *premoves* :: $\text{State} \rightarrow \text{Move} \rightarrow [[\text{Move}]]$, такую что каждый вариант *pms* в списке *premoves* $q m$ содержит предварительные ходы, которые необходимо выполнить, чтобы m стал возможен. В свою очередь, ходы из *pms* сами могут потребовать предварительных ходов, так что мы расширим план с помощью повторного применения *premoves*:

$$\begin{aligned} \text{newplans} &:: \text{State} \rightarrow \text{Plan} \rightarrow [\text{Plan}] \\ \text{newplans } q \text{ ms} &= \text{mkplans ms} \\ \text{where} \\ \text{mkplans ms} \mid \text{null ms} &= [] \\ \mid m \in qms &= [ms] \\ \mid \text{otherwise} &= \text{concat} [\text{mkplans} (\text{pms} + ms) \mid \\ &\quad \text{pms} \leftarrow \text{premoves } q \text{ m}, \\ &\quad \text{all} (\not\in ms) \text{ pms}] \\ \text{where } m &= \text{head ms}; qms = \text{moves } q \end{aligned}$$

Результат *newplans q ms* представляет собой возможно пустой список непустых планов, первый ход которых может быть выполнен из состояния q .

Для затравки планирования предположим, что головоломка решается из состояния q выполнением ходов в $goalmoves\ q$, где $goalmoves::State \rightarrow Plan$.

С помощью всего двух новых функций $goalmoves$ и $premoves$ можно определить процедуру поиска иначе, основываясь на понятиях усиленного пути и фронта:

```
type APath    = ([Move], State, Plan)
type AFrontier = [APath]
```

Усиленный путь состоит из ходов, выполненных из некоторого начального состояния, достигнутого состояния и плана последующих ходов. Поиск заключается в исследовании усиленных путей до тех пор, пока не найдётся путь, ведущий к успеху, или все пути не провалятся.

```
psearch :: [State] → AFrontier → Maybe [Move]
psearch qs [] = Nothing
psearch qs (p@(ms, q, plan) : ps)
| solved q = Just ms
| q ∈ qs = psearch qs ps
| otherwise = psearch (q : qs) (asuccs p ++ ps ++ bsuccs p)
```

где

```
asuccs, bsuccs :: APath → [APath]
asuccs (ms, q, plan)
= [(ms ++ [m], move q m, plan') | m : plan' ← newplans q plan]
bsuccs (ms, q, _)
= [(ms ++ [m], q', goalmoves q') | m ← moves q, let q' = move q m]
```

В $psearch$ все планы фронта ps проверяются в стиле поиска в глубину. Если все они закончатся неудачей, добавляются новые планы, полученные некоторым допустимым шагом и сконструированные с некоторой новой целью. Эти дополнительные планы, выраженные функцией $bsuccs$, необходимы для полноты процесса. Простые головоломки можно решить с подходящим планом, но в общем случае, даже если решение существует, планирование может ни к чему не привести. Это следствие жадности процесса планирования: те ходы, которые можно сделать, делаются. Чтобы гарантировать полноту, нужно приготовиться строить дополнительные планы на каждом этапе.

Как и в случае с поиском в ширину, $psearch$ можно ускорить, если ввести аккумулирующий параметр:

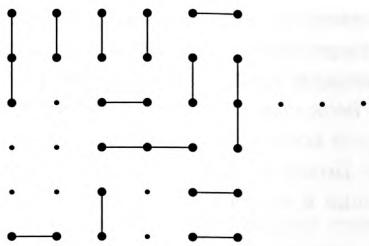


Рис. 18.1: поле в «Час пик»

```

psearch' :: [State] → AFrontier → AFrontier → Maybe [Move]
psearch' qs [] [] = Nothing
psearch' qs rs [] = psearch' qs [] rs
psearch' qs rs (p@(ms, q, plan) : ps)
| solved q      = Just (reverse ms)
| q ∈ qs       = psearch' qs rs ps
| otherwise     = psearch' (q : qs) (bsuccs p ++ rs) (asuccs p ++ ps)

```

Функция *psolve* теперь может быть определена так:

```

psolve    :: State → Maybe [Move]
psolve q = psearch' [] [] ([][], q, goalmoves q)]

```

Функция *psolve* реализует *solve* посредством планирования. Можно определить версию *psolve*, которая исследует планы в стиле поиска в ширину, но мы оставим эти детали читателю. Отметим, что *psearch* найдёт решение, если оно существует, хотя и не обязательно кратчайшее.

«Час пик»

Теперь применим изложенные выше мысли к «Часу пик». Как упоминалось ранее, эта игра состоит из поля 6×6 в 36 клеток. Некоторые из клеток заняты транспортом, который может быть ориентирован вертикально или горизонтально и занимать две клетки, если это легковая машина, или три клетки, если это грузовик. Горизонтальный транспорт может двигаться влево и вправо, а вертикальный — вверх и вниз. Одна фиксированная клетка, третья сверху вдоль правой вертикальной стороны, имеет специальный смысл и называется выездом. Одно транспортное средство также

имеет особое назначение, оно расположено горизонтально слева от выезда. Цель игры переместить этот специальный транспорт к выезду. Пример начальной расстановки машин приведён на рис. 18.1.

Существует несколько способов представления игрового поля, простейший из которых состоит в обозначении каждой клетки парой её декартовых координат. Более экономный в отношении памяти вариант (важное свойство для поиска в ширину) представлен следующей нумерацией клеток:

1	2	3	4	5	6
8	9	10	11	12	13
15	16	17	18	19	20
22	23	24	25	26	27
29	30	31	32	33	34
36	37	38	39	40	41

Левую и правую границу составляют клетки с числами кратными 7, верхнюю границу — клетки с отрицательными числами, нижнюю — клетки с числами больше 42. Выезд обозначен числом 20. Положение на поле может быть закодировано списком пар клеток, обозначающих начало и конец машины соответственно. Транспорт имеет неявные номера, соответствующие позициям в этом списке, причём специальный автомобиль имеет номер 0, так что его представляет первая пара. К примеру, поле на рис. 18.1 кодируется так:

```
g1 = [(17, 18), (1, 15), (2, 9), (3, 10), (4, 11), (5, 6), (12, 19),
      (13, 27), (24, 26), (31, 38), (33, 34), (36, 37), (40, 41)]
```

Это представление отражено в следующих синонимах типов:

```
type Cell    = Int
type Grid   = [(Cell, Cell)]
type Vehicle = Int
type Move    = (Vehicle, Cell)
type State   = Grid
```

Занятые клетки можно перечислить по возрастанию, если заполнять интервалы, соответствующие каждому автомобилю, и производить слияние:

```
occupied     :: Grid → [Cell]
occupied     = foldr (merge ∙ fillcells) []
fillcells (r, f) = if r > f - 7 then [r .. f] else [r, r + 7 .. f]
```

Автомобиль, занимающий клетки в интервале (r, f) (от англ. rear — задняя часть, front — передняя часть), расположен горизонтально, если $r > f - 7$ и вертикально, если $r \leq f - 7$. Свободные клетки определяются так:

$$\begin{aligned} \text{freecells} &:: \text{Grid} \rightarrow [\text{Cell}] \\ \text{freecells } g &= \text{allcells} \setminus \text{occupied } g \end{aligned}$$

где $\text{allcells} = [c \mid c \leftarrow [1..41], c \bmod 7 \neq 0]$. Опустим стандартные определения *merge* и операции разности упорядоченных списков.

Функция *moves* реализуется так:

$$\begin{aligned} \text{moves} &:: \text{Grid} \rightarrow [\text{Move}] \\ \text{moves } g &= [(v, c) \mid (v, i) \leftarrow \text{zip } [0..] g, c \leftarrow \text{adjs } i, c \in \text{fs}] \\ &\quad \text{where } \text{fs} = \text{freecells } g \\ \text{adjs } (r, f) &= \text{if } r > f - 7 \text{ then } [f + 1, r - 1] \text{ else } [f + 7, r - 7] \end{aligned}$$

Ход (v, c) допустим тогда и только тогда, когда клетка c не занята и находится следом за клетками, занятymi v , вдоль подходящей оси координат.

Функция *move* выглядит так:

$$\begin{aligned} \text{move } g(v, c) &= g1 \uparrow\downarrow \text{adjust } i \ c : g2 \\ &\quad \text{where } (g1, i : g2) = \text{splitAt } v \ g \end{aligned}$$

а *adjust* так:

$$\begin{aligned} \text{adjust } (r, f) \ c \\ | \ r > f - 7 &= \text{if } c > f \text{ then } (r + 1, c) \text{ else } (c, f - 1) \\ | \ \text{otherwise} &= \text{if } c < r \text{ then } (c, f - 7) \text{ else } (r + 7, c) \end{aligned}$$

Арифметика здесь говорит сама за себя, так что пояснения мы опустим.

Головоломка решена, если автомобиль с номером 0 находится у выезда:

$$\begin{aligned} \text{solved} &:: \text{Grid} \rightarrow \text{Bool} \\ \text{solved } g &= \text{snd } (\text{head } g) = 20 \end{aligned}$$

Теперь можно реализовать поиск в ширину:

$$\begin{aligned} \text{bfsolve} &:: \text{Grid} \rightarrow \text{Maybe } [\text{Move}] \\ \text{bfsolve } g &= \text{bfsearch}' [] [] ([[], g]) \end{aligned}$$

где *bfsearch'* это функция из предыдущего раздела.

Чтобы получить *psearch*, нужно определить две дополнительные функции, *goalmoves* и *premoves*. Первая не составит труда:

goalmoves :: *Grid* → *Plan*

goalmoves g = [(0, *c*) | *c* ← [*snd* (*head g*) + 1..20]]

Таким образом, *goalmoves* это список ходов, которые переместят автомобиль 0 к выезду.

Значение *premoves g m* требуется только тогда, когда *m* это ход в клетку, которая на данный момент занята. Единственным образом определена пара (*v, i*) из списка *zip* [0..] *g*, в которой интервал *i* содержит клетку *c*. Функция *blocker* находит такую пару:

blocker :: *Grid* → *Cell* → (*Vehicle*, (*Cell*, *Cell*))

blocker g c = *search* (*zip* [0..] *g*) *c*

search ((v, i) : vis) c = if *covers c i* then (*v, i*) else *search vis c*

covers c (r, f) = *r* ≤ *c* ∧ *c* ≤ *f* ∧ (*r* > *f* - 7 ∨ (*c* - *r*) mod 7 == 0)

Блокирующий автомобиль *v* занимает интервал *i* = (*r, f*) и должен быть смещён, чтобы освободить клетку *c*; для этого *v* перемещается за подходящее количество ходов влево или вправо, если он стоит горизонтально, и вверх или вниз, если вертикально. Эти ходы вычисляются функцией *freeingmoves*:

freeingmoves :: *Cell* → (*Vehicle*, (*Cell*, *Cell*)) → [[*Move*]]

freeingmoves c (v, (r, f))

| *r* > *f* - 7 = [[[*(v, j)*] | *j* ← [*f* + 1..*c* + *n*]] | *c* + *n* < *k* + 7] +
| [[[*(v, j)*] | *j* ← [*r* - 1, *r* - 2..*c* - *n*]] | *c* - *n* > *k*]]

| otherwise = [[[*(v, j)*] | *j* ← [*r* - 7, *r* - 14..*c* - *m*]] | *c* - *m* > 0] +
| [[[*(v, j)*] | *j* ← [*f* + 7, *f* + 14..*c* + *m*]] | *c* + *m* < 42]]

where (*k, m, n*) = (*f* - *f* mod 7, *f* - *r* + 7, *f* - *r* + 1)

Если *v* стоит горизонтально, что значит *r* > *f* - 7, то его длина *n* = *f* - *r* + 1 и, чтобы освободить *c*, надо сдвинуть *v* или вправо к клетке *c* + *n*, или влево к клетке *c* - *n*, если таковые находятся в пределах поля. Если *v* ориентирован вертикально, то его длина *n* = (*f* - *r*) div 7 + 1 и смещаться надо вверх к клетке *c* - 7*n* либо вниз к клетке *c* + 7*n*, снова в предположении, что эти клетки входят в поле.

Теперь можно описать *premoves*:

premoves :: *Grid* → *Move* → [[*Move*]]

premoves g (v, c) = *freeingmoves c (blocker g c)*

Между тем определение *newplans* из прошлого раздела нуждается в изменении, чтобы «работать в час пик». Понять это можно, представив, что текущий план выражен в ходах $[(0, 19), (0, 20)]$, а машина 0 занимает клетки $[17, 18]$ на исходном поле. Пусть первый ход $(0, 19)$ не допустим до тех пор, пока не сделаны подготовительные ходы *pms*. В этот момент вполне возможно, что $(0, 16)$, который сдвинет нулевой автомобиль влево, попадёт в список *pms*. После выполнения ходов *pms*, готовясь к $(0, 19)$, мы видим, что $(0, 19)$ не является более корректным ходом на полученном поле, так как он требует теперь перемещения 0 на два шага вперёд, и должен быть, таким образом, сперва разложен в два «нормальных» одношаговых хода $[(0, 18), (0, 19)]$. В связи с этим переопределим *newplans*:

```

newplans :: Grid → Plan → [Plan]
newplans g [] = []
newplans g (m : ms) = mkplans (expand g m ++ ms)
where mkplans ms = if m ∈ gms then [ms] else
    concat [mkplans (pms ++ ms) |
    pms ← premoves g m,
    all (∉ ms) pms]
where m = head ms; gms = moves g
```

Новая функция *expand*, которая раскладывает возможно некорректный ход в последовательность корректных, определяется так:

```

expand :: Grid → Move → [Move]
expand g (v, c)
  | r > f - 7 = if c > f then  $[(v, p) | p \leftarrow [f + 1..c]]$ 
    else  $[(v, p) | p \leftarrow [r - 1, r - 2..c]]$ 
  | otherwise = if c > f then  $[(v, p) | p \leftarrow [f + 7, f + 14..c]]$ 
    else  $[(v, p) | p \leftarrow [r - 7, r - 14..c]]$ 
where  $(r, f) = g !! v$ 
```

Теперь можно реализовать алгоритм планирования:

```

psolve :: Grid → Maybe [Move]
psolve g = psearch' [] [] [([], g, goalmoves g)]
```

где *psearch'* это функция из прошлого раздела с точностью до пересмотренного определения *newplans*.

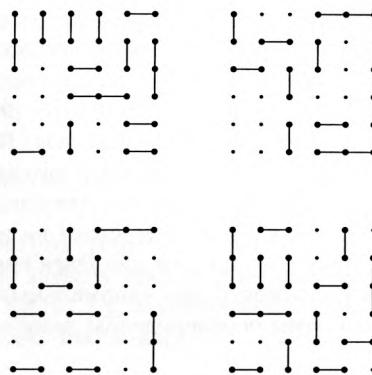


Рис. 18.2: четыре варианта головоломки «Час пик»

Таблица 18.1. Замеры времени для решения четырёх вариантов головоломки «Час пик»

Вариант	<i>bfsolve</i>	Ходы	<i>psolve</i>	Ходы	<i>dfsolve</i>	Ходы
1	9.11	34	0.23	38	3.96	1228
2	4.71	18	0.04	27	3.75	2126
3	1.70	55	0.91	75	0.97	812
4	9.84	81	2.36	93	2.25	1305

Результаты

Насколько же лучше *psolve*, чем *bfsolve*? Четыре варианта «Часа пик» (рис. 18.2) были взяты с сайта Ника Бакстера (Nick Baxter), посвящённого головоломкам: www.puzzleworld.org/SlidingBlockPuzzles/rushhour.htm, и решались на Pentium 3, 1000MHz с использованием GHCi. Результаты представлены в таблице 18.1 (время в секундах). Как видно из таблицы, *psolve* заметно быстрей *bfsolve*, от двух до ста раз. С другой стороны, *psolve* может делать лишние ходы. Для сравнения мы также привели время и количество ходов *dfsolve*.

Заключительные замечания

Результат о PSPACE-полноте «Часа пик» можно найти в (Flake and Baum, 2002). Автор познакомился с головоломкой на летней школе по функциональному программированию (Jones, 2008), где Марк Джонс (Mark Jones) читал серию лекций о преимуществах функционального мышления в решении задач. Он приводил решение с поиском в ширину, но предлагал участникам летней школы потрудиться над более быстрым алгоритмом. Эта жемчужина родилась в ответ на предложение Джонса.

Литература

- Flake, G. W. and Baum, E. B. (2002). Rush Hour is PSPACE-complete, or “Why you should generously tip parking lot attendants”. *Theoretical Computer Science* **270** (1), 895–911.
- Jones, M. P. (2008). Functional thinking. *Advanced Functional Programming Summer School*, Boxmeer, The Netherlands.

19

Простой алгоритм решения судоку

ПРАВИЛА ИГРЫ: заполните сетку так, чтобы в каждой строке, каждом столбце и каждой площадке 3×3 стояли цифры 1–9. Никакой математики. Головоломка решается с помощью логики и рассуждений.

Советы как играть в судоку, газета *The Independent Newspaper*

Введение

В судоку играют на сетке 9×9 . Пусть дана матрица, например такая, как изображена на рис. 19.1, задача — заполнить пустые клетки цифрами от 1 до 9 так, чтобы каждая строка, столбец и площадка 3×3 содержали все числа от 1 до 9. В общем случае может существовать одно или несколько решений, кроме того, их может не быть вовсе, хотя в хорошей головоломке решение всегда есть, причём единственное. Наша цель в этой жемчужине — сконструировать программу на Haskell, которая решает судоку. Более точно, мы определим функцию *solve*, которая находит все возможные способы корректного заполнения заданной начальной сетки. Если требуется только одно решение, можно взять голову итогового списка. Ленивые вычисления гарантируют, что в этом случае только первый вариант и будет посчитан. Начнём с некоторой спецификации, а затем, используя эквациональные рассуждения, выведем более эффективную версию. Никакой математики — только логика и рассуждения!

		4		5	7		
				9	4		
3	6						8
7	2		6				
		4		2			
			8		9	3	
4					5	6	
		5	3				
	6	1		9			

Рис. 19.1: сетка для судоку

Спецификация

Для начала определим несколько простейших типов данных. Матрицы:

```
type Matrix a = [Row a]
type Row a    = [a]
```

Матрица $m \times n$ представляется списком из m строк, в котором все строки имеют одинаковую длину n . Сетка это матрица 9×9 из цифр:

```
type Grid  = Matrix Digit
type Digit = Char
```

Допускаются цифры от 1 до 9 и 0, обозначающий пробел:

```
digits = ['1'..'9']
blank  = ('==' '0')
```

Мы предполагаем, что заданная исходная сетка содержит только пробелы и цифры. Кроме того, считается, что эта сетка корректна, то есть ни одна цифра не повторяется более одного раза ни в одном столбце, строке или площадке.

Теперь перейдём к спецификации. Цель состоит в том, чтобы записать простейшую и наиболее понятную спецификацию *solve* без учёта эффективности результата. Первый вариант мог бы выглядеть так: сначала

создать список всех корректно заполненных сеток, а затем проверить, какие из них подходят для исходной сетки (у них должны совпадать цифры, заданные в непустых клетках этой сетки). Другая версия — и мы воспользуемся именно ей — заключается в том, чтобы начать из заданной сетки и проставить все возможные варианты в каждую пустую клетку. Далее надо вычислить все сетки, появляющиеся для каждого варианта, а затем из них отфильтровать корректные. Такая спецификация формализуется следующим образом:

$$\text{solve} = \text{filter valid} \cdot \text{expand} \cdot \text{choices}$$

Вспомогательные функции обладают типами:

$$\begin{aligned}\text{choices} &:: \text{Grid} \rightarrow \text{Matrix Choices} \\ \text{expand} &:: \text{Matrix Choices} \rightarrow [\text{Grid}] \\ \text{valid} &:: \text{Grid} \rightarrow \text{Bool}\end{aligned}$$

Простейший выбор для *Choices* это *type Choices = [Digit]*. В таком случае мы получаем

$$\begin{aligned}\text{choices} &:: \text{Grid} \rightarrow \text{Matrix Choices} \\ \text{choices} &= \text{map} (\text{map choice}) \\ \text{choice } d &= \text{if blank } d \text{ then digits else } [d]\end{aligned}$$

Если клетка пуста, то все цифры признаются допустимыми, иначе выбора нет, и возвращается список из одного элемента.

Далее, полученный результат разлагается в декартово произведение матриц:

$$\begin{aligned}\text{expand} &:: \text{Matrix Choices} \rightarrow [\text{Grid}] \\ \text{expand} &:: \text{cp} \cdot \text{map cp}\end{aligned}$$

Декартово произведение списка списков вычисляется так:

$$\begin{aligned}\text{cp} &:: [[a]] \rightarrow [[a]] \\ \text{cp} [] &= [[]] \\ \text{cp} (xs : xss) &= [x : ys \mid x \leftarrow xs, ys \leftarrow \text{cp } xss]\end{aligned}$$

Например, $\text{cp} [[1, 2], [3], [4, 5]] = [[1, 3, 4], [1, 3, 5], [2, 3, 4], [2, 3, 5]]$. Таким образом, *map cp* возвращает список всех возможных вариантов для каждой строки, а *cp · map cp* собирает все варианты для строк всеми возможными способами.

Наконец, рассмотрим *valid*. Корректной является сетка, в которой ни одна строка, столбец или площадка не содержат повторяющихся цифр:

$$\begin{aligned} \textit{valid} &:: \textit{Grid} \rightarrow \textit{Bool} \\ \textit{valid } g &= \textit{all nodups} (\textit{rows } g) \wedge \\ &\quad \textit{all nodups} (\textit{cols } g) \wedge \\ &\quad \textit{all nodups} (\textit{boxs } g) \end{aligned}$$

Стандартная функция *all p*, применённая к конечному списку *xs*, возвращает *True*, если все элементы *xs* удовлетворяют *p*, и *False* в противном случае. Функция *nodups* может быть определена так:

$$\begin{aligned} \textit{nodups} &:: \textit{Eq } a \Rightarrow [a] \rightarrow \textit{Bool} \\ \textit{nodups } [] &= \textit{True} \\ \textit{nodups } (x : xs) &= \textit{all } (\neq x) xs \wedge \textit{nodups } xs \end{aligned}$$

Функция *nodups* имеет квадратичную сложность по времени. В качестве альтернативы допустимо отсортировать список цифр и проверить, является ли он строго возрастающим. Сортировку можно провести за $\Theta(n \log n)$ операций. Однако, при $n = 9$ не очевидно, что сортировка цифр принесёт выигрыш. Что бы предпочли вы: $2n^2$ или $100n \log_2 n$ операций?

Остаётся определить функции *rows*, *cols* и *boxs*. Если матрица задана списком своих строк, то *rows* представляет собой просто тождественную функцию на множестве матриц:

$$\begin{aligned} \textit{rows} &:: \textit{Matrix } a \rightarrow \textit{Matrix } a \\ \textit{rows} &= \textit{id} \end{aligned}$$

Функция *cols* выполняет транспонирование матрицы. Одно возможное определение:

$$\begin{aligned} \textit{cols} &:: \textit{Matrix } a \rightarrow \textit{Matrix } a \\ \textit{cols } [xs] &= [[x] \mid x \leftarrow xs] \\ \textit{cols } (xs : xss) &= \textit{zipWith } (:) xs (\textit{cols } xss) \end{aligned}$$

Функция *boxs* немного интересней:

$$\begin{aligned} \textit{boxs} &:: \textit{Matrix } a \rightarrow \textit{Matrix } a \\ \textit{boxs} &= \textit{map ungroup} \cdot \textit{ungroup} \cdot \textit{map cols} \cdot \textit{group} \cdot \textit{map group} \end{aligned}$$

Функция *group* разбивает список на группы по три элемента:

$$\begin{aligned} \textit{group} &:: [a] \rightarrow [[a]] \\ \textit{group } [] &= [] \\ \textit{group } xs &= \textit{take } 3 xs : \textit{group } (\textit{drop } 3 xs) \end{aligned}$$

Функция *ungroup* принимает сгруппированный список и разгруппировывает его:

$$\begin{aligned} \textit{ungroup} &:: [[a]] \rightarrow [a] \\ \textit{ungroup} &= \textit{concat} \end{aligned}$$

Действие *boxs* в случае матрицы 4×4 , когда *group* разбивает список на группы по два, проиллюстрировано ниже:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \rightarrow \begin{pmatrix} (ab & cd) \\ (ef & gh) \\ (ij & kl) \\ (mn & op) \end{pmatrix} \rightarrow \begin{pmatrix} (ab & ef) \\ (cd & gh) \\ (ij & mn) \\ (kl & op) \end{pmatrix}$$

Функция *group · map group* создаёт список матриц; транспонирование каждой и разгруппировка даёт искомые площадки.

Заметьте, что вместо того, чтобы думать о матрицах в терминах индексов и проводить вычисления с этими индексами для выделения строк, столбцов и площадок, мы пришли к определению функций, которые воспринимают матрицу как единое целое. Герэйт Джонс (Geraint Jones) метко окрестил этот стиль цельнозерновое программирование (*wholemeal programming*). Цельнозерновое программирование очень полезно: оно способствует предотвращению таких недугов, как индекситит, и поощряет здоровое построение программ.

Приведём три закона, справедливых для сеток судоку 9×9 , а на самом деле и для произвольных $N^2 \times N^2$ матриц:

$$\begin{aligned} \textit{rows} \cdot \textit{rows} &= \textit{id} \\ \textit{cols} \cdot \textit{cols} &= \textit{id} \\ \textit{boxs} \cdot \textit{boxs} &= \textit{id} \end{aligned}$$

По-другому: все три функции являются инволюциями. Два закона доказать легко, третий — намного сложней. Сложный закон касается не функции *boxs*, как можно было бы подумать, а инволютивности *cols*. Хотя он интуитивно очевиден, доказательство напрямую из определения довольно мудрёное. Свойство инволюции *boxs* показывается простыми вычислениями с использованием инволютивного свойства *cols*, простых свойств *map* и того факта, что *group · ungroup = id*.

Вот ещё три закона, справедливые для матриц с вариантами $N^2 \times N^2$:

$$\textit{map rows} \cdot \textit{expand} = \textit{expand} \cdot \textit{rows} \tag{19.1}$$

$$\textit{map cols} \cdot \textit{expand} = \textit{expand} \cdot \textit{cols} \tag{19.2}$$

$$\textit{map boxs} \cdot \textit{expand} = \textit{expand} \cdot \textit{boxs} \tag{19.3}$$

Вскоре они нам понадобятся.

Отсечения в матрицах с вариантами

Вполне пригодная теоретически, спецификация *solve* просто безнадёжна на практике. Если предположить, что примерно половина из 81 позиции заполнена изначально (преддая оценка), имеется около 9^{40} или

147 808 829 414 345 923 316 083 210 206 383 297 601

сеток для проверки! Значит нам нужен подход получше. Для построения более эффективного алгоритма хорошей идеей было бы удалять те варианты цифр из клетки c , которые уже появлялись в одноэлементных списках в смежной с c строке, столбце и площадке. Нам понадобится функция

prune :: *Matrix Choices* → *Matrix Choices*

такая, что

filter valid · expand = *filter valid · expand · prune*

Как определить *prune*? Поскольку матрица задана списком строк, для начала научимся проводить отсечения внутри одной строки. Функция *pruneRow* определяется так:

```
pruneRow      :: Row Choices → Row Choices
pruneRow row = map (remove fixed) row
    where fixed = [d | [d] ← row]
```

Функция *remove* удаляет заданные варианты из любого набора вариантов, кроме тех, что были фиксированы изначально:

remove xs ds = **if** singleton *ds* **then** *ds* **else** *ds* \\ *xs*

Функция *pruneRow* удовлетворяет условию

filter nodups · cp = *filter nodups · cp · pruneRow* (19.4)

Доказательство остаётся в качестве упражнения.

Теперь почти всё готово для вычисления, которое определит функцию *prune*. «Почти», потому что нам понадобятся два закона относительно функций *filter*. Первый состоит в том, что если $f \cdot f = id$, то

filter (p · f) = *map f · filter p · map f* (19.5)

По-другому, $\text{filter } (p \cdot f) \cdot \text{map } f = \text{map } f \cdot \text{filter } p$. Второй закон гласит:

$$\text{filter } (\text{all } p) \cdot cp = cp \cdot \text{map } (\text{filter } p) \quad (19.6)$$

Доказательства (19.5) и (19.6) вновь предстаиваютя для упражнений.

Теперь можно приступить к обещанному вычислению. Отправная точка состоит в преобразовании выражения $\text{filter valid} \cdot \text{expand}$:

$$\begin{aligned} \text{filter valid} \cdot \text{expand} &= \text{filter } (\text{all nodups} \cdot \text{boxs}) \cdot \\ &\quad \text{filter } (\text{all nodups} \cdot \text{cols}) \cdot \\ &\quad \text{filter } (\text{all nodups} \cdot \text{rows}) \cdot \text{expand} \end{aligned}$$

Порядок фильтров в правой части равенства не существуетен. План действий состоит в том, чтобы направить каждый из этих фильтров на борьбу с expand . Например, в случае boxs выводим:

$$\begin{aligned} &\text{filter } (\text{all nodups} \cdot \text{boxs}) \cdot \text{expand} \\ &= \{ (19.5), \text{ поскольку } \text{boxs} \cdot \text{boxs} = id \} \\ &\quad \text{map } \text{boxs} \cdot \text{filter } (\text{all nodups}) \cdot \text{map } \text{boxs} \cdot \text{expand} \\ &= \{ (19.3) \} \\ &\quad \text{map } \text{boxs} \cdot \text{filter } (\text{all nodups}) \cdot \text{expand} \cdot \text{boxs} \\ &= \{ \text{определение } \text{expand} \} \\ &\quad \text{map } \text{boxs} \cdot \text{filter } (\text{all nodups}) \cdot cp \cdot \text{map } cp \cdot \text{boxs} \\ &= \{ (19.6) \text{ и } \text{map } f \cdot \text{map } g = \text{map } (f \cdot g) \} \\ &\quad \text{map } \text{boxs} \cdot cp \cdot \text{map } (\text{filter nodups} \cdot cp) \cdot \text{boxs} \\ &= \{ (19.4) \} \\ &\quad \text{map } \text{boxs} \cdot cp \cdot \text{map } (\text{filter nodups} \cdot cp \cdot \text{pruneRow}) \cdot \text{boxs} \\ &= \{ (19.6) \} \\ &\quad \text{map } \text{boxs} \cdot \text{filter } (\text{all nodups}) \cdot cp \cdot \text{map } cp \cdot \text{map } \text{pruneRow} \cdot \text{boxs} \\ &= \{ \text{определение } \text{expand} \} \\ &\quad \text{map } \text{boxs} \cdot \text{filter } (\text{all nodups}) \cdot \text{expand} \cdot \text{map } \text{pruneRow} \cdot \text{boxs} \\ &= \{ (19.5) \text{ в форме } \text{map } f \cdot \text{filter } p = \text{filter } (p \cdot f) \cdot \text{map } f \} \\ &\quad \text{filter } (\text{all nodups} \cdot \text{boxs}) \cdot \text{map } \text{boxs} \cdot \text{expand} \cdot \text{map } \text{pruneRow} \cdot \text{boxs} \\ &= \{ (19.3) \} \\ &\quad \text{filter } (\text{all nodups} \cdot \text{boxs}) \cdot \text{expand} \cdot \text{boxs} \cdot \text{map } \text{pruneRow} \cdot \text{boxs} \end{aligned}$$

Мы показали, что

$$\begin{aligned} \text{filter } (\text{all nodups} \cdot \text{boxs}) \cdot \text{expand} \\ = \text{filter } (\text{all nodups} \cdot \text{boxs}) \cdot \text{expand} \cdot \text{pruneBy boxs} \end{aligned}$$

где $\text{pruneBy } f = f \cdot \text{map } \text{pruneRow} \cdot f$. Проведя аналогичные вычисления для rows и cols , получим

$$\text{filter valid} \cdot \text{expand} = \text{filter valid} \cdot \text{expand} \cdot \text{prune}$$

где

$$\text{prune} = \text{pruneBy boxs} \cdot \text{pruneBy cols} \cdot \text{pruneBy rows}$$

Наконец, проплое определение solve можно заменить:

$$\text{solve} = \text{filter valid} \cdot \text{expand} \cdot \text{prune} \cdot \text{choices}$$

На самом деле вместо однократного prune мы можем использовать столько prune , сколько пожелаем. Это имеет смысл, так как после одного раунда отсечений в некоторых наборах вариантов может остаться по одному элементу, и следующий раунд сможет отсечь с их помощью ещё больше некорректных вариантов. Простейшие задачи судоку как раз и решаются отсечениями в матрице вариантов, повторяющимися до тех пор, пока не останутся только одноэлементные наборы вариантов.

Одноклеточное порождение

Для головоломок посложнее мы можем совместить отсечения с ещё одной идеей: раскрывать варианты только для одной клетки за раз. Тогда как expand порождает сетки, используя все возможные варианты из каждой клетки матрицы вариантов, в один приём, одноклеточное порождение выбирает единственную клетку и порождает матрицы путём подстановки каждого варианта в эту и только эту клетку. Будем надеяться, что перемежая отсечения с одноклеточными порождениями, мы придём к решению скорее.

Итак, определим функцию expand1 , которая раскрывает варианты для единственной клетки. От неё требуется удовлетворять следующему свойству с точностью до некоторой перестановки ответов:

$$\text{expand} = \text{concat} \cdot \text{map } \text{expand} \cdot \text{expand1} \tag{19.7}$$

Хорошим кандидатом для раскрытия будет клетка с наименьшим числом вариантов (если оно не равно единице, конечно). Клетка с пустым

набором вариантов означает, что головоломка не разрешима, потому желательно выявлять такие клетки как можно раньше. Пусть клетка, содержащая список вариантов cs , расположена где-то в строке row , так что $row = row1 ++ [cs] ++ row2$, а сама строка находится в матрице вариантов, где сверху её имеются строки $rows1$, а снизу — $rows2$. Тогда можно определить:

```

expand1      :: Matrix Choices → [Matrix Choices]
expand1 rows = [rows1 ++ [row1 ++ [c] : row2] ++ rows2 | c ← cs]
where (rows1, row : rows2) = break (any smallest) rows
      (row1, cs : row2)    = break smallest row
      smallest cs          = length cs == n
      n                    = minimum (counts rows)
      counts               = filter (≠ 1) · map length · concat

```

Число n это наименьшее количество вариантов среди всех клеток матрицы вариантов, в которых имеется не один вариант. Если матрица содержит только одноэлементные наборы вариантов, то n это минимум из пустого списка целых чисел, то есть не определено. Стандартная функция $break p$ разбивает список на два:

$$break p xs = (takeWhile (not · p) xs, dropWhile (not · p) xs)$$

То есть $break (any smallest) rows$ разбивает матрицу на два списка строк, причём голова второго содержит клетку с минимальным числом вариантов. Далее, второй вызов $break$ делит строку на две подстроки, и голова второй, cs , представляет клетку с наименьшим числом вариантов. В конце при помощи подстановки каждого варианта из cs строится итоговый список матриц. Если в матрице найдётся пустой список вариантов, $expand1$ также вернёт пустой список.

Из определения n следует, что (19.7) выполняется только тогда, когда матрица вариантов имеет хотя бы один набор, содержащий не один вариант. Будем говорить, что матрица полна, если все наборы вариантов одноэлементные; матрица называется неудачной, если одноэлементные наборы вариантов в некоторой строке, столбце или площадке содержат повторяющиеся между собой варианты. Неполные неудачные матрицы никогда не приведут к верной сетке. Полная удачная матрица вариантов определяет единственную верную сетку. Эти два условия могут быть реализованы так:

$$complete = all (all single)$$

где $single$ это проверка того, что в списке содержится ровно один элемент;

$\text{safe } m = \text{all ok}(\text{rows } m) \wedge \text{all ok}(\text{cols } m) \wedge \text{all ok}(\text{boxs } m)$

где $\text{ok row} = \text{nodups}[d | [d] \leftarrow \text{row}]$.

Предполагая, что матрица неполная, но удачная, можем вычислить

$$\begin{aligned} & \text{filter valid} \cdot \text{expand} \\ = & \{ \text{поскольку для неполных матриц} \\ & \quad \text{expand} = \text{concat} \cdot \text{map expand} \cdot \text{expand1} \} \\ & \text{filter valid} \cdot \text{concat} \cdot \text{map expand} \cdot \text{expand1} \\ = & \{ \text{поскольку filter p} \cdot \text{concat} = \text{concat} \cdot \text{map}(\text{filter p}) \} \\ & \text{concat} \cdot \text{map}(\text{filter valid} \cdot \text{expand}) \cdot \text{expand1} \\ = & \{ \text{поскольку filter valid} \cdot \text{expand} = \text{filter valid} \cdot \text{expand} \cdot \text{prune} \} \\ & \text{concat} \cdot \text{map}(\text{filter valid} \cdot \text{expand} \cdot \text{prune}) \cdot \text{expand1} \end{aligned}$$

Введя $\text{search} = \text{filter valid} \cdot \text{expand} \cdot \text{prune}$, для удачных неполных матриц получим

$$\text{search} \cdot \text{prune} = \text{concat} \cdot \text{map search} \cdot \text{expand1}$$

Теперь мы можем представить третью версию *solve*:

$$\begin{aligned} \text{solve} &= \text{search} \cdot \text{choices} \\ \text{search } m &\mid \text{not}(\text{safe } m) = [] \\ &\mid \text{complete } m' = [\text{map}(\text{map head}) m'] \\ &\mid \text{otherwise} = \text{concat}(\text{map search}(\text{expand1 } m')) \\ &\mid \text{where } m' = \text{prune } m \end{aligned}$$

Это окончательная версия нашего простого алгоритма решения судоку.

Заключительные замечания

Мы протестировали наш алгоритм на 36 головоломках, приведённых на сайте <http://haskell.org/haskellwiki/Sudoku>. Он решил их за 8.8 секунды (на компьютере с 1 GHz Pentium 3). Мы также проверили его на шести самых маленьких версиях судоку (каждая с 17 непустыми клетками), выбранных случайно из 32000 штук, помешённых на том же сайте. Алгоритм справился с ними за 111.4 секунды. На указанном сайте размещено около дюжины различных вариантов решений судоку на Haskell. Все они, включая славный алгоритм Леннарта Аугустссона (Lennart Augustsson), используют вычисления в координатах. Многие используют массивы, и

большинство — монады. Наша версия в два раза медленней, чем алгоритм Аугустссона, на «гнусном примере» (особо сложной задаче с минимальными 17 заполненными клетками), но примерно в 30 раз быстрее, чем алгоритм Ица Гейла (Yitz Gale), на простых задачах. Нам известны алгоритмы, которые сводят головоломку к задаче проверки выполнимости булевой формулы, поиска с ограничениями, проверки на моделях и так далее. Я бы сказал, что алгоритм, представленный выше, действительно один из простейших и наиболее коротких. И, по меньшей мере, он был получен частично с помощью эквациональных рассуждений.

20

Задача «Обратного отсчёта»

Введение

«Обратный отсчёт» это название игры из популярной телепрограммы на британском телевидении; во Франции она называется «Сказка хороша» (*Le Conte est Bon*). Участники получают шесть исходных чисел, необязательно различных, и одно целевое число, все числа представляют из себя положительные целые. Цель в том, чтобы составить из исходных чисел с помощью арифметических операций выражение, значение которого будет как можно ближе к целевому числу. Выражения строятся с помощью четырёх простейших операций сложения, вычитания, умножения и деления. Участникам даётся 30 секунд на раздумья. К примеру, для исходных чисел $[1, 3, 7, 10, 25, 50]$ и целевого 831 точного решения не существует; одно из выражений, которое подбирается ближе всего: $7 + (1 + 10) \times (25 + 30) = 832$. Наша цель в этой жемчужине описать различные способы решения «Обратного отсчёта», каждый из которых тем или иным образом полагается на полный перебор. Игра привлекательна как учебный пример на тему переборных алгоритмов, потому что легко формулируется, а различные её решения показывают разные формы компромисса между памятью и временем, что играет важную роль при сравнении программ на функциональных языках программирования.

Простое решение

Вот незамысловатый алгоритм для «Обратного отсчёта»:

$$\begin{aligned} \text{countdown1} &:: \text{Int} \rightarrow [\text{Int}] \rightarrow (\text{Expr}, \text{Value}) \\ \text{countdown1 } n &= \text{nearest } n \cdot \text{concatMap } \text{mkExprs} \cdot \text{subseqs} \end{aligned}$$

Прежде всего, исходные числа заданы списком, их порядок не важен, а вот повторы имеют значение. Предположим, что числа отсортированы по возрастанию, это будет использовано позднее. Каждый вариант отбора чисел представляется некоторой подпоследовательностью. Для каждой подпоследовательности xs определяются всевозможные арифметические выражения, полученные из неё вместе со значениями этих выражений¹. Результаты собираются в один список, из которого выбирается ближайший к целевому числу ответ.

Составные части countdown1 определяются следующим образом. Для начала приведём subseqs , которая возвращает список всех непустых подпоследовательностей непустого списка:

$$\begin{aligned} \text{subseqs } [x] &= [[x]] \\ \text{subseqs } (x : xs) &= xss \uparrow\uparrow [x] : \text{map } (x:) xss \\ &\quad \text{where } xss = \text{subseqs } xs \end{aligned}$$

Далее, объявим типы данных для выражений и их значений:

$$\begin{aligned} \text{data Expr} &= \text{Num Int} \mid \text{App Op Expr Expr} \\ \text{data Op} &= \text{Add} \mid \text{Sub} \mid \text{Mul} \mid \text{Div} \\ \text{type Value} &= \text{Int} \end{aligned}$$

Значение выражений вычисляется так:

$$\begin{aligned} \text{value} &\quad :: \text{Expr} \rightarrow \text{Value} \\ \text{value } (\text{Num } x) &= x \\ \text{value } (\text{App op e1 e2}) &= \text{apply op} (\text{value e1}) (\text{value e2}) \end{aligned}$$

где $\text{apply Add} = (+)$, $\text{apply Sub} = (-)$ и так далее. Однако не все выражения допустимы для «Обратного отсчёта». Например, результат вычитания должен быть положительным, деление разрешено лишь тогда, когда оно может быть проведено нацело без остатка. Выражение допустимо, если допустимы его подвыражения и если самая внешняя операция проходит проверку *legal*:

¹Строго говоря, нет необходимости возвращать и выражения, и их значения, так как последние могут быть посчитаны из первых. Но, как мы видели в главе «Делаем сотню» (жемчужина 6), стоит избегать повторяющихся вычислений, так что, наученные опытом, эту оптимизацию включим с самого начала.

```

legal          :: Op → Value → Value → Bool
legal Add v1 v2 = True
legal Sub v1 v2 = (v2 < v1)
legal Mul v1 v2 = True
legal Div v1 v2 = (v1 mod v2 == 0)

```

Следующая составляющая это *mkExprs*, она создаёт список всех допустимых выражений, которые могут быть построены из данной подпоследовательности:

```

mkExprs      :: [Int] → [(Expr, Value)]
mkExprs [x]  = [(Num x, x)]
mkExprs xs   = [ev | (ys, zs) ← unmerges xs,
                  ev1 ← mkExprs ys,
                  ev2 ← mkExprs zs,
                  ev ← combine ev1 ev2]

```

Для данного упорядоченного списка *xs* длины больше, чем единица, *unmerges xs* это список всех пар *(ys, zs)* непустых списков, таких что *merge ys zs = xs*, где *merge* соединяет два упорядоченных списка в один (спецификация *unmerges* требует упорядоченности аргументов). Один из способов определения *unmerges* таков:

```

unmerges      :: [a] → [[a], [a]]
unmerges [x, y] = [[x], [y]], ([y], [x])
unmerges (x : xs) = [[x], xs], (xs, [x]) ++
                     concatMap (add x) (unmerges xs)
                     where add x (ys, zs) = [(x : ys, zs), (ys, x : zs)]

```

Поучительным упражнением станет вычисление *unmerges* из её спецификации, мы оставляем это удовольствие читателю.

Функция *combine* определяется так:

```

combine :: (Expr, Value) → (Expr, Value) → [(Expr, Value)]
combine (e1, v1) (e2, v2)
= [(App op e1 e2, apply op v1 v2) | op ← ops, legal op v1 v2]

```

где *ops = [Add, Sub, Mul, Div]*.

Наконец, функция *nearest n* принимает непустой список выражений и возвращает выражение, значение которого ближе всего к *n*. Этот поиск имеет смысл останавливать, если нашлось выражение, значение которого в точности равно *n*:

$\text{nearest } n ((e, v) : \text{evs})$	$= \text{if } d == 0 \text{ then } (e, v)$ $\quad \text{else search } n \ d (e, v) \ \text{evs}$ $\quad \text{where } d = \text{abs} (n - v)$
$\text{search } n \ d \ \text{ev} []$	$= \text{ev}$
$\text{search } n \ d \ \text{ev} ((e, v) : \text{evs})$	$ \quad d' == 0 = (e, v)$ $ \quad d' < d = \text{search } n \ d' (e, v) \ \text{evs}$ $ \quad d' \geq d = \text{search } n \ d \ \text{ev} \ \text{evs}$ $\quad \text{where } d' = \text{abs} (n - v)$

В GHCi (версия 6.8.3 на 2394 Mhz ноутбуке под управлением Windows XP) получились такие результаты:

```
> display (countdown1 831 [1,3,7,10,25,50])
(7+((1+10)*(25+50))) = 832
(42.28 secs, 4198816144 bytes)
> length $ concatMap mkExprs $ subseques [1,3,7,10,25,50]
4672540
```

Функции *countdown1* потребовалось 42 секунды, чтобы построить и проанализировать около 4.5 миллионов выражений, около 100 000 выражений в секунду. Это не укладывается в тридцатисекундный лимит времени, а значит, недостаточно хорошо.

Две оптимизации

Имеются две простые оптимизации, которые помогут улучшить положение дел. Первая касается проверки допустимости выражения. Среди примерно 33 миллионов выражений, которые можно построить из шести чисел, содержится лишь от четырёх до пяти миллионов (в зависимости от конкретных значений входных чисел) допустимых. Однако тут присутствует огромная избыточность. Например, каждая из следующих пар представляет по сути одинаковые выражения:

$$x + y \text{ и } y + x, \quad x * y \text{ и } y * x, \quad (x - y) + z \text{ и } (x + z) - y$$

Более точная проверка допустимости реализуется так:

```
legal Add v1 v2 = (v1 ≤ v2)
legal Sub v1 v2 = (v2 < v1)
legal Mul v1 v2 = (1 < v1) ∧ (v1 ≤ v2)
legal Div v1 v2 = (1 < v2) ∧ (v1 mod v2 == 0)
```

```

comb1 (e1, v1) (e2, v2)
= [(App Add e1 e2, v1 + v2), (App Sub e2 e1, v2 - v1)] ++
  if 1 < v1 then
    [(App Mul e1 e2, v1 * v2)] ++
    [(App Div e2 e1, q) | r = 0]
  else []
  where (q, r) = divMod v2 v1
comb2 (e1, v1) (e2, v2)
= [(App Add e1 e2, v1 + v2)] ++
  if 1 < v1 then
    [(App Mul e1 e2, v1 * v2), (App Div e1 e2, 1)]
  else []

```

Рис. 20.1: определения *comb1* и *comb2*

Эта версия учитывает коммутативность $+$ и $*$ с помощью требования упорядоченности аргументов, а также учитывает свойства нейтральности $*$ и $/$ с помощью запрета на единичные значения аргументов. Эта проверка снижает число допустимых выражений примерно до 300 000. Можно было бы пойти дальше и продолжить усиливать эту проверку, но мы оставим это для следующего раздела.

Вторая оптимизация касается *unmerges* и *combine*. По определению выше *unmerges xs* возвращает все пары (ys, zs) , такие что $\text{merge } ys \text{ } zs = xs$, а это означает, что каждая пара порождается дважды: один раз в виде (ys, zs) и один раз — в виде (zs, ys) . Нет причин удваивать работу, поэтому переопределим *unmerges* так, чтобы она возвращала только существенно различные пары списков:

```

unmerges [x, y] = [[x], [y]]
unmerges (x : xs) = [[x], xs] ++
  concatMap (add x) (unmerges xs)
  where add x (ys, zs) = [(x : ys, zs), (ys, x : zs)]

```

Функция *combine* может быть легко переписана с учётом новой версии *unmerges*:

```

combine (e1, v1) (e2, v2)
= [(App op e1 e2, apply op v1 v2) | op  $\leftarrow$  ops, legal op v1 v2] ++
  [(App op e2 e1, apply op v2 v1) | op  $\leftarrow$  ops, legal op v2 v1]

```

Однако более быстрое решение состоит во включении усиленной проверки допустимости прямо в определение *combine*:

$$\begin{aligned} \text{combine } (e1, v1) \ (e2, v2) \\ | \ v1 < v2 = \text{comb1 } (e1, v1) \ (e2, v2) \\ | \ v1 == v2 = \text{comb2 } (e1, v1) \ (e2, v2) \\ | \ v1 > v2 = \text{comb1 } (e2, v2) \ (e1, v1) \end{aligned}$$

Функция *comb1* используется в случае, когда значение первого выражения строго больше значения второго, а *comb2* — когда они равны. Определения этих функций приведены на рис. 20.1. Проведя такие изменения, получим функцию *countdown2*, определение которой в остальном не отличается от *countdown1*. Вот её результаты:

```
> display (countdown2 831 [1,3,7,10,25,50])
(7+((1+10)*(25+50))) = 832
(1.77 secs, 168447772 bytes)
> length $ concatMap mkExprs $ subseqs [1,3,7,10,25,50]
240436
```

Уже лучше, ведь потребовалось всего около двух секунд на определение и анализ около 250 000 выражений, но улучшения ещё не закончены.

Ещё более строгая проверка допустимости

В попытке ещё сильней ограничить число допустимых выражений определим выражение в *нормальной форме* как имеющее следующий вид:

$$[(e_1 + e_2) + \dots + e_m] - [(f_1 + f_2) + \dots + f_n],$$

где $m \geq 1$ и $n \geq 0$, обе последовательности выражений e_1, e_2, \dots , и f_1, f_2, \dots упорядочены в порядке увеличения их значений, а каждое e_i и f_j имеет вид

$$[(g_1 * g_2) * \dots * g_p]/[(h_1 * h_2) * \dots * h_q],$$

где $p \geq 1$ и $q \geq 0$, обе последовательности выражений g_1, g_2, \dots , и h_1, h_2, \dots упорядочены в порядке увеличения их значений, а каждое g_i и h_j представляет собой число либо выражение в нормальной форме.

С точностью до перестановки подвыражений с равными значениями каждое выражение имеет единственную нормальную форму. Среди 300 000 выражений из шести чисел, допустимых в смысле прошлого определения,

только от 30 00 до 70 000 находятся в нормальной форме. Однако нормальная форма не устраниет избыточность полностью. Например, выражения $2 + 5 + 7$ и $2 * 7$ имеют одинаковое значение, но второе составлено из чисел, представляющих подпоследовательность чисел первого. То есть рассматривать первое вовсе не нужно. Однако мы не будем исследовать дальнейшие способы сокращения числа порождаемых выражений. Эксперименты показывают, что эта игра не стоит свеч: выигрыш, получаемый сокращением до действительно существенных выражений, меньше, чем усилия для их выявления.

Мы можем отобрать выражения в нормальной форме, усилив проверку допустимости, но на этот раз придётся рассматривать сами выражения, а не только их значения. Для начала определим функцию *non*:

$$\begin{aligned} \textit{non} &:: \textit{Op} \rightarrow \textit{Expr} \rightarrow \textit{Bool} \\ \textit{non op} (\textit{Num } x) &= \textit{True} \\ \textit{non op1} (\textit{App op2 } e1 \textit{ e2}) &= \textit{op1} \neq \textit{op2} \end{aligned}$$

Тогда усиленная проверка допустимости может быть определена так:

$$\begin{aligned} \textit{legal} &:: \textit{Op} \rightarrow (\textit{Expr}, \textit{Value}) \rightarrow (\textit{Expr}, \textit{Value}) \rightarrow \textit{Bool} \\ \textit{legal Add} (e1, v1) (e2, v2) &= (v1 \leq v2) \wedge \textit{non Sub } e1 \wedge \textit{non Add } e2 \wedge \textit{non Sub } e2 \\ \textit{legal Sub} (e1, v1) (e2, v2) &= (v2 < v1) \wedge \textit{non Sub } e1 \wedge \textit{non Sub } e2 \\ \textit{legal Mul} (e1, v1) (e2, v2) &= (1 < v1 \wedge v1 \leq v2) \wedge \textit{non Div } e1 \wedge \textit{non Mul } e2 \wedge \textit{non Div } e2 \\ \textit{legal Div} (e1, v1) (e2, v2) &= (1 < v2 \wedge v1 \bmod v2 == 0) \wedge \textit{non Div } e1 \wedge \textit{non Div } e2 \end{aligned}$$

Как и прежде, встроим проверку допустимости в новую версию *combine*. Необходимо только поменять определения *comb1* и *comb2*. Уточнённые определения приведены на рис. 20.2.

Используя эти изменения для функции *countdown3*, получим:

```
> display (countdown3 831 [1,3,7,10,25,50])
(7+((1+10)*(25+50))) = 832
(1.06 secs, 88697284 bytes)
> length $ concatMap mkExprs $ subsequs [1,3,7,10,25,50]
36539
```

Всего за одну секунду проверяется около 36 000 выражений, что примерно в два раза быстрее, чем *countdown2*.

```

comb1 (e1, v1) (e2, v2)
= (if non Sub e1 ∧ non Sub e2 then
  [(App Add e1 e2, v1 + v2) | non Add e2] ++ [(App Sub e2 e1, v2 - v1)]
  else []) ++
  (if 1 < v1 ∧ non Div e1 ∧ non Div e2 then
  [(App Mul e1 e2, v1 * v2) | non Mul e2] ++ [(App Div e2 e1, q) | r 0]
  else [])
  where (q, r) = divMod v2 v1
comb2 (e1, v1) (e2, v2)
= [(App Add e1 e2, v1 + v2) | non Sub e1, non Add e2, non Sub e2] ++
  (if 1 < v1 ∧ non Div e1 ∧ non Div e2 then
  [(App Mul e1 e2, v1 * v2) | non Mul e2] ++ [(App Div e1 e2, 1)]
  else [])

```

Рис. 20.2: новые определения *comb1* и *comb2*

Мемоизация

Даже без учёта избыточности во множестве порождаемых выражений вычисления повторяются, так как каждая подпоследовательность рассматривается как независимая задача. Например, для исходных чисел $[1..6]$ выражения на основе чисел $[1..5]$ будут вычисляться дважды: один раз для подпоследовательности $[1..5]$, а другой — для $[1..6]$. Выражения на основе $[1..4]$ построятся четырежды, по разу для каждой из подпоследовательностей:

$$[1, 2, 3, 4], \quad [1, 2, 3, 4, 5], \quad [1, 2, 3, 4, 6], \quad [1, 2, 3, 4, 5, 6]$$

По сути, выражения на основе k чисел из n исходных будут посчитаны 2^{n-k} раз.

Один из способов избежать повторяющихся вычислений состоит в мемоизации функции *mkExprs*. В этом случае исходящая структура *mkExprs* сохраняется, но полученные результаты сохраняются в специальной таблице для последующего использования. Для реализации этой идеи понадобится тип данных *Memo*, который поддерживает такие операции:

```

empty :: Memo
fetch :: Memo → [Int] → [(Expr, Value)]
store :: [Int] → [(Expr, Value)] → Memo → Memo

```

Значение *empty* соответствует пустой таблице, *fetch* принимает список исходных чисел и проверяет наличие вычисленных выражений для этого списка в таблице, а *store* принимает аналогичный список чисел вместе со списком выражений, построенных из этих чисел, и сохраняет эту информацию в таблице.

Соответствующее определение *mkExprs* выглядит так:

```
mkExprs           :: Memo → [Int] → [(Expr, Value)]
mkExprs memo [x] = [(Num x, x)]
mkExprs memo xs   = [ev | (ys, zs) ← unmerges xs,
                      ev1 ← fetch memo ys,
                      ev2 ← fetch memo zs,
                      ev ← combine ev1 ev2]
```

Здесь предполагается, что для любой подпоследовательности *xs* все арифметические выражения на основе *ys* и *zs* при любом таком разбиении *xs* вычислены и помечены заранее в таблицу. Это предположение справедливо, если порождение подпоследовательностей из исходных чисел обладает тем свойством, что для любых двух подпоследовательностей *xs*, *ys* исходных чисел, если *xs* является подпоследовательностью *ys*, то она появляется в списке раньше *ys*. К счастью, данное ранее определение *subseqs* этим свойством обладает. Теперь можно записать

```
countdown4     :: Int → [Int] → (Expr, Value)
countdown4 n = nearest n · extract · memoise · subseqs
```

где *memoise* определяется так:

```
memoise        :: [[Int]] → Memo
memoise        = foldl insert empty
insert memo xs = store xs (mkExprs memo xs) memo
```

Функция *extract* «сглаживает» таблицу, возвращая список выражений из неё. Приведём эту функцию ниже, как только определимся со структурой *Memo*.

Одна возможная реализация *Memo* может быть построена на основе префиксного дерева:

```
data Trie a = Node a [(Int, Trie a)]
type Memo = Trie [(Expr, Value)]
```

Префиксное дерево это разновидность сильно ветвящегося (не бинарного) дерева, известного также как «розовый куст», узлы которого помечены,

как в данном случае, целыми числами. Пустой таблице соответствует значение $\text{empty} = \text{Node} [] []$. Поиск в таблице производится в соответствии с метками узлов:

$$\begin{array}{lll} \text{fetch} & :: \text{Memo} \rightarrow [\text{Int}] \rightarrow [(\text{Expr}, \text{Value})] \\ \text{fetch} (\text{Node es } xms) [] & = \text{es} \\ \text{fetch} (\text{Node es } xms) (x : xs) & = \text{fetch} (\text{follow } x \text{ } xms) \text{ } xs \\ \text{follow} & :: \text{Int} \rightarrow [(\text{Int}, \text{Memo})] \rightarrow \text{Memo} \\ \text{follow } x \text{ } xms & = \text{head} [m \mid (x', m) \leftarrow xms, x == x'] \end{array}$$

Отметим, что результат поиска в таблице записи, помеченной xs , не определён (равен голове пустого списка), если на префиксном дереве не существует пути, метки которого составляют xs . Но здесь нет проблемы, так как определение *subseqs* гарантирует, что записи таблицы вычисляются в нужном порядке, так что вся необходимая информация имеется к тому моменту, когда она может быть востребована.

Способ сохранения новых записей в таблице:

$$\begin{array}{ll} \text{store} :: [\text{Int}] \rightarrow [(\text{Expr}, \text{Value})] \rightarrow \text{Memo} \rightarrow \text{Memo} \\ \text{store } [x] \text{ } es \text{ } (\text{Node fs } xms) = \text{Node fs } ((x, \text{Node es } []) : xms) \\ \text{store } (x : xs) \text{ } es \text{ } (\text{Node fs } xms) \\ = \text{Node fs } (yms ++ (x, \text{store } xs \text{ } es \text{ } m) : zms) \\ \text{where } (yms, (z, m) : zms) = \text{break } (\text{equals } x) \text{ } xms \\ \text{equals } x \text{ } (z, m) = (x == z) \end{array}$$

Определение *store* предполагает, что в момент создания записи для списка $xs ++ [x]$ в таблице уже имеется запись для xs . Функция Haskell *break* была определена в прошлой жемчужине.

Наконец, мы научимся извлекать все записи из таблицы с помощью

$$\begin{array}{ll} \text{extract} & :: \text{Memo} \rightarrow [(\text{Expr}, \text{Value})] \\ \text{extract } (\text{Node es } xms) & = es ++ \text{concatMap } (\text{extract} \cdot \text{snd}) \text{ } xms \end{array}$$

Теперь получаем:

```
> display (countdown4 831 [1,3,7,10,25,50])
(10*((1+7)+(3*25))) = 830
(0.66 secs, 55798164 bytes)
```

В результате иного порядка обработки мы получили другое выражение по цене, почти в два раза меньшей, чем в случае *countdown3*.

Структурные деревья

Мемоизация *countdown* имеет свою цену: построение таблицы требует частого обращения к куче и намного больше времени тратится на сборку мусора. Как избавиться от больших затрат по памяти, сохранив преимущества мемоизации?

Попробуем сконцентрироваться на структуре арифметического выражения без учёта конкретных операций, которые в нём применяются. Сколько различных ориентированных бинарных деревьев можно построить в данном случае? В ориентированном дереве порядок поддеревьев не имеет значения. Мы уже использовали эту идею в «ориентированном» определении *unmerges*. Оказывается, имеется всего 1881 ориентированных бинарных деревьев на основе шести исходных чисел. Ориентированные бинарные деревья можно назвать также *структурными* деревьями. Для построения экономного в отношении памяти алгоритма можно было бы сперва строить такие деревья, а затем вставлять в их узлы конкретные операции.

Воплощая эту идею, введём тип данных бинарного дерева с метками только на листьях:

```
data Tree = Tip Int | Bin Tree Tree
```

Вместо мемоизации выражений будем заниматься мемоизацией деревьев:

```
type Memo = Trie [Tree]
```

Порождать деревья можно точно так же, как выражения:

```
mkTrees :: Memo → [Int] → [Tree]
mkTrees memo [x] = [Tip x]
mkTrees memo xs = [Bin t1 t2 | (ys, zs) ← unmerges xs,
                     t1 ← fetch memo ys,
                     t2 ← fetch memo zs]
```

Преобразовать дерево в список выражений возможно путём подстановки операций всеми допустимыми способами:

```
toExprs :: Tree → [(Expr, Value)]
toExprs (Tip x) = [(Num x, x)]
toExprs (Bin t1 t2) = [ev | ev1 ← toExprs t1, ev2 ← toExprs t2,
                           ev ← combine ev1 ev2]
```

В итоге получаем последнюю версию функции *countdown*:

Таблица 20.1. Время исполнения разных версий функции *countdown* в случаях шести, семи и восьми исходных чисел

Файл	<i>countdown1</i>		<i>countdown2</i>		<i>countdown3</i>		<i>countdown4</i>		<i>countdown5</i>	
	Всего	GC								
d6	1.56	0.78	0.19	0.08	0.09	0.05	0.08	0.02	0.05	0.00
d7	77.6	36.9	2.03	1.19	0.44	0.09	0.53	0.30	0.33	0.02
d8	—	—	99.8	57.2	13.8	7.30	16.9	9.02	7.22	0.31

countdown5 n

= nearest n · concatMap toExprs · extract · memoise · subseqs

в которой *memoise* определяется следующим образом:

```

memoise      :: [[Int]] → Memo
memoise      = foldl insert empty
insert memo xs = store xs (mkTrees memo xs) memo

```

Результат на нашем примере таков:

```

> display (countdown5 831 [1,3,7,10,25,50])
(10*((1+7)+(3*25))) = 830
(1.06 secs, 88272332 bytes)

```

Это выглядит доказательством того, что мемоизация структурных деревьев вместо выражений оказалась не такой уж хорошей идеей. Однако ситуация заслуживает более пристального рассмотрения.

Дальнейшие эксперименты

Посмотрим, как пять версий *countdown*, описанных выше, ведут себя в присутствии оптимизирующего компилятора. Мы скомпилировали пять программ с помощью GHC версии 6.8.3 с флагом -O2. Статистика была собрана системой времени выполнения GHC с флагом -s. Использовались три файла, *d6*, *d7* и *d8*, содержащих шесть, семь и восемь исходных чисел соответственно. В каждом случае заранее установлено, что точных совпадений быть не может, а потому будет обработано всё множество возможных выражений. Результаты собраны в таблице 20.1. Приведено общее время и время на сборку мусора (GC — Garbage Collection); всё — в секундах. Программа *countdown1* не проверялась на *d8*.

Три главных вывода, которые можно сделать из этих цифр, таковы. Первое и самое очевидное, компиляция имеет ощутимое преимущество перед интерпретацией. Второе, для шести и семи чисел нет заметной разницы между *countdown3* (версия со строгой проверкой допустимости), *countdown4* (версия со строгой проверкой допустимости и мемоизацией) и *countdown5* (версия со строгой проверкой допустимости и мемоизацией структурных деревьев вместо выражений). Однако для восьми исходных чисел итоговая версия, *countdown5*, стала ощутимо выигрывать, работая примерно в два раза быстрее в основном благодаря сократившемуся времени сборки мусора, которое заняло 5% от всего времени исполнения, в отличие от 50% для *countdown3* и *countdown4*.

Заключительные замечания

Эта жемчужина основана на материале из (Bird and Mu, 2005), где описана спецификация «Обратного отсчёта» в терминах отношений и на основе алгебраических законов для свёртки и развёртки (*unfold*) выведены несколько алгоритмов решения задачи. Эти вычисления здесь не использовались. Впервые «Обратный отсчёт» был исследован в ранней жемчужине (Hutton, 2002) в качестве иллюстрации того, как доказывать соответствие функциональных программ их спецификациям. Цель Хаттона была не в том, чтобы получить наилучший алгоритм, а в том, чтобы продемонстрировать тот, доказательство корректности которого потребует лишь обычной индукции. По сути доказательства в этой работе имели дело с программой *countdown2*.

Литература

- Bird, R. S. and Mu, S.-C. (2005). *Countdown*: a case study in origami programming. *Journal of Functional Programming* 15 (6), 679–702.
- Hutton, G. (2002). The *Countdown* problem. *Journal of Functional Programming* 12 (6), 609–16.

21

Хиломорфизмы и нексусы

Введение

Понятие хиломорфизма для алгоритма, состоящего из свёртки, следующей за развёрткой, ввёл не кто иной, как Эрик Мейджер (Erik Meijer). Развёртка порождает структуру данных, а свёртка поглощает её. Эта промежуточная структура может быть исключена из вычислительного процесса, что называется *усечением* (deforestation). В результате получается шаблон рекурсивного алгоритма, который подходит для большинства случаев, встречающихся на практике. Тем не менее, промежуточная структура совсем не бесполезна. Она соответствует дереву рекурсивных вызовов хиломорфизма и может быть положена в основу альтернативной, иногда более быстрой его реализации. Ускорение происходит, когда дерево вызовов содержит разделяемые узлы, то есть те, которые имеют более одной входящей дуги. Такое «дерево» называется *нексус*, оно появляется в любом рекурсивном вычислении, где рекурсивно решаемые подзадачи имеют пересечения. Типичным примером здесь могут служить алгоритмы динамического программирования. Цель этой жемчужины в том, чтобы рассмотреть построение нексусов на двух-трёх занимательных примерах.

Свёртки, развёртки и хиломорфизмы

Вместо абстрактных рассуждений рассмотрим один пример промежуточной структуры данных, а именно дерево с помеченными листьями такого вида:

```
data Tree a = Leaf a | Node [Tree a]
```

Функции свёртки и развёртки для $\text{Tree } a$ полагаются на следующий изоморфизм:

$$\text{Tree } a \approx \text{Either } a [\text{Tree } a]$$

Чтобы свернуть дерево, необходимо предоставить функцию накопления с типом $\text{Either } a [b] \rightarrow b$, а для развёртки в дерево потребуется функция с типом $b \rightarrow \text{Either } a [b]$. Более точно,

$fold$	$:: (\text{Either } a [b] \rightarrow b) \rightarrow \text{Tree } a \rightarrow b$
$fold f t$	$= \text{case } t \text{ of}$
	$\quad \text{Leaf } x \rightarrow f(\text{Left } x)$
	$\quad \text{Node } ts \rightarrow f(\text{Right}(\text{map}(\text{fold } f) ts))$
$unfold$	$:: (b \rightarrow \text{Either } a [b]) \rightarrow b \rightarrow \text{Tree } a$
$unfold g x$	$= \text{case } g x \text{ of}$
	$\quad \text{Left } y \rightarrow \text{Leaf } y$
	$\quad \text{Right } xs \rightarrow \text{Node}(\text{map}(\text{unfold } g) xs)$

Определение $hylo f g = fold f \cdot unfold g$ после усечения будет выглядеть так:

$hylo f g x$	$= \text{case } g x \text{ of}$
	$\quad \text{Left } y \rightarrow f(\text{Left } y)$
	$\quad \text{Right } xs \rightarrow f(\text{Right}(\text{map}(\text{hylo } f g) xs))$

Вид такого рекурсивного определения выглядит не слишком знакомым в основном из-за участия типа Either , затемняющего существо дела. Упростим этот код, не теряя общности. Функция с типом $\text{Either } a [b] \rightarrow b$ может быть представлена в виде двух функций, что повлияет на определение $fold$:

$fold$	$:: (a \rightarrow b) \rightarrow ([b] \rightarrow b) \rightarrow \text{Tree } a \rightarrow b$
$fold f g (\text{Leaf } x)$	$= f x$
$fold f g (\text{Node } ts)$	$= g(\text{map}(\text{fold } f g) ts)$

Аналогично, функцию с типом $b \rightarrow \text{Either } a [b]$ можно заменить тремя функциями и поменять определение $unfold$ соответствующим образом:

$unfold$	$:: (b \rightarrow \text{Bool}) \rightarrow (b \rightarrow a) \rightarrow (b \rightarrow [b]) \rightarrow b \rightarrow \text{Tree } a$
$unfold p v h x$	$= \text{if } p x \text{ then Leaf } (v x) \text{ else}$
	$\quad \text{Node}(\text{map}(\text{unfold } p v h) (h x))$

С использованием этих новых определений $hylo = fold f g \cdot unfold p v h$ будет усечён следующим образом:

$$hylo x = \text{if } p x \text{ then } f (v x) \text{ else } g (\text{map } hylo (h x))$$

Это уже лучше, но теперь видно, что функция v не требуется, так как её действие можно включить в определение f . Удаление v даст

$$hylo x = \text{if } p x \text{ then } f x \text{ else } g (\text{map } hylo (h x)) \quad (21.1)$$

что является общей формой хиломорфизма над $\text{Tree } a$. Словом, если аргумент x элементарен ($p x$), результат $f x$ вычисляется непосредственно; иначе x раскладывается на подзадачи ($h x$), вычисляется результат каждой из них ($\text{map } hylo (h x)$), и все результаты собираются функцией g . Мы пришли к привычному рекурсивному вычислению.

Определение (21.1) даёт усечённую версию выражения $fold f g \cdot unfold pid h$. В противовес усечению рассмотрим идею аннотирования: в соответствии с ней структура дерева сохраняется до конца вычислений, при этом к каждому узлу дерева добавляются метки, которые представляют значение хиломорфизма поддерева, определяемого данным узлом. Более точно, определим размеченный вариант Tree под названием $LTree$:

$$\text{data } LTree a = LLeaf a | LNode a [LTree a]$$

Затем определим функцию $fill$:

$$\begin{aligned} fill &:: (a \rightarrow b) \rightarrow ([b] \rightarrow b) \rightarrow \text{Tree } a \rightarrow LTree b \\ fill f g &= fold (\text{lleaf } f) (\text{lnode } g) \end{aligned}$$

где «умные» конструкторы lleaf и lnode заданы так:

$$\begin{aligned} \text{lleaf } f x &= LLeaf (f x) \\ \text{lnode } g ts &= LNode (g (\text{map } \text{label } ts)) ts \end{aligned}$$

а label так:

$$\begin{aligned} \text{label } (LLLeaf x) &= x \\ \text{label } (LNode x ts) &= x \end{aligned}$$

Функция $fill$ перерабатывает дерево в размеченное дерево, сохраняя структуру, при этом метка каждого узла является результатом свёртки поддерева с корнем в этом узле. Метка корня дерева даёт результат всего хиломорфизма, так что:

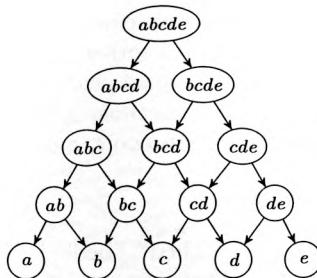


Рис. 21.1: нексус

$$hylo = \text{label} \cdot \text{fill } f \cdot g \cdot \text{unfold } p \text{ id } h$$

Это определение подводит нас к центральной идеи жемчужины. Предположим, что $\text{unfold } p \text{ id } h$ представляет собой нексус, а $\text{fill } f \cdot g$ можно применить к нему, не разрушая связей, которые отвечают за разделение узлов. В этом случае $hylo$ может быть вычислена более эффективно, чем рекурсивным методом (21.1).

Остаётся разобраться, как эта идея работает на практике. Во всех примерах ниже x из (21.1) представляет собой непустой список, а p — проверку на то, что список содержит ровно один элемент. Таким образом, все наши хиломорфизмы имеют вид:

$$\begin{aligned} hylo &:: ([a] \rightarrow b) \rightarrow ([b] \rightarrow b) \rightarrow ([a] \rightarrow [[a]]) \rightarrow [a] \rightarrow b \\ hylo \, f \, g \, h &= \text{fold } f \, g \cdot \text{mkTree } h \end{aligned}$$

где $\text{mkTree } h = \text{unfold single id } h$, а *single* это проверка включения ровно одного элемента. В частности, h принимает список хотя бы из двух элементов и возвращает список непустых списков.

Три примера

Для первого примера положим $h = \text{split}$, где

$$\text{split } xs = [\text{take } n \, xs, \text{drop } n \, xs] \text{ where } n = \text{length } xs \text{ div } 2$$

В случае списка xs длины 2^n результат $\text{mkTree split } xs$ представляет собой совершенное бинарное дерево из $2^n - 1$ внутренних узлов и 2^n листьев, которые помечены одноэлементными списками, по одному листу для каждого

элемента xs . Здесь отсутствует разделение узлов, и потому нексус будет иметь такое же количество узлов и листьев. В качестве конкретного примера укажем *hylo id merge split*, который реализует стандартный алгоритм сортировки слиянием только для списков, длина которых равна степени двойки. Сравним это со вторым примером, для которого $h = isegs$, где

$$isegs \ xs \ = \ [init \ xs, tail \ xs]$$

Например, $isegs "abcde" = ["abcd", "bcde"]$. Функция *isegs* (от англ. immediate segments) возвращает список последовательных отрезков списка длины не меньше двух. Результат *mkTree isegs xs* снова представляет собой совершенное бинарное дерево, листья которого помечены одноэлементными списками. Оно имеет $2^n - 1$ узлов, где $n = length \ xs$, так что в лучшем случае именно столько времени уйдёт на построение. Однако, в отличие от первого примера, поддеревья могут быть разделены между несколькими узлами, что даёт нам нексус, пример которого приведён на рис. 21.1. Нексус размечен различными непустыми отрезками строки *abcde*. Более точно, он был заполнен с помощью *fill id recover*, где

$$\begin{aligned} recover &:: [[a]] \rightarrow [a] \\ recover \ xs &= head \ (head \ xs) : last \ xs \end{aligned}$$

Функция *recover* удовлетворяет условию $recover \cdot isegs = id$. Нексус имеет $n(n + 1)/2$ узлов для входа размером n , то есть разделение узлов даёт ощущимое сокращение размера дерева.

Для третьего примера положим $h = minors$, где

$$\begin{aligned} minors \ [x, y] &= [[x], [y]] \\ minors \ (x : xs) &= map \ (x:) \ (minors \ xs) ++ [xs] \end{aligned}$$

Например, $minors "abcde" = ["abcd", "abce", "abde", "acde", "bcde"]$. Функция *minors* возвращает те подпоследовательности заданного списка, в которых исключён ровно один его элемент¹. Результат *mkTree minors xs*, если xs это непустой список, снова не что иное, как дерево, в листах которого находятся одноэлементные списки. Для входа длины n дерево имеет размер $S(n)$, где $S(0) = 0$ и $S(n+1) = 1 + (n+1)S(n)$. Решая рекуррентное соотношение, получаем:

$$S(n) = n! \sum_{k=1}^n \frac{1}{k!}$$

¹Функция *minors* встретится нам в следующих двух жемчужинах в связи с матрицами.

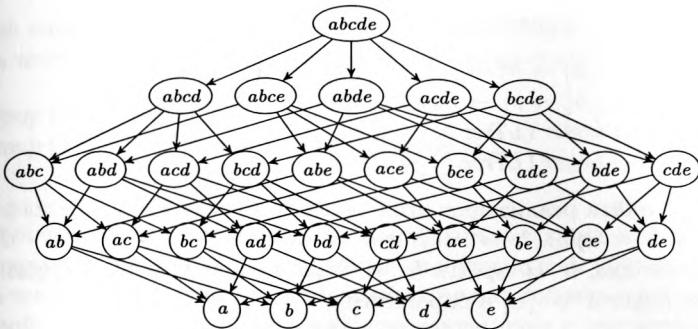


Рис. 21.2: ещё один нексус

так что $S(n)$ заключена между $n!$ и $en!^2$. Снова есть возможность для разделения узлов, соответствующий нексус представлен на рис. 21.2. Этот нексус размечен непустыми подпоследовательностями строки $abcde$; более точно, он был заполнен с помощью *fill id recover*, где функция *recover* взята из второго примера. Нексус имеет всего $2^n - 1$ узлов — намного меньше, чем в было в дереве.

Построение нексуса

Теперь обратимся к задаче построения нексуса и заполнения его с учётом информации о разделяемых узлах. Во всех наших примерах все листья каждого нексуса расположены на одной и той же глубине, так что одна очевидная мысль заключается в том, чтобы строить нексус $\text{fill } f \ g \cdot \text{mkTree } h$ снизу вверх уровней за уровнем. Процесс завершается, когда достигнут уровни с одним элементом. Казалось бы также очевидным, что каждый уровень должен представлять из себя список помеченных деревьев, то есть $[LTree a]$, но не будем торопиться. Вместо этого введём тип *Layer a* и определим представление каждого уровня как *Layer* ($LTree a$). Восходящая схема создания нексуса $\text{fill } f \ g \cdot \text{mkTree } h$ в общем виде для произвольной h выглядит так:

$$\text{mkNexus } f \ g = \text{label} \cdot \text{extractL} \cdot \text{until singleL} (\text{stepL } g) \cdot \text{initialL}$$

²Эта последовательность имеет номер A002627 в списке целочисленных последовательностей Слоана, точная формула выглядит так: $S(n) = \lfloor (e - 1)n! \rfloor$.

вспомогательные функции имеют типы:

$$\begin{aligned} initialL &:: ([a] \rightarrow b) \rightarrow [a] \rightarrow Layer(LTree b) \\ stepL &:: ([b] \rightarrow b) \rightarrow Layer(LTree b) \rightarrow Layer(LTree b) \\ singleL &:: Layer(LTree b) \rightarrow Bool \\ extractL &:: Layer(LTree b) \rightarrow LTree b \end{aligned}$$

Наша цель найти реализацию этих четырёх функций для каждого из трёх примеров, $h = split$, $h = isegs$ и $h = minors$.

Первые два, $h = split$ и $h = isegs$, не представляют труда: просто положим $Layer a = [a]$ и определим:

$$\begin{aligned} initialL f &= map(lleaf f \cdot wrap) \\ singleL &= single \\ extractL &= head \end{aligned}$$

где $wrap x = [x]$. Так что стартовый уровень это список листов. Определение $stepL$ для $h = split$ таково:

$$stepL g = map(lnode g) \cdot group$$

где $group :: [a] \rightarrow [[a]]$ собирает элементы списка по два и определена так:

$$\begin{aligned} group [] &= [] \\ group (x : y : xs) &= [x, y] : group xs \end{aligned}$$

С этими определениями $mkNexus id merge xs$ для списка xs , длина которого есть степень двойки, представляет из себя восходящее определение сортировки слиянием, при котором элементы объединяются сначала в упорядоченные пары, затем в четвёрки и т. д.

В случае $h = isegs$ нужно только поменять определение $group$:

$$\begin{aligned} group [x] &= [] \\ group (x : y : xs) &= [x, y] : group (y : xs) \end{aligned}$$

Таким образом, $group xs$ теперь возвращает список последовательных пар элементов из списка xs . Этот выбор становится очевиден при взгляде на рис. 21.1, формальное доказательство мы опускаем.

Как можно было ожидать, случай $h = minors$ существенно сложнее. Необходимо найти способ группировки деревьев на одном уровне для получения деревьев на следующем. Начнём с простого случая самого нижнего уровня, то есть списка листьев. Один способ объединить листья в пары покажем на примере пяти листьев a, b, c, d и e :

$$(ab\ ac\ ad\ ae)\ (bc\ bd\ be)\ (cd\ ce)\ (de)$$

Здесь ab означает $[a, b]$ и т. п. Без учёта информации о скобках второй уровень получается изменением определения *group* следующим образом:

$$\begin{aligned} \text{group } [x] &= [] \\ \text{group } (x : xs) &= \text{map} (\text{bind } x) xs \uparrow\! \text{group } xs \\ &\text{where bind } x y = [x, y] \end{aligned}$$

На следующем уровне придётся объединять пары в тройки, это можно сделать, если учитывать группы, созданные на предыдущем уровне и в явном виде на текущем уровне не представленные: нужно собрать в пары элементы первой группы и объединить каждую такую пару с соответствующим элементом оставшихся групп. Потом повторить процедуру с оставшимися группами. Это даст третий уровень:

$$((abc \ abd \ abe) \ (acd \ ace) \ (ade)) \ ((bcd \ bce) \ (bde)) \ ((cde))$$

где abc означает $[[a, b], [a, c], [b, c]]$ и т. п. Четвёртый уровень, то есть группы четырёрок

$$(((abcd \ abce) \ (abde)) \ ((acde))) \ (((bcde)))$$

получается аналогичным образом: собрать в тройки элементы первой группы и добавить к ним соответствующую тройку из остальных групп, а затем повторить этот процесс для оставшихся групп.

Необходимую информацию о группах можно сохранить, если представить каждый уровень как список деревьев, то есть лес. Форма этого леса ограничена, она может быть определена двумя числами: длиной, то есть числом деревьев n , и общей для всех деревьев глубиной d . Лес глубиной 0 состоит из списка листьев. Лес длиной n и глубиной $d + 1$ представляет собой список из n деревьев, в котором каждый ребёнок корня первого дерева является лесом длины n и глубины d , каждый ребёнок корня второго дерева — лесом длины $n - 1$ и глубины d , и так далее до последнего дерева, дети корня которого это лес длины 1 и глубины d . Нижний уровень нексуса представлен лесом длины n и глубины 0, следующий уровень — лесом длины $n - 1$ и глубины 1, и так далее до самого верхнего уровня, леса длины 1 и глубины $n - 1$. Тип данных для леса это $[\text{Tree } a]$, так что положим $\text{Layer } a = [\text{Tree } a]$. То, что мы начали жемчужину с определения типа *Tree a*, совершенно случайно, но нам всё равно бы пришлось дать это определение, чтобы описать лес.

Реализация *initialL*, *singleL* и *extractL* для построения нексуса в случае $h = \text{minors}$ приведена ниже:

```

initialL f = map (Leaf · lleaf f · wrap)
singleL   = single
extractL  = extract · head
    where extract (Leaf x)   = x
          extract (Node [t]) = extract t

```

Функция *initialL* создаёт лес глубины 0, метки которого взяты из листьев размеченного дерева (тип *LTree a*). Функция *extractL* принимает лес длины 1 и некоторой глубины *d* и извлекает его метки. Конечно, определённую сложность для понимания представляет то, что вычисления проводятся в терминах такой структуры данных, как [*Tree (LTree a)*], списка деревьев помеченных деревьев.

Остаётся определить функцию *stepL*, она задаётся так:

$$stepL g = map (mapTree (lnode g)) \cdot group$$

где *mapTree* это функция отображения для типа *Tree a* и

```

group :: [Tree a] → [Tree [a]]
group [t] = []
group (Leaf x : vs)
  = Node [Leaf [x, y] | Leaf y ← vs] : group vs
group (Node us : vs)
  = Node (zipWith combine (group us) vs) : group vs

combine (Leaf xs) (Leaf x)   = Leaf (xs ++ [x])
combine (Node us) (Node vs) = Node (zipWith combine us vs)

```

Эти определения формализуют описание, приведённое выше. Чтобы подтвердить их корректность, необходимо доказать, что

$$mkNexus f g = fill f g \cdot mkTree minors$$

Однако доказательство довольно объёмно, и мы опустим его.

Зачем строить нексусы?

Хороший вопрос. Всё, что было сказано о восходящем построении нексусов уровень за уровнем, остаётся справедливым, если отказаться от нексусов и просто продолжать рассчитывать значения меток. Возьмём случай *h = isegs* и рассмотрим функцию *solve*:

$$\begin{aligned} solve &:: [a] \rightarrow b \rightarrow ([b] \rightarrow b) \rightarrow [a] \rightarrow b \\ solve\ f\ g &= head \cdot until\ single\ (map\ g \cdot group) \cdot map\ (f \cdot wrap) \end{aligned}$$

для *group* выбрано определение, связанное с *isegs*. Функция *solve f g* реализует хиломорфизм *hylo f g isegs* без создания нексуса. Аналогично, пусть *solve* имеет вид

$$\begin{aligned} solve\ f\ g &= extractL \cdot until\ singleL\ (step\ g) \cdot map\ (Leaf \cdot f \cdot wrap) \\ step\ g &= map\ (mapTree\ g) \cdot group \end{aligned}$$

а *extractL*, *singleL* и *group* определены так, как это было показано для *minors*. Вновь *solve f g* представляет хиломорфизм *hylo f g minors* без построения нексуса.

Ответ на поставленный вопрос в том, что нексус полезен в задачах, которые можно рассматривать как разновидности задач обсуждавшихся выше. Например, в качестве стандартной задачи, связанной с сегментами списка, можно указать задачу оптимальной расстановки скобок, в которой требуется расставить скобки в выражении $x_1 \oplus x_2 \oplus \dots \oplus x_n$ наилучшим образом. Предполагается, что операция \oplus ассоциативна, так что способ расстановки не влияет на значение выражения. Однако разные варианты связаны с разными затратами. Стоимость вычисления $x \oplus y$ зависит от размеров x и y , а рекурсивное решение использует не *isegs*, а функцию *uncats*:

$$\begin{aligned} uncats\ [x, y] &= [[x], [y]] \\ uncats\ (x : xs) &= ([x], xs) : map\ (cons\ x)\ (uncats\ xs) \\ &\text{where } cons\ x\ (ys, zs) = (x : ys, zs) \end{aligned}$$

К примеру, *uncats "abcde"* равно

$$[("a", "bcde"), ("ab", "cde"), ("abc", "de"), ("abcd", "e")]$$

Каждая из этих пар соответствует одному из возможных способов расстановки самых внешних скобок, оптимальное решение получается рекурсивным подсчётом минимальной стоимости каждой компоненты каждой пары и вычислением минимума среди всех вариантов для объединения компонент.

Использование *uncats* вместо *isegs* приводит к тому, что работа ведётся с элементами более сложного, чем *Tree a*, типа, а именно, с разновидностью дерева, в которой каждое «поддерево» является списком пар поддеревьев. Тем не менее, задачу расстановки скобок можно решить вычислением нексуса для *isegs*, изменив умный конструктор *lnode* следующим образом:

$$\textit{lnode } g [u, v] = \textit{LNode} (g (\textit{zip} (\textit{lspine} u) (\textit{rspine} v))) [u, v]$$

Функции *lspine*, *rspine* :: *LTree a* → [*a*] определяются так:

$$\begin{aligned}\textit{lspine} (\textit{LLeaf} x) &= [x] \\ \textit{lspine} (\textit{LNode} x [u, v]) &= \textit{lspine} u ++ [x] \\ \textit{rspine} (\textit{LLeaf} x) &= [x] \\ \textit{rspine} (\textit{LNode} x [u, v]) &= [x] ++ \textit{rspine} v\end{aligned}$$

Например, левая и правая ось (spine) двух поддеревьев дерева на рис. 21.1 это [*a*, *ab*, *abc*, *abcd*] и [*bcd*, *cde*, *de*, *e*] соответственно. Их попарное объединение (*zip*) и даёт результат *uncats* "abcde". Данное определение *lspine* требует квадратичного времени, но с помощью аккумулирующего параметра можно легко обойтись и линейным временем.

В качестве второго примера вернёмся к некусу подпоследовательностей с рис. 21.2. Задача «Обратный отсчёт» из предыдущей жемчужины относится к классу задач, связанных с подпоследовательностями. Там мы использовали функцию *unmerges*, определяемую так:

$$\begin{aligned}\textit{unmerges} [x, y] &= [[x], [y]] \\ \textit{unmerges} (x : xs) &= [[x], xs] ++ \textit{concatMap} (\textit{add} x) (\textit{unmerges} xs) \\ \textbf{where} \quad \textit{add} x (ys, zs) &= [(x : ys, zs), (ys, x : zs)]\end{aligned}$$

Например, *unmerges* "abcd" равно

$$[("a", "bcd"), ("ab", "cd"), ("b", "acd"), ("abc", "d"), ("bc", "ad"), ("ac", "bd"), ("c", "abd")]$$

Порядок, в котором пары подпоследовательностей располагаются в этом списке, не важен, так же как и порядок компонент каждой пары. Действительно важно лишь то, что каждая подпоследовательность объединена в пару со своим дополнением. В «Обратном отсчёте» множество всех возможных выражений, которые могут быть составлены из данного списка *xs* целых чисел, расположенных по возрастанию, вычисляется рекурсивно, путём генерации всех выражений для каждого списка в каждой компоненте всех пар *unmerges xs* и последующего комбинирования результатов.

Использование *unmerges* вместо *minors* даёт более сложную структуру данных, чем *Tree a*. Тем не менее, как и раньше, «Обратный отсчёт» можно решить вычислением некуса для *minors*, если заменить умный конструктор *lnode*. В результате наша задача состоит в том, чтобы «извлечь» *unmerges* из меток некуса, связанного с каждым узлом. Это означает обработку каждой метки некуса. Это могло бы быть сделано с помощью

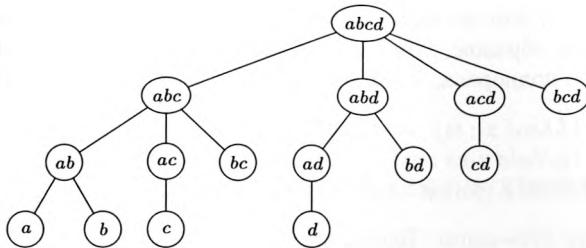


Рис. 21.3: биномиальное оствное дерево

обхода нексуса в ширину. Например, обход в ширину нексуса для `abcd` с рис. 21.2 за вычетом первого элемента выглядит так:

`abc, abd, acd, bcd, ab, ac, bc, ad, bd, cd, a, b, c, d`

Если этот список разбить на две половины и попарно объединить первую половину со второй, развернутой задом наперёд, получится результат *unmerges* "`abcd`", хотя пары будут следовать в другом порядке, так же как и компоненты в отдельных парах.

Однако при любом обходе графа требуется следить за тем, какие узлы были посещены, а это невозможно в случае нексуса, так как его узлы не могут быть проверены на равенство. Возможное решение этой проблемы состоит в том, чтобы сначала построить оствное дерево нексуса, и обходить в ширину уже его. Обход леса реализуется так:

```

traverse :: [LTree a] → [a]
traverse [] = []
traverse ts = map label ts ++ traverse (concatMap subtrees ts)
subtrees (LLeaf x) = []
subtrees (LNode x ts) = ts
  
```

Одно оствное дерево нексуса, связанного со строкой `abcd` рис. 21.2 изображено на рис. 21.3. Это биномиальное дерево ранга 4. Биномиальные деревья близкие родственники деревьев леса, использовавшегося при построении нексуса. Биномиальное дерево ранга n имеет n дочерних узлов, которые, в свою очередь, являются биномиальными деревьями рангов $n-1, n-2, \dots, 0$. Чтобы получить биномиальное дерево из нексуса, нужно удалить нуль дочерних вершин первого поддерева, одну вершину второго

поддерева и т. д. Эта же процедура повторяется рекурсивно для дочерних вершин. Таким образом, для k -го ребёнка необходимо исключить k детей из его первого поддерева, $k + 1$ из второго и т. д. Функция $forest k$

$$\begin{aligned} forest k (LLeaf x : ts) &= LLeaf x : ts \\ forest k (LNode x us : vs) \\ &= LNode x (forest k (drop k us)) : forest (k + 1) vs \end{aligned}$$

выполняет эти отсечения. Теперь можно определить

$$\begin{aligned} lnode g ts &= LNode (g (zip xs) (reverse ys)) ts \\ \text{where } (xs, ys) &= halve (traverse (forest 0 ts)) \end{aligned}$$

где $halve xs = splitAt (length xs \text{ div } 2) xs$.

Заключительные замечания

Название хиломорфизм впервые появилось в (Meijer, 1992), см. также (Meijer et al., 1991). Содержание этой жемчужины имеет два главных источника. Построение нексусов впервые было описано в (Bird and Hinze, 2003), где приводился другой способ создания нексуса для *minors*, он использовал циклические деревья с указателями вверх и вниз по дереву. Позже, в (Bird, 2008), было показано, что для некоторых задач — правда, из весьма ограниченного класса — основная функция для построения каждого уровня нексуса, *group*, может быть выражена как обратная к функции декомпозиции h хиломорфизма.

Литература

- Bird, R. S. and Hinze, R. (2003). Trouble shared is trouble halved. *ACM SIGPLAN Haskell Workshop*, Uppsala, Sweden.
- Bird, R. S. (2008). Zippy tabulations of recursive functions. In *LNCS 5133: Proceedings of the Ninth International Conference on the Mathematics of Program Construction*, ed. P. Audebaud and C. Paulin-Mohring. pp. 92–109.
- Meijer, E. (1992). Calculating compilers. PhD thesis, Nijmegen University, The Netherlands.
- Meijer, E., Fokkinga, M. and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. New York, NY: Springer-Verlag, pp. 124–44.

22

Три способа вычисления определителей

Введение

Определитель матрицы $A = (a_{ij})$ размеров $n \times n$, обозначаемый $\det(A)$ или $|A|$, может быть выражен формулой Лейбница:

$$|A| = \sum_{\pi} \text{sign}(\pi) \prod_{1 \leq j \leq n} a_{j\pi(j)}$$

Суммирование производится по всем перестановкам π чисел $[1 \dots n]$, а $\text{sign}(\pi) = 1$ для чётной перестановки (имеющей чётное число инверсий) и -1 для нечётной. Непосредственный подсчёт $|A|$ требует $\Theta(n \times n!)$ операций. Чтобы снизить сложность до $\Theta(n^3)$, можно преобразовать матрицу к верхнетреугольному виду с помощью метода Гаусса. Последний использует деление, и если на входе имеется целочисленная матрица, а определитель требуется подсчитать точно, то деление не должно вносить погрешностей. Следовательно, необходимо использовать рациональные дроби. Рациональная арифметика включает нормализацию числителей и знаменателей, то есть операцию подсчёта наибольшего общего делителя двух целых чисел.

Есть метод, позволяющий избежать рационального деления, это метод последовательной конденсации Чио (Chió's pivotal condensation). По сути это версия метода Гаусса, которая использует только целочисленное деление. Метод Чио требует $\Theta(n^3)$ умножений, но только $\Theta(n)$ делений (и возведений в степень). Недостатком является то, что размер промежуточных результатов растёт экспоненциально. Однако есть вариант этого

алгоритма, где размер выкладок удаётся свести к разумному объёму, при этом число целочисленных делений возрастает до $\Theta(n^3)$.

Наконец, известны достаточно быстрые методы подсчёта определителя, которые вообще не требуют деления. Например, метод, основанный на многократном умножении матриц. Промежуточные результаты требуют не так много памяти, но количество операций возрастает до $\Theta(n^4)$. Так как нам понадобится считать определители в следующей жемчужине, эту мы посвятим описанию и сравнению трёх перечисленных вариантов решения задачи.

Школьный метод

Для тренировки запрограммируем школьный метод вычисления определителей. Он связан с рекурсивным вычислением миноров данной матрицы. Например,

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{21} \begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} + a_{31} \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$$

Если матрица представлена списком строк, школьный метод реализуется так:

```
det      :: [[Integer]] → Integer
det [[x]] = x
det xss  = foldr1 (-) (zipWith (*) col1 (map det (minors cols)))
           where col1 = map head xss
                 cols = map tail xss
```

Случай 1×1 вычисляется непосредственно. В общем случае каждый элемент первой строки умножается на соответствующий минор, составленный из оставшихся столбцов, результаты суммируются с чередующимися знаками. Функция *minors*, впервые появившаяся в прошлой жемчужине, определяется так:

```
minors      :: [a] → [[a]]
minors []   = []
minors (x : xs) = xs : map (x:) (minors xs)
```

Например, *minors "abcd"* = *["bcd", "acd", "abd", "abc"]*.

Определение *det* привлекательно своей лаконичностью, однако соответствующие вычисления занимают экспоненциальное время. Рекуррентное соотношение для $T(n)$, числа шагов, необходимых для вычисления

определителя матрицы $n \times n$, выглядит так: $T(n) = nT(n - 1) + \Theta(n)$, его решение: $T(n) = \Theta(n!)$. Тем не менее, это вполне подходит для случаев $n = 2$ или $n = 3$.

Используем рациональное деление

Метод Гаусса полагается на то, что добавление любого кратного любой строки к любой другой строке не меняет значение определителя. Если первый элемент первой строки не нуль, добавление подходящих кратных первой строки к остальным строкам позволяет свести все остальные элементы первого столбца к нулям. Повторение данной процедуры к подматрице, образованной после удаления первой строки и столбца, приводит матрицу к верхнетреугольной форме. Определитель верхнетреугольной матрицы равен произведению элементов на диагонали.

Положение усложняется тем, что ведущий элемент может быть нулем. В этом случае необходимо найти подходящую строку, первый элемент которой, так называемый опорный элемент, не равен нулю. Функция *det* определяется так:

```
det :: [[Ratio Integer]] → Ratio Integer
det [[x]] = x
det xss =
  case break ((≠ 0) · head) xss of
    (yss, []) → 0
    (yss, zs : zss) → let x = head zs * det (reduce zs (yss ++ zss))
                        in if even (length yss) then x else -x
```

Выражение *break* ($((\neq 0) \cdot head)$) *xss* разбивает матрицу на две части, (yss, zss) , причём *zss* либо пусто, либо голова её первой строки *zs* не равна нулю. В первом случае матрица вырождена и её определитель равен нулю. В противном случае оставшиеся строки $(yss ++ zss)$ редуцируются к матрице $(n - 1) \times (n - 1)$ добавлением подходящих кратных строки *zs* к остальным строкам и удалением первого столбца:

<i>reduce xs yss</i>	= <i>map</i> (<i>reduce1 xs</i>) <i>yss</i>
<i>reduce1</i> (<i>x : xs</i>) (<i>y : ys</i>)	= <i>zipWith</i> ($\lambda a b \rightarrow b - d * a$) <i>xs ys</i>
	where <i>d</i> = <i>y / x</i>

Определитель редуцированной матрицы берётся со знаком минус, если позиция ведущей строки в матрице нечётна. Под операцией $(/)$ подразумевается рациональное деление.

Используем целочисленное деление

Ещё один способ подсчёта определителя $|A|$ основан на следующем факте. Определим матрицу X , положив $x_{jk} = a_{11} * a_{jk} - a_{1k} * a_{j1}$, где $2 \leq j, k \leq n$. По-другому:

$$x_{jk} = \begin{vmatrix} a_{11} & a_{1k} \\ a_{j1} & a_{jk} \end{vmatrix}$$

Матрица X имеет размеры $(n - 1) \times (n - 1)$. Тогда $|A| = |X|/a_{11}^{n-2}$, если $a_{11} \neq 0$. Это равенство Чио: определитель матрицы $n \times n$ выражен в терминах определителя «конденсированной» матрицы $(n - 1) \times (n - 1)$, составленной из определителей порядка два. Хотя равенство Чио использует деление, это точная операция и может выполняться полностью в целых числах. Обратим внимание на предположение о том, что a_{11} отлично от нуля. Если оно не выполнено, как и в методе Гаусса, приходится искать ненулевой опорный элемент. Страна, содержащая его, меняется местами с первой строкой. Такая перестановка меняет знак определителя на противоположный, если строка с опорным элементом перемещается на нечётное число позиций. Всё это приводит к следующему определению *det*:

```

det      :: [[Integer]] → Integer
det [[x]] = x
det xss =
  case break ((≠ 0) · head) xss of
    (yss, []) → 0
    (yss, zs : zss) → let x = det (condense (zs : yss ++ zss))
                        d = head zs ↑ (length xss - 2)
                        y = x div d
      in if even (length yss) then y else - y
  
```

В данном случае (\uparrow) обозначает возведение в степень. Функция *condense* определяется так:

```

condense = map (map det · pair · uncurry zip) · pair
  where pair (x : xs) = map ((,) x) xs
        det ((a, b), (c, d)) = a * d - b * c
  
```

Первая строка матрицы комбинируется в пару с каждой из оставшихся строк. Каждая пара строк $([a_1, a_2, \dots, a_n], [b_1, b_2, \dots, b_n])$, в свою очередь, порождает пары такого вида:

$[((a_1, b_1), (a_2, b_2)), ((a_1, b_1), (a_3, b_3)), \dots, ((a_1, b_1), (a_n, b_n))]$

В конце вычисляется определитель 2×2 , полученный из каждой пары пар.

Что касается сложности вычислений, конденсация матрицы $n \times n$ требует $\Theta(n^2)$ операций, то есть рекуррентное соотношение для $T(n)$ выглядит так: $T(n) = T(n - 1) + \Theta(n^2)$, которое имеет решение $T(n) = \Theta(n^3)$. Хотя рациональное деление не используется, целые числа в алгоритме растут очень быстро. Было бы разумней не откладывать деление до самого конца, а чередовать его с конденсацией.

Чередование

Чередование конденсации и деления возможно благодаря одному из многих любопытных свойств определителя. Пусть матрица X получена конденсацией матрицы A , и Y является результатом конденсации X . Тогда Y имеет размеры $(n - 2) \times (n - 2)$. В предположении $a_{11} \neq 0$ каждый элемент Y должен делиться на a_{11} . Доказательство остаётся в качестве упражнения. Сказанное означает, что множитель $1/a_{11}^{n-2}$ в методе Чио может быть исключён делением каждой дважды конденсированной матрицы на a_{11} . Это приводит к реализации $\det = \det' 1$, где

```

 $\det' :: \text{Integer} \rightarrow [[\text{Integer}]] \rightarrow \text{Integer}$ 
 $\det' k [[x]] = x$ 
 $\det' k xss =$ 
  case break ((\neq 0) \cdot head) xss of
    (yss, [])      \rightarrow 0
    (yss, zs : zss) \rightarrow let x = \det' (head zs) (cd k (zs : yss ++ zss))
                                in if even (length yss) then x else -x
  
```

Функция cd здесь определена так:

```

 $cd k = \text{map} (\text{map} \det \cdot \text{pair} \cdot \text{uncurry} \text{zip}) \cdot \text{pair}$ 
  where  $\text{pair} (x : xs) = \text{map} ((,) x) xs$ 
         $\det ((a, b), (c, d)) = (a * d - b * c) \text{ div } k$ 
  
```

Конечно, в таком случае число целочисленных делений возрастает до $\Theta(n^3)$.

Не используем деление

Наконец, приведём ещё один метод вычисления \det , который на этот раз обходится совсем без деления. Этот способ похож на магию, и мы не будем разоблачать её, приводя доказательства. Для матрицы $X = (x_{ij})$

размеров $n \times n$ определим $\text{MUT}(X)$ (от англ. make upper triangular — привести к треугольному виду):

$$\text{MUT}(X) = \begin{pmatrix} -\sum_{j=2}^n x_{jj} & x_{12} & \dots & x_{1n} \\ 0 & -\sum_{j=3}^n x_{jj} & \dots & x_{2n} \\ \dots & 0 & \dots & -\sum_{j=n+1}^n x_{jj} \\ 0 & 0 & \dots & -\sum_{j=n+1}^n x_{jj} \end{pmatrix}$$

Таким образом, элементы X ниже главной диагонали равны нулю, элементы выше главной диагонали остаются неизменными, каждый элемент диагонали заменяется на сумму всех элементов диагонали ниже данного, взятую со знаком минус. Заметим, что $\sum_{j=n+1}^n x_{jj} = 0$.

Далее положим $F_A(X) = \text{MUT}(X) \times A$ и $B = F_A^{n-1} A'$, где $A' = A$, если n нечётно, и $A' = -A$ в противном случае. Словом, матрица B это результат $(n-1)$ -кратного применения F_A к A' , она содержит нули во всех позициях, кроме b_{11} , равного $|A|$. Вычисление $\text{MUT}(X) \times A$ требует $\Theta(n^3)$, и, поскольку это действие повторяется $n-1$ раз, общее время подсчёта $|A|$ составит $\Theta(n^4)$.

Следующее определение точно воспроизводит приведённое описание:

```

det      :: [[Integer]] → Integer
det ass = head (head bss)
  where
    bss   = foldl (matmult ∙ mut) ass' (replicate (n - 1) ass)
    ass' = if odd n then ass else map (map negate) ass
    n     = length ass
  
```

Функция *mut* реализует MUT :

```

mut xss
  = zipWith (++) zeros (zipWith (:) ys (zipWith drop [1..] xss))
  where ys = map negate (tail (scanr (+) 0 (diagonal xss)))
  
```

Значение *zeros* представляет из себя бесконечную нижнетреугольную матрицу нулей, начинающуюся с пустой строки:

```
zeros = [take j (repeat 0) | j ← [0..]]
```

Функция *diagonal* возвращает элементы на главной диагонали:

```

diagonal []      = []
diagonal (xs : xss) = head xs : diagonal (map tail xss)
  
```

Наконец, *matmult* реализует матричное умножение:

$$\begin{aligned} \text{matmult } xss \ yss &= \text{zipWith} (\text{map} \cdot \text{dp}) \ xss (\text{repeat} (\text{transpose} \ yss)) \\ \text{dp } xs \ ys &= \text{sum} (\text{zipWith} (*) \ xs \ ys) \end{aligned}$$

Функция *dp* вычисляет скалярное произведение двух векторов.

Заметим, что значение $\text{MUT}(X)$ не зависит от элементов X , расположенных ниже главной диагонали. При ленивом исполнении их значения никогда не будут вычислены. Тем не менее, для большей эффективности стоит изменить определение *mult* с помощью специальной операции умножения *trimult*, которая умножает верхнетреугольную матрицу на любую, давая в результате верхнетреугольную. Пусть *xss* это список строк верхнетреугольной матрицы, а *yss* произвольная матрица. Тогда

$$\text{trimult } xss \ yss = \text{zipWith} (\text{map} \cdot \text{dp}) \ xss (\text{submats} (\text{transpose} \ yss))$$

даёт на выходе верхнетреугольную матрицу. Функция *submats* возвращает список главных подматриц:

$$\begin{aligned} \text{submats} &:: [[a]] \rightarrow [[[a]]] \\ \text{submats } [[x]] &= [[[x]]] \\ \text{submats } xss &= xss : \text{submats} (\text{map tail} (\text{tail} \ xss)) \end{aligned}$$

Для случая верхнетреугольной матрицы *xss* определение *mut xss* можно упростить:

$$\begin{aligned} \text{mut } xss &= \text{zipWith} (:) \ ys (\text{map tail} \ xss) \\ \text{where } ys &= \text{map} \ \text{negate} (\text{tail} (\text{scanr} (+) 0 (\text{map head} \ xss))) \end{aligned}$$

Диагональ верхнетреугольной матрицы это *map head xss*, а элементы выше неё — *map tail xss*.

Теперь *det* можно переписать так:

$$\begin{aligned} \text{det} &:: [[\text{Integer}]] \rightarrow \text{Integer} \\ \text{det ass} &= \text{head} (\text{head} \ bss) \\ \text{where} \\ bss &= \text{foldl} (\text{trimult} \cdot \text{mut}) \ ass' (\text{replicate} (n - 1) \ ass) \\ ass' &= \text{if odd } n \text{ then upper ass} \\ &\quad \text{else map} (\text{map} \ \text{negate}) (\text{upper ass}) \\ n &= \text{length} \ ass \end{aligned}$$

где *upper* = *zipWith drop [0..]*.

Беглое сравнение

Какой же из трёх приведённых вариантов лучший? Рациональное деление (метод Гаусса), целочисленное деление (две версии, одна с использованием равенства Чио, одна с поочерёдными конденсацией и делением) или без деления (с многократным умножением матриц)?

Мы провели беглое сравнение этих методов с использованием случайных матриц различных размеров n , содержащих элементы в интервале $(-20, 20)$. Как можно было ожидать, оригинальная версия метода Чио без надёжна, в то время как её модификация с попеременными конденсацией и делением стала бесспорным лидером. Для $n = 150$ метод Гаусса занял около 30 секунд, модифицированный метод Чио 10 секунд, многократное умножение 40 секунд.

Заключительные замечания

Описание метода Чио, насчитывающего более 150 лет, можно найти по адресу: <http://mathworld.wolfram.com/ChioPivotalCondensation.html>. Модифицированная версия дана в (Bareiss, 1968), автор главным образом использовал равенство Сильвестра — более общую версию равенства Чио. История метода с многократным умножением менее ясна. Главный факт, на котором основан метод, всё ещё не имеет чисто алгебраического доказательства. Приведённая версия получена из алгоритма в (Mahajan and Vinay, 1997), где рассуждения связаны со специальным обобщением циклового разложения перестановок. В этом обобщении каждый цикл может содержать повторяющиеся элементы. Махаян и Винай показали, что все слагаемые, которые не соответствуют каким бы то ни было перестановкам, взаимно уничтожаются, остаются лишь те $a_{1\pi(1)}a_{2\pi(2)} \dots a_{n\pi(n)}$, которые соответствуют. Доказательство этого факта нетривиально. Произведение матричных элементов такого вида может быть выражено в форме пути на некотором ориентированном ациклическом графе, а вычисление суммы таких произведений сводится к задаче о путях на графе. При переводе соответствующего рекурсивного определения обратно к матричным операциям и было получено равенство, использованное выше. Хотя нам не удалось найти никаких ссылок, это равенство наверняка было известно ранее.

Ещё одно обстоятельство не было отмечено в тексте: ни один Haskell-массив не пострадал в процессе вычислений, будь то неизменяемый массив или нет. Вместо этого матрица была по умолчанию представлена списком строк, что позволило описать каждый алгоритм достаточно ясно. Но воз-

можно, лучшее было бы определить подходящий для матриц абстрактный тип данных, вместе с которым бы предоставлялись все необходимые операции наподобие обращения к первому столбцу, первой строке, диагонали, главным подматрицам и т. д.

Литература

- Bareiss, E. H. (1968). Sylvester's identity and multi-step integer preserving Gaussian elimination. *Mathematics of Computation* **22** (103), 565–78.
- Mahajan, M. and Vinay, V. (1997). Determinant: combinatorics, algorithms and complexity. *Chicago Journal of Theoretical Computer Science*, Article 5.

23

Внутри выпуклой оболочки

Введение

Вычисление выпуклой оболочки множества точек является центральным моментом многих задач вычислительной геометрии, ему посвящено большое число исследований. Отыскание оболочки имеет смысл в любой размерности d , однако большинство учебников рассматривают случаи $d = 2$ и $d = 3$. В этой жемчужине мы сформулируем d -мерный вариант задачи и опишем прямолинейный инкрементный алгоритм её решения. Этот алгоритм хорошо известен, для него предлагалось множество замысловатых усовершенствований, но мы обсудим лишь основную идею. Вместо полного вывода алгоритма будет показано, как протестировать его с помощью библиотеки QuickCheck Классена (Claessen) и Хьюза (Hughes). Тестирование обнаружит наличие ошибки в коде, которая была оставлена умышленно, чтобы проверить внимательность читателя.

Предварительные сведения

Многие геометрические алгоритмы рассыпаются на глазах в присутствии вычислительной погрешности, поэтому разумным было бы оставаться в пределах целочисленной арифметики и ограничить рассмотрение подмножеством $Q(d)$ евклидова пространства $E(d)$ размерности d , состоящего из точек с рациональными декартовыми координатами. Точка в $Q(d)$ может быть представлена списком из $d + 1$ целого числа $[x_0, x_1, \dots, x_d]$, где $x_d \neq 0$; этот список соответствует d рациональным декартовым коор-

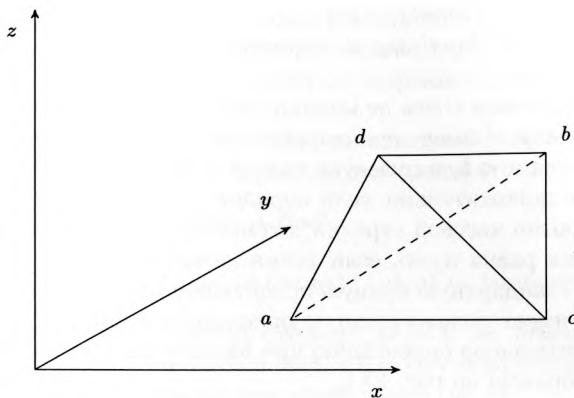


Рис. 23.1: положительно ориентированный тетраэдр; точки a, b и c лежат в плоскости (x, y) , а точка d над ней.

динатам $[x_0/x_d, x_1/x_d, \dots, x_{d-1}/x_d]$. Таким образом, определим $\text{Point} = [\text{Integer}]$. Размерность точки задана так:

```
dimension :: Point → Int
dimension ps = length ps - 1
```

По определению d -симплексом является список из $d+1$ точки в $Q(d)$ вместе со значением $+1$ или -1 , отражающим ориентацию симплекса, которая зависит от порядка перечисления точек:

```
type Simplex = ([Point], Int)
```

Точки или, как их ещё называют, вершины d -симплекса должны находиться в «общем положении», то есть никакие две точки не должны совпадать, никакие три лежать на одной прямой, никакие четыре в общей плоскости и т. д. 1-симплекс в $Q(1)$ это отрезок, 2-симплекс в $Q(2)$ треугольник, 3-симплекс в $Q(3)$ тетраэдр. Формально точки $[v_0, v_1, \dots, v_d]$, где точка v_j имеет координаты $[x_{j0}, x_{j1}, \dots, x_{jd}]$, находятся в общем положении, если определитель матрицы $X = (x_{ij})$ отличен от нуля. Значение определителя пропорционально объёму симплекса, взятому с соответствующим знаком, а ориентация симплекса и есть по определению этот знак:¹

¹Определение \det дано в предшествующей жемчужине.

```
orientation :: [Point] → Int
orientation = fromIntegral · signum · det
```

Чётные перестановки точек не меняют ориентации в отличие от нечётных. В $Q(1)$ отрезок $[a, b]$ имеет положительную ориентацию, если $a > b$, отрицательную, если $a < b$, и нулевую, если $a = b$. В $Q(2)$ треугольник $[a, b, c]$ ориентирован положительно, если порядок $[a, b, c]$ перечисляет точки треугольника против часовой стрелки, отрицательно, если по часовой стрелке, ориентация равна нулю, если точки лежат на одной прямой. В $Q(3)$, предполагая стандартную правую ориентацию координатных осей, тетраэдр $[a, b, c, d]$ имеет положительную ориентацию, если треугольник $[b, c, d]$ имеет положительную ориентацию при взгляде на него из точки a . Такой тетраэдр изображён на рис. 23.1.

Каждый симплекс *smp* определяет выпуклую область $CS(smp)$ пространства $Q(d)$, а именно, множество рациональных точек внутри *smp* или на его границе. Чтобы определять принадлежность точки к $CS(smp)$, научимся сначала вычислять грани симплекса *smp*. Грань d -симплекса задаётся списком d точек вместе с ориентацией, унаследованной от симплекса. Грани отрезка это его конечные точки, грани треугольника — три его стороны и т. д. В общем случае грани симплекса и сопутствующие ориентации вычисляются так:

```
facets          :: Simplex → [Facet]
facets (us, b) = zip (minors us) (cycle [b, -b])
minors        :: [a] → [[a]]
minors []      = []
minors (x : xs) = xs : map (x:) (minors xs)
```

где $Facet = ([Point], Int)$. Минорами списка являются подпоследовательности. Например, $minors "abcd" = ["bcd", "acd", "abd", "abc"]$. Эта функция встречалась в двух прошлых жемчужинах. Для отрезка $[a, b]$ в $Q(1)$ с положительной ориентацией гранями выступают 0-симплексы $[b]$ и $[a]$ с ориентациями $+1$ и -1 соответственно. Точка p находится строго внутри области симплекса $[a, b]$, если симплекс $[p, b]$ ориентирован так же, как $[b]$, то есть положительно, а симплекс $[p, a]$ — так же, как $[a]$, то есть отрицательно. Другими словами, если $b < p < a$. Аналогичные рассуждения применимы в старших размерностях. Таким образом, положительно ориентированный треугольник $[a, b, c]$ в $Q(2)$ имеет три грани:

$([b, c], +1), \quad ([a, c], -1), \quad ([a, b], +1)$

Точка p находится строго внутри области симплекса $[a, b, c]$, если три симплекса $[p, b, c]$, $[p, a, c]$ и $[p, a, b]$ имеют ориентацию $+1$, -1 и $+1$ соответственно, что означает расположение p слева от стороны bc , справа от стороны ac и слева от ab .

Область $CS(smp)$ тех точек, которые находятся строго внутри симплекса smp или на его границе, определяется предикатом:

$$\begin{aligned} insideCS &:: Simplex \rightarrow Point \rightarrow Bool \\ insideCS smp p &= and [0 \leq b * orientation (p : us) | (us, b) \leftarrow facets smp] \end{aligned}$$

Точка p находится строго внутри $CS(smp)$, если $b = orientation (p : us)$ для каждой грани (us, b) симплекса smp , и находится на границе smp , если $orientation (p : us) = 0$ хотя бы для одной грани (us, b) .

Выпуклые оболочки

Выпуклая оболочка $CH(vs)$ множества точек vs в пространстве $Q(d)$ это некоторая область в $Q(d)$. Она может быть определена различными эквивалентными способами, включая следующий: $CH(vs)$ представляет собой объединение множеств $CS(smp)$ для всех d -симплексов smp , определённых точками из vs . Например, в $Q(2)$ выпуклая оболочка vs это объединение областей, ограниченных треугольниками, вершины которых берутся в vs . Множество $CH(vs)$ может быть, таким образом, задано предикатом:

$$\begin{aligned} insideCH &:: [Point] \rightarrow Point \rightarrow Bool \\ insideCH vs p &= or [insideCS smp p | smp \leftarrow simplexes vs] \end{aligned}$$

где $simplexes vs$ перечисляет все симплексы, которые могут быть построены на точках vs :

$$\begin{aligned} simplexes &:: [Point] \rightarrow [Simplex] \\ simplexes vs &= [(us, b) | us \leftarrow tuples (d + 1) vs, \\ &\quad let b = orientation us, b \neq 0] \\ &\quad where d = dimension (head vs) \end{aligned}$$

Значение $tuples n vs$ представляет собой все n -ки точек из vs , то есть все подпоследовательности vs длины n . Определение $tuples$ остаётся в качестве упражнения.

В соответствии с данным определением $insideCH$, множество $CH(vs)$ пусто, если из элементов vs нельзя составить ни одного симплекса. Множество точек vs в $Q(1)$ не даёт 1-симплексов, если все точки совпадают, в

$Q(2)$ — если все точки лежат на одной прямой, в $Q(3)$ — если все точки лежат в одной плоскости. Для подобных множеств возможно определить выпуклую оболочку, понизив размерность, но мы оставим данное определение как есть.

Инкрементный алгоритм

На n точках можно построить $\Omega(n^{d+1})$ d -симплексов, так что вычисление *insideCH* vs p потребует $\Omega(n^{d+1})$ операций. Большинство этих симплексов перекрываются, а некоторые просто совпадают — следует рассматривать только те, которые покрывают оболочку без пересечений. Значит более эффективный алгоритм получится заменой *simplexes* другой функцией *partition* с таким же типом, которая разбивает оболочку на части. Заменим *insideCH* на *insideCH'*:

```
insideCH'      :: [Point] → Point → Bool
insideCH' vs p = or [insideCS smp p | smp ← partition vs]
```

Функция *partition* представляет собой процедуру, которая начинает с одного симплекса, а затем последовательно добавляет новые симплексы, анализируя каждую дополнительную точку вне текущей выпуклой оболочки:

```
partition :: [Point] → [Simplex]
partition vs
  = case findSimplex vs of
    Nothing   → []
    Just [smp] → foldl update [smp] (vs \\ vertices smp)
```

Вершины симплекса перечисляются функцией *vertices*:

```
vertices :: Simplex → [Point]
vertices = sort · fst
```

Вершины передаются в упорядоченном виде, так что если для *vs* также поддерживать упорядоченность, то $\backslash\backslash$ может быть реализована эффективно, как разность упорядоченных списков. Функция *findSimplex* пытается найти некоторый симплекс. Если попытка кончается неудачей, то разбиения не существует и оболочка пуста. В противном случае, этот симплекс используется как исходные данные, его точки удаляются из списка дополнительных точек, которые предстоит рассмотреть. Остается определить *findSimplex* и *update*. Мы разберёмся с этими функциями по отдельности.

Отыскание симплекса

Один очевидный способ определения *findSimplex* состоит в следующем:

```
findSimplex vs = if null smps then Nothing else Just (head smps)
  where smps = simplexes vs
```

Но в худшем случае это будет стоить $\Omega(n^{d+1})$ операций, что сводит на нет наше желание ускорить алгоритм. Худший случай редко проявится на практике, так что определение выше можно считать достаточно хорошим, но существует и другой метод. Идея состоит в том, чтобы взять первую точку v_0 из *vs*, а затем выполнить однократный просмотр оставшихся элементов *vs* в поисках оставшихся точек для симплекса. Сначала найдём вторую точку v_1 , которая не лежит на одной прямой с v_0 . Затем просмотр продолжится среди точек, следующих за v_1 , в поисках третьей точки v_2 , которая не лежит в одной плоскости с v_0 и v_1 , и так далее, пока не будет найдена $d + 1$ точка в общем положении. Тонкий момент заключается не в поиске, а в том, что невырожденность $k + 1$ точки в $Q(d)$ не может быть проверена с помощью обычного подсчёта определителя: соответствующая матрица координат точек имеет размеры $(k+1) \times (d+1)$, то есть не является квадратной в случае $k < d$.

Необходимо научиться рассматривать квадратные подматрицы данной матрицы. Пусть X это матрица размеров $(k + 1) \times (d + 1)$, полученная из первых $k + 1$ вершин, подматрицы размеров $(k + 1) \times (k + 1)$ получаются выбором любой комбинации k столбцов из первых d столбцов вместе с последним столбцом (где стоят знаменатели рациональных координат вершин). Вырожденным считается набор из $k + 1$ точки, для которого определитель каждой такой квадратной подматрицы равен нулю. Эта проверка реализуется следующим образом:

```
degenerate k      = all (== 0) · map det · submatrices k · transpose
submatrices k vs = map (+[last vs]) (tuples k (init vs))
```

Функция *transpose* транспонирует матрицу, так что *submatrices* выбирает столбцы матрицы, рассматривая строки транспонированной матрицы. Определитель матрицы равен определителю транспонированной. Поскольку всего имеется $O(d^k)$ подматриц транспонированной матрицы, а подсчёт определителя требует $O(k^3)$ операций, вычисление *degenerate k vs* для списка из $k + 1$ точки *vs* в $Q(d)$, где $k \leq d$, займёт $O(k^3 d^k) = O(d^{d+3})$ операций.

Функция *findSimplex* реализуется так:

```


$$\begin{aligned} \text{findSimplex} &:: [\text{Point}] \rightarrow \text{Maybe Simplex} \\ \text{findSimplex} [] &= \text{Nothing} \\ \text{findSimplex} (v : vs) &= \text{search} (\text{length } v - 1) 1 [v] vs \end{aligned}$$


```

где функция *search* определяется так:

```


$$\begin{aligned} \text{search } d \ k \ us \ vs & \\ | \ k == d + 1 &= \text{Just } (us, \text{orientation } us) \\ | \ \text{null } vs &= \text{Nothing} \\ | \ \text{degenerate } k \ (v : us) &= \text{search } d \ k \ us \ (\text{tail } vs) \\ | \ \text{otherwise} &= \text{search } d \ (k + 1) \ (v : us) \ (\text{tail } vs) \\ \text{where } v &= \text{head } vs \end{aligned}$$


```

Временная сложность *findSimplex vs* для списка *vs* из *n* точек составляет $O(d^{d+3}n)$, то есть линейна по *n*, хотя и с большим константным множителем.

Добавление симплекса

Чтобы определить оставшуюся функцию *update*, рассмотрим множество *smps* симплексов, разбивающих выпуклую оболочку точек, рассмотренных на некотором этапе. Границ этих симплексов делятся на две группы: внутренние грани, которые встретились ровно дважды (с противоположными ориентациями); внешние грани, которые появились лишь однажды. Например, возьмём вершины квадрата $[a, b, c, d]$ в $Q(2)$. Существует два возможных способа разбиения на треугольники: $[a, b, c]$ и $[c, d, a]$ или $[a, b, d]$ и $[b, c, d]$. В первой триангуляции внутренней является сторона $[a, c]$, во второй — сторона $[b, d]$. Внешние грани вычисляются так:

```


$$\begin{aligned} \text{external} &:: [\text{Simplex}] \rightarrow [\text{Facet}] \\ \text{external} &= \text{foldr } op [] \cdot \text{sort} \cdot \text{concatMap facets} \end{aligned}$$


```

где

```


$$\begin{aligned} op \ smp [] &= [] \\ op \ smp (smp' : smps) &= \text{if } \text{vertices } smp == \text{vertices } smp' \text{ then } smps \\ &\quad \text{else } smp : smp' : smps \end{aligned}$$


```

Стоимость вычисления *external smps* составляет $O(dS \log dS)$, где *S* это размер *smps*, так как наиболее затратным действием является сортировка граней, а их $O(dS)$ штук.

Каждая новая точка v разбивает внешние грани на две группы: **видимые и невидимые**. Представим лампочку, установленную в v ; она будет освещать только видимые грани. Грань (us, b) видима для v , если v находится строго снаружи от неё, то есть $\text{orientation}(v : us) \neq 0$.

$$\begin{aligned} \text{visible} &:: \text{Point} \rightarrow [\text{Facet}] \rightarrow [\text{Facet}] \\ \text{visible } v \text{ } fs &= [(us, b) \mid (us, b) \leftarrow fs, b * \text{orientation}(v : us) < 0] \end{aligned}$$

Видимые грани отсутствуют, если v находится внутри или на поверхности текущей оболочки. В частности, если v совпадает с одной из уже рассмотренных вершин, то текущая оболочка останется неизменной, поэтому наличие повторяющихся точек в vs не имеет значения (а также нет необходимости в удалении вершин исходного симплекса из исходного набора точек).

Обновление оболочки заключается в добавлении к $smps$ нового симплекса для каждой видимой грани:

$$\begin{aligned} \text{newSimplex} &:: \text{Point} \rightarrow \text{Facet} \rightarrow \text{Simplex} \\ \text{newSimplex } v \text{ } (us, b) &= (v : us, -b) \end{aligned}$$

Задаваемая ориентация корректна, поскольку если (us, b) видна для v , то $b * \text{orientation}(v : us) < 0$, и потому $\text{orientation}(v : us) = -b$.

Теперь $update$ можно определить так:

$$\begin{aligned} update &:: [\text{Simplex}] \rightarrow \text{Point} \rightarrow [\text{Simplex}] \\ update \text{ } smps \text{ } v &= smps \uplus \text{map}(\text{newSimplex } v)(\text{visible } v \text{ } (\text{external } smps)) \end{aligned}$$

Время вычисления $update \text{ } smps$ определяется временем отыскания видимых граней, а оно составляет $O(dS \log dS)$ шагов, где S это размер $smps$. Сложность $insideCH'$ как функции от n , количества точек в vs , таким образом, $O(dnS \log dS)$, где S это максимальное число симплексов, рассматриваемых на каждом этапе. Известно, что $S = O(n^e)$, где $e = \lfloor d/2 \rfloor$, так что вычисление $insideCH'$ vs требует $O(n^{e+1} \log n)$ операций. Лучше, чем $insideCH$, но всё ещё далеко от идеала.

Улучшение

Как показано выше, инкрементный алгоритм вычисляет множество симплексов, представляющих разбиение оболочки. На каждом шаге определяются внешние грани текущей оболочки, чтобы затем выделить видимые грани для очередной точки и добавить соответствующие симплексы к

оболочке. Очевидно, что более разумным было бы хранить внешние грани симплексов вместо самих симплексов. Положив $\text{faces} = \text{external} \cdot \text{partition}$, можем заменить $\text{insideCH}'$ на $\text{insideCH}''$:

$$\text{insideCH}'' \text{ vs } p = \text{and} [0 \leq b * \text{orientation}(p : us) \mid (us, b) \leftarrow \text{faces} \text{ vs}]$$

Функция faces имеет тип $[\text{Point}] \rightarrow [\text{Facet}]$. Один из привычных способов описания выпуклой оболочки в вычислительной геометрии состоит в перечислении внешних граней.

Эффективный алгоритм вычисления faces может быть выведен с помощью закона слияния для foldl . Необходимо найти такую функцию $\text{update}' :: [\text{Facet}] \rightarrow \text{Point} \rightarrow [\text{Facet}]$, что

$$\text{external} (\text{update smps} v) = \text{update}' (\text{external smps}) v$$

Поскольку внешними гранями одного симплекса являются все его грани, получаем:

$$\begin{aligned} & \text{faces vs} \\ &= \text{case findSimplex vs of} \\ &\quad \text{Nothing} \rightarrow [] \\ &\quad \text{Just [smp]} \rightarrow \text{foldl update}' (\text{facets smp}) (\text{vs} \setminus\setminus \text{vertices vs}) \end{aligned}$$

Мы не будем вдаваться в детали вывода update' , а просто покажем результат:

$$\begin{aligned} \text{update}' fs v &= (fs \setminus\setminus fs') \uplus \text{map} (\text{newFacet} v) (\text{external} fs') \\ &\quad \text{where } fs' = \text{visible } v fs \\ \text{newFacet } v (us, b) &= (v : us, b) \end{aligned}$$

Словом, грани, видимые из очередной точки, удаляются из текущего множества граней, после чего добавляются новые грани. Видимые грани образуют связное множество, их граница состоит из множества их граней, то есть $(d - 2)$ -симплексов, встречающихся по одному разу. Например, в $Q(3)$ гранями являются треугольники, граница связного множества видимых треугольников представляет собой множество внешних сторон этих треугольников. Ориентация каждой новой грани определяется ориентацией соответствующих границ граней. Чтобы пояснить этот факт, рассмотрим отрезок $([a, b], +1)$ в $Q(2)$, который виден из точки c и в котором точка b является граничной (то есть следующий отрезок, начинающийся в b , невидим). Связанный с b 0-симплекс имеет положительную ориентацию,

и новый отрезок $[c, b]$ должен быть направлен к b , то есть снова иметь положительную ориентацию.

Время выполнения *faces* определяется временем отыскания граней, видимых из каждой следующей точки. Чтобы найти эти грани, просматривается каждая грань оболочки; разумеется, это неэффективный подход, потому что видимые грани образуют небольшое локально связное множество. Именно в этой части предлагают улучшения более изощрённые алгоритмы, такие как алгоритм бульдозера (Blelloch *et al.*, 2001), однако мы не станем вдаваться в детали.

QuickCheck

Коэн Классен и Джон Хьюз разработали очень полезный набор функций под названием *QuickCheck* для тестирования программ на языке Haskell, см. (Claessen and Hughes, 2000). Обсуждение тонкостей *QuickCheck* заняло бы слишком много места, но мы кратко покажем, как можно использовать функции из этого набора, чтобы проверить корректность двух версий алгоритма вычисления выпуклой оболочки, описанных выше.

Во-первых, нам понадобится генератор точки в $Q(d)$:

```
point    :: Int → Gen [Integer]
point d = do { xs ← vector d; return (xs ++ [1]) }
```

Вызов *vector d* возвращает список из d случайных значений, в данном случае — целых чисел. Таким образом, *point d* представляет список из $d + 1$ целого числа, в котором последний элемент равен единице.

Далее потребуется генератор списка n точек:

```
points      :: Int → Int → Gen [[Integer]]
points d 0   = return []
points d (n + 1) = do { p ← point d; ps ← points d n;
                        return (p : ps) }
```

Теперь можно определить свойство *prop_Hull*, которое проверяет соответствие инкрементного алгоритма спецификации:

```
prop_Hull    :: Int → Int → Property
prop_Hull d n = forAll (points d n) $ λ vs →
                  forAll (point d) $ λ v →
                  insideCH vs v == insideCH' vs v
```

К примеру, запуск *quickCheck (prop_Hull 3 10)* даёт:

OK, passed 100 tests.

Однако замена $insideCH'$ на $insideCH''$ в $prop_Hull$ обнаруживает ошибку:

```
Main> quickCheck (prop_Hull 2 4)
Falsifiable, after 2 tests:
[[0,0,1],[0,0,1],[0,0,1],[-1,-1,1]]
[1,0,1]
```

Батеньки, что же пошло не так? В общем, трудность в том, что четыре точки лежат на одной прямой, так что не существует разбиения и отсутствуют внешние грани. В случае, когда vs коллинеарны в $Q(2)$, $insideCH'' vs$, как и следует, возвращает $False$, тогда как $insideCH' vs$ возвращает $True$. Необходимо поправить $insideCH''$ следующим образом:

$$\begin{aligned}insideCH'' \text{ vs } v = & \text{ if } null fs \text{ then } False \text{ else} \\& \text{ and } [0 \leq b * orientation (v : us) \mid (us, b) \leftarrow fs] \\& \text{ where } fs = faces vs\end{aligned}$$

Теперь *QuickCheck* счастлив. А вам удалось заметить ошибку?

Заключительные замечания

Огромное число учебников по вычислительной геометрии рассматривают алгоритмы построения выпуклой оболочки; вот два из них: (1998, O'Rourke) и (Preparata and Shamos, 1985). В частности, отличная книга O'Рурка содержит две аккуратно написанных главы по этой теме, а список литературы включает большую часть соответствующей литературы, хотя разбор трёхмерного случая, принадлежащий Дейкстре (Dijkstra, 1976), упущен. Наша жемчужина возникла как результат желания помериться силами с (Karimipour and Frank, 2009), хотя в деталях мы сильно расходимся. Я хотел бы поблагодарить Ирину Войкулеску (Irina Voiculescu) за ряд полезных обсуждений выпуклых оболочек и методов их вычисления.

Литература

- Claessen, K. and Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN International Conference of Functional Programming*, Montreal, Canada, pp. 268–79. См. также <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.
- Blelloch, G., Burch, H., Crary, K., et al. (2001). Persistent triangulations. *Journal of Functional Programming* 11 (5), 441–66.

- Dijkstra, E. W. (1976). *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall. [Имеется русский перевод: Дейкстра Э. Дисциплина программирования. М.: Мир. 1978.]
- Karimipour, F. and Frank, A. U. (2009). A dimension independent convex hull algorithm. Unpublished.
- O'Rourke, J. (1998). *Computational Geometry*, second edition. Cambridge, UK: Cambridge University Press.
- Preparata, F. P. and Shamos, M. I. (1985). *Computational Geometry*. New York, NY: Springer-Verlag. [Имеется русский перевод: Препарата Ф., Шеймос М. Вычислительная геометрия: Введение. М.: Мир. 1989.]

24

Рациональное арифметическое кодирование

Введение

Эта и следующая жемчужины посвящены вопросу арифметического кодирования — специального способа сжатия данных. В отличие от других методов оно не предполагает представления каждого символа текста некоторым целым числом битов; вместо этого весь текст целиком преобразуется в двоичную дробь из единичного интервала. Хотя сама идея довольно стара, об арифметическом кодировании нельзя было говорить как о серьёзном явлении в мире сжатия данных до публикации «доступной реализации» алгоритма Виттеном (Witten), Нилом (Neal) и Клири (Cleary) в 1987 году. За последние два десятилетия метод был отточен, выявлены его достоинства и недостатки по сравнению с конкурирующими схемами кодирования. Арифметическое кодирование может быть более эффективным, чем кодирование по Хаффману или алгоритм Шеннона—Фано, оно хорошо подходит для учёта статистических показателей символов текста. С другой стороны, процессы кодирования и декодирования занимают больше времени, чем в случае с другими методами.

Справедливо считается, что арифметическое кодирование сложно реализовать; тем не менее, наша цель в двух данных жемчужинах состоит в том, чтобы разобрать основные алгоритмы. Наша реализация проста и элегантна, хотя весьма затратна как по времени, так и по памяти. В следующей жемчужине кодирование и декодирование будут реализованы заново с помощью целых чисел фиксированного размера. Как раз в этом месте кроется большая часть неочевидных сторон задачи.

Арифметическое кодирование с рациональной арифметикой

Основные этапы арифметического кодирования состоят в следующем:

1. Разбить входной текст на символы, под которыми понимается некоторое логическое объединение букв, например, в слова или просто буквы по отдельности. Для простоты можно считать, что число возможных символов конечно.
2. Сопоставить каждому символу полуоткрытый интервал внутри единичного интервала $[0, 1]$. Это сопоставление задаётся моделью.
3. Последовательно сужать единичный интервал на величины, которые определяются интервалами, сопоставленными символам текста.
4. Выбрать достаточно короткую дробь в итоговом интервале.

Следующие определения отражают основные типы данных алгоритма:

```
type Fraction = Ratio Integer
type Interval = (Fraction, Fraction)
```

Дробь представляется отношением двух целых чисел произвольного размера (тип *Integer*), а интервал — парой дробей. Допустимыми считаются только дроби f , для которых выполнено $0 \leq f < 1$. Единичный интервал обозначается парой $(0, 1)$; будем писать $f \in (\ell, r)$, если $\ell \leq f < r$, то есть интервалы считаются замкнутыми слева и открытыми справа. Запись $i \subseteq j$ означает, что i является подинтервалом j .

Сужение

Запись $i \triangleright j$ означает сужение интервала i с помощью интервала j , в результате чего получается подинтервал k интервала i , который относится к интервалу i так же, как интервал j относится к единичному интервалу:

$$\begin{array}{rcl} (\triangleright) & :: & \text{Interval} \rightarrow \text{Interval} \rightarrow \text{Interval} \\ (\ell_1, r_1) \triangleright (\ell_2, r_2) & = & (\ell_1 + (r_1 - \ell_1) * \ell_2, \ell_1 + (r_1 - \ell_1) * r_2) \end{array}$$

Операция \triangleright ассоциативна, интервал $(0, 1)$ выступает для неё в качестве единичного элемента, это оправдывает принятное нами инфиксное обозначение. Можно проверить, что если $f \in i \triangleright j$, то $f \in i$. Следовательно, $i \triangleright j \subseteq i$.

Кроме того, если $f \in i \triangleright j$, то $(f \triangleleft i) \in j$, где операция (\triangleleft) «растягивает» отношение:

$$\begin{array}{rcl} (\triangleleft) & :: & Fraction \rightarrow Interval \rightarrow Fraction \\ f \triangleleft (\ell, r) & = & (f - \ell) / (r - \ell) \end{array}$$

Подытоживая:

$$f \in i \triangleright j \Rightarrow f \in i \wedge (f \triangleleft i) \in j \quad (24.1)$$

На деле (24.1) составляют эквивалентные условия. Более того, если расширить \triangleleft до операции на интервалах так: $(\ell, r) \triangleleft j = (\ell \triangleleft j, r \triangleleft j)$, то $(i \triangleright j) \triangleleft i = j$, то есть \triangleright обладает всеми свойствами групповой операции.

Модели

Чтобы кодировать текст, нужно задать соответствие каждого возможного символа некоторому интервалу. В данной задаче *Model* обозначает абстрактный тип данных, задающий конечное отображение из конечного множества *Symbols* во множество *Intervals* и снабжённый функциями:

$$\begin{array}{l} interval :: Model \rightarrow Symbol \rightarrow Interval \\ symbol :: Model \rightarrow Fraction \rightarrow Symbol \end{array}$$

То есть *interval m x* возвращает интервал, сопоставляемый символу *x* в модели *m*, а *symbol m f* представляет символ, который соответствует единственному интервалу, содержащему дробь *f*. Предполагается, что интервалы, сопоставленные символам, разбивают единичный интервал:

$$x = symbol m f \equiv f \in interval m x \quad (24.2)$$

для любой модели *m* и допустимой дроби *f*.

Важным уточнением основных идей алгоритма является возможность изменения модели по мере чтения текста. Такой подход называется адаптивным кодированием. Например, можно начать с простой модели, в которой все символы отображаются в интервалы равной длины, затем позволить модели адаптироваться с помощью растяжения интервалов для тех символов, которые встречаются чаще. Чем шире интервал, тем проще найти в нём «короткую» дробь. Возможны и более хитроумные адаптации. Например, в русском языке после буквы «с» чаще всего идёт буква «т». Следовательно, встречая букву «с», можно было бы расширять интервал для «т», в ожидании, что именно она попадётся следующей.

В наши цели не входит пристальное изучение адаптации моделей. Вместо этого просто предположим существование дополнительной функции:

$$\text{adapt} :: \text{Model} \rightarrow \text{Symbol} \rightarrow \text{Model}$$

Функция $\text{intervals} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Interval}]$ в этом случае будет определяться так:

$$\begin{aligned}\text{intervals } m [] &= [] \\ \text{intervals } m (x : xs) &= \text{interval } m x : \text{intervals} (\text{adapt } m x) xs\end{aligned}$$

Каждый символ текста преобразуется в интервал с помощью применения функции interval к последовательно изменяющейся модели. Поскольку декодер знает исходную модель и функцию adapt , он может применить необходимые адаптации к модели, чтобы корректно восстановить каждый символ. Важно, что нет необходимости передавать множество моделей вместе с текстом.

Кодирование

После того как определены нужные типы данных и вспомогательные функции, можно описать функцию encode :

$$\begin{aligned}\text{encode} &:: \text{Model} \rightarrow [\text{Symbol}] \rightarrow \text{Fraction} \\ \text{encode } m &= \text{pick} \cdot \text{foldl} (\triangleleft) (0, 1) \cdot \text{intervals } m\end{aligned}$$

где $\text{pick } i \in i$. Интервалы, соответствующие символам текста, используются для сужения единичного интервала к некоторому заключительному интервалу, из которого выбирается некоторая дробь.

Приведём простой пример. Возьмём статичную модель m , которая содержит пять символов с сопоставленными им интервалами:

$$[(e, (0, 3/8)), (g, (3/8, 1/2)), (n, (1/2, 5/8)), (r, (5/8, 7/8)), (v, (7/8, 1))]$$

Тогда

$$\begin{aligned}\text{encode } m \text{ "evergreen"} &= \text{pick} ((0, 1) \triangleright (0, 3/8) \triangleright (7/8, 1) \cdots \triangleright (1/2, 5/8)) \\ &= \text{pick} (11445828/2^{25}, 11445909/2^{25})\end{aligned}$$

Наилучшее определение для pick вернёт $(89\,421, 2^{18})$, единственную дробь в данном интервале с кратчайшим двоичным представлением, а именно

010101110101001101. Так что девять символов "evergreen" могут быть закодированы 18 битами или тремя символами. На самом деле, так как числитель кратчайшей дроби должен быть нечётным, последний бит всегда равен 1 и может быть опущен, так что в этом примере нужно выдавать в качестве ответа всего 17 битов. Лучшее, чего может добиться тут кодирование по Хаффману, это 19 битов. Мы вернёмся к подходящему выбору $pick$ позже, пока предположим лишь, что $pick i \in i$.

Декодирование

Очевидный способ задания $decode$ представляется таким условием:

$$xs = decode m (encode m xs)$$

для всех конечных списков символов xs . Однако в силу причин, которые будут изложены ниже, спецификация ослабляется следующим образом:

$$xs \sqsubseteq decode m (encode m xs) \tag{24.3}$$

где \sqsubseteq обозначает отношение префикса для списков, то есть $xs \sqsubseteq ys$, если $ys = xs ++ zs$ для некоторого zs . Таким образом, $decode$ является левой обратной к $encode$ в том смысле, что она выдаёт последовательность символов, которые были закодированы $encode$, но необязательно только их.

Для явного определения $decode$ обозначим вход $encode$ следующим образом: $xs = [x_0, x_1, \dots, x_{n-1}]$. Пусть m_0 обозначает исходную модель, а $j_0 = (0, 1)$ — исходный интервал. Обозначим

$$\begin{aligned} m_{k+1} &= adapt m_k x_k \\ i_k &= interval m_k x_k \\ j_{k+1} &= j_k \triangleright i_{k+1} \end{aligned}$$

где $0 \leq k < n$. По определению $encode$, если $f = encode m_0 xs$, то $f \in j_n$. Теперь можно провести рассуждения для $n > 0$:

$$\begin{aligned} f &\in j_n \\ &\equiv \{ \text{определение } j_n \} \\ f &\in (j_{n-1} \triangleright i_n) \\ \Rightarrow &\{ (24.1) \} \\ f &\in j_{n-1} \wedge (f \triangleleft j_{n-1}) \in i_n \\ &\equiv \{ \text{определение } i_n \} \end{aligned}$$

$$\begin{aligned}
 f \in j_{n-1} \wedge (f \triangleleft j_{n-1}) &\in \text{interval } m_n \ x_n \\
 \equiv \quad \{ (24.2) \} \\
 f \in j_{n-1} \wedge x_n &= \text{symbol } m_n (f \triangleleft j_{n-1})
 \end{aligned}$$

Таким образом, по индукции получается:

$$x_k = \text{symbol } m_k (f \triangleleft j_{k-1}) \quad (24.4)$$

для $k = n - 1, n - 2, \dots, 0$. Ничто не мешает вычислять эти значения в порядке $k = 0, 1, \dots, n - 1$. Однако поскольку декодер не знает количества символов, он будет порождать их в неограниченном количестве. Заметим, что в выкладках выше не использовалась ассоциативность \triangleright .

Реализуем процесс декодирования с помощью стандартной функции Haskell *unfoldr*:

$$\begin{aligned}
 \text{unfoldr} &:: (b \rightarrow \text{Maybe}(a, b)) \rightarrow b \rightarrow [a] \\
 \text{unfoldr } f \ b &= \text{case } f \ b \text{ of} \\
 &\quad \text{Just}(a, b') \rightarrow a : \text{unfoldr } f \ b' \\
 &\quad \text{Nothing} \rightarrow []
 \end{aligned}$$

Функция *decode* определяется так:

$$\begin{aligned}
 \text{decode} &:: \text{Model} \rightarrow \text{Fraction} \rightarrow [\text{Symbol}] \\
 \text{decode } m \ f &= \text{unfoldr step}(m, (0, 1), f) \\
 \text{step } (m, i, f) &= \text{Just}(x, (\text{adapt } m \ x, i \triangleright \text{interval } m \ x, f)) \\
 &\quad \text{where } x = \text{symbol } m (f \triangleleft i)
 \end{aligned}$$

Доказательство того, что это определение удовлетворяет (24.3), проводится индукцией по длине *xs*. Детали не добавят ничего нового к неформальному обсуждению выше, и мы их опустим.

Остаётся разобраться с остановкой декодирования. Есть два метода для решения этой задачи. Если количество символов в тексте известно заранее, его можно переслать до начала кодирования. Тогда функция *decode* может быть остановлена после получения этого количества символов. Второй вариант заключается в использовании специального символа конца файла EOF, который добавляется в конец каждого текста. В этом случае *decode* останавливается, как только будет получен указанный специальный символ. Второй метод обычно применяется на практике, хотя обладает тем недостатком, что каждая модель должна резервировать, хоть и небольшой, интервал для EOF, сужая тем самым общую длину интервалов, доступную для других символов.

Инкрементное кодирование и декодирование

Какими бы простыми и элегантными ни были определения *encode* и *decode*, они порождают и обрабатывают дроби. Знаменатели этих дробей растут очень быстро. Более предпочтительной была бы работа *encode* и *decode* со списками битов — не в последнюю очередь, потому что это открыло бы возможность к выдаче части результата до завершения обработки всех входных данных и сокращению размера знаменателя.

Разложим *pick* на две части, функции *toBits* :: *Interval* → [Bit] и *toFrac* :: [Bit] → Fraction, так что *pick* = *toFrac* · *toBits*. Исправим определения *encode* и *decode*:

```

encode      :: Model → [Symbol] → [Bit]
encode m   = toBits · foldl ( $\triangleright$ ) (0, 1) · intervals m
decode     :: Model → [Bit] → [Symbol]
decode m bs = unfoldr step (m, (0, 1), toFrac bs)
step (m, i, f) = Just (x, (adapt m x, i  $\triangleright$  interval m x, f))
                  where x = symbol m (f  $\triangleleft$  i)

```

Новая версия *encode* считывает символы и выдаёт биты, а *decode* считывает биты и выдаёт символы. Функции *toBits* и *toFrac* пока остаются без определения, однако если *toFrac (toBits i)* ∈ *i* для любого интервала *i*, то (24.3) выполнено.

Новое определение *encode* считывает все символы, прежде чем выдать какой-либо результат. Мы покажем, как сделать функцию *encode* инкрементной, это же подскажет определения для *toBits* и *toFrac*.

Потоковая обработка

Рассмотрим функцию *stream*:

```

stream f g s xs = unfoldr step (s, xs)
where step (s, xs) = case f s of
                           Just (y, s') → Just (y, (s', xs))
                           Nothing → case xs of
                               x : xs' → step (g s x, xs')
                               [] → Nothing

```

Эта функция описывает чередование между считыванием входных данных и вычислением результата. В состоянии *s* управление передаётся вначале

порождающей функции f , которая выдаёт результат, пока ей хватает данных. Затем управление переходит к функции g , которая считывает вход x и создаёт новое состояние. Цикл продолжается, пока вход не будет исчерпан.

Следующая теорема, называемая теоремой о потоковой обработке, описывает связь между *stream* и композицией *unfoldr* с *foldl*.

Теорема 24.1. Пусть функции f и g удовлетворяют потоковому условию:

$$f s = \text{Just}(y, s') \Rightarrow f(g s x) = \text{Just}(y, g s' x)$$

для всех s и x . Тогда $\text{unfoldr } f (\text{foldl } g s xs) = \text{stream } f g s xs$ для всех s и всех конечных списков xs .

Доказательство теоремы о потоковой обработке оставим для приложения. Чтобы применить её к *encode*, предположим $\text{toBits} = \text{unfoldr } bit$ для некоторой функции *bit*, которая удовлетворяет условию:

$$bit i = \text{Just}(b, i_b) \Rightarrow bit(i \triangleright j) = \text{Just}(b, i_b \triangleright j) \quad (24.5)$$

Тогда имеем:

$$\text{encode } m = \text{stream } bit(\triangleright)(0, 1) \cdot \text{intervals } m$$

В итоге получился инкрементный алгоритм для *encode*.

Чтобы удовлетворить (24.5), нужно подобрать подходящее определение *bit*. Кроме того, нужно выполнить условие $\text{toFrac}(\text{toBits } i) \in i$. Отметим, (24.5) требует, чтобы если *bit* i возвращает бит b , то тот же самый бит должен быть значением *bit* i' для любого подинтервала i' интервала i . Это серьёзное ограничение на определение *bit*, один из вариантов которого выглядит так:

$$\begin{aligned} bit(\ell, r) \mid r \leqslant 1/2 &= \text{Just}(0, (2 * \ell, 2 * r)) \\ \mid 1/2 \leqslant \ell &= \text{Just}(1, (2 * \ell - 1, 2 * r - 1)) \\ \mid \text{otherwise} &= \text{Nothing} \end{aligned}$$

Таким образом, *bit* i возвращает *Nothing*, если i содержит внутри себя точку $1/2$; иначе получится 0, если $i \subseteq (0, 1/2)$, и 1, если $i \subseteq (1/2, 1)$. Разумность этого выбора подтверждается тем, что бинарное представление дробей, лежащих в $(0, 1/2)$, начинается с нуля, а лежащих в $(1/2, 1)$ — с единицы.

Если *bit* i возвращает b , то так же происходит и с *bit* i' для любого подинтервала i' интервала i , включая $i \triangleright j$. Более того,

$$\begin{aligned} (2 * \ell, 2 * r) &= (0, 2) \triangleright (\ell, r) \\ (2 * \ell - 1, 2 * r - 1) &= (-1, 1) \triangleright (\ell, r) \end{aligned}$$

Следовательно, если $bit\ i$ выдаёт бит b , то $bit\ i = Just\ (b, j_b \triangleright i)$, где $j_0 = (0, 2)$ и $j_1 = (-1, 1)$. Кроме того, $j_b \triangleright (i \triangleright j) = (j_b \triangleright i) \triangleright j$, поскольку \triangleright ассоциативна. Таким образом, (24.5) выполняется для $i_b = j_b \triangleright i$.

Длина $toBits\ i$ конечна; в самом деле,

$$length\ (toBits\ (\ell, r)) \leq \lfloor \log_2 1/(r - \ell) \rfloor$$

Для доказательства этого нужно заметить, что $toBits$, применённая к интервалу длины, большей $1/2$, возвращает пустую последовательность бит, потому что такой интервал обязательно содержит точку $1/2$. Кроме того, каждый последующий вызов bit производится для интервала в два раза большего, чем предыдущий. Следовательно, если $1/2^{k+1} < r - \ell \leq 1/2^k$, или, эквивалентно, $k = \lfloor \log_2 [1/(r - \ell)] \rfloor$, то остановка гарантирована, самое большое, через k шагов.

С указанным выбором $toBits$ дополняющая её функция $toFrac$ определяется так:

$$toFrac = foldr (\lambda b f \rightarrow (b + f) / 2) (1/2)$$

Или, что же: $toFrac\ bs = foldr (\lambda b f \rightarrow (b + f) / 2) 0 (bs ++ [1])$. Таким образом, $toFrac\ bs$ добавляет бит 1 в конец bs и преобразует результат в дробь обычным способом. Несложно проверить, что выполняется $toFrac\ bs = (2n + 1)/2^{k+1}$, где $k = length\ bs$ и $n = toInt\ bs$, двоичное целое, представленное bs .

Чтобы показать, что $pick\ i \in i$, где $pick = toFrac \cdot toBits$ заметим, что $pick$ является композицией функции, которая принимает список, и функции, которая возвращает список. Промежуточный список может быть и ключён:

$$\begin{aligned} pick\ (\ell, r) \mid r \leq 1/2 &= pick\ (2 * \ell, 2 * r) / 2 \\ \mid 1/2 \leq \ell &= (1 + pick\ (2 * \ell - 1, 2 * r - 1)) / 2 \\ \mid \text{otherwise} &= 1/2 \end{aligned}$$

Доказательство $pick\ i \in i$ (на самом деле $pick\ i$ строго содержится в i) проводится по индукции неподвижной точки. В таком типе индукции сохранение справедливости гипотезы проверяется для рекурсивного вызова. Таким образом, доказательство по индукции неподвижной точки это доказательство индукцией по глубине рекурсии. Детали остаются в качестве упражнения.

Не обсуждён лишь способ определить декодирование инкрементно. Сделать это можно, но более подробное обсуждение не приводится во избежание затраты времени: другая реализация *encode* и *decode* в терминах целых чисел фиксированного размера, обсуждаемая в следующей жемчужине, потребует совсем иного подхода.

Заключительные замечания

Материал для этих двух жемчужин позаимствован из (Bird and Gibbons, 2003) и (Stratford, 2005). (Witten *et al.*, 1987) описывает упомянутую «доступную реализацию». Подробное описание кодов Хаффмана и Шеннона—Фано см. в (Huffman, 1952) и (Fano, 1961). За современным взглядом на арифметическое кодирование можно обратиться в (Moffat *et al.*, 1998) и (Mackay, 2003). Теорема о потоковой обработке является новым результатом и была придумана специально для формулировки инкрементной версии кодирования, но имеет и другие приложения; см. (Gibbons, 2007). Это хороший пример, когда практика приводит к развитию теории.

Литература

- Bird, R. S. and Gibbons, J. (2003). Arithmetic coding with folds and unfolds. *Advanced Functional Programming 4, Volume 2638 of Lecture Notes in Computer Science*, ed. J. Jeuring and S. Peyton Jones. Springer-Verlag, pp. 1–26.
- Fano, R. M. (1961). *Transmission of Information*. Cambridge, MA/New York, NY: MIT Press/Wiley. [Имеется русский перевод: Фано Р. Передача информации. М.: Мир. 1965.]
- Gibbons, J. (2007). Metamorphisms: streaming representation-changers. *Science of Computer Programming* **65**, 108–39.
- Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers* **40** (9), 1098–101.
- Mackay, D. (2003). *Information Theory, Learning and Inference Algorithms*. Cambridge, UK: Cambridge University Press.
- Moffat, A., Neal, R. M. and Witten, I. H. (1998). Arithmetic coding revisited. *ACM Transactions on Information Systems* **16** (3), 256–94.
- Stratford, B. (2005). *A formal treatment of lossless data compression*. DPhil thesis, Oxford University Computing Laboratory, Oxford, UK.

Witten, I. H., Neal, R. M. and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM* **30** (6), 520–40.

Приложение

Теорема о потоковой обработке может быть доказана с отсылкой к более общей теореме об *unfoldr*. Последняя утверждает, что $\text{unfoldr } f \cdot g = \text{unfoldr } h$, если выполнены два условия:

$$\begin{aligned} h x &= \text{Nothing} \Rightarrow f(g x) = \text{Nothing} \\ h x &= \text{Just}(y, x') \Rightarrow f(g x) = \text{Just}(y, g x') \end{aligned}$$

Этот результат известен как закон слияния для *unfoldr*. В частности, условия слияния для

$$\text{unfoldr } \text{step}(s, xs) = \text{unfoldr } f(\text{foldl } g s xs)$$

где *xs* соответствует конечному списку и

$$\begin{aligned} \text{step}(s, xs) &= \text{case } f s \text{ of} \\ &\quad \text{Just}(y, s') \rightarrow \text{Just}(y, (s', xs)) \\ &\quad \text{Nothing} \rightarrow \text{case } xs \text{ of} \\ &\quad \quad x : xs \rightarrow \text{step}(g s x, xs) \\ &\quad \quad [] \rightarrow \text{Nothing} \end{aligned}$$

принимают следующий вид:

$$\text{step}(s, xs) = \text{Nothing} \Rightarrow f(\text{foldl } g s xs) = \text{Nothing}$$

и

$$\begin{aligned} \text{step}(s, xs) &= \text{Just}(y, (s', xs')) \\ &\Rightarrow f(\text{foldl } g s xs) = \text{Just}(y, \text{foldl } g s' xs') \end{aligned}$$

для всех конечных списков *xs*. Первое условие проверить просто, второе следует отсюда:

$$f s = \text{Just}(y, s') \Rightarrow f(\text{foldl } g s xs) = \text{Just}(y, \text{foldl } g s' xs)$$

для всех конечных списков *xs*, что может быть доказано индукцией по *xs* с опорой на то, что условия потоковой обработки выполнены для *f* и *g*.

25

Целочисленное арифметическое кодирование

Введение

Эта жемчужина продолжает изучение арифметического кодирования, начатое в прошлой главе. Наша цель теперь состоит в том, чтобы заменить рациональную арифметику целочисленной. Основная мысль заключается в представлении сужаемого интервала парой целых чисел (ℓ, r) ограниченного размера, где $0 \leq \ell < r \leq 2^e$ и e это заданное целое. Интервалы, создаваемые моделью, имеют тот же самый вид, но с другой экспонентой d вместо e . Как мы увидим позже, необходимо $d \leq e - 2$, так что эти два числа действительно не могут совпадать. Значения e и d будем считать глобальными константами, достаточно малыми, чтобы можно было производить все вычисления с целыми числами фиксированного размера, например, с типом данных *Int* языка Haskell.

Новые определения

Теперь положим $\text{Interval} = (\text{Int}, \text{Int})$, так что $\text{interval } m\,x$ возвращает пару (p, q) целых чисел ограниченного размера, обозначающих интервал $(p/2^d, q/2^d)$. Функция $\text{symbol } m$ принимает целое число n в промежутке $0 \leq n < 2^d$ и возвращает символ x . Как и раньше, $x = \text{symbol } m\,n$ тогда и только тогда, когда $n \in \text{interval } m\,x$, лишь с тем изменением, что n теперь это целое число.

Далее, изменим определение операции сужения, задав функцию \blacktriangleright вместо \triangleright :

$(\blacktriangleright) :: Interval \rightarrow Interval \rightarrow Interval$

$$(\ell, r) \blacktriangleright (p, q) = (\ell + \lfloor (r - \ell) * p / 2^d \rfloor, \ell + \lfloor (r - \ell) * q / 2^d \rfloor)$$

Наибольшее целое, которое может появиться в вычислениях \blacktriangleright это 2^{e+d} (это потенциальный размер $(r - \ell) * q$); если оно умещается в тип *Int*, то и все операции с интервалами удовлетворяют этому требованию.

Теперь обратимся к функции *toBits* из предыдущей жемчужины. Она преобразует дробный интервал в список битов и определяется так: $toBits = unfoldr bit$, где

$$\begin{aligned} bit(\ell, r) \mid r \leqslant 1/2 &= Just(0, (2 * \ell, 2 * r)) \\ \mid 1/2 \leqslant r &= Just(1, (2 * \ell - 1, 2 * r - 1)) \\ \mid \text{otherwise} &= Nothing \end{aligned}$$

Для целочисленного кодирования требуется заменить *bit* на *ibit*, которая бы использовала целые числа фиксированной длины:

$$\begin{aligned} ibit(\ell, r) \mid r \leqslant 2^{e-1} &= Just(0, (2 * \ell, 2 * r)) \\ \mid 2^{e-1} \leqslant r &= Just(1, (2 * \ell - 2^e, 2 * r - 2^e)) \\ \mid \text{otherwise} &= Nothing \end{aligned}$$

Функция *ibit* представляет собой масштабированный вариант *bit*, так что $2^e * toFrac(unfoldr ibit i) \in i$.

Исправленные версии функций, приведённые выше, позволяют получить новое определение *encode*:

$$encode_1 m = unfoldr ibit \cdot foldl (\blacktriangleright) (0, 2^e) \cdot intervals m \quad (25.1)$$

Словом, символы текста преобразуются в интервалы, которые затем используются для сужения интервала $(0, 2^e)$ к некоторому заключительному интервалу i ; по нему определяется битовая строка, которая после преобразования к дроби и умножения на 2^e даёт число в интервале i . Всё это выглядит достаточно незатейливо.

Правда, сложность с определением (25.1) состоит в том, что оно не работает! Сужение с помощью \blacktriangleright приведёт рано или поздно к пустому интервалу, чего не могло случиться с \triangleright . Поясним это на таком примере: пусть $e = 5$, $d = 3$ и m сопоставляет интервал $(3, 5)$ букве «а» и $(5, 6)$ букве «б». Если $adapt m x = m$, то есть m является статичной моделью, то

$$\begin{aligned} & encode_1 m "bba" \\ &= foldl (\blacktriangleright) (0, 32) [(5, 6), (5, 6), (3, 5)] \\ &= foldl (\blacktriangleright) (20, 24) [(5, 6), (3, 5)] \end{aligned}$$

$$\begin{aligned}
 &= foldl (\blacktriangleright) (22, 23) [(3, 5)] \\
 &= (22, 22)
 \end{aligned}$$

Более того, *unfoldr ibit* (22, 22) порождает неограниченное количество мусора. Ой!

Инкрементное кодирование и расширение интервала

Выйти из положения помогает сочетание двух идей: инкрементного кодирования и расширения интервала. Во-первых, заменим (25.1) на

$$encode_2 m = stream ibit (\blacktriangleright) (0, 2^e) \cdot intervals m \quad (25.2)$$

Операции *ibit* и \blacktriangleright не удовлетворяют потоковому условию, так как \blacktriangleright не ассоциативна, и потому $encode_2 \neq encode_1$. Действительно, вернёмся к примеру выше:

$$\begin{aligned}
 &encode_2 m "bba" \\
 &= stream ibit (\blacktriangleright) (0, 32) [(5, 6), (5, 6), (3, 5)] \\
 &= stream ibit (\blacktriangleright) (20, 24) [(5, 6), (3, 5)] \\
 &= 101 : stream ibit (\blacktriangleright) (0, 32) [(5, 6), (3, 5)] \\
 &= 101101 : stream ibit (\blacktriangleright) (0, 32) [(3, 5)] \\
 &= 101101 : stream ibit (\blacktriangleright) (12, 20) [] \\
 &= 101101
 \end{aligned}$$

Интервал не сводится к пустому множеству. Это хорошие новости. С другой стороны:

$$\begin{aligned}
 &encode_2 m "aab" \\
 &= stream ibit (\blacktriangleright) (0, 32) [(3, 5), (3, 5), (5, 6)] \\
 &= stream ibit (\blacktriangleright) (12, 20) [(3, 5), (5, 6)] \\
 &= stream ibit (\blacktriangleright) (15, 17) [(5, 6)] \\
 &= stream ibit (\blacktriangleright) (16, 16) [] \\
 &= 0111 ...
 \end{aligned}$$

Интервал становится пустым, потому что промежуточные интервалы постепенно стягиваются к 16, а *ibit* в таких случаях возвращает *Nothing*.

Вывод в том, что инкрементное кодирование само по себе не спасает от схлопывания интервалов. Сложность возникает с интервалами, стягивающимися к 2^{e-1} , и вторая мысль, расширение интервалов, состоит в увеличении ширины таких интервалов самое меньшее до 2^{e-2} . Суженный интервал $(\ell, r) \blacktriangleright (p, q)$ не схлопнется, если:

$$\lfloor (r - \ell) * p / 2^d \rfloor < \lfloor (r - \ell) * q / 2^d \rfloor$$

для всех $p < q$, или, что то же, если $\lfloor (r - \ell) * p / 2^d \rfloor < \lfloor (r - \ell) * (p + 1) / 2^d \rfloor$ для всех p . Поскольку $\lfloor x \rfloor < \lfloor y \rfloor$, если $x + 1 \leq y$, то данное условие выполнено в случае $2^d \leq r - \ell$. Так что схлопывания не происходит, если $r - \ell \leq 2^{e-2}$, то есть если $d \leq e - 2$.

Расширение интервалов

Расширение интервалов это уточнение данных, при котором интервал (ℓ, r) представляется тройкой $(n, (\ell', r'))$, где $\ell' = \text{widen } n \ell$ и $r' = \text{widen } n r$, кроме того:

$$\text{widen } n x = 2^n(x - 2^{e-1}) + 2^{e-1}$$

Вполне расширенным называется интервал, в котором n максимально с учётом границ $0 \leq \ell' < r' \leq 2^e$. Например, при $e = 5$ интервал $(13, 17)$ длины 4 может быть представлен как вполне расширенный интервал $(2, (4, 20))$ длины 16.

Функция *expand* принимает интервал и возвращает соответствующий вполне расширенный интервал. Чтобы определить её и во избежание дробей и возведений в степень в будущем, определим четыре целых числа e_i , $1 \leq i \leq 4$, так: $e_i = (i/4)2^e$. Заметим, что

$$\begin{aligned} 0 \leq 2 * (\ell - e_2) + e_2 &\equiv e_1 \leq \ell \\ 2 * (r - e_2) + e_2 \leq e_4 &\equiv r \leq e_3 \end{aligned}$$

Таким образом, можно далее расширять $(n, (\ell, r))$ если $e_1 \leq \ell$ и $r \leq e_3$. Это приводит к определению *expand* в терминах вспомогательной функции *extend*:

$$\begin{aligned} \text{expand } i &= \text{extend}(0, i) \\ \text{extend } (n, (\ell, r)) \\ | e_1 \leq \ell \wedge r \leq e_3 &= \text{extend}(n + 1, 2 * \ell - e_2, 2 * r - e_2) \\ | \text{otherwise} &= (n, (\ell, r)) \end{aligned}$$

Обратная к *expand* функция *contract* определяется так:

$$\text{contract } (n, (\ell, r)) = (\text{shorten } n \ell, \text{shorten } n r)$$

где $\text{shorten } n x = (x - e_2) / 2^n + e_2$. Имеем $\text{shorten } n \cdot \text{widen } n = id$, из чего следует $\text{contract} \cdot \text{expand} = id$, однако в общем случае $\text{expand} \cdot \text{contract} \neq id$. Это обычная ситуация с уточнением данных.

Далее мы определим *enarrow*:

$$\begin{aligned} \text{enarrow} &:: (\text{Int}, \text{Interval}) \rightarrow \text{Interval} \rightarrow (\text{Int}, \text{Interval}) \\ \text{enarrow } ei \ j &= (n, i \blacktriangleright j) \text{ where } (n, i) = \text{extend } ei \end{aligned}$$

Таким образом, *enarrow* принимает частично расширенный интервал, вполне расширяет его и затем сужает с помощью \blacktriangleright . Следовательно, (ℓ, r) сужается только в случае $\ell < e_1$ или $e_3 < r$. Если, кроме того, $\ell < e_2 < r$, то имеет место либо $\ell < e_1$ и $e_2 < r$, либо $\ell < e_2$ и $e_3 < r$. В любом случае $e_1 < r - \ell$, что и требовалось показать.

Новое определение

Заменим (25.2) третьим, совершенно новым определением:

$$\text{encode}_3 m = \text{stream } ebit \text{ enarrow } (0, (0, 2^e)) \cdot \text{intervals } m \quad (25.3)$$

Функция *ebit* соответствует *ibit*, но работает на расширенных интервалах и определяется свойством:

$$\text{unfoldr } ebit = \text{unfoldr } ibit \cdot \text{contract} \quad (25.4)$$

где *ibit* определена, как показано выше. Явное определение *ebit* рассматривается ниже. Функция *ebit* будет возвращать *Nothing* на интервалах, которые стягиваются к e_2 , так *encode*₃ убеждается, что интервалы, сужаются \blacktriangleright , только если их ширина не меньше e_1 , что позволяет избежать склонения интервалов при условии, что $d \leq e - 2$.

Функция *encode*₃ отличается от всех предыдущих версий *encode*. Это означает, что нам снова нужно задуматься об определении *decode*. Отложим дискуссию о связи *encode*₃ *m xs* и *xs* до тех пор, пока не будет явно задана функция *ebit*.

Уравнение (25.4) подсказывает идею обратиться к закону слияния для *unfoldr* (использованному в Приложении к прошлой жемчужине). Он гласит:

$$\text{unfoldr } h = \text{unfoldr } f \cdot g$$

если выполнено:

$$\begin{aligned} h x &= \text{Nothing} \Rightarrow f(g x) = \text{Nothing} \\ h x &= \text{Just}(y, x') \Rightarrow f(g x) = \text{Just}(y, g x') \end{aligned}$$

Положив $h = ebit$, $f = ibit$ и $g = contract$, нужно показать:

$$\begin{aligned} ebit\ x &= \text{Nothing} \Rightarrow ibit(\text{contract } x) = \text{Nothing} \\ ebit\ x &= \text{Just}(y, x') \Rightarrow ibit(\text{contract } x) = \text{Just}(y, \text{contract } x') \end{aligned}$$

Следующее определение *ebit* удовлетворяет этим условиям:

$$\begin{aligned} ebit(0, (\ell, r)) &= \begin{cases} r \leqslant e_2 & = \text{Just}(0, (0, (2 * \ell, 2 * r))) \\ e_2 \leqslant \ell & = \text{Just}(1, (0, (2 * \ell - e_4, 2 * r - e_4))) \\ \text{otherwise} & = \text{Nothing} \end{cases} \\ ebit(n + 1, (\ell, r)) &= \begin{cases} r \leqslant e_2 & = \text{Just}(0, (n, (\ell + 2^n * e_2, r + 2^n * e_2))) \\ e_2 \leqslant \ell & = \text{Just}(1, (n, (\ell - 2^n * e_2, r - 2^n * e_2))) \\ \text{otherwise} & = \text{Nothing} \end{cases} \end{aligned}$$

Ниже эта запись будет упрощена. Пусть $\text{contract}(n, (\ell, r)) = (\ell', r')$, легко проверить, что $r \leqslant e_2 \equiv r' \leqslant e_2$ и $e_2 \leqslant \ell \equiv e_2 \leqslant \ell'$, так что первое условие слияния выполнено. Второе получается непосредственно в случае $n = 0$, так как $\text{contract}(0, i) = i$. Оставшийся случай сводится к равенству:

$$2 * \text{shorten}(n + 1)x - e_4 * b = \text{shorten } n(x + (1 - 2b) * 2^n * e_2)$$

для $b = 0$ и $b = 1$, в чём легко убедиться.

Определение *ebit* неэффективно, кроме того, что неуклюже, однако оно может быть улучшено. Заметим, что $e_2 \leqslant \ell + 2^n e_2$ и $r - 2^n e_2 \leqslant e_2$ для всех $n \geqslant 0$. То есть в случае $r \leqslant e_2$ вычисление *unfoldr ebit*($n, (\ell, r)$) выглядит так:

$$\begin{aligned} &\text{unfoldr ebit}(n, (\ell, r)) \\ &= 0 : \text{unfoldr ebit}(n - 1, (\ell + 2^{n-1}e_2, r + 2^{n-1}e_2)) \\ &= 01 : \text{unfoldr ebit}(n - 2, (\ell + 2^{n-2}e_2, r + 2^{n-2}e_2)) \\ &= \dots \\ &= 01^{n-1} : \text{unfoldr ebit}(0, (\ell + e_2, r + e_2)) \\ &= 01^n : \text{unfoldr ebit}(0, (2\ell, 2r)) \end{aligned}$$

где 01^n означает нуль, за которым следует n единиц. Аналогично, если $e_2 \leqslant \ell$, то

$$\text{unfoldr ebit}(n, (\ell, r)) = 10^n : \text{unfoldr ebit}(0, (2\ell - e_4, 2r - e_4))$$

Таким образом, *unfoldr ebit* = *concat* · *unfoldr ebits*, где

$$\begin{aligned} \text{ebits}(n, (\ell, r)) &= \begin{cases} r \leqslant e_2 & = \text{Just}(\text{bits } n\ 0, (0, (2 * \ell, 2 * r))) \end{cases} \end{aligned}$$

$$\begin{cases} e_2 \leq \ell & = \text{Just}(\text{bits } n 1, (0, (2 * \ell - e_4, 2 * r - e_4))) \\ \text{otherwise} & = \text{Nothing} \end{cases}$$

и $\text{bits } n b = b : \text{replicate } n (1 - b)$ возвращает b , за которым следует n экземпляров $1 - b$. Из всего сказанного вытекает, что (25.3) можно заменить эквивалентным, но более эффективным определением:

$$\text{encode}_3 m = \text{concat} \cdot \text{stream ebits enarrow} (0, (0, 2^e)) \cdot \text{intervals } m \quad (25.5)$$

Выражение (25.5) является нашим окончательным алгоритмом для encode .

Ключевой вопрос

Но что в сущности делает функция encode_3 ? Как её выход связан со входом? Версия encode из прошлой жемчужины удовлетворяла условию:

$$\text{toFrac} (\text{encode } m xs) \in \text{foldl} (\triangleright) (0, 1) (\text{intervals } m xs)$$

Но это неверно для encode_3 .

Чтобы ответить на этот ключевой вопрос, определим версию encode'_3 функции encode_3 , которая принимает в качестве дополнительного аргумента начальный интервал (что можно было бы сделать с самого начала):

$$\text{encode}'_3 m ei = \text{concat} \cdot \text{stream ebits enarrow} ei \cdot \text{intervals } m$$

Тогда имеем:

$$2^e * \text{toFrac} (\text{encode}'_3 m ei xs) \in \text{contract} ei \quad (25.6)$$

для всех моделей m , расширенных интервалов ei и списков символов xs . Свойство (25.6), доказательство которого дано в Приложении, является решающим в реализации decode .

Последняя трудность

К сожалению, в (25.5) приведена версия encode , которая в некоторых случаях может некорректно работать с целыми числами фиксированного размера. Трудность связана с числом n во вполне расширенном интервале (n, i) . Легко видеть, что n может достигать очень больших значений, в том числе таких, которые не представляются типом Int . Например, рассмотрим многократное сужение интервала $(0, e_4)$ с помощью $(3/8, 5/8)$. Сужаемый

интервал строго стремится к e_2 , так что результатом *encode* станет пустой список битов. Если на каждом шаге применять расширение интервала, то в итоге будет получен расширенный интервал $(n, (0, e_4))$, где n может превышать максимальное значение целых чисел ограниченного размера. Конечно, такое вряд ли случится на практике, но в теории это возможно. Легко убедиться, что никакая версия арифметического кодирования не может работать с какой бы то ни было формой ограниченных целых чисел в любом случае. Если возникает ситуация, как описанная выше, то выход в том, чтобы подать соответствующее сообщение об ошибке либо переключиться обратно на рациональное арифметическое кодирование.

Обращаем потоки

Теперь займёмся задачей декодирования. Функция *decode* задаётся условием $xs \sqsubseteq decode m (encode m xs)$. С определением *encode*, данным в (25.3) или эквивалентно в (25.5), единственный способ удовлетворить этому условию состоит в обращении потоков. Воспользуемся функцией *destream*, определённой так:

```
destream f g h s ys = unfoldr step (s, ys)
where step (s, ys) = case f s of
    Just (y, s') → step (s', ys ↓ [y])
    Nothing → Just (x, (g s x, ys))
where x = h s ys
```

Операция \downarrow определяется так: $(us \dagger\! \dagger vs) \downarrow us = vs$. Функция *destream* двойственна к *stream*: если $f s$ возвращает некоторый y , то y удаляется из головы входа ys ; если $f s$ не возвращает ничего, очередная порция результата порождается вспомогательной функцией h .

Связь между *stream* и *destream* даётся следующей теоремой, которая называется теоремой об извлечении из потока. Её доказательство приведено в Приложении.

Теорема 25.1. Пусть $stream f g s xs$ возвращает конечный список, а функция h удовлетворяет условию: $h s (stream f g s (x : xs)) = x$, если $f s = Nothing$. В этих предположениях имеет место:

$$xs \sqsubseteq destream f g h s (stream f g s xs)$$

Чтобы применить теорему к *encode*, положим $f = ebit$, $g = enarrow$ и $s = ei$. Тогда

$$\text{decode } m = \text{destream ebit enarrow } h(m, (0, (0, e_4)))$$

при условии, что вспомогательная функция h удовлетворяет

$$h(m, ei) (\text{encode}'_3 m ei (x : xs)) = x \quad (25.7)$$

для всех интервалов ei , которые стягиваются к e_2 .

Определение decode может быть улучшено по аналогии с тем, как (25.3) было переведено в (25.4) заменой $ebit$ на $ebits$. В результате получится:

$$\begin{aligned} \text{decode } m bs &= \text{unfoldr step}(m, (0, (0, e_4)), bs) \\ \text{step}(m, (n, (\ell, r)), bs) &= \begin{cases} r \leq e_2 & = \text{step}(m, (0, (2 * \ell, 2 * r)), bs \downarrow \text{bits } n \ 0) \\ e_2 \leq \ell & = \text{step}(m, (0, (2 * \ell - e_4, 2 * r - e_4)), bs \downarrow \text{bits } n \ 1) \\ \text{otherwise} & = \text{Just}(x, (\text{adapt } m \ x, \\ &\quad \text{enarrow}(n, (\ell, r))(\text{interval } m \ x), bs)) \end{cases} \\ \text{where } x &= h(m, (n, (\ell, r))) \ bs \end{aligned}$$

Остается найти функцию h .

Вспомогательная функция

Начнём с вычисления, приводящего к определению \blacktriangleleft , операции, которая относится к \blacktriangleright так же, как \lhd к \rhd . Напомним, что

$$f \lhd (\ell, r) = (f - \ell) / (r - \ell)$$

и $f \in i \triangleright j \equiv (f \lhd i) \in j$. Вычисление использует важное свойство, известное как правило округления: $n \leq f \equiv n \leq \lfloor f \rfloor$ для всех целых n и вещественных f . Пусть k, ℓ, r, p и q это произвольные числа. Имеем:

$$\begin{aligned} k &\in (\ell, r) \triangleright (p, q) \\ &\equiv \{ \text{определение } \triangleright \} \\ &\quad \ell + \lfloor (r - \ell) * p / 2^d \rfloor \leq k < \ell + \lfloor (r - \ell) * q / 2^d \rfloor \\ &\equiv \{ \text{арифметика} \} \\ &\quad \lfloor (r - \ell) * p / 2^d \rfloor < k - \ell + 1 \leq \lfloor (r - \ell) * q / 2^d \rfloor \\ &\equiv \{ \text{правило округления} \} \\ &\quad (r - \ell) * p / 2^d < k - \ell + 1 \leq (r - \ell) * q / 2^d \\ &\equiv \{ \text{арифметика} \} \end{aligned}$$

$$\begin{aligned}
 p &\leq ((k - \ell + 1) * 2^d - 1) / (r - \ell) < q \\
 &\equiv \{ \text{правило округления} \} \\
 p &\leq \lfloor ((k - \ell + 1) * 2^d - 1) / (r - \ell) \rfloor < q
 \end{aligned}$$

Таким образом, $k \in (i \blacktriangleright j) \equiv (k \blacktriangleleft i) \in j$, где

$$\begin{aligned}
 (\blacktriangleleft) &\quad :: \text{Int} \rightarrow \text{Interval} \rightarrow \text{Int} \\
 k \blacktriangleleft (\ell, r) &= ((k - \ell + 1) * 2^d - 1) \text{ div } (r - \ell)
 \end{aligned}$$

Далее, напомним свойство (25.6) из прошлого раздела: для всех ei и xs

$$2^e * \text{toFrac}(\text{encode}'_3 m ei xs) \in \text{contract} ei$$

Равносильная формулировка с использованием определений *contract* и *widen*:

$$\text{widen } n (2^e * \text{toFrac}(\text{encode}'_3 m (n, i) xs)) \in ei$$

В предположении, что интервал i имеет целочисленные границы, и вновь по правилу округления это эквивалентно:

$$\lfloor \text{widen } n (2^e * \text{toFrac}(\text{encode}'_3 m (n, i) xs)) \rfloor \in i$$

Далее, пусть вполне расширенный интервал (n, i) стягивается к e_2 , так что

$$\begin{aligned}
 \text{encode}'_3 m (n, i) (x : xs) \\
 = \text{encode}'_3 (\text{adapt } m x) (n, i \blacktriangleright \text{interval } m x) xs
 \end{aligned}$$

В этом случае имеем

$$\lfloor \text{widen } n (2^e * \text{toFrac}(\text{encode}'_3 m (n, i) (x : xs))) \rfloor \in i \blacktriangleright \text{interval } m xs$$

Наконец, вспомним о связи *symbol* и *interval*, а именно, о том, что $x = \text{symbol } m n$, если и только если $n \in \text{interval } m x$. Отсюда следует, что h может быть определена следующим образом:

$$\begin{aligned}
 h(m, ei) bs &= \text{symbol } m (\lfloor \text{widen } n (2^e * \text{toFrac} bs) \rfloor \blacktriangleleft i) \\
 \text{where } (n, i) &= \text{extend } ei
 \end{aligned}$$

Инкрементное декодирование

Текущее определение *decode* имеет несколько недостатков: оно использует рациональную арифметику (при подсчёте h , так как *toFrac* bs это дробь), оно не инкрементно и очень неэффективно. Вычисление *extend* повторяется в h и *enarrow*, функция *toFrac* выполняется заново для каждого выходного символа, а *widen n* использует возведение в степень, которое представляет собой дорогую операцию. В общем, *decode*, мягко говоря, нельзя признать удачной. Однако если сделать функцию *decode* инкрементной, можно преодолеть все перечисленные недостатки.

Переход к инкрементному определению *decode* выполняется в три этапа. Во-первых, удалим все зависимости от функции *expand*, включив соответствующие вычисления в новую версию *step*:

$$\begin{aligned} \text{step } (m, (n, (\ell, r)), bs) &= \text{step } (m, (0, (2 * \ell, 2 * r)), bs \downarrow \text{bits } n \ 0) \\ | \ r \leqslant e_2 &= \text{step } (m, (0, (2 * \ell - e_4, 2 * r - e_4)), bs \downarrow \text{bits } n \ 1) \\ | \ e_2 \leqslant \ell &= \text{step } (m, (n + 1, (2 * \ell - e_2, 2 * r - e_2)), bs) \\ | \ e_1 \leqslant \ell \wedge r \leqslant e_3 &= \text{Just } (x, \\ | \ \text{otherwise} &= \text{adapt } m \ x, (n, (\ell, r) \blacktriangleright \text{interval } m \ x), bs)) \\ \text{where } x &= \text{symbol } m \ (\lfloor \text{widen } n \ (e_4 * \text{toFrac } bs) \rfloor \blacktriangleleft (\ell, r)) \end{aligned}$$

Идея в том, что если $\text{step } (m, ei, bs)$ возвращает хоть что-то, то только в том случае, когда ei представляет собой вполне расширенный интервал, так что *enarrow* можно заменить на \blacktriangleright .

Далее, покажем, как избежать повторных вычислений *toFrac*. Зададим f и step' такими правилами:

$$\begin{aligned} f \ n \ bs &= \text{widen } n \ (e_4 * \text{toFrac } bs) \\ \text{step}' \ (m, (n, i), f \ n \ bs) &= \text{step } (m, (n, i), bs) \end{aligned}$$

Идея в том, чтобы в качестве третьего аргумента *step* принимать $f \ n \ bs$ вместо bs , где n это коэффициент расширения во втором аргументе. В качестве упражнения остаётся показать, что

$$\begin{aligned} f \ 0 \ (bs \downarrow \text{bits } n \ b) &= 2 * f \ n \ bs - e_4 * b \\ f \ (n + 1) \ bs &= 2 * f \ n \ bs - e_2 \end{aligned}$$

Это приводит к следующей версии *decode*, в которой *step'* переименована в *step*:

$$\text{decode } m \ bs = \text{unfoldr } \text{step } (m, (0, (0, e_4)), e_4 * \text{toFrac } bs) \quad (25.8)$$

```

decode m bs = unfoldr step (m, (0, e4), toInt (take e bs'), drop e bs')
  where bs' = bs ++ 1 : repeat 0

step (m, (ℓ, r), n, b : bs)
| r ≤ e2           = step (m, (2 * ℓ, 2 * r), 2 * n + b, bs)
| e2 ≤ ℓ           = step (m, (2 * ℓ - e4, 2 * r - e4), 2 * n - e4 + b, bs)
| e1 ≤ ℓ ∧ r ≤ e3 = step (m, (2 * ℓ - e2, 2 * r - e2), 2 * n - e2 + b, bs)
| otherwise          = Just (x,
                        (adapt m x, (ℓ, r) ▷ interval m x, n, b : bs))
  where x = symbol m (n ▲ (ℓ, r))

```

Рис. 25.1: заключительная версия decode

где

```

step (m, (n, (ℓ, r)), f)
| r ≤ e2           = step (m, (0, (2 * ℓ, 2 * r)), 2 * f)
| e2 ≤ ℓ           = step (m, (0, (2 * ℓ - e4, 2 * r - e4)), 2 * f - e4)
| e1 ≤ ℓ ∧ r ≤ e3 = step (m, (n + 1, (2 * ℓ - e2, 2 * r - e2)), 2 * f - e2)
| otherwise          = Just (x,
                        (adapt m x, (n, (ℓ, r)) ▷ interval m x), f))
  where x = symbol m (lf] ▲ (ℓ, r))

```

Здесь видно, что присутствие n избыточно, мы можем избавиться от него:

```

step (m, (ℓ, r), f)
| r ≤ e2           = step (m, (2 * ℓ, 2 * r), 2 * f)
| e2 ≤ ℓ           = step (m, (2 * ℓ - e4, 2 * r - e4), 2 * f - e4)
| e1 ≤ ℓ ∧ r ≤ e3 = step (m, (2 * ℓ - e2, 2 * r - e2), 2 * f - e2)
| otherwise          = Just (x,
                        (adapt m x, (ℓ, r) ▷ interval m x, f))
  where x = symbol m (lf] ▲ (ℓ, r))

```

Наконец, всё готово для записи инкрементного вычисления. Заметим, что в (25.8) в силу $e_4 = 2^e$ выражение $\lfloor e_4 * \text{toFrac } bs \rfloor$ зависит лишь от первых e элементов bs . В самом деле,

$$\lfloor e_4 * \text{toFrac } bs \rfloor = \text{toInt} (\text{take } e (bs ++ 1 : \text{repeat } 0))$$

где $\text{toInt} = \text{foldl} (\lambda n b \rightarrow 2 * n + b) 0$ переводит битовую строку в целое число. Стока bs дополняется достаточным количеством элементов списка

$1 : repeat 0$, чтобы итоговая длина составляла не менее, чем e символов.
Более того, если $bs' = bs \uparrow\downarrow 1 : repeat 0$, то

$$\lfloor 2 * e_4 * toFrac bs \rfloor = toInt(take(e + 1) bs') \\ = 2 * toInt(take e bs') + head(drop e bs')$$

Это означает, что третий аргумент f функции *step* можно заменить парой (n, ds) , где $n = toInt(take e bs')$ и $ds = drop e bs'$, а $bs' = bs \uparrow\downarrow 1 : repeat 0$, что приводит к заключительной версии *decode*, которая отображена на рис. 25.1.

Заключительные замечания

Читатель, дошедший до победного конца, оценит, как много арифметики кроется в арифметическом кодировании. Включая арифметику свёрток и разверток наряду с обычной числовой арифметикой. Как и было сказано в прошлой жемчужине, арифметическое кодирование представляет собой простую идею, которая требует деликатного обращения в реализации с использованием целых чисел ограниченного размера.

Приложение

Доказательство теоремы 25.1 полагается на следующие два свойства, каждое из которых является простым следствием определений *stream* и *destream*:

$$f s = Nothing \Rightarrow \\ stream f g s (x : xs) = stream f g (g s x) xs \wedge \\ destream f g h s ys = x : destream f g h (g s x) ys \\ f s = Just(y, s') \Rightarrow \\ stream f g s xs = y : stream f g s' xs \wedge \\ destream f g h s (y : ys) = destream f g h s' ys$$

В первом свойстве x определяется так: $x = h s ys$. Докажем, что

$$xs \sqsubseteq destream f g h s (stream f g s xs)$$

парной индукцией по xs и n , где n представляет длину $stream f g s xs$.

Случай []: следует немедленно, так как [] является префиксом любого списка.

Случай $x : xs$: сначала рассмотрим подслучай $f s = Nothing$. Первое свойство выше даёт:

$$\begin{aligned} & \text{destream } f \ g \ h \ s \ (\text{stream } f \ g \ s \ (x : xs)) \\ &= x : \text{destream } f \ g \ h \ (g \ z \ x) \ (\text{stream } f \ g \ (g \ z \ x) \ xs) \end{aligned}$$

Поскольку $x : xs \sqsubseteq x : xs'$ тогда и только тогда, когда $xs \sqsubseteq xs'$, используем предположение индукции.

В случае $f \ s = \text{Just } (y, s')$ второе свойство даёт:

$$\begin{aligned} & \text{destream } f \ g \ h \ s \ (\text{stream } f \ g \ s \ (x : xs)) \\ &= \text{destream } f \ g \ h \ s' \ (\text{stream } f \ g \ s' \ (x : xs)) \end{aligned}$$

Но поскольку $\text{length } (\text{stream } f \ g \ z' \ (x : xs)) = n - 1$, мы снова обращаемся к предположению индукции и завершаем доказательство.

Последняя задача состоит в том, чтобы доказать

$$e_4 * \text{toFrac } (\text{encode}'_3 m ei xs) \in \text{contract } ei$$

Доказательство ведётся парной индукцией по xs и n , где n представляет собой длину $\text{encode}'_3 m ei xs$.

Случай []: здесь имеем $\text{encode}'_3 m ei [] = \text{concat } (\text{unfoldr } ebits ei)$. Тогда

$$\begin{aligned} & e_4 * (\text{toFrac } (\text{concat } (\text{unfoldr } ebits ei))) \in \text{contract } ei \\ &\equiv \{ \text{определение } ebit \} \\ & e_4 * (\text{toFrac } (\text{unfoldr } ebit ei)) \in \text{contract } ei \\ &\equiv \{ \text{в силу } \text{unfoldr } ebit = \text{unfoldr } ibit \cdot \text{contract} \} \\ & e_4 * (\text{toFrac } (\text{unfoldr } ibit (\text{contract } ei))) \in \text{contract } ei \\ &\Leftarrow \{ \text{определение } ibit \} \\ & \text{true} \end{aligned}$$

Что и показывает справедливость включения в данном случае.

Случай $x : xs$: нам потребуется альтернативное определение encode'_3 :

$$\begin{aligned} & \text{encode}'_3 m (n, (\ell, r)) (x : xs) \\ & | r \leqslant e_2 = \text{bits } n \ 0 \mathbin{++} \text{encode}'_3 m (0, (2\ell, 2r)) (x : xs) \\ & | e_2 \leqslant \ell = \text{bits } n \ 1 \mathbin{++} \text{encode}'_3 m (0, (2\ell - e_4, 2r - e_4)) (x : xs) \\ & | \text{otherwise} = \text{encode}'_3 (\text{adapt } m x) ej xs \\ & \quad \text{where } ej = \text{enarrow } (n, (\ell, r)) (\text{interval } m x) \end{aligned}$$

По индукции имеем: $\text{encode}'_3 (\text{adapt } m x) ej xs \in \text{contract } ej$. Но

$$\text{contract } (\text{enarrow } ej j) \subseteq \text{contract } (\text{extend } ei) = \text{contract } ei$$

Это подтверждает справедливость третьей части определения $encode'_3$.
Для доказательства оставшихся двух частей заметим, что длина

$$encode'_3 m (0, (2\ell - be_4, 2r - be_4)) (x : xs)$$

меньше n . Отсюда по индукции получается:

$$\begin{aligned} e_4 * toFrac (encode'_3 m (0, (2\ell - be_4, 2r - be_4)) (x : xs)) \\ \in (2\ell - be_4, 2r - be_4) \end{aligned}$$

Наконец, поскольку $toFrac (bits n b ++ bs) = (2^n + (b - 1) + toFrac bs) / 2^{n+1}$,
простой подсчёт показывает, что $e_4 * toFrac bs \in (2\ell - be_4, 2r - be_4)$, если и
только если

$$e_4 * toFrac (bits n b ++ bs) \in contract (n, (\ell, r))$$

что подтверждает справедливость принадлежности в рассматриваемом
случае и завершает доказательство.

26

Алгоритм Шора—Вейта

Введение

Алгоритм Шора—Вейта представляет собой метод разметки вершин ориентированного графа, достижимых из заданной начальной вершины. Рассматриваются только графы с полустепенью исхода, равной двум. Алгоритм состоит из итеративного цикла, который реализует поиск в глубину на графе, но не использует явного стека для контроля за обходом. Вместо этого стек симулируется модификацией самого графа, который, однако, восстанавливается к первоначальному виду в конце процесса разметки. Алгоритм весьма запутан. В (Morris, 1982) он описан как «наиболее непокорный алгоритм», а в (Borod, 2000) как «первое существенное препятствие для любого формализма анализа псевдонимов в присутствии указателей».

Цель нашей жемчужины в том, чтобы представить алгоритм Шора—Вейта как упражнение в объяснении алгоритмов с помощью преобразования программ. Мы начнём с простой версии алгоритма разметки, а затем изложим три преобразования, которые приведут к итоговому алгоритму. Каждое преобразование описывает различное представление стека, в завершении оно сводится к связному списку, встроенному в граф.

Спецификация

В постановке задачи Шора и Вейта граф задаёт S-выражение Маккарти (McCarthy, 1960) и потому имеет полустепень исхода, равную двум.

Вершины представлены целыми числами. Определим

```
type Node = Int
type Graph = Node → (Node, Node)
```

Операции $left, right :: Graph \rightarrow Node \rightarrow Node$ извлекают информацию, связанную с вершиной, а

```
setl, setr :: Graph → Node → Node → Graph
```

модифицируют эту информацию. Так, $left (setl g x y) x = y$, и аналогичное справедливо для $setr$.

Функция разметки $mark$ принимает граф g и начальную вершину $root$ и возвращает булево-значочную функцию m , такую что $m x = True$, если и только если вершина x достижима из $root$. Реализуем функцию $mark$ g таким образом, чтобы она возвращала g и m . Причина такого решения в том, что итоговый алгоритм модифицирует g в процессе работы; таким образом, свидетельство эквивалентности различных версий $mark$ гарантирует не только то, что итоговое значение m одинаково в разных версиях, но что это же справедливо и для g . Итак, $mark$ имеет тип

```
mark :: Graph → Node → (Graph, Node → Bool)
```

Вполне разумной альтернативой является встраивание функции разметки в граф, так что он станет отображением из вершин в тройки. Тогда $mark$ должна возвращать лишь итоговый граф. Однако настоящая версия, в которой функция разметки хранится отдельно, одновременно проще и удобней для формальных рассуждений.

Функция $mark$ реализуется с помощью стандартного алгоритма поиска в глубину, опирающегося на стек:

```
mark g root = seek0 (g, const False) [root]
seek0 (g, m) [] = (g, m)
seek0 (g, m) (x : xs)
| not (m x) = seek0 (g, set m x) (left g x : right g x : xs)
| otherwise = seek0 (g, m) xs
```

Функции set и $unset$ (которые понадобятся позже) определяются так:

```
set, unset :: (Node → Bool) → Node → (Node → Bool)
set f x = λy → if y == x then True else f y
unset f x = λy → if y == x then False else f y
```

Это определение $mark$ служит нам отправной точкой.

Безопасная замена

В рассуждениях об алгоритмах, которые включают работу с указателями, рано или поздно приходится сталкиваться с задачей безопасной замены. Чтобы проиллюстрировать это, определим функцию *replace*:

$$\begin{aligned} \text{replace} &:: \text{Eq } a \rightarrow (a \rightarrow b) \rightarrow a \rightarrow b \rightarrow (a \rightarrow b) \\ \text{replace } f \ x \ y &= \lambda z \rightarrow \text{if } z == x \text{ then } y \text{ else } f z \end{aligned}$$

То есть *replace* является обобщением функции *set*, введённой выше. Когда *replace* можно использовать безопасно? Например, когда следующие равенства

$$\begin{aligned} \text{map } f \ xs &= \text{map} (\text{replace } f \ x \ y) \ xs \\ \text{filter } p \ xs &= \text{filter} (\text{replace } p \ x \ y) \ xs \end{aligned}$$

выполняются? Ответ не удивителен: когда x отсутствует в списке xs . Другой вариант: когда $y = f x$ или $y = p x$, — тоже допустим, но не представляет интереса. Также неинтересно доказательство справедливости первого ответа. Ниже мы обратимся к безопасной замене неоднократно, каждый раз снабжая это соответствующим указанием.

Удаление повторяющихся элементов

Вернёмся к нашей задаче, стек устраниется последовательно с помощью эквивалентных преобразований *mark*. Цель первого преобразования состоит в избавлении от повторяющихся элементов стека, если использовать его для хранения лишь тех вершин, которые были размечены. Это преобразование подготавливает сцену для последующего обращения к безопасной замене. Функция *seek0* приводится к новой функции *seek1*, определённой так:

$$\text{seek1 } (g, m) \ x \ xs = \text{seek0 } (g, m) (x : \text{map} (\text{right } g) \ xs)$$

и удовлетворяющей инварианту: $\text{clean } m \ xs = \text{all } m \ xs \wedge \text{nodups } xs$. Получение непосредственного определения *seek1* не составляет труда и приводит к следующей версии *mark*:

$$\begin{aligned} \text{mark } g \ root &= \text{seek1 } (g, \text{const } \text{False}) \ root [] \\ \text{seek1 } (g, m) \ x \ xs \\ | \text{not } (m \ x) &= \text{seek1 } (g, \text{set } m \ x) (\text{left } g \ x) (x : xs) \end{aligned}$$

$$\begin{array}{lcl} \mid \text{null } xs & = & (g, m) \\ \mid \text{otherwise} & = & \text{seek1}(g, m) (\text{right } g (\text{head } xs)) (\text{tail } xs) \end{array}$$

Поскольку x добавляется в стек только при получении метки, а вершины размечаются, самое большое, однажды, стек не замусоривается повторяющимися значениями — он чист.

Согласование стека

Второе преобразование спроектировано для того, чтобы убедиться, что стек не только чист, но и согласован, это означает, что для каждой пары последовательных элементов x и y в нём выполнено $x = \text{left } g y$ или $x = \text{right } g y$. Это неверно для seek1 . Рассмотрим выражение $\text{seek1}(g, m) (\text{left } g x) (x : y : xs)$, в котором $\text{left } g x$ размечен, а $\text{right } g x$ нет. Тогда seek1 заменит x на стеке элементом $\text{right } g x$, который в общем случае не является левой или правой дочерней вершиной для y . Однако если бы удалось сохранить x на стеке, взведя для него некоторый флаг, чтобы избежать повторной обработки, то добавление на стек $\text{right } g x$ удовлетворило бы ограничению. Элемент x будет удалён позднее, после обработки всех его потомков.

Такое поведение можно организовать при помощи второй размечающей функции p . Определим функцию seek2 следующим образом:

$$\text{seek2}(g, m) p x xs = \text{seek1}(g, m) x (\text{filter } p xs)$$

чтобы она удовлетворяла инварианту: $\text{threaded } g mp x xs$, где

$$\begin{aligned} \text{threaded } g mp x xs &= \text{clean } m xs \wedge \\ &\quad \text{and } [\text{link } u v \mid (u, v) \leftarrow \text{zip } (x : xs) xs] \\ \text{where } \text{link } u v &= \text{if } p v \text{ then } u == \text{left } g v \text{ else } u == \text{right } g v \end{aligned}$$

Ниже мы будем обращаться к следующему факту с указанием «согласованность»: если $m x$ и $x \notin xs$, то

$$\begin{aligned} \text{threaded } g mp x xs \Rightarrow \text{threaded } g m (\text{set } p x) (\text{left } g x) (x : xs) \wedge \\ \text{threaded } g m (\text{unset } p x) (\text{right } g x) (x : xs) \end{aligned}$$

Теперь мы составим новую версию mark , основанную на seek2 . Очевидно, что

$$\text{mark } g x = \text{seek2}(g, \text{const False}) (\text{const False}) x []$$

так что остаётся лишь вычислить определение $seek2$. В случае $not(m \ x)$ справедливо такое рассуждение:

$$\begin{aligned}
 & seek2(g, m) p x xs \\
 = & \quad \{ \text{определение} \} \\
 & seek1(g, m) x (\text{filter } p xs) \\
 = & \quad \{ \text{предположение } not(m \ x) \} \\
 & seek1(g, set m x) (\text{left } g x) (x : \text{filter } p xs) \\
 = & \quad \{ \text{безопасная замена, в силу } x \notin xs \} \\
 & seek1(g, set m x) (\text{left } g x) (x : \text{filter}(\text{set } p x) xs) \\
 = & \quad \{ \text{в силу } \text{set } p x \ x = True \} \\
 & seek1(g, set m x) (\text{left } g x) (\text{filter}(\text{set } p x) (x : xs)) \\
 = & \quad \{ \text{определение } seek2 \text{ и согласованность} \} \\
 & seek2(g, set m x) (\text{set } p x) (\text{left } g x) (x : xs)
 \end{aligned}$$

Таким образом,

$$seek2(g, m) p x xs = seek2(g, set m x) (\text{set } p x) (\text{left } g x) (x : xs)$$

В случае $m \ x$ необходимо найти в стеке первый элемент, удовлетворяющий p , так как именно его надо обрабатывать следующим. Введём для этого функцию $find2$, определённую так:

$$find2(g, m) p xs = seek1(g, m) x (\text{filter } p xs)$$

для любой помеченной вершины x и затем получим непосредственное определение $find2$, которое не полагается на $seek1$. В случае пустого списка xs имеем:

$$find2(g, m) p [] = (g, m)$$

Если $xs = y : ys$ и $not(p \ y)$, получается:

$$find2(g, m) p (y : ys) = find2(g, m) p ys$$

В оставшемся случае справедливы следующие рассуждения:

$$\begin{aligned}
 & find2(g, m) p (y : ys) \\
 = & \quad \{ \text{определение } find2 \text{ и } seek1 \text{ в случае } p \ y \} \\
 & seek1(g, m) (\text{right } g y) (\text{filter } p ys)
 \end{aligned}$$

$$\begin{aligned}
 &= \{ \text{безопасная замена, так как } y \notin ys \} \\
 &\quad seek1(g, m) (right g y) (filter (unset p y) ys) \\
 &= \{ \text{в силу } unset p v v = False \} \\
 &\quad seek1(g, m) (right g y) (filter (unset p y) (y : ys)) \\
 &= \{ \text{определение } seek2 \text{ и согласованность} \} \\
 &\quad seek2(g, m) (unset p y) (right g y) (y : ys)
 \end{aligned}$$

Мы убедились, что

$$\begin{aligned}
 mark\ g\ root &= seek2(g, const\ False) (const\ False)\ root\ [] \\
 seek2(g, m)\ p\ x\ xs \\
 | not(m\ x) &= seek2(g, set\ m\ x) (set\ p\ x) (left\ g\ x) (x : xs) \\
 | otherwise &= find2(g, m)\ p\ xs \\
 find2(g, m)\ p\ [] &= (g, m) \\
 find2(g, m)\ p\ (y : ys) \\
 | not(p\ y) &= find2(g, m)\ p\ ys \\
 | otherwise &= seek2(g, m) (unset\ p\ y) (right\ g\ y) (y : ys)
 \end{aligned}$$

Две взаимно рекурсивные функции *seek2* и *find2* содержат хвостовую рекурсию и могут быть реализованы в императивном стиле как простой цикл.

Представление стека связным списком

Заключительное преобразование заключается в представлении стека связным списком. Идея Шора и Вейта состоит в том, чтобы хранить связи прямо в графе. Хотя результат работает не быстрее исходной версии, он использует меньше памяти. Такое представление стека не использует отдельной функции, отвечающей за связи. Вместо этого имеется дополнительная размечающая функция *p*, которая, наряду с *m*, занимает всего два бита в каждой вершине графа.

В качестве подготовки к заключительному преобразованию получим следующие два ингредиента. Первый представляет собой абстрагирующую функцию *stack*, которая восстанавливает стек из связного представления:

$$\begin{aligned}
 stack &:: Graph \rightarrow (Node \rightarrow Bool) \rightarrow Node \rightarrow [Node] \\
 stack\ g\ p\ x\ | x == 0 &= [] \\
 | p\ x &= x : stack\ g\ p\ (left\ g\ x) \\
 | not(p\ x) &= x : stack\ g\ p\ (right\ g\ x)
 \end{aligned}$$

Новая вершина 0 имеет специальный смысл, она служит для обозначения конца списка. Вот как можно добавить новую ненулевую вершину x в стек:

$$x : \text{stack } g p y = \text{stack} (\text{setl } g x y) (\text{set } p x) x \quad (26.1)$$

Доказательство (26.1) выглядит так:

$$\begin{aligned} & \text{stack} (\text{setl } g x y) (\text{set } p x) x \\ = & \quad \{ \text{определение stack, так как } \text{set } p x x = \text{True} \} \\ x : \text{stack} & (\text{setl } g x y) (\text{set } p x) (\text{left} (\text{setl } g x y) x) \\ = & \quad \{ \text{в силу left } (\text{setl } g x y) x = y \} \\ x : \text{stack} & (\text{setl } g x y) (\text{set } p x) y \\ = & \quad \{ \text{безопасная замена, так как } x \notin \text{stack} (\text{setl } g x y) (\text{set } p x) y \} \\ x : \text{stack } g p y & \end{aligned}$$

Второй ингредиент это функция *restore*, определяемая так:

$$\begin{aligned} \text{restore} :: \text{Graph} & \rightarrow (\text{Node} \rightarrow \text{Bool}) \rightarrow \text{Node} \rightarrow [\text{Node}] \rightarrow \text{Graph} \\ \text{restore } g p x [] & = g \\ \text{restore } g p x (y : ys) & \mid p y = \text{restore} (\text{setl } g y x) p y ys \\ & \mid \text{not} (p y) = \text{restore} (\text{setr } g y x) p y ys \end{aligned}$$

Функция *restore* предназначена для приведения графа к первоначальному состоянию по завершении процесса разметки. Мотивацией к определению служит свойство $\text{restore } g p x xs = g$, при условии $\text{threaded } g p x xs$, которое остаётся в качестве упражнения.

Получив определения *stack* и *restore*, мы можем дать определения *seek3* и *find3*:

$$\begin{aligned} \text{seek3 } (g, m) p x y & = \text{seek2} (\text{restore } g p x xs, m) p x xs \\ & \quad \text{where } xs = \text{stack } g p y \\ \text{find3 } (g, m) p x y & = \text{find2} (\text{restore } g p x xs, m) p xs \\ & \quad \text{where } xs = \text{stack } g p y \end{aligned}$$

По-настоящему тяжёлой работой становится определение *mark* в терминах *seek3* и *find3*. Первый шаг довольно прост:

$$\begin{aligned} \text{mark } g \text{ root} & \\ = & \quad \{ \text{текущее определение mark} \} \\ \text{seek2 } (g, \text{const False}) & (\text{const False}) \text{ root } [] \end{aligned}$$

$$\begin{aligned}
 &= \{ \text{в силу } \text{restore } g \ p \ x [] = g \} \\
 &\quad \text{seek2} (\text{restore } g \ p \ \text{root} [], \text{const False}) (\text{const False}) \text{root} [] \\
 &= \{ \text{определения seek3 и stack} \} \\
 &\quad \text{seek3} (g, \text{const False}) (\text{const False}) \text{root} 0
 \end{aligned}$$

Таким образом, $\text{mark } g \text{ root} = \text{seek3} (g, \text{const False}) (\text{const False}) \text{root} 0$.

Далее необходимо определить seek3 непосредственно. Предположим сперва $\text{not} (m x)$.

$$\begin{aligned}
 &\text{seek3} (g, m) \ p \ x \ y \\
 &= \{ \text{полагая } xs = \text{stack } g \ p \ y \text{ и } g' = \text{restore } g \ p \ x \ xs \} \\
 &\quad \text{seek2} (g', m) \ p \ x \ xs \\
 &= \{ \text{предположение } \text{not} (m x) \} \\
 &\quad \text{seek2} (g', \text{set } m \ x) (\text{set } p \ x) (\text{left } g' \ x) (x : xs) \\
 &= \{ \text{безопасная замена, так как } x \notin xs \} \\
 &\quad \text{seek2} (g', \text{set } m \ x) (\text{set } p \ x) (\text{left } g \ x) (x : xs) \\
 &= \{ \text{факт ниже} \} \\
 &\quad \text{seek2} (\text{restore} (\text{setl } g \ x \ y) (\text{set } p \ x) (\text{left } g \ x) (x : xs), \text{set } m \ x) \\
 &\quad \quad (\text{set } p \ x) (\text{left } g \ x) (x : xs) \\
 &= \{ (26.1) \text{ и определение seek3} \} \\
 &\quad \text{seek3} (\text{setl } g \ x \ y, \text{set } m \ x) (\text{set } p \ x) (\text{left } g \ x) \ x
 \end{aligned}$$

Таким образом,

$$\text{seek3} (g, m) \ p \ x \ y = \text{seek3} (\text{setl } g \ x \ y, \text{set } m \ x) (\text{set } p \ x) (\text{left } g \ x) \ x$$

Проверим:

$$\text{restore } g \ p \ x \ xs = \text{restore} (\text{setl } g \ x \ y) (\text{set } p \ x) (\text{left } g \ x) (x : xs)$$

Для этого проведём рассуждения:

$$\begin{aligned}
 &\text{restore} (\text{setl } g \ x \ y) (\text{set } p \ x) (\text{left } g \ x) (x : xs) \\
 &= \{ \text{определение restore, так как set } p \ x \ x = \text{True} \} \\
 &\quad \text{restore} (\text{setl} (\text{setl } g \ x \ y) \ x (\text{left } g \ x)) (\text{set } p \ x) \ x \ xs \\
 &= \{ \text{определение setl} \} \\
 &\quad \text{restore } g (\text{set } p \ x) \ x \ xs
 \end{aligned}$$

$$= \{ \text{безопасная замена ввиду } x \notin xs \}$$

restore g p x xs

В случае $m x$ имеем:

$$\begin{aligned} & seek3(g, m) p x y \\ = & \{ \text{как выше, при } xs = stack g p y \text{ и } g' = restore g p x xs \} \\ & seek2(g', m) p x xs \\ = & \{ \text{предположение } m x \} \\ & find2(g', m) p xs \\ = & \{ \text{определение } find3 \} \\ & find3(g, m) p x y \end{aligned}$$

Разберём $find3$ по случаям. Во-первых, $find3(g, m) p x 0 = (g, m)$. В случае $y \neq 0$ и $not(p y)$ справедливо следующее:

$$\begin{aligned} & find3(g, m) p x y \\ = & \{ \text{определение } find3 \text{ и } stack \text{ с учётом } ys = stack g p (right g y) \} \\ & find2 restore g p x (y : ys), m) p (y : ys \\ = & \{ \text{определение } restore \text{ в случае } not(p y) \} \\ & find2 restore (setr g y x) p y ys, m) p (y : ys \\ = & \{ \text{определение } find2 \text{ в случае } not(p y) \} \\ & find2 restore (setr g y x) p y ys, m) p ys \\ = & \{ \text{безопасная замена: } ys = stack (setr g y x) p (right g y) \} \\ & find3 (setr g y x, m) p y (right g y) \end{aligned}$$

Таким образом, $find3(g, m) p x y = find3(setr g y x, m) p y (right g y)$.

Наконец, в наиболее сложном случае $y \neq 0$ и $p y$ имеем:

$$\begin{aligned} & find3(g, m) p x y \\ = & \{ \text{определение } find3 \text{ и } stack, \text{ с учётом } ys = stack g p (left g y) \} \\ & find2 restore g p x (y : ys), m) p (y : ys \\ = & \{ \text{определение } restore \text{ в случае } p y \} \\ & find2 restore (setl g y x) p y ys, m) p (y : ys \\ = & \{ \text{определение } find2 \text{ в случае } p y \} \\ & seek2 restore (setl g y x) p y ys, m) (unset p y) (right g y) (y : ys) \\ = & \{ \text{факт (смотри ниже): при } \\ & \quad swing g y x = setr (setl g y x) y (left g y) \} \end{aligned}$$

$$\begin{aligned}
 & seek2 (restore (swing g y x) (unset p y) (right g y) (y : ys), m) \\
 & \quad (unset p y) (right g y) (y : ys) \\
 = & \quad \{ \text{безопасная замена: } y : ys = stack (swing g y x) (unset p y) y \} \\
 & seek3 (swing g y x, m) (unset p y) (right g y) y
 \end{aligned}$$

Таким образом,

$$find3 (g, m) p x y = seek3 (swing g y x, m) (unset p y) (right g y) y$$

Главное утверждение таково:

$$\begin{aligned}
 & restore (setl g y x) p y ys \\
 & = restore (swing g y x) (unset p y) (right g y) (y : ys)
 \end{aligned}$$

где $swing g y x = setr (setl g y x) y (left g y)$. Вот его доказательство:

$$\begin{aligned}
 & restore (swing g y x) (unset p y) (right g y) (y : ys) \\
 = & \quad \{ \text{определение restore в силу } unset p y y = False \} \\
 & restore (setr (swing g y x) y (right g y)) (unset p y) y ys \\
 = & \quad \{ \text{поскольку } setr (swing g y x) y (right g y) = setl g y x \} \\
 & restore (setl g y x) (unset p y) y ys \\
 = & \quad \{ \text{безопасная замена в силу } y \notin ys \} \\
 & restore (setl g y x) p y ys
 \end{aligned}$$

Подытоживая:

$$\begin{aligned}
 mark g root & = seek3 (g, const False) (const False) root 0 \\
 seek3 (g, m) p x y & \\
 | not (m x) & = seek3 (setl g x y, set m x) (set p x) (left g x) x \\
 | otherwise & = find3 (g, m) p x y \\
 find3 (g, m) p x y & \\
 | y == 0 & = (g, m) \\
 | p y & = seek3 (swing g y x, m) (unset p y) (right g y) y \\
 | otherwise & = find3 (setr g y x, m) p y (right g y) \\
 & \quad \text{where } swing g y x = setr (setl g y x) y (left g y)
 \end{aligned}$$

Это и есть алгоритм разметки Шора—Вейта.

Заключительные замечания

Алгоритм Шора—Вейта впервые описан в (Schorr and Waite, 1967). Неполный список источников, где обсуждается формальное изложение

алгоритма с использованием инвариантов цикла, таков: (Bornat, 2000), (Butler, 1999), (Gries, 1979), (Morris, 1982) и (Topor, 1979). В (Möller, 1997, 1999) для формального описания алгоритма используются отношения и реляционные алгебры, в то время как в (Mason, 1988) применяется *Lisp* и модифицирующие функции с изменением состояния.

Любое изложение алгоритма обречено быть чрезвычайно детализированным и наше — не исключение. Особую важность представляет использование свойства безопасной замены в разных случаях. Альтернативный подход с использованием сепарационной логики представлен в (O’Hearn *et al.*, 2004). Мы утверждаем, однако, что каждое последующее определение *seek* в терминах предыдущего имеет чёткую мотивацию, достаточно просто для понимания и является удачным способом изложения алгоритма.

Литература

- Bornat, R. (2000). Proving pointer programs in Hoare logic. *LNCS 1837: 5th Mathematics of Program Construction Conference*, pp. 102–26.
- Butler, M. (1999). Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming* **33** (3), 221–60.
- Gries, D. (1979). The Schorr–Waite graph marking algorithm. *Acta Informatica* **11**, 223–32.
- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* **3**, 184.
- Mason, I. A. (1988). Verification of programs that destructively manipulate data. *Science of Computer Programming* **10** (2), 177–210.
- Möller, B. (1997). Calculating with pointer structures. *IFIP TC2/WG2.1 Working Conference on Algorithmic Languages and Calculi*. Chapman and Hall, pp. 24–48.
- Möller, B. (1999). Calculating with acyclic and cyclic lists. *Information Sciences* **119**, 135–54.
- Morris, J. M. (1982). A proof of the Schorr–Waite algorithm. *Proceedings of the 1981 Marktoberdorf Summer School*, ed. M. Broy and G. Schmidt. Reidel, pp. 25–51.
- O’Hearn, P. W., Yang, H. and Reynolds, J. C. (2004). Separation and information hiding. *31st Principles of Programming Languages Conference*. ACM Publications, pp. 268–80.

- Schorr, H. and Waite, W. M. (1967). An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM* **10**, 501–6.
- Topor, R. W. (1979). The correctness of the Schorr–Waite marking algorithm. *Acta Informatica* **11**, 211–21.

27

Упорядоченная вставка

Введение

Рассмотрим задачу вставки элементов списка длины N , все из которых различны и принадлежат некоторому упорядоченному типу A , по-очерёдно в пустой массив размера N таким образом, что на каждом шаге вставленные элементы расположены по возрастанию, хотя между ними допустимы пустоты. Можно думать о списке, как о груде книг, сваленных на полу, и о массиве, как о книжной полке с гнёздами. На каждом шаге книги на полке должны быть расставлены в алфавитном порядке. Нужно взять ближайшую книгу в груде на полу и поставить её на полку. В общем случае это можно сделать, только сдвинув некоторые книги на полке, чтобы освободить нужное место. Цель упражнения состоит в сведении к минимуму общего числа сдвигов, при этом расстояние сдвигов не учитывается. Например, если $A = \text{Char}$, один из способов вставки символов строки "КРЕСЛО" в массив размера 6 описывается таблицей:

1	2	3	4	5	6	Сдвиги
—	K	—	—	—	—	0
—	K	P	—	—	—	0
E	K	P	C	—	—	0
E	K	L	P	C	—	2
E	K	L	O	P	C	2

Общее количество сдвигов равно четырём. В этой жемчужине мы собираемся получить функцию, которая выполняет задачу за $\Theta(N \log^3 N)$

сдвигов, что, как предполагается, является наилучшим решением. Сам алгоритм не так сложен, но вычисление временной границы связано с довольно запутанными выкладками.

Наивный алгоритм

Начнём с записи минимального числа сдвигов $m(N)$, которое требуется для размещения N элементов произвольного списка в массиве такой же длины для $1 \leq N \leq 12$:

N	=	1	2	3	4	5	6	7	8	9	10	11	12
$m(N)$	=	0	1	2	4	6	8	11	14	17	21	24	29

Отдельная любопытная задача состоит в отыскании стратегии, которая гарантированно располагает список из шести элементов в упорядоченном массиве самое большое за восемь сдвигов (из того, что "КРЕСЛО" может быть размещено с четырьмя сдвигами, ещё не следует, что для любого списка это верно). Неизвестно асимптотической формулы для $m(N)$, однако есть серьёзные основания полагать, что $m(N) = \Theta(N \log^3 N)$.

Имеется один очевидный алгоритм для решения задачи, хотя его показатели в худшем случае плачевны. Поставить первый элемент на первую позицию, а затем вставлять оставшиеся элементы, выполняя сдвиги на столько позиций, чтобы хватало места. Так, если k -ый элемент меньше $k - 1$ предыдущего значения, то каждое из них должно быть сдвинуто на одну позицию. Наивный алгоритм требует $N(N - 1)/2$ сдвигов в худшем случае.

Чуть лучшая, хотя не намного, стратегия предполагает вставку каждого элемента в середину имеющихся свободных гнёзд, если они есть. Первые $\lfloor \log N \rfloor$ элементов могут быть размещены без сдвигов. Однако после того, как будет обработана половина элементов, сдвиги начнут появляться весьма часто, приводя всё к той же квадратичной производительности в худшем случае.

Одно разумное уточнение основной стратегии: периодически вставлять элементы таким образом, чтобы сохранять равномерное распределение свободных гнёзд. Пусть задано некоторое число M , назовём элементы iM , где $1 \leq i \leq k$ и $kM \leq N < (k + 1)M$, специальными. Вставка специального элемента iM сопровождается необходимыми сдвигами, чтобы получить равномерное распределение, и может потребовать до $iM - 1$ сдвига.

Общая стоимость не превосходит

$$\sum_{i=1}^k (iM - 1) = \Theta(N^2/M)$$

сдвигов для вставки всех специальных элементов.

Когда специальный элемент iM вставлен, предположим, что $N - iM$ свободных гнёзд равномерно распределены с примерно $C_i = iM/(N - iM)$ элементами по каждую сторону от каждого гнезда. Наивный алгоритм для вставки следующих $M - 1$ (или меньше, если $i = k$) неспециальных элементов может потребовать не более

$$\sum_{j=1}^{M-1} (jC_i + j - 1) = \Theta(C_i M^2)$$

сдвигов. Общее число сдвигов для вставки всех неспециальных элементов:

$$\Theta\left(\sum_{i=1}^{N/M} C_i M^2\right) = \Theta\left(M^2 \sum_{i=1}^{N/M} \frac{iM}{N - iM}\right) = \Theta(MN \log N)$$

Теперь, если выбрать M таким, чтобы $N^2/M \approx MN \log N$, то есть $M \approx \sqrt{N/\log N}$, общее количество сдвигов составит $\Theta(\sqrt{N^3 \log N})$, что немного хуже, чем $\Theta(N^{1.5})$.

Улучшенный алгоритм

Стратегия предыдущего раздела может быть улучшена двумя способами: можно выбрать другую последовательность специальных элементов и найти лучший метод вставки неспециальных. Два улучшения не зависят друг от друга, но оба потребуются нам для достижения нужной асимптотической оценки.

Вместо равномерного распределения специальных элементов можно выбрать такое, где их количество увеличивается по мере заполнения гнёзд. Скажем точнее, пусть специальные элементы пронумерованы числами n_1, n_2, \dots, n_k , где $n_1 = \lfloor N/2 \rfloor$, $n_2 = \lfloor 3N/4 \rfloor$, $n_3 = \lfloor 7N/8 \rfloor$ и так далее до $n_k = \lfloor (2^k - 1)N/2^k \rfloor$, где $2^{k-1} < N \leq 2^k$, так что $k = \lceil \log N \rceil$. Например, если $N = 1000$, специальными станут

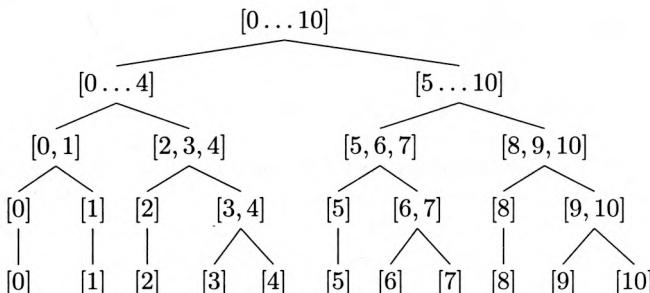
500, 750, 875, 937, 968, 984, 992, 996, 998, 999

Последний элемент вставляется с не более чем $N - 1$ сдвигом. Поскольку $k = \lceil \log N \rceil$, общее число сдвигов при вставке специальных элементов не превосходит

$$\sum_{i=1}^k (n_i - 1) \leq \sum_{i=1}^k N = \Theta(N \log N)$$

Теперь нужно разобраться с неспециальными элементами. Рассмотрим неспециальный элемент с номером n на этапе i , что значит $n_{i-1} < n < n_i$, при $n_0 = 0$. Предположим, что этот элемент должен быть вставлен после элемента в позиции $p - 1$, но перед элементом в позиции q ; другими словами — в интервале $[p, q]$. Если $p < q$, так что имеются свободные гнёзда, элемент может быть расположен в позиции $\lfloor (p+q)/2 \rfloor$. Если $p = q$, то есть пустые гнёзда отсутствуют, элементы в некотором окружении p должны быть сдвинуты, чтобы освободить необходимое место. Ключевой вопрос: каким должно быть выбрано это окружение.

Ответ, достаточно ожидаемый, звучит так: окружение должно быть не сильно заполненным. Представим, что N гнёзд расположены как листья на сбалансированном дереве. Например, для $N = 11$ одно из возможных деревьев выглядит так:



Каждая позиция вставки p определяет уникальную последовательность интервалов v_0, v_1, \dots, v_k , где $v_0 = [p]$, размечённых вдоль пути от p к корню дерева. Обозначим $L(v_j)$ длину интервала v_j и $S(v_j)$ текущее количество занятых гнёзд. Способ вставки элемента n на этапе i заключается в помещении n в интервал v_j , где j это наименьшее целое, удовлетворяющее условию плотности:

$$S(v_j) < \delta(i, j)L(v_j)$$

1	2	3	4	5	6	7	8	9	Сдвиги
—	—	—	—	К	—	—	—	—	0
—	—	А	—	К	—	—	—	—	0
—	—	А	—	К	—	—	Н	—	0
—	А	—	Д	—	К	—	Н	—	2
—	А	—	Д	Е	К	—	Н	—	0
—	А	Д	Е	—	К	Л	—	Н	3
—	А	Д	Е	К	Л	Н	—	Я	4
А	Б	Д	Е	К	Л	Н	—	Я	1
А	Б	Д	Е	К	Л	Н	Р	Я	0

Рис. 27.1: Способ разместить "КАНДЕЛЯБР"

где

$$\delta(i, j) = \left(\frac{2^i - 1}{2^i} \right)^{j/k}$$

Более того, элемент n и $S(v_j)$ вставленных ранее элементов равномерно распределены на интервале v_j по цене в $S(v_j)$ сдвигов. Можно заметить, что $1 = \delta(i, 0)$ и $\delta(i, j) > \delta(i, k)$, если $j < k$. Отметим, кроме того, что условие плотности выполнено в корне дерева:

$$S(v_k) < n_i \leq \left(\frac{2^i - 1}{2^i} \right) N = \delta(i, k) L(v_k)$$

Таким образом, всегда имеется хотя бы один интервал, удовлетворяющий условию плотности. Если оно выполнено, то $S(v_j) < L(v_j)$, поскольку $\delta(i, j) \leq 1$. Так что в v_j всегда имеется пустое гнездо. Наконец, заметим, что условие плотности выполнено для каждого листа v_0 , такого что $S(v_0) = 0$, в этом случае требуемое для вставки гнездо не занято.

В качестве примера приведём результат вставки девяти букв слова "КАНДЕЛЯБР" в массив соответствующего размера на рис. 27.1.

Что касается анализа сложности, предположим, что на этапе i вставляется C элементов, тогда $C \leq N/2^i$. В худшем случае каждый из этих элементов предназначен для вставки в одну и ту же позицию, то есть требует вставки в один из интервалов вдоль некоторого фиксированного пути v_0, v_1, \dots, v_k от позиции вставки до корня. Пусть элемент p , где $1 \leq p \leq C$, требует перемещений в интервале v_{j_p} . Обозначим $S_0(v_{j_p})$ количество занятых гнёзд сразу после последнего перераспределения элементов, затра-

нувшего v_{j_p} (что могло произойти при вставке последнего специального элемента), и пусть $\Delta(v_{j_p}) = S(v_{j_p}) - S_0(v_{j_p})$, так что $\Delta(v_{j_p}) \geq 0$.

Доказательства таких двух оценок даны в следующем разделе:

$$S(v_{j_p}) < 4k2^i(\Delta(v_{j_p}) + 1) \quad (27.1)$$

$$\sum_{p=1}^C \Delta(v_{j_p}) \leq kC \quad (27.2)$$

Поскольку $C \leq N/2^i$, свойства (27.1) и (27.2) дают:

$$\sum_{p=1}^C S(v_{j_p}) < \sum_{p=1}^C 4k2^i(\Delta(v_{j_p}) + 1) \leq 4k2^i(k+1)C \leq 4k(k+1)N$$

Общая стоимость расстановки всех неспециальных элементов, таким образом, не превосходит

$$\sum_{i=1}^k 4k(k+1)N = 4k^2(k+1)N = \Theta(N \log^3 N)$$

Вспоминая, что общая стоимость для специальных элементов составляет $\Theta(N \log N)$, получаем границу $\Theta(N \log^3 N)$ для всего алгоритма. На самом деле проведённый анализ доказывает нечто большее: первые $N/2$ элементов могут быть вставлены самое большое за $4N \log^2 N$ сдвигов, первые $3N/4$ элементов — за $8N \log^2 N$ и так далее. На это можно посмотреть с другой стороны, если предположить, что массив имеет размер $(1+\varepsilon)N$ вместо N , тогда N элементов могут быть вставлены в него за $\Theta((N \log^2 N)/\varepsilon)$ операций.

Доказательства

Если обозначить j_p как j , то неравенство (27.1) примет следующий вид: $S(v_j) < 4k2^i(\Delta(v_j)+1)$. Напомним, что $\Delta(v_j) = S(v_j) - S_0(v_j)$, где $S(v_j)$ это число занятых гнёзд в интервале v_j , $S_0(v_j)$ представляет количество гнёзд v_j , которые были заняты после последнего перераспределения места в некотором интервале, содержащем v_j . Перечислим важные факты, на которых основывается доказательство:

- 1) $S(v_j) < \delta(i, j)L(v_j)$ и $S(v_{j-1}) \geq \delta(i, j-1)L(v_{j-1})$, потому что v_j это наименьший интервал, удовлетворяющий условию плотности;

- 2) $2L(v_{j-1}) \geq L(v_j) - 1$, так как дерево сбалансировано;
- 3) $2S_0(v_{j-1}) \leq S_0(v_j) + 1$, потому что в результате предыдущего переворота интервала, содержащего v_j , образовалось равномерное распределение;
- 4) $\Delta(v_j) \geq \Delta(v_{j-1})$, потому что v_{j-1} является подинтервалом v_j .

С помощью этих утверждений проведём следующее рассуждение:

$$\begin{aligned}
& 2\Delta(v_j) \\
\geq & \{ \text{в силу } \Delta(v_j) \geq \Delta(v_{j-1}) \text{ и определения } \Delta \} \\
& 2S(v_{j-1}) - 2S_0(v_{j-1}) \\
\geq & \{ \text{в силу } 2S_0(v_{j-1}) \leq S_0(v_j) + 1 \leq S(v_j) + 1 \} \\
& 2S(v_{j-1}) - S(v_j) - 1 \\
\geq & \{ \text{в силу } S(v_{j-1}) \geq \delta(i, j-1)L(v_{j-1}) \} \\
& 2\delta(i, j-1)L(v_{j-1}) - S(v_j) - 1 \\
\geq & \{ \text{в силу } 2L(v_{j-1}) \geq L(v_j) - 1 \} \\
& \delta(i, j-1)(L(v_j) - 1) - S(v_j) - 1 \\
> & \{ \text{в силу } \delta(i, j)L(v_j) > S(v_j) \text{ и } \delta(i, j) \leq 1 \} \\
& (\delta(i, j-1)/\delta(i, j) - 1)S(v_j) - 2
\end{aligned}$$

Таким образом,

$$\Delta(v_j) + 1 > \frac{1}{2} \left(\frac{\delta(i, j-1)}{\delta(i, j)} - 1 \right) S(v_j)$$

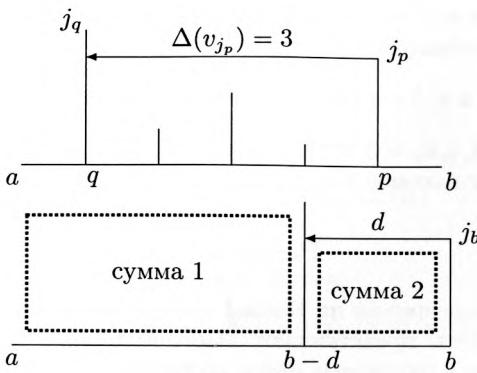
Из определения $\delta(i, j)$ получаем:

$$\frac{\delta(i, j-1)}{\delta(i, j)} - 1 = \left(\frac{2^i}{2^i - 1} \right)^{1/k} - 1$$

Поскольку $x^{1/k} = 2^{(1/k)\log x}$ и $2^y - 1 \geq y/2$ для $0 < y < 1$, имеем:

$$\frac{\delta(i, j-1)}{\delta(i, j)} - 1 \geq \frac{1}{2k} \log \left(\frac{2^i}{2^i - 1} \right)$$

Наконец, так как $2^i \log[2^i/(2^i - 1)] \geq 1$, то справедливым оказывается неравенство $4k2^i(\Delta(v_j) + 1) > S(v_j)$, а это и есть (27.1).

Рис. 27.2: разложение $P(a, b, k)$ в две суммы

Утверждение (27.2) состоит в том, что $\sum_{p=1}^C \Delta(v_{j_p}) \leq kC$. Покажем более общий факт, а именно:

$$\sum_{p=a}^b \Delta(v_{j_p}) \leq k(b-a+1)$$

для $1 \leq a \leq b \leq C$. Пусть $P(a, b, k)$ обозначает сумму в левой части неравенства, здесь явно указана зависимость от высоты дерева k . Рассмотрим последовательность точек j_a, \dots, j_b , где $0 \leq j_p \leq k$. Каждая точка соответствует интервалу, в который производится вставка. Величина $\Delta(v_{j_p})$ представляет количество элементов, которые были вставлены в интервал v_{j_p} со времени последнего перераспределения в интервале, содержащем v_{j_p} , имеющего номер, скажем, j_q ; таким образом, эта величина равна количеству точек между q и p . Описанная ситуация изображена на верхней диаграмме рис. 27.2.

Чтобы оценить $P(a, b, k)$, предположим, что $\Delta(v_{j_b}) = d$, что значит $0 \leq d \leq b-a$. Тогда P может быть разложено в две суммы, как показано на рис. 27.2: сумма от a до $b-d-1$ и сумма от $b-d$ до $b-1$, причём для второй суммы нам известно, что максимальное значение j составляет $j_b - 1$. Это приводит к следующей верхней оценке $P(a, b, k)$:

$$(\max d : 0 \leq d \leq b-a : P(a, b-d-1, k) + P(b-d, b-1, j_b-1) + d)$$

Поскольку $j_b \leq k$ и P монотонна по третьему аргументу, верхняя оценка для $P(a, b, k)$ преобразуется к виду

$$(\max d : 0 \leq d \leq b - a : P(a, b - d - 1, k) + P(b - d, b - 1, k - 1) + d)$$

Заметим, что $P(a, b, k) = 0$, если $b - a \leq 1$ или $k = 0$, и простое рассуждение по индукции даёт искомое $P(a, b, k) \leq k(b - a + 1)$.

Реализация

Реализация алгоритма на Haskell проводится непосредственно; единственную трудность представляют многочисленные значения, которые нужно передавать в различные места программы. Например, в ряде расчётов требуются номер этапа i и размер массива n . Первый поддерживается с помощью отметок об этапах на элементах, второй включается дополнительным аргументом некоторых функций.

Массив представляется простым списком пар индекс—значение:

```
type Array a = [(Int, a)]
```

Инвариант, который накладывается на такой массив, состоит в том, что индекс i находится в промежутке $0 \leq i < n$, а значения расположены по возрастанию. Основная функция вставки определяется так:

```
insertAll n = scanl (insert n) [] · label n
```

Функция $insert n$ выполняет однократную вставку, а $label n$ размечает элементы номерами этапов. На выходе $insertAll$ появляется список массивов, который затем может быть использован для отображения результатов или подсчёта количества сдвигов.

Код $label$ включает в себя определение специальных элементов. Им ставится в соответствие этап под номером нуль, остальным же приписывается этап i , где $i > 0$. Специальные элементы задаются функцией $specials n$:

```
specials    :: Int → [Int]
specials n = scanl1 (+) (halves n)
halves n   = if n == 1 then [] else m : halves (n - m)
            where m = n div 2
```

К примеру, $specials 11 = [5, 8, 9, 10]$. Функция $label$ определяется так:

```
label      :: Int → [a] → [(Int, a)]
label n xs = replace 1 (zip [1..] xs) (specials n)
```

где *replace* заменяет позиции номерами этапов:

```
replace i [] ns = []
replace i ((k, x) : kxs) ns
| null ns = [(0, x)]
| k < n = (i, x) : replace i kxs ns
| k == n = (0, x) : replace (i + 1) kxs (tail ns)
where n = head ns
```

Например, *label 11 [1..11]* возвращает список:

```
[(1, 1), (1, 2), (1, 3), (1, 4), (0, 5), (2, 6), (2, 7), (0, 8), (0, 9), (0, 10), (0, 11)]
```

Теперь разберёмся с *insert n*, которая реализуется так:

```
insert :: Ord a => Int -> Array a -> (Int, a) -> Array a
insert n as (i, x) = if i == 0
                     then relocate (0, n) x as
                     else relocate (l, r) x as
where (l, r) = ipick n as (i, x)
```

Если элемент *x* специальный, то ему приписан нулевой этап и вставлен он будет с помощью повторного размещения (*relocate*) элементов массива из интервала $(0, n]$, к которым добавляется сам *x*. Если *x* неспециальный, функция *ipick* выбирает интервал (l, r) , на котором будет произведено повторное размещение.

Функция *relocate* определяется следующим образом:

```
relocate :: Ord a => (Int, Int) -> a -> Array a -> Array a
relocate (l, r) x as = distribute (add x (entries (l, r) as)) (l, r) as
```

где *entries* возвращает (упорядоченный) список элементов массива из заданного интервала, а *add* вставляет *x* в этот список. Эти две функции определяются так:

```
entries (l, r) as = [x | (i, x) ← as, l ≤ i ∧ i < r]
add x xs          = takeWhile (<x) xs ++ [x] ++ dropWhile (<x) xs
```

Функция *distribute* равномерно распределяет элементы заданного списка по заданному интервалу в данном массиве:

```
distribute :: [a] -> (Int, Int) -> Array a -> Array a
distribute xs (l, r) as = takeWhile (λ(i, x) → i < l) as ++
```

$$\begin{aligned} & \text{spread } xs (\ell, r) ++ \\ & \text{dropWhile } (\lambda(i, x) \rightarrow i < r) \text{ as } \end{aligned}$$

Один из способов определения *spread* заключается в делении как списка, так и интервала на две половины и рекурсивного распределения левой половины элементов на левой половине интервала и аналогично с правыми половинами.

$$\begin{aligned} \text{spread} & :: [a] \rightarrow (\text{Int}, \text{Int}) \rightarrow \text{Array } a \\ \text{spread } xs (\ell, r) & | \text{ null } xs = [] \\ & | n == 0 = [(m, \text{head } xs)] \\ & | n > 0 = \text{spread } ys (\ell, m) ++ \text{spread } zs (m, r) \\ & \text{where } (n, m) = (\text{length } xs \text{ div } 2, (\ell + r) \text{ div } 2) \\ & (ys, zs) = \text{splitAt } n \text{ xs} \end{aligned}$$

Следующая функция на очереди это *ipick*. Её определение таково:

$$\begin{aligned} \text{ipick} & :: \text{Ord } a \Rightarrow \text{Int} \rightarrow \text{Array } a \rightarrow (\text{Int}, a) \rightarrow (\text{Int}, \text{Int}) \\ \text{ipick } n \text{ as } (i, x) & = \text{if } p < q \text{ then } (p, q) \text{ else} \\ & \quad \text{head } [(\ell, r) \mid (j, (\ell, r)) \leftarrow \text{zip } [0..] (\text{ipath } n \text{ p}), \\ & \quad \quad \text{let } s = \text{length } (\text{entries } (\ell, r) \text{ as }), \\ & \quad \quad \quad \text{densityTest } i \ j \ n \ s \ (r - \ell)] \\ & \text{where } (p, q) = \text{ipoint } n \ x \text{ as } \end{aligned}$$

Вначале с помощью *ipoint* определяется позиция для вставки *x*. Результатом служит интервал (p, q) , в котором нет вставленных элементов. Если $p < q$, то есть этот интервал не пуст, то результатом *ipick* становится (p, q) . Следующее перераспределение на интервале (p, q) обеспечит вставку *x* ровно в середину этого интервала. Если $p = q$, то интервал пуст, в этом случае посредством *ipath* вычисляется путь из интервалов от точки вставки *p* до корня воображаемого дерева. На этом пути выбирается первый интервал, удовлетворяющий условию плотности, которое проверяется *densityTest*.

Функция *ipoint* реализуется так:

$$\begin{aligned} \text{ipoint} & :: \text{Ord } a \Rightarrow \text{Int} \rightarrow a \rightarrow \text{Array } a \rightarrow (\text{Int}, \text{Int}) \\ \text{ipoint } n \text{ x as} & = \text{search } (0, n) \text{ as} \\ \text{where } \text{search } (p, q) [] & = (p, q) \\ \text{search } (p, q) ((i, y) : \text{as}) & = \text{if } x < y \text{ then } (p, i) \\ & \quad \text{else search } (i + 1, q) \text{ as} \end{aligned}$$

Значение *ipath* получается обращением пути от корня $(0, n)$ до *p*:

$$\text{ipath } n \ p = \text{reverse } (\text{intervals } (0, n) \ p)$$

где

$$\begin{aligned}
 \text{intervals } (\ell, r) \ p \mid \ell + 1 == r &= [(\ell, r)] \\
 \mid p < m &= (\ell, r) : \text{intervals } (\ell, m) \ p \\
 \mid m \leq p &= (\ell, r) : \text{intervals } (m, r) \ p \\
 \text{where } m &= (\ell + r) \text{ div } 2
 \end{aligned}$$

Остается разобраться с функцией плотности, для которой вместо вещественной арифметики будет использована целочисленная арифметика произвольной точности. Все пять аргументов функции *densityTest* являются целыми числами фиксированного размера, так что их необходимо преобразовать в целые произвольного размера:

$$\begin{aligned}
 \text{densityTest } i' j' n s' w' &= 2 \uparrow (i * j) * s \uparrow k < (2 \uparrow i - 1) \uparrow j * w \uparrow k \\
 \text{where } (i, j, s, w) &= \text{convert toInteger} (i', j', s', w') \\
 k &= \text{toInteger} (\text{ceiling} (\log_{\text{Base}} 2 (\text{fromIntegral} n)))
 \end{aligned}$$

Функция *convert* определяется так:

$$\text{convert } f (a, b, c, d) = (f \ a, f \ b, f \ c, f \ d)$$

Заключительные замечания

Задача вставки элементов списка длины N , на которых введено отношение порядка, в массив размера N является частным случаем более общей задачи, известной как *онлайновая разметка списка* (Bender *et al.*, 2002); цель последней состоит в поддержании отображения из динамически изменяющегося множества элементов M (то есть допускается как вставка, так и удаление из массива), принадлежащего некоторому линейно упорядоченному множеству, во множество целых чисел промежутка от 0 до $N - 1$, так чтобы порядок меток соответствовал порядку соответствующих элементов. Основное применение онлайновая разметка списка находит в связанный задаче, известной как *поддержание порядка*, она, в свою очередь, подразумевает обработку списка, который изначально содержит единственный базовый элемент и для которого допускаются вставка, удаление и сравнение. Запрос вставки имеет вид $\text{insert}(x, y)$, это означает вставку нового элемента x непосредственно следом за (уже имеющимся) элементом y . Операция $\text{delete}(x)$ удаляет элемент x из списка, а $\text{query}(x, y)$ возвращает значение истины или лжи в зависимости от того, следует ли y за x . Хотя поддержание порядка не требует приписывания меток каждому

элементу, большинство решений используют в качестве одной из составных частей онлайновую разметку.

Предметом интереса в данной задаче является, конечно, не реализация на Haskell, а занимательный выбор функции плотности и анализ того, как достигается поведение $\Theta(N \log^3 N)$. Открыт вопрос о том, составляется ли $\Omega(N \log^3 N)$ нижнюю границу, хотя в (Zhang, 1993) доказано, что это имеет место для любого алгоритма, который допускает лишь сдвиги в позиции, где свободные слоты равномерно распределены. В этом отношении представленный алгоритм «гладок», и маловероятно существование негладкого алгоритма, использующего меньшее число сдвигов, однако оказывается затруднительным строго доказать отсутствие такого алгоритма.

В заключении отметим, что наша жемчужина была создана на основе (Bird and Sadnicki, 2007), где можно найти ссылки на дополнительную литературу по задаче.

Литература

- Bender, M. A., Cole, R., Demaine, E. D., Frach-Colton, M. and Zito, J. (2002). Two simplified algorithms for maintaining order in a list. *Lecture Notes in Computer Science, Volume 2461*. Springer-Verlag, pp. 139–51.
- Bird, R. S. and Sadnicki, S. (2007). Minimal on-line labelling. *Information Processing Letters* **101** (1), 41–5.
- Zhang, J. (1993). Density control and on-line labeling problems. Technical Report 481 and PhD thesis, Computer Science Department, University of Rochester, New York, USA.

28

Бесцикловые функциональные алгоритмы

Введение

Представим программу для порождения комбинаторных объектов некоторого вида, скажем, подпоследовательностей или перестановок списка. Предположим, что каждый объект может быть получен из предыдущего с помощью одного *перехода*. Переход i для подпоследовательностей может означать «вставить или удалить элемент в позицию i », тогда как для перестановок — «поменять местами элементы, расположенные в позициях i и $i - 1$ ». Алгоритм порождения всех объектов называется *бесцикловым*, если первый переход создаётся за линейное время, а каждый последующий за константное. Заметьте, что за константное время создаются именно переходы, а не сами объекты. Порождение объекта за константное время обычно невозможно.

Бесцикловые алгоритмы обычно формулируются в императивном стиле, для их построения используются различные хитрости, такие как со-программы (*cotoutine*) и двусвязные списки с указателями на текущий элемент. В этой и двух последующих жемчужинах мы выясним, что может дать в этом вопросе функциональный подход. Мы выведем функциональные версии бесцикловых алгоритмов порождения перестановок Джонсона—Троттера (Johnson—Trotter), порождения всех префиксов леса Коды—Раски (Koda—Ruskey) и его обобщения до алгоритма *прядения паутинь* Кнута, в котором генерируются все битовые строки, удовлетворяющие заданным ограничениям неравенств. Эти совершенно новые алгоритмы опираются не более чем на списки, деревья и очереди. Настоящая

жемчужина посвящена большей частью исследованию вопроса и разбору упражнений для разогрева. Имейте, однако, в виду, что бесцикловые алгоритмы вовсе не обязательно быстрее своих же версий с циклами. Вот что пишет об этом Кнут (Knuth, 2001):

Все приёмы, необходимые для удаления циклов, редко бывают полезными сами по себе, поскольку они обычно увеличивают общее время выполнения по сравнению с более непосредственными алгоритмами. Однако они представляют академический интерес. Поэтому помимо прочего это задание можно рассматривать как препятствие, могущее способствовать развитию наших навыков алгоритмизации.

Замените последние слова на «навыки формального вывода» и вы ощутите реальный смысл упражнения.

Бесцикловые алгоритмы

Сформулируем идею бесциклового алгоритма в терминах стандартной функции *unfoldr*, определяемой следующим образом:

```
unfoldr      :: (b → Maybe (a, b)) → b → [a]
unfoldr step b = case step b of
    Just (a, b') → a : unfoldr step b'
    Nothing       → []
```

Бесцикловыми алгоритмами будем называть такие алгоритмы, которые можно представить в форме:

unfoldr step · prolog

где *step* выполняется за константное время, а *prolog x* — за $O(n)$ операций, *n* здесь это некоторая характеристика для размера *x*. Каждый бесциклический алгоритм должен иметь приведённую выше форму, а его компоненты должны удовлетворять указанным ограничениям.

В данном определении скрывается довольно тонкая проблема: в случае использования ленивого языка, такого как Haskell, исполнение бесциклической программы вообще говоря не приводит к бесцикловому вычислению с константным временем задержки перед каждым выводом. В ленивом языке работа, выполняемая функцией *prolog*, распределяется по всему вычислению, а вовсе не концентрируется в его начале. Следовательно,

на самом деле мы должны трактовать операцию композиции (\cdot) между *unfoldr step* и *prolog* как строгую, т.е. предполагать, что вычисление *prolog* происходит до запуска *unfoldr*. Хотя в языке Haskell и невозможно определить достаточно общую строгую операцию композиции, мы всё же будем всякий раз стараться убеждаться, что *prolog* и *step* выполняются за линейное и константное время соответственно как при строгом, так и при ленивом порядке вычисления.

Четыре упражнения для разогрева

Простейшим упражнением будет тождественная функция на списках. Добуквенно следуя требуемым условиям, получаем:

```

id :: [a] → [a]
id = unfoldr uncons · prolog
prolog :: [a] → [a]
prolog = id
uncons      :: [a] → Maybe (a, [a])
uncons []    = Nothing
uncons (x : xs) = Just (x, xs)

```

Это было несложно, поэтому теперь давайте рассмотрим функцию *reverse*, которая обращает конечный список. В языке Haskell эта функция определена следующим образом:

```

reverse :: [a] → [a]
reverse = foldl (flip (:)) []

```

Комбинатор *flip* определён как $\text{flip } f \ x \ y = f \ y \ x$. Данное определение обращает список за линейное время. Программа бесциклового обращения списка будет выглядеть так:

```
reverse = unfoldr uncons · foldl (flip (:)) []
```

Собственно, вся работа делается в прологе.

В следующем упражнении посмотрим на функцию *concat*, соединяющую элементы списка списков. Вот обсуждаемая далее бесцикловая версия:

```

concat      :: [[a]] → [a]
concat      = unfoldr step · filter (not · null)

```

```

step          :: [[a]] → Maybe (a, [[a]])
step []       = Nothing
step (x : xs) : xss) = Just (x, consList xs xss)
consList     :: [a] → [[a]] → [[a]]
consList xs xss = if null xs then xss else xs : xss

```

В прологе из входного списка отбираются непустые списки, это требует линейного по размеру списка времени. Функция *step* сохраняет инвариант: она принимает и возвращает *непустые* списки. Пустые списки действительно следует удалять, поскольку в противном случае функции *step* не хватало бы линейного времени. Рассмотрим, к примеру, список $[[1],[],[],\dots,[],[2]]$, в котором между первым и последним одноэлементными списками содержится n пустых. После получения первого элемента списка 1 потребовалось бы n операций для получения второго элемента результирующего списка 2.

Вдумчивые читатели могут возразить, что такое определение *concat* избыточно, так как альтернативное определение

```
concat = unfoldr uncons · foldr (++) []
```

так же бесцикловое. Здесь вся работа производится в прологе, который требует линейного времени относительно общего размера входных данных, а именно суммы длин списков, составляющих заданный. Спор тут заключается в выборе используемой характеристики для размера данных. Поскольку в нас нет стремления кодировать все входные данные в виде строк над конечным алфавитом, мы оставляем неформальное понятие размера и считаем оба определения *concat* бесцикловыми алгоритмами.

В качестве последнего четвёртого упражнения рассмотрим задачу префиксного обхода леса «розовых кустов» (сильно ветвящихся деревьев):

```

type Forest a = [Rose a]
data Rose a   = Node a (Forest a)

```

Префиксный обход можно определить следующим образом:

```

preorder        :: Forest a → [a]
preorder []     = []
preorder (Node x xs : ys) = x : preorder (xs ++ ys)

```

Далее, *preorder* = *unfoldr step*, где

```

step           :: Forest a → Maybe (a, Forest a)
step []        = Nothing
step (Node x xs : ys) = Just (x, xs ++ ys)

```

Функции *step* недостаточно константного времени, поскольку его недостаточно ++ , но этого можно добиться, изменив тип. Вместо того, чтобы принимать в качестве аргумента лес, она будет брать список лесов, поэтому её определение изменится так:

$$\begin{aligned} \text{step} &:: [\text{Forest } a] \rightarrow \text{Maybe } (a, [\text{Forest } a]) \\ \text{step } [] &= \text{Nothing} \\ \text{step } ((\text{Node } x \text{ xs} : ys) : zss) &= \text{Just } (x, \text{consList } xs (\text{consList } ys zss)) \end{aligned}$$

В сущности, здесь тот же приём, который мы использовали для *concat*. Теперь имеем:

$$\text{preorder} = \text{unfoldr step} \cdot \text{wrapList}$$

где $\text{wrapList } xs = \text{consList } xs []$. Это и есть бесцикловый алгоритм для *preorder*. Разумеется, он подвержен той же проблеме, что и *concat*. Поскольку длина выхода пропорциональна размеру входа, аналогичным образом вся работа могла быть проделана и в прологе.

Бустрофедоновое произведение

Многие алгоритмы порождения комбинаторных объектов содержат проходы по списку в прямом и обратном направлениях, перемежающиеся с генерацией последовательных элементов результирующего списка. Всё это похоже на движение челнока в ткацком станке или быка, вспахивающего поле. Собственно, Кнут именно для этой операции использовал название *бустрофедоновое произведение*. Мы будем называть её *box* и обозначать инфиксным символом \square , поскольку такое имя короче и проще произносится. Вот определение:

$$\begin{aligned} (\square) &:: [a] \rightarrow [a] \rightarrow [a] \\ [] \square ys &= ys \\ (x : xs) \square ys &= ys \text{ ++ } [x] \text{ ++ } (xs \square \text{reverse } ys) \end{aligned}$$

Например,

$$[3, 4] \square [0, 1, 2] = [0, 1, 2, 3, 2, 1, 0, 4, 0, 1, 2]$$

Определение \square приводит к неэффективным вычислениям, поскольку *ys* обращается на каждом шаге. Следующая версия, в которой обращение происходит лишь однажды, лучше:

$$\begin{aligned} xs \square ys &= mix\ xs\ (ys, reverse\ ys) \\ mix\ []\ (ys, sy) &= ys \\ mix\ (x : xs)\ (ys, sy) &= ys ++ [x] ++ mix\ xs\ (sy, ys) \end{aligned}$$

Операция \square ассоциативна, в качестве нейтрального элемента выступает пустой список. Это хорошая причина для использования инфиксной записи. Доказательство ассоциативности мы оставим для читателей, выпишем сейчас лишь два вспомогательных тождества, на которых оно основывается и которые понадобятся нам в дальнейшем:

$$(xs ++ [y] ++ ys) \square zs = (xs \square zs) ++ [y] ++ (ys \square zs') \quad (28.1)$$

$$reverse\ (xs \square ys) = (reverse\ xs) \square ys' \quad (28.2)$$

где zs' и ys' определены следующим образом:

$$\begin{aligned} zs' &= \text{if even}\ (\text{length}\ xs) \text{ then } reverse\ zs \text{ else } zs \\ ys' &= \text{if even}\ (\text{length}\ xs) \text{ then } reverse\ ys \text{ else } ys \end{aligned}$$

Таким образом, и (28.1), и (28.2) зависят от чётности длины списка xs .

Аналогично дистрибутивному закону для *concat* и ++ над списками списков, имеется такой же закон для *boxall* относительно \square :

$$\begin{aligned} boxall &:: [[a]] \rightarrow [a] \\ boxall &= foldr\ (\square)\ [] \end{aligned}$$

Результат операции *boxall* на n -элементном списке m -элементных списков имеет длину $(m+1)^n - 1$, которая экспоненциальна относительно mn , общей длины входных данных. Поставим теперь задачу на последнее упражнение, которое заключается в выводе бесцикловой версии *boxall*.

Применение кортежей

Поскольку $boxall\ (xs : xss) = xs \square (boxall\ xss)$, а определение $xs \square ys$ включает $reverse\ ys$, ясно, что вместе с вычислением *boxall* нам необходимо вычислять и $reverse \cdot boxall$. Это подсказывает нам, что выполнять такие вычисления стоит одновременно. Для этого воспользуемся правилом кортежей для *foldr*. Это правило утверждает:

$$(foldr\ f\ a\ xs, foldr\ g\ b\ xs) = foldr\ h\ (a, b)\ xs$$

где $h\ x\ (y, z) = (f\ x\ y, g\ x\ z)$.

Предположим, что нам удалось найти такую операцию \boxtimes , что

$$\text{reverse} \cdot \text{boxall} = \text{foldr} (\boxtimes) []$$

Теперь правило кортежей для foldr даст нам:

$$(\text{boxall } xs, \text{reverse} (\text{boxall } xs)) = \text{foldr } op ([], []) xs$$

где

$$op xs (ys, sy) = (xs \square ys, xs \boxtimes sy) \quad (28.3)$$

Последовательности ys и sy являются обратными друг к другу, отсюда и их имена.

Для обнаружения \boxtimes необходимо обратиться к закону слияния для foldr . Этот закон утверждает, что для всех конечных списков xs имеет место $f (\text{foldr } g a xs) = \text{foldr } h b xs$, при условии, что $f a = b$, и для всех x и y выполняется $f (g x y) = h x (f y)$. Так как $\text{reverse} [] = []$, остаётся найти такое определение \boxtimes , которое удовлетворяло бы тождеству

$$\text{reverse} (xs \square ys) = xs \boxtimes (\text{reverse} ys)$$

Для любого конечного списка xs выполняется $\text{reverse} (\text{reverse} xs) = xs$, поэтому

$$xs \boxtimes sy = \text{reverse} (xs \square (\text{reverse} sy))$$

Можно также дать непосредственное рекурсивное определение \boxtimes :

$$\begin{aligned} [] \boxtimes sy &= sy \\ (x : xs) \boxtimes sy &= (xs \boxtimes (\text{reverse} sy)) ++ [x] ++ sy \end{aligned}$$

Иначе, \boxtimes можно выразить посредством (28.2):

$$xs \boxtimes sy = \text{if even} (\text{length} xs) \text{then} (\text{reverse} xs) \square (\text{reverse} sy) \\ \text{else} (\text{reverse} xs) \square sy$$

С учётом (28.3) и определения \square первое рекурсивное определение \boxtimes ведёт к следующему определению op , переименованной здесь в $op1$:

$$\begin{aligned} op1 [] (ys, sy) &= (ys, sy) \\ op1 (x : xs) (ys, sy) &= (ys ++ [x] ++ zs, sz ++ [x] ++ sy) \\ &\quad \text{where } (zs, sz) = op1 xs (sy, ys) \end{aligned}$$

С учётом (28.3) и определения \square в терминах mix второе рекурсивное определение \boxtimes ведёт к другому определению op , переименованной в $op2$:

$$\begin{aligned} op2 \ xs \ (ys, sy) = & \text{ if } even \ (length \ xs) \\ & \text{then } (mix \ xs \ (ys, sy), mix \ (reverse \ xs) \ (sy, ys)) \\ & \text{else } (mix \ xs \ (ys, sy), mix \ (reverse \ xs) \ (ys, sy)) \end{aligned}$$

Разница между $op1$ и $op2$ в том, что последняя в отличие от первой явно вызывает $reverse$. Если пренебречь затратами на вычисление операции ++ , то выполнение как $op1 \ xs$, так и $op2 \ xs$ происходит за время, пропорциональное длине xs , а значит, $foldr \ op1 \ ([], [])$ и $foldr \ op2 \ ([], [])$ будут выполняться за время, пропорциональное общему размеру входных данных. Оба определения, $op1$ и $op2$, останутся в игре, поскольку они приведут к двум различным бесцикловым версиям $boxall$.

Деревья и очереди

Последний шаг состоит в избавлении от дорогостоящих операций ++ . Эта цель достигается в два этапа, первый из которых заключается в представлении списков лесами «розовых кустов» относительно функции $preorder$. При этом получаем:

$$\begin{aligned} boxall &= preorder \cdot fst \cdot foldr \ op1' \ ([], []) \\ boxall &= preorder \cdot fst \cdot foldr \ op2' \ ([], []) \end{aligned}$$

где спецификация $op1'$ выглядит так:

$$pair \ preorder \ (op1' \ xs \ (ys, sy)) = op1 \ xs \ (preorder \ ys, preorder \ sy)$$

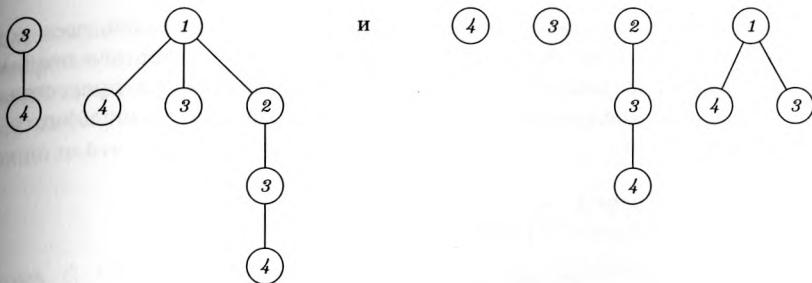
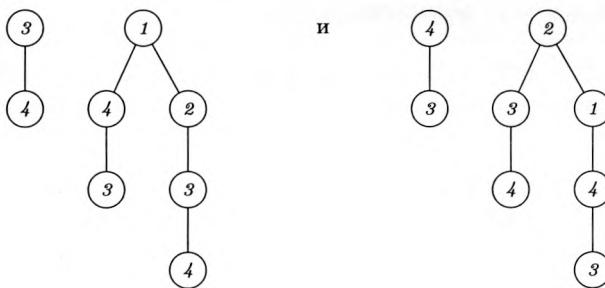
и $pair f \ (x, y) = (f \ x, f \ y)$. Похожим образом обстоит дело с $op2'$ и $op2$. Из проделанных ранее упражнений мы знаем, как сделать $preorder$ бесцикловым алгоритмом, поэтому при условии, что $op1' \ xs$ и $op2' \ xs$ выполняются за время, пропорциональное длине xs , каждая из представленных выше реализаций $boxall$ является бесцикловым алгоритмом.

Определения $op1'$ и $op2'$ можно вывести формально, но результаты достаточно очевидны для того, чтобы не вдаваться в детали. Во-первых,

$$\begin{aligned} op1' &\quad :: [a] \rightarrow (Forest \ a, Forest \ a) \rightarrow (Forest \ a, Forest \ a) \\ op1' \ [] \ (ys, sy) &= (ys, sy) \\ op1' \ (x : xs) \ (ys, sy) &= (ys \text{ ++ } [Node \ x \ zs], sz \text{ ++ } [Node \ x \ sy]) \\ &\quad \text{where } (zs, sz) = op1' \ xs \ (sy, ys) \end{aligned}$$

Во-вторых, с новой версией mix :

$$\begin{aligned} op2' \ xs \ (ys, sy) = & \text{ if } even \ (length \ xs) \\ & \text{then } (mix \ xs \ (ys, sy), mix \ (reverse \ xs) \ (sy, ys)) \end{aligned}$$

Рис. 28.1: два леса, построенные $foldr\ op1'\ ([], [])\ [[1, 2], [3, 4]]$ Рис. 28.2: два леса, построенные $foldr\ op2'\ ([], [])\ [[1, 2], [3, 4]]$

```

else (mix xs (ys, sy), mix (reverse xs) (ys, sy))
mix [] (ys, sy)      = ys
mix (x : xs) (ys, sy) = ys ++ [Node x (mix xs (sy, ys))]
```

Определения $op1'$ и $op2'$ приводят к различным парам лесов. Посмотрим, к примеру, на рис. 28.1 и 28.2. В каждом случае результаты прямого обхода пары лесов совпадают, поэтому оба определения нам подходят.

Мы пока ещё не совсем закончили, поскольку добавление «розового куста» к лесу (в конец списка) не является операцией с константным временем. Очевидные идеи для решения проблемы типа добавления в начало или введения аккумулирующего параметра не работают. Самым подходящим методом, которым мы также воспользуемся в несколько иных целях

в последующих жемчужинах, оказывается введение *очередей*, леса розовых кустов становятся при этом очередями, а не списками, как это было до сих пор. Реализация очередей, принадлежащая Окасаки, предоставляет тип *Queue a*, все следующие операции для которого выполняются за константное время:

```
insert :: Queue a → a → Queue a
remove :: Queue a → (a, Queue a)
empty :: Queue a
isempty :: Queue a → Bool
```

Чтобы воспользоваться очередями, переопределим тип *Forest*:

```
type Forest a = Queue (Rose a)
data Rose a = Node a (Forest a)
```

Функции *op1'* и *op2'* такие же, что и прежде, за тем исключением, что выражения вида *as ++ [Node x bs]* заменяются на *insert as (Node x bs)*. Теперь получаем:

```
boxall = unfoldr step · wrapQueue · fst · foldr op1' (empty, empty)
boxall = unfoldr step · wrapQueue · fst · foldr op2' (empty, empty)
```

где

```
step           :: [Forest a] → Maybe (a, [Forest a])
step []        = Nothing
step (zs : zss) = Just (x, consQueue xs (consQueue ys zss))
                  where (Node x xs, ys) = remove zs
consQueue     :: Queue a → [Queue a] → [Queue a]
consQueue xs xss = if isempty xs then xss else xs : xss
wrapQueue     :: Queue a → [Queue a]
wrapQueue xs  = consQueue xs []
```

Оба получившихся определения *boxall* являются бесцикловыми алгоритмами.

Заключительные замечания

Термин бесциклового алгоритма был предложен в (Ehrlich, 1973). Некоторое количество таких алгоритмов для порождения комбинаторных

объектов появилось в опубликованных Кнутом черновиках трёх разделов четвёртого тома Искусства программирования (Knuth, 2005). В этих выпусках имеется множество ссылок на соответствующую литературу. Цитата из Кнута появилась в (Knuth, 2001). Реализацию очередей Окасаки можно найти в (Okasaki, 1995).

Литература

- Ehrlich, G. (1973). Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM* **20**, 500–13.
- Knuth, D. E. (2001). SPIDERS: a program downloadable from www-cs-faculty.stanford.edu/~knuth/programs.html.
- Knuth, D. E. (2005). *The Art of Computer Programming, Volume 4, Fascicles 2,3,4*. Reading, MA: Addison-Wesley. [Имеется русский перевод: Кнут Д. Искусство программирования, том 4, выпуск 2, 3, 4. М.: Вильямс. 2007–2008.]
- Okasaki, C. (1995). Simple and efficient purely functional queues and deques. *Journal of Functional Programming* **5** (4), 583–92.

29

Алгоритм Джонсона—Троттера

Введение

Алгоритм Джонсона—Троттера это метод генерации всех перестановок заданного списка в порядке, при котором переход от некоторой перестановки к следующей выполняется переменой мест двух соседних элементов. В этой жемчужине мы получим бесцикловую версию этого алгоритма. Основная идея состоит в использовании одной из программ для обобщённого бустрофедонового произведения *boxall*, разработанной в предыдущей жемчужине.

Рекурсивная формулировка

В алгоритме Джонсона—Троттера переходы длиной более единицы для списка длины n определяются через переходы для списка длины $n - 1$. Пометим элементы списка их позициями от 0 до $n - 1$, и пусть сам список обозначается $xs \uparrow [x]$. Начинаем с исходящего прогона $[n - 1, n - 2, \dots, 1]$, где переход i означает «поменять местами элементы, находящиеся в позициях i и $i - 1$ ». Эффект этих действий состоит в перемещении элемента x к началу списка, т.е. в результате получаем список $[x] \uparrow xs$. Например, переходы $[3, 2, 1]$, применённые к строке "abcd" дают следующие три перестановки "abdc", "adbc" и "dabc". Предположим теперь, что переходы, генерирующие перестановки xs , задаются списком $[j_1, j_2, \dots]$. Применим переход $j_1 + 1$ к текущей перестановке $[x] \uparrow xs$. Увеличение на единицу необходимо, поскольку список xs сдвинут теперь на одну позицию вправо

от элемента x . Теперь снова переносим x в последнюю позицию, применяя к списку последовательность переходов $[1, 2, \dots, n-1]$ (восходящий прогон). В результате получаем окончательную перестановку $ys \uparrow\downarrow [y]$, где ys это результат применения перестановки j_1 к списку xs . К примеру, последовательность переходов $[3, 1, 2, 3]$, применённая к "dabc" даёт ещё четыре перестановки "dacb", "adcb", "acdb" и "acbd". На следующем шаге применяем к xs переход j_2 и перегоняем элемент x снова в начало списка. Переход j_2 модифицировать не нужно, так как после восходящего прогона соответствующая перестановка находится левее x . Продолжаем в том же духе, перемежая прогоны элемента x вниз и вверх с переходами вплоть до $(n - 1)$ -го.

Данное выше описание легко записывается с применением определённого в предыдущей жемчужине бустрофедонового произведения (\square):

```
jcode :: Int → [Int]
jcode 1 = []
jcode n = (bumpBy 1 (jcode (n - 1))) □ [n - 1, n - 2 .. 1]
```

Функция $bumpBy k$ прибавляет k ко всем элементам в чётных позициях:

$bumpBy k []$	=	[]
$bumpBy k [a]$	=	$[a + k]$
$bumpBy k (a : b : as)$	=	$(a + k) : b : bumpBy k as$

Наша цель заключается в приведении $jcode$ к бесциклической форме.

План

Общий план состоит в выражении $jcode$ через $boxall$, обобщённое бустрофедоновое произведение, введённое в предыдущей жемчужине, а затем в обращении к бесциклическому алгоритму для $boxall$. Чтобы это проделать, следует обобщить $jcode$. Рассмотрим функцию $code$, определённую следующим образом:

$$code (k, n) = bumpBy k (*jcode* n)$$

Ясно, что $jcode n = code (0, n)$. При положительном нечётном n рассуждаем так:

$$\begin{aligned} code (k, n) \\ = \{ \text{определение} \} \end{aligned}$$

```

 $bumpBy k (jcode n)$ 
= { определение  $jcode$  }
 $bumpBy k (bumpBy 1 (jcode (n - 1)) \square [n - 1, n - 2..1])$ 
= { утверждение, см. ниже }
 $bumpBy (k + 1) (jcode (n - 1)) \square bumpBy k [n - 1, n - 2..1]$ 
= { определение  $code$  и  $bumpDn$  (см. ниже) }
 $code (k + 1, n - 1) \square bumpDn (k, n)$ 

```

где $bumpDn$ (толчок вниз) определяется так:

$$bumpDn (k, n) = bumpBy k [n - 1, n - 2..1] \quad (29.1)$$

Первое утверждение состоит в том, что

```

 $bumpBy k (xs \square ys)$ 
= if even (length ys) then  $bumpBy k xs \square bumpBy k ys$ 
else  $xs \square bumpBy k ys$ 

```

Оно может быть доказано на основе определения \square и того факта, что

```

 $bumpBy k (xs \square [y] \square ys)$ 
= if even (length xs) then  $bumpBy k xs \square bumpBy k ([y] \square ys)$ 
else  $bumpBy k xs \square [y] \square bumpBy k ys$ 

```

Несложные детали оставлены в качестве упражнения.

Случай чётного n рассматривается аналогично, и в итоге мы получаем следующее определение $code$:

```

 $code (k, 1) = []$ 
 $code (k, n) = code (k', n - 1) \square bumpDn (k, n)$ 
    where  $k' = \text{if odd } n \text{ then } k + 1 \text{ else } 1$ 

```

Например, учитывая, что операция \square ассоциативна, имеем:

$$code (0, 4) = bumpDn (2, 2) \square bumpDn (1, 3) \square bumpDn (0, 4)$$

Вспоминая, что $boxall = foldr (\square) []$, мы можем теперь переписать $code$ в следующем виде:

$$code = boxall \cdot map bumpDn \cdot pairs$$

где

$$\begin{aligned}
 pairs &:: (Int, Int) \rightarrow [(Int, Int)] \\
 pairs(k, 1) &= [] \\
 pairs(k, n) &= pairs(k', n - 1) \uplus [(k, n)] \\
 &\quad \text{where } k' = \text{if odd } n \text{ then } k + 1 \text{ else } 1
 \end{aligned}$$

И вновь несложные детали оставлены в качестве упражнения. Вычисление $pairs(k, n)$ выполняется за $\Theta(n^2)$ операций, однако эту сложность можно сократить до $\Theta(n)$ с помощью аккумулирующего параметра. Определим $addpair$:

$$addpair(k, n) ps = pairs(k, n) \uplus ps$$

Построение прямого определения $addpair$ даст нам:

$$\begin{aligned}
 pairs(k, n) &= addpair(k, n) [] \\
 addpair(k, 1) ps &= ps \\
 addpair(k, n) ps &= addpair(k', n - 1)((k, n) : ps) \\
 &\quad \text{where } k' = \text{if odd } n \text{ then } k + 1 \text{ else } 1
 \end{aligned}$$

Следовательно, в силу $jcode n = code(0, n)$, получаем:

$$\begin{aligned}
 jcode &= boxall \cdot map bumpDn \cdot pairs \\
 \text{where } pairs n &= addpair(0, n) []
 \end{aligned}$$

Так как мы знаем, как реализовать бесцикловый $boxall$, это определение $jcode$ даёт нам бесцикловый алгоритм. Или нет?

Бесцикловый алгоритм

Нет, не даёт: частью пролога является $map bumpDn(pairs n)$, что требует $\Theta(n^2)$ операций. Правила игры допускают только те прологи, которым достаточно $\Theta(n)$ операций.

Что нам действительно нужно, так это превратить $boxall \cdot map bumpDn$ в бесцикловый алгоритм. Реализовать бесцикловую $bumpDn$ нетрудно, поэтому давайте начнём именно с неё. Вспомним определение (29.1):

$$bumpDn(k, n) = bumpBy k [n - 1, n - 2..1]$$

Для удаления циклов из $bumpDn$ мы можем организовать состояние, содержащее четвёрку (j, k, m, n) , в которой первый компонент j начинается с k , а затем чередует значения 0 и k , второй и четвёртый компоненты k

и n не меняются, а третий компонент m отсчитывает шаги от $n - 1$ до 1. Теперь получаем:

$$\begin{aligned} bumpDn &= unfoldr stepDn \cdot prologDn \\ prologDn(k, n) &= (k, k, n - 1, 1) \\ stepDn(j, k, m, n) &= \begin{cases} \text{if } m < n \text{ then Nothing} \\ \text{else Just } (m + j, (k - j, k, m - 1, n)) \end{cases} \end{aligned}$$

Похожим образом удаляем циклы из $reverse \cdot bumpDn$:

$$\begin{aligned} reverse \cdot bumpDn &= unfoldr stepUp \cdot prologUp \\ prologUp(k, n) &= (\text{if even } n \text{ then } k \text{ else } 0, k, 1, n - 1) \\ stepUp(j, k, m, n) &= \begin{cases} \text{if } m > n \text{ then Nothing} \\ \text{else Just } (m + j, (k - j, k, m + 1, n)) \end{cases} \end{aligned}$$

Функции $stepDn$ и $stepUp$ можно объединить в одну, скажем, $bump$ (имя $step$ нам понадобится позднее для других целей), если добавить пятый компонент i , полагая $i = -1$ для шага вниз и $i = 1$ для шага вверх. Это даёт нам:

$$\begin{aligned} bumpDn &= unfoldr bump \cdot prologDn \\ reverse \cdot bumpDn &= unfoldr bump \cdot prologUp \end{aligned}$$

где

$$\begin{aligned} bump(i, j, k, m, n) &= \begin{cases} \text{if } i * (n - m) < 0 \text{ then Nothing} \\ \text{else Just } (m + j, (i, k - j, k, m + i, n)) \end{cases} \\ prologDn(k, n) &= (-1, k, k, n - 1, 1) \\ prologUp(k, n) &= (1, \text{if even } n \text{ then } k \text{ else } 0, k, 1, n - 1) \end{aligned}$$

Вспомним теперь, что в предыдущей жемчужине одно из бесциклических определений $boxall$ имело форму $boxall = unfoldr step \cdot prolog$, где

$$prolog = wrapQueue \cdot fst \cdot foldr op (empty, empty)$$

Функция op определялась следующим образом:

$$\begin{aligned} op xs (ys, sy) &= \begin{cases} \text{if even } (length xs) \\ \text{then } (mix xs (ys, sy), mix (reverse xs) (sy, ys)) \\ \text{else } (mix xs (ys, sy), mix (reverse xs) (ys, sy)) \end{cases} \\ mix [] (ys, sy) &= ys \\ mix (x : xs) (ys, sy) &= insert ys (\text{Node } x (mix xs (sy, ys))) \end{aligned}$$

Функция *step* определялась так:

$$\begin{aligned}
 \text{step} [] &= \text{Nothing} \\
 \text{step} (\text{zs} : \text{zss}) &= \text{Just} (x, \text{consQueue xs} (\text{consQueue ys zss})) \\
 &\quad \text{where } (\text{Node } x \text{ xs}, \text{ys}) = \text{remove zs} \\
 \text{consQueue} &:: \text{Queue } a \rightarrow [\text{Queue } a] \rightarrow [\text{Queue } a] \\
 \text{consQueue xs xss} &= \text{if } \text{isempty xs} \text{ then } xss \text{ else } \text{xs} : xss \\
 \text{wrapQueue} &:: \text{Queue } a \rightarrow [\text{Queue } a] \\
 \text{wrapQueue xs} &= \text{consQueue xs} []
 \end{aligned}$$

Рассуждаем:

$$\begin{aligned}
 jcode & \\
 = & \{ \text{определение } jcode \text{ через } \text{boxall} \} \\
 & \text{boxall} \cdot \text{map bumpDn} \cdot \text{pairs} \\
 = & \{ \text{бесцикловое определение } \text{boxall} \} \\
 & \text{unfoldr step} \cdot \text{wrapQueue} \cdot \text{fst} \cdot \text{foldr op} (\text{empty}, \text{empty}) \cdot \\
 & \text{map bumpDn} \cdot \text{pairs} \\
 = & \{ \text{закон слияния} \} \\
 & \text{unfoldr step} \cdot \text{wrapQueue} \cdot \text{fst} \cdot \text{foldr op}' (\text{empty}, \text{empty}) \cdot \text{pairs}
 \end{aligned}$$

где

$$\text{op}' (k, n) (\text{ys}, \text{sy}) = \text{op} (\text{bumpDn} (k, n)) (\text{ys}, \text{sy})$$

Разворачивая это определение и используя тот факт, что *bumpDn* (*k*, *n*) имеет чётную длину при нечётном *n*, а также пользуясь определениями *prologDn* и *prologUp*, находим:

$$\begin{aligned}
 \text{op}' (k, n) (\text{ys}, \text{sy}) & \\
 = & \text{if odd } n \\
 & \text{then } (\text{mix} (\text{unfoldr bump} (-1, k, k, n - 1, 1)) (\text{ys}, \text{sy}), \\
 & \quad \text{mix} (\text{unfoldr bump} (1, 0, k, 1, n - 1)) (\text{sy}, \text{ys})) \\
 & \text{else } (\text{mix} (\text{unfoldr bump} (-1, k, k, n - 1, 1)) (\text{ys}, \text{sy}), \\
 & \quad \text{mix} (\text{unfoldr bump} (1, k, k, 1, n - 1)) (\text{ys}, \text{sy}))
 \end{aligned}$$

Функция *op'* (*k*, *n*) выполняется за $\Theta(n)$ операций, поэтому для вычисления *foldr op' (empty, empty)* необходимо квадратичное время. Задачи *op'* можно немного сократить, вытащив *unfoldr bump* из её определения и поручив всю работу модифицированной версии *step*. Как результат, мы отложим

вычисление первого аргумента *mix*. Для представления отложенного вычисления нам понадобится новый тип данных:

```
type Forest a = Queue (Rose a)
data Rose a = Node a (Forest a, Forest a)
```

В новом определении «розовый куст» состоит вместо одного леса из двух лесов как потомков. Рассмотрим теперь новую версию *step*, определённую так:

```
type State = (Int, Int, Int, Int, Int)
type Pair a = (a, a)
step :: [Forest (Int, State)] → Maybe (Int, [Forest (Int, State)])
step [] = Nothing
step (zs : zss)
    = Just (x, consQueue (mix q (sy, ys)) (consQueue zs' zss))
      where (Node (x, q) (ys, sy), zs') = remove zs
```

где *mix* изменена следующим образом:

```
mix :: State → Pair (Forest (Int, State)) → Forest (Int, State)
mix (i, j, k, m, n) (ys, sy)
    = if i * (n - m) < 0 then ys
      else insert ys (Node (m + j, (i, k - j, k, m + i, n)) (ys, sy))
```

Функция *step* генерирует следующий переход *x* и передаёт состояние *q* (как пятёрку) функции *mix*, которая вычисляет результат этого перехода, если он есть, и новое состояние. Теперь можно утверждать, что

```
jcode = unfoldr step · wrapQueue · fst · foldr op' (empty, empty) · pairs
```

где *op'* переопределена так:

```
op' :: (Int, Int) → Pair (Forest (Int, State))
      → Pair (Forest (Int, State))
op' (k, n) (ys, sy) = if odd n
    then (mix (-1, k, k, n - 1, 1) (ys, sy),
          mix (1, 0, k, 1, n - 1) (sy, ys))
    else (mix (-1, k, k, n - 1, 1) (ys, sy),
          mix (1, k, k, 1, n - 1) (ys, sy))
```

Детали вновь оставлены в качестве упражнения. Довольно длинный прог

$$\text{prolog} = \text{wrapQueue} \cdot \text{fst} \cdot \text{foldr } op' (\text{empty}, \text{empty}) \cdot \text{pairs}$$

выполняется за $\Theta(n)$ операций, при условии, что применяется к n , а *step* выполняется за константное время, поэтому в результате мы-таки получили чистейшую, в 24 карата, бесцикловую программу для *jcode*.

Заключительные замечания

Если бы перед нами не стояло жёсткое требование относительно линейности пролога, мы могли бы остановиться сразу после получения следующего определения:

$$\text{jcode} = \text{boxall} \cdot \text{map bumpDn} \cdot \text{pairs}$$

Что это определение реально показывает, так это полезность обобщённого бустрофедонового произведения — функции *boxall* — в задачах генерации комбинаторных объектов различных видов. В заключительной жемчужине мы посмотрим на другие способы его применения.

Алгоритм Джонсона—Троттера был независимо описан в (Johnson, 1963) и (Trotter, 1962). Как уже указывалось в предыдущей жемчужине, в работе (Ehrlich, 1973), где вводится идея бесциклового алгоритма, основной задачей являлось написание бесцик洛вой программы как раз для алгоритма Джонсона—Троттера.

Литература

- Ehrlich, G. (1973). Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM* **20**, 500–13.
- Johnson, S. M. (1963). Generation of permutations by adjacent transpositions. *Mathematics of Computation* **17**, 282–5.
- Trotter, A. F. (1962). *Perm* (Algorithm 115). *Communications of the ACM* **5**, 434–5.

30

Прядение паутины для чайников

Не ведаем, какую сеть плетём,

Начав лишь выводить.

(С извинениями сэру Вальтеру Скотту)

Введение

Рассмотрим задачу порождения всех битовых строк $a_1a_2 \dots a_n$ длины n , удовлетворяющих заданным ограничениям вида $a_i \leq a_j$ при различных i и j . Строки будем генерировать в порядке кодов Грэя, т.е. каждая строка должна получаться из предыдущей изменением в точности одного бита. Преобразующим кодом называется список целых чисел, обозначающих номера тех битов, которые следует менять на каждом шаге. Например, при $n = 3$ и с ограничениями $a_1 \leq a_2$ и $a_3 \leq a_2$ одним из возможных кодов Грэя будет последовательность 000, 010, 011, 111, 110, где преобразующий код это $[2, 3, 1, 3]$, а начальная строка 000.

Есть, правда, в этой задаче одна загвоздка: у неё не всегда имеется решение. Например, при $n = 4$ и с двумя ограничениями $a_1 \leq a_2 \leq a_4$ и $a_1 \leq a_3 \leq a_4$ шесть подходящих битовых строк 0000, 0001, 0011, 0101, 0111 и 1111 невозможно расположить в требуемом порядке. Среди этих строк четыре строки чётного веса (по количеству единиц) и две нечётного, тогда как в любом коде Грэя чётность весов должна чередоваться.

Ограничения вида $a_i \leq a_j$ на битовых строках длины n можно представить ориентированным графом с n вершинами, в котором наличие дуги $i \leftarrow j$ соответствует ограничению $a_i \leq a_j$. Кнут (Knuth) и Раски (Ruskey)

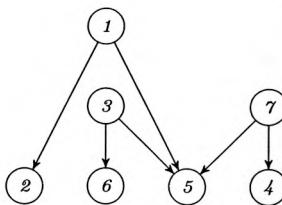


Рис. 30.1: трёхлапый паук

показали, как построить код Грэя в случае, если орграф является сильно ациклическим, т.е. соответствующий ему неориентированный граф остаётся ациклическим после удаления всей информации о направлениях дуг. Они назвали связный сильно ациклический ориентированный граф пауком, потому что если рисовать дугу $i \leftarrow j$ так, чтобы i оказывалось под j , то орграф можно сделать похожим на паукообразное (на рис. 30.1 можно увидеть трёхногого паука). Сильно ациклический орграф они называют САО, но так как его компоненты связности являются пауками, мы продолжим соответствующую метафору и будем называть его паучьим гнездом (*nest*).

Кнут обозначил задачу генерации битовых строк кода Грэя термином *плющение пауков*. Несколько более формально она звучит так: «построение всех идеалов¹ сильно ациклического частично упорядоченного множества в порядке кода Грэя». Поскольку пауки полезны для окружающей среды, и их никогда не следует плющить, мы вместо этого будем использовать для этой задачи название «прядение паутины».

О задаче прядения паутины полезно думать в терминах раскрасок. Предположим, что вершины паука на рис. 30.1 выкрашены в чёрный цвет, если соответствующий бит равен 1, и в белый в противном случае. Таким образом, каждый потомок белой вершины должен быть белым. Например, если вершина 1 является белой, то вершины 2 и 5 также должны быть белыми. Теперь задача прядения паутины заключается в перечислении всех допустимых раскрасок, начиная с некоторой, причём на каждом шаге должен меняться цвет только одной вершины. Как будет видно позднее, начальную раскраску вообще говоря нельзя выбирать так, чтобы все вершины были либо белыми, либо чёрными.

¹Идеалом частично упорядоченного множества S называется такое подмножество I множества S , для которого из $x \in I$ и $x \leq y$ следует, что $y \in I$.

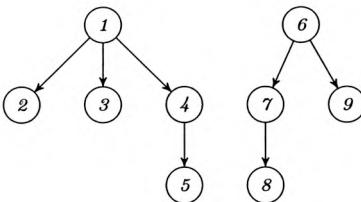


Рис. 30.2: гнездо с двумя древесными пауками

Наша цель в этой жемчужине состоит в выведении бесциклового алгоритма для решения задачи прядения паутины. Кнут и Раски приводят алгоритм решения, но он содержит циклы. Правда, на сайте Кнута есть программа SPIDERS (пауки), в которой реализован бесцик洛вый алгоритм. Она довольно сложна, и Кнут с готовностью признаёт:

Но прежде я должен принести извинения в том, что алгоритм представляется довольно искусным, а я не в состоянии придумать способ его объяснения для «чайников».

Отсюда наш заголовок. Наша цель состоит в получении бесциклового алгоритма прядения паутины. Я не имею ни малейшего представления о том, как связан мой алгоритм с алгоритмом Кнута, поскольку я тоже не могу объяснить его алгоритм.

Прядение паутины древесными пауками

Давайте для начала рассмотрим более простую задачу, в которой каждый паук является деревом, т.е. все паучьи лапки направлены вниз. Этот случай рассматривался Кодой (Koda) и Раски. Вот подходящие типовые определения:

```

type Nest   = [Spider]
data Spider = Node Int Nest
  
```

Гнездо с двумя древесными пауками показано на рис. 30.2. Мы будем предполагать, что метками вершин в паучьем гнезде размера n являются элементы списка $[1 \dots n]$ в некотором порядке. Можно определить функции

ncode и *scode*, вычисляющие преобразующие коды для гнезда и одного паука соответственно, воспользовавшись для этого функцией обобщённого бустрофедонового произведения *boxall*:

$$\begin{aligned} ncode &:: \text{Nest} \rightarrow [\text{Int}] \\ ncode &= \text{boxall} \cdot \text{map } scode \\ scode &:: \text{Spider} \rightarrow [\text{Int}] \\ scode (\text{Node } a \text{ xs}) &= a : ncode \text{ xs} \end{aligned}$$

Преобразующий код для одного паука состоит из начального перехода с изменением цвета корневой вершины (самостоянно, её перекрашивания из белого в чёрный), за которым следует полный список переходов для гнезда его подпауков. Определение функции *ncode* короткое и довольно приятное, но в нём есть цикл.

Бесцикловая программа

Первый шаг на пути к бесцикловому решению полностью задаётся формой определения функции *ncode*. Вспомнив, что $\text{boxall} = \text{foldr}(\square) []$, и применив закон слияния для *foldr*, получаем:

$$ncode = \text{foldr} ((\square) \cdot scode) []$$

Сфокусируемся теперь на функции $(\square) \cdot scode$. Рассуждаем:

$$\begin{aligned} scode (\text{Node } a \text{ xs}) \square bs &= \{ \text{определение } scode \} \\ &= (a : ncode \text{ xs}) \square bs \\ &= \{ \text{определение } \square \} \\ &bs \sqcup [a] \sqcup (ncode \text{ xs} \square (reverse \text{ bs})) \\ &= \{ \text{исходное определение } ncode \} \\ &bs \sqcup [a] \sqcup (\text{boxall} (\text{map } scode \text{ xs}) \square (reverse \text{ bs})) \end{aligned}$$

Третий терм этого выражения имеет вид

$$(\text{foldr} (\square) [] ass) \square cs$$

в котором $ass = \text{map } scode \text{ xs}$, а $cs = reverse \text{ bs}$. Это подсказывает нам возможность применения закона слияния для свёртки *foldr*. Полагая $f as = as \square cs$, получаем, что f является строгой, а $f [] = cs$, поскольку

$[]$ является единицей относительно \square . Следовательно, закон слияния для свёртки даёт нам:

$$(foldr (\square) [] ass) \square cs = foldr h cs ass$$

при условии, что мы сможем найти такую функцию h , что $(as \square bs) \square cs = h as (bs \square cs)$. Но операция \square ассоциативна, поэтому в качестве h можно взять $h = (\square)$.

Собирая вместе эти вычисления, получаем:

$$\begin{aligned} & scode (Node a xs) \square bs \\ &= \{ \text{см. выше} \} \\ & bs \uparrow\!\! [a] \uparrow\!\! (boxall (map scode xs) \square (reverse bs)) \\ &= \{ \text{слияние для свёртки} \} \\ & bs \uparrow\!\! [a] \uparrow\!\! foldr (\square) (reverse bs) (map scode xs) \\ &= \{ \text{слияние для map} \} \\ & bs \uparrow\!\! [a] \uparrow\!\! foldr ((\square) \cdot scode) (reverse bs) xs \end{aligned}$$

Таким образом, положив $op = (\square) \cdot scode$, мы можем получить следующее решение:

$$\begin{aligned} ncode &= foldr op [] \\ op (Node a xs) bs &= bs \uparrow\!\! [a] \uparrow\!\! foldr op (reverse bs) xs \end{aligned}$$

Оставшиеся шаги состоят в удалении $reverse$ посредством одновременного вычисления $ncode$ и $reverse \cdot ncode$ и в представлении их результатов очередью, элементами которой являются «розовые кусты» относительно абстрагирующей функции $preorder$. «Розовый куст» определяется так:

$$\begin{aligned} \text{type Forest } a &= Queue (Rose a) \\ \text{data Rose } a &= Fork a (Forest a) \end{aligned}$$

Мы выполняли эти шаги во время вывода бесцик洛вой программы для функции $boxall$, поэтому сейчас просто покажем результат:

$$\begin{aligned} ncode &= unfoldr step \cdot wrapQueue \cdot fst \cdot foldr op (empty, empty) \\ op (Node a xs) (bs, sb) &= (insert bs (Fork a cs), insert sc (Fork a sb)) \\ \text{where } (cs, sc) &= foldr op (sb, bs) xs \end{aligned}$$

Функции $step$ и $wrapQueue$ в точности такие же, как и в решении для $boxall$. Поскольку $foldr op (empty, empty)$ выполняется за линейное по размеру гнезда время, эта программа является бесцикловой программой для $ncode$.

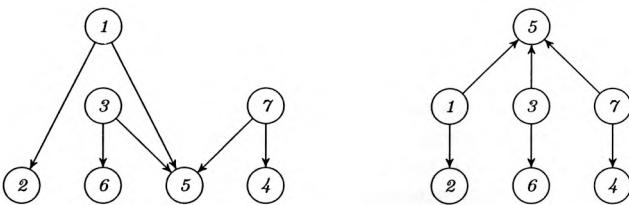


Рис. 30.3: паук и связанное с ним дерево

Прядение паутины пауками общего вида

Теперь мы готовы заняться задачей о прядении паутины в общем случае. Во-первых, заметим, что приподнимая паука за одну из его вершин, мы получаем дерево с направленными дугами, например, такими, как показано на рис. 30.3. В зависимости от выбора вершины дерева получаются разными, но все они отражают одни и те же ограничения. Следовательно, паука общего вида можно моделировать следующими типовыми определениями:

```

type Nest    = [Spider]
data Spider = Node Int [Leg]
data Leg     = Dn Spider | Up Spider

```

У паука лапки, а не дуги. Паучья лапка направлена либо вверх, либо вниз к другому пауку.

При работе с пауками общего вида возникает одно затруднение, которого не было с видами попроще: начальная битовая строка необязательно состоит из всех нулей. Например, при $n = 3$ и с ограничениями $a_1 \geq a_2 \leq a_3$ допустимы пять строк, а именно 000, 001, 100, 101 и 111, причём их можно организовать в код Грэя, только начав с одной из строк нечётного веса: 001, 100 или 111. Мы пока, однако, отложим рассмотрение функции $seed : Nest \rightarrow [Bit]$, которая определяет начальную строку, на более позднее время.

Как и с древесными пауками, определяем:

```

ncode :: Nest → [Int]
ncode = boxall · map scode

```

Функция $scode$ определяется как конкатенация двух списков, белого кода и чёрного кода:

$$\begin{aligned} s\text{code} &:: \text{Spider} \rightarrow [\text{Int}] \\ s\text{code} (\text{Node } a \text{ legs}) &= w\text{code legs} ++ [a] ++ b\text{code legs} \end{aligned}$$

Белый код, $w\text{code}$, для паука $\text{Node } a \text{ legs}$ является допустимой последовательностью переходов, когда корневая вершина a выкрашена белым (соответствует нулевому биту), тогда как чёрный код является допустимой последовательностью, если эта вершина выкрашена чёрным (соответствует единичному биту). Таким образом, $s\text{code}$ можно определить как последовательность, которая проходит по белому коду, изменяет цвет a с белого на чёрный, а затем идёт по чёрному коду. Заметим, что когда пауки являются древесными, т.е. все лапки направлены вниз, белый код является пустой последовательностью.

Чтобы $s\text{code}$ работала верно, последняя раскраска после выполнения $w\text{code legs}$ должна быть начальной раскраской, на которой начинается работа $b\text{code legs}$. Чтобы раскраски друг другу соответствовали, нам нужно определить $w\text{code}$ посредством некоторой вариации \square , которую мы обозначим \diamond . Операция \diamond является сопряжённой с \square :

$$as \diamond bs = \text{reverse} ((\text{reverse } as) \square (\text{reverse } bs))$$

Тогда как $as \square bs$ начинается с bs и заканчивается либо bs , либо $\text{reverse } bs$ в зависимости от того, является ли длина as чётной, $as \diamond bs$ заканчивается на bs и начинается либо с bs , либо с $\text{reverse } bs$. Например:

$$\begin{aligned} [2, 3, 4] \square [0, 1] &= [0, 1, 2, 1, 0, 3, 0, 1, 4, 1, 0] \\ [2, 3, 4] \diamond [0, 1] &= [1, 0, 2, 0, 1, 3, 1, 0, 4, 0, 1] \end{aligned}$$

Операцию \square можно выразить через \diamond посредством сопряжения, но есть и другой способ:

$$as \square bs = \text{if even}(\text{length } as) \text{then } as \diamond bs \text{ else } as \diamond (\text{reverse } bs) \quad (30.1)$$

Похожим образом можно определить \diamond через \square . Утверждение (30.1) понадобится нам в дальнейшем.

Операция \diamond является ассоциативной, роль единицы играет пустая последовательность. Доказательство оставляется в качестве упражнения, оно основывается на следующем факте:

$$(as ++ [b] ++ bs) \diamond cs = (as \diamond cs') ++ [b] ++ (bs \diamond cs)$$

где $cs' = \text{if even}(\text{length } bs) \text{then reverse } cs \text{ else } cs$. Похожее свойство требовалось для доказательства ассоциативности \square .

Полагая $\text{coxall} = \text{foldr} (\diamond) []$, можно определить:

$$\begin{aligned} wcode, bcode &:: [\text{Leg}] \rightarrow [\text{Int}] \\ wcode &= \text{coxall} \cdot \text{map } wc \\ bcode &= \text{boxall} \cdot \text{map } bc \end{aligned}$$

где функции $wc, bc :: \text{Leg} \rightarrow [\text{Int}]$ ещё ждут своего определения. Использование coxall в определении $wcode$ гарантирует, что окончательная раскраска в результате выполнения $wcode$ будет объединением раскрасок, полученных после переходов wc , а использование boxall в определении $bcode$ — что эта раскраска также будет объединением раскрасок, которые являются начальными для переходов bc .

Остаётся определить wc и bc . В соответствии с приведёнными выше условиями получаем следующие определения:

$$\begin{aligned} wc(Up(\text{Node } a \text{ legs})) &= wcode \text{ legs} ++ [a] ++ bcode \text{ legs} \\ wc(Dn(\text{Node } a \text{ legs})) &= \text{reverse}(wcode \text{ legs}) \\ bc(Up(\text{Node } a \text{ legs})) &= \text{reverse}(bcode \text{ legs}) \\ bc(Dn(\text{Node } a \text{ legs})) &= wcode \text{ legs} ++ [a] ++ bcode \text{ legs} \end{aligned}$$

Посмотрим сначала на $wc(Up x)$. Если вершина исходного паука белая и соединяется с x направленной вверх дугой, то никаких ограничений на $wc(Up x)$ нет, поэтому результат можно определить равным либо списку $scode x$, либо обратному ему. Однако последующие действующие на x переходы берутся из списка $bc(Up x)$, а значит, единственным способом удовлетворить требованию окончательной раскраски первых с учётом начальной окраски последних оказываются приведённые выше определения. Аналогичные рассуждения применимы и в случае с $bc(Dn x)$ и $wc(Dn x)$.

Покажем, наконец, что $ncode$ можно выразить через $bcode$:

$$\begin{aligned} ncode \text{ xs} &= \{ \text{определение } ncode \} \\ &= \text{boxall}(\text{map } scode \text{ xs}) \\ &= \{ \text{определение } scode \text{ и } bc \} \\ &= \text{boxall}[\text{bc}(Dn x) | x \leftarrow xs] \\ &= \{ \text{определение } bcode \} \\ &= bcode[(Dn x) | x \leftarrow xs] \end{aligned}$$

Программа для $ncode$ без реализации $boxall$ и $coxall$ приведена на рис. 30.4. Наша цель заключается в получении бесциклической реализации $ncode$.

```

 $\begin{aligned} ncode &:: \text{Nest} \rightarrow [\text{Int}] \\ ncode &= bcode \cdot \text{map } Dn \\ bcode, wcode &:: [\text{Leg}] \rightarrow [\text{Int}] \\ bcode &= boxall \cdot \text{map } bc \\ wcode &= coxall \cdot \text{map } wc \\ bc, wc &:: \text{Leg} \rightarrow [\text{Int}] \\ bc (\text{Up} (\text{Node } a \text{ legs})) &= \text{reverse} (bcode \text{ legs}) \\ bc (\text{Dn} (\text{Node } a \text{ legs})) &= wcode \text{ legs} \uparrow\uparrow [a] \uparrow\uparrow bcode \text{ legs} \\ wc (\text{Up} (\text{Node } a \text{ legs})) &= wcode \text{ legs} \uparrow\uparrow [a] \uparrow\uparrow bcode \text{ legs} \\ wc (\text{Dn} (\text{Node } a \text{ legs})) &= \text{reverse} (wcode \text{ legs}) \end{aligned}$ 

```

Рис. 30.4: исходная программа для *ncode*

Бесцикловый алгоритм

Преобразование алгоритма к бесцик洛вой форме следует тем же путём, которым мы решали более простую задачу о гнезде древесных пауков. А именно, необходимо:

- а) Избавиться в определении *ncode* от *boxall* и *coxall*, воспользовавшись законами слияния для *map* и *foldr*.
- б) Заменить *reverse* на одновременные вычисления в кортеже.
- в) Применить очереди с целью преодоления проблем с остающейся сложностью.

Самым хитрым оказывается использование законов слияния. Первым делом применим закон слияния для *map* к определениям *wcode* и *bcode*, получив:

$$\begin{aligned} bcode &= \text{foldr} ((\square) \cdot bc) [] \\ wcode &= \text{foldr} ((\diamond) \cdot wc) [] \end{aligned}$$

Сфокусируемся теперь на термах $(\square) \cdot bc$ и $(\diamond) \cdot wc$. Всё, что мы выясним относительно первого, с очевидными изменениями переносится на второй. Будем рассуждать как можно ближе к случаю древесных пауков.

В определении *bc* две части, рассмотрим их по очереди. Для первой имеем:

$$bc (\text{Up} (\text{Node } a \text{ legs})) \square cs$$

$$\begin{aligned}
 &= \{ \text{определение } bc \} \\
 &\quad \text{reverse} (bcode \ legs) \square cs \\
 &= \{ \text{определение } bcode \} \\
 &\quad \text{reverse} (\text{boxall} (\text{map } bc \ legs)) \square cs
 \end{aligned}$$

Как и в ситуации с древесными пауками, на следующем шаге необходимо применить закон слияния для свёртки: если можно найти такую функцию h , для которой

$$\text{reverse} (as \square bs) \square cs = h as ((\text{reverse } bs) \square cs) \quad (30.2)$$

то

$$\text{reverse} (\text{boxall} (\text{map } bc \ legs)) \square cs = \text{foldr } h \ cs (\text{map } bc \ legs)$$

Проблема в отсутствии функции, удовлетворяющей (30.2). Причина в неинъективности \square , к примеру:

$$\text{"abab"} \square \text{"aaaba"} = \text{"ab"} \square \text{"aaabaaaaba"}$$

Если бы функция h , удовлетворяющая (30.2), существовала, то мы бы потребовали

$$\begin{aligned}
 &\text{reverse} (as \square \text{"bab"}) \square \text{"aaaba"} \\
 &= \text{reverse} (as \square \text{"ba"}) \square \text{"aaabaaaaba"}
 \end{aligned}$$

для всех as . Однако приведённое выше утверждение ложно: достаточно взять, например, $as = \text{"c"}$.

Что мы можем сделать, так это найти такую h , что:

$$\text{reverse} (as \square bs) \diamond cs = h as ((\text{reverse } bs) \diamond cs) \quad (30.3)$$

Утверждение (30.3) имеет ту же форму, что и (30.2), за тем исключением, что последнее вхождение \square с каждой стороны заменено на \diamond . Чтобы обнаружить h , будем рассуждать так:

$$\begin{aligned}
 &\text{reverse} (as \square bs) \diamond cs \\
 &= \{ \text{определение } \diamond \} \\
 &\quad (\text{reverse } as \diamond \text{reverse } bs) \diamond cs \\
 &= \{ \text{так как } \diamond \text{ ассоциативна} \} \\
 &\quad \text{reverse } as \diamond (\text{reverse } bs \diamond cs)
 \end{aligned}$$

Таким образом, можно взять $h \text{ as } bs = \text{reverse as } \diamond \text{bs}$. Применение слияния для свёртки теперь даёт нам:

$$\text{reverse} (\text{boxall} (\text{map } bc \text{ legs})) \diamond cs = \text{foldr } h cs (\text{map } bc \text{ legs})$$

Всё это, правда, сработает, если нам удастся заменить \square на \diamond . По счастью, спасение приходит от свойства (30.1). Положив

$$cs' = \text{if even} (\text{length} (bcode \text{ legs})) \text{ then } cs \text{ else reverse } cs$$

мы можем рассуждать так:

$$\begin{aligned} & bc (\text{Up} (\text{Node } a \text{ legs})) \square cs \\ &= \{ \text{ см. выше} \} \\ & \text{reverse} (\text{boxall} (\text{map } bc \text{ legs})) \square cs \\ &= \{ \text{ используем (30.1)} \} \\ & \text{reverse} (\text{boxall} (\text{map } bc \text{ legs})) \diamond cs' \\ &= \{ \text{ слияние для свёртки} \} \\ & \text{foldr} ((\diamond) \cdot \text{reverse}) cs' (\text{map } bc \text{ legs}) \end{aligned}$$

Преобразовав \square к \diamond , теперь можно выполнить обратное преобразование, снова воспользовавшись слиянием для свёртки. Условие слияния

$$\text{reverse} ((\text{reverse as}) \diamond cs) = as \square (\text{reverse cs})$$

представляет собой просто свойство сопряжённости \square и \diamond , его применение даёт

$$\text{reverse} \cdot \text{foldr} ((\diamond) \cdot \text{reverse}) cs' = \text{foldr} (\square) (\text{reverse cs}')$$

Следовательно:

$$\begin{aligned} & bc (\text{Up} (\text{Node } a \text{ legs})) \square cs \\ &= \{ \text{ см. выше} \} \\ & \text{foldr} ((\diamond) \cdot \text{reverse}) cs' (\text{map } bc \text{ legs}) \\ &= \{ \text{ слияние для свёртки} \} \\ & \text{reverse} (\text{foldr} ((\square) \cdot bc) (\text{reverse cs}')) \text{ legs} \end{aligned}$$

Если обозначить $bop = (\square) \cdot bc$, то мы показали, что $bcode = \text{foldr } bop []$, где

$$\begin{aligned} bop (Up (Node a legs)) cs &= reverse (foldr bop cs' legs) \\ \text{where } cs' &= \begin{cases} \text{if even} (\text{length} (bcode legs)) \\ \quad \text{then reverse } cs \text{ else } cs \end{cases} \end{aligned}$$

Полностью аналогичные рассуждения с $wop = (\diamondsuit) \cdot wc$ приводят к определению $wcode = foldr wop []$, где

$$\begin{aligned} wop (Dn (Node a legs)) cs &= reverse (foldr wop cs' legs) \\ \text{where } cs' &= \begin{cases} \text{if even} (\text{length} (wcode legs)) \\ \quad \text{then reverse } cs \text{ else } cs \end{cases} \end{aligned}$$

Это было довольно утомительно, но пока мы разобрали только два случая, так что работа ещё остаётся. Рассмотрим теперь другой вариант определения bc ($Dn (Node a legs)$) \square cs , начав со следующих рассуждений:

$$\begin{aligned} bc (Dn (Node a legs)) \square cs &= \{ \text{определение } bc \} \\ (wcode legs + [a] + bcode legs) \square cs &= \{ \text{дистрибутивный закон для } + \text{ и } \square \} \\ (wcode legs \square cs) + [a] + (bcode legs \square cs') \end{aligned}$$

где

$$cs' = \text{if even} (\text{length} (wcode legs)) \text{ then reverse } cs \text{ else } cs$$

Дистрибутивный закон для $+$ и \square обсуждался в первой жемчужине о бесцикловых алгоритмах:

$$\begin{aligned} (xs + [y] + ys) \square zs &= (xs \square zs) + [y] + (ys \square zs) \\ \text{where } zs' &= \text{if even} (\text{length } xs) \text{ then reverse } zs \text{ else } zs \end{aligned}$$

Рассмотрим по очереди термы $bcode legs \square cs'$ и $wcode legs \square cs$. Для первого имеем:

$$\begin{aligned} bcode legs \square cs' &= \{ \text{определение } bcode \} \\ foldr (\square) [] (map bc legs) \square cs' &= \{ \text{слияние для свёртки (упражнение)} \} \\ foldr (\square) cs' (map bc legs) &= \{ \text{слияние для map и определение bop} \} \end{aligned}$$

foldr bop cs' legs

Для второго:

$$\begin{aligned}
 & wcode \text{ } legs \square \text{ } cs \\
 = & \{ \text{ используем (30.1) } \} \\
 & wcode \text{ } legs \diamond \text{ } reverse \text{ } cs' \\
 = & \{ \text{ определение } wcode \} \\
 & foldr (\diamond) [] (\text{map } wc \text{ } legs) \diamond \text{ } reverse \text{ } cs' \\
 = & \{ \text{ слияние для свёртки (упражнение) } \} \\
 & foldr (\diamond) (\text{reverse } cs') (\text{map } bc \text{ } legs) \\
 = & \{ \text{ слияние для } map \text{ и определение } wop \} \\
 & foldr wop (\text{reverse } cs') \text{ } legs
 \end{aligned}$$

Следовательно, результат вывода таков:

$$\begin{aligned}
 bop (Dn (Node a legs)) \text{ } cs &= foldr wop (\text{reverse } cs') \text{ } legs ++ [a] ++ \\
 &\quad foldr bop cs' \text{ } legs \\
 \text{where } cs' &= \text{if even (length (wcode legs)) then reverse cs else cs}
 \end{aligned}$$

Аналогичные рассуждения дают похожий результат для двойственного выражения *wop (Up (Node a legs)) cs*, что в целом приводит нас к программе, представленной на рис. 30.5. Она не слишком привлекательна и уж точно мало эффективна, прежде всего из-за повторяющихся вычислений информации о чётности.

Пауки с информацией о чётности

Вместо того, чтобы всякий раз перевычислять информацию о чётности, добавим её в данные паука так, чтобы в каждой вершине имелось два булевых значения:

$$\begin{aligned}
 \text{data } Spider' &= Node' (Bool, Bool) \text{ } Int [Leg'] \\
 \text{data } Leg' &= Dn' Spider' \mid Up' Spider'
 \end{aligned}$$

Для паука *Node' (w, b) a legs* имеется следующий инвариант:

$$\begin{aligned}
 w &= \text{even (length (wcode legs))} \\
 b &= \text{even (length (bcode legs))}
 \end{aligned}$$

```

nnode          :: Nest → [Int]
nnode          = foldr bop [] · map Dn
bop, wop       :: Leg → [Int] → [Int]
bop (Up (Node a legs)) cs = reverse (foldr bop cs' legs)
where cs' = if even (length (foldr bop [] legs)) then reverse cs else cs
bop (Dn (Node a legs)) cs = foldr wop (reverse cs') legs ++ [a] ++
                           foldr bop cs' legs
where cs' = if even (length (foldr wop [] legs)) then reverse cs else cs
wop (Up (Node a legs)) cs = foldr wop cs' legs ++ [a] ++
                           foldr bop (reverse cs') legs
where cs' = if even (length (foldr bop [] legs)) then reverse cs else cs
wop (Dn (Node a legs)) cs = reverse (foldr wop cs' legs)
where cs' = if even (length (foldr wop [] legs)) then reverse cs else cs

```

Рис. 30.5: код после удаления \square и \diamond

где *wcode* и *bcode* возвращают для таких пауков белый и чёрный коды соответственно.

Информация о чётности может быть добавлена к обычному пауку с помощью декорирования следующими функциями:

```

decorate          :: Spider → Spider'
decorate (Node a legs) = node' a (map (mapLeg decorate) legs)
mapLeg f (Up x)    = Up' (f x)
mapLeg f (Dn x)    = Dn' (f x)

```

«Умный» конструктор *node'* определён так:

$$\text{node}' a \text{ legs} = \text{Node}' (\text{foldr } op (\text{True}, \text{True}) \text{ legs}) a \text{ legs}$$

где *op* :: *Leg'* → (*Bool*, *Bool*) → (*Bool*, *Bool*) это

$$\begin{aligned} op (Up' (\text{Node}' (w, b) - -)) (w', b') &= ((w \neq b) \wedge w', b \wedge b') \\ op (Dn' (\text{Node}' (w, b) - -)) (w', b') &= (w \wedge w', (w \neq b) \wedge b') \end{aligned}$$

Для обоснования определения *op*, обозначив предварительно *even · length* через *el*, будем рассуждать так:

$$el (wcode (leg : legs))$$

```

bop, wop :: Leg' → [Int] → Int
bop (Up' (Node' (w, b) a legs)) cs
  = reverse (foldr bop (revif b cs) legs)
bop (Dn' (Node' (w, b) a legs)) cs
  = foldr wop (revif (not w) cs) legs ++ [a] ++ foldr bop (revif w cs) legs
wop (Up' (Node' (w, b) a legs)) cs
  = foldr wop (revif b cs) legs ++ [a] ++ foldr bop (revif (not b) cs) legs
wop (Dn' (Node' (w, b) a legs)) cs
  = reverse (foldr wop (revif w cs) legs)
revif b cs = if b then reverse cs else cs

```

Рис. 30.6: прядение паутины для пауков с информацией о чётности

```

= { определение wcode (для пауков с информацией о чётности) }
el (wc leg ∘ wcode legs)
= { так как el (as ∘ bs) = el as ∧ el bs }
el (wc leg) ∧ el (wcode legs)
= { в предположении, что leg = Up' (Node' a legs') }
el (wcode legs' ++ [a] ++ bcode legs') ∧ el (wcode legs)

```

Но длина $as \text{ ++ } [a] \text{ ++ } bs$ является нечётной в том, и только в том случае, когда чётность длин as и bs противоположна. Похожим образом рассматриваются остальные случаи.

Добавление информации о чётности выполняется за линейное по размеру паука время и ведёт к слегка более простым и гораздо более эффективным определениям bop и wop , данным на рис. 30.6.

Оставшиеся шаги

Последние шаги заключаются в удалении $reverse$ и представлении каждого компонента пары результатов bop и wop префиксным обходом очереди розовых кустов тем же способом, что и в случае с древесными пауками. Чтобы избавиться от $reverse$, мы представим последовательность as парой (as, sa) , где $sa = reverse as$. Конкатенацию пар можно реализовать так:

$$cat a (ws, sw) (bs, sb) = (ws \text{ ++ } [a] \text{ ++ } bs, sb \text{ ++ } [a] \text{ ++ } sw)$$

```

 $\text{ncode} = \text{unfoldr step} \cdot \text{prolog}$ 
 $\text{prolog} = \text{wrapQueue} \cdot \text{fst} \cdot \text{foldr bop} (\text{empty}, \text{empty}) \cdot \text{map} (\text{Dn}' \cdot \text{decorate})$ 
 $\text{bop} (\text{Up}' (\text{Node}' (w, b) a legs)) ps$ 
 $= \text{swap} (\text{foldr bop} (\text{swapif } b ps) legs)$ 
 $\text{bop} (\text{Dn}' (\text{Node}' (w, b) a legs)) ps$ 
 $= \text{cat a} (\text{foldr wop} (\text{swapif } (\text{not } w) ps) legs) (\text{foldr bop} (\text{swapif } w ps) legs)$ 
 $\text{wop} (\text{Up}' (\text{Node}' (w, b) a legs)) ps$ 
 $= \text{cat a} (\text{foldr wop} (\text{swapif } b ps) legs) (\text{foldr bop} (\text{swapif } (\text{not } b) ps) legs)$ 
 $\text{wop} (\text{Dn}' (\text{Node}' (w, b) a legs)) ps$ 
 $= \text{swap} (\text{foldr wop} (\text{swapif } w ps) legs)$ 
 $\text{cat a} (ws, sw) (bs, sb)$ 
 $= (\text{insert ws} (\text{Fork a bs}), \text{insert sb} (\text{Fork a sw}))$ 
 $\text{swap} (xs, ys) = (ys, xs)$ 
 $\text{swapif } b (xs, ys) = \text{if } b \text{ then } (ys, xs) \text{ else } (xs, ys)$ 

```

Рис. 30.7: окончательная бесцикловая программа

Обращение последовательности теперь реализуется просто переменой мест списков в паре. Затем каждый компонент пары следует представить очередью розовых кустов способом, который мы уже наблюдали дважды. Результат всех этих манёвров даёт нам окончательный вид бесцик洛вой программы, приведённый на рис. 30.7.

Хотя пролог и стал теперь пьесой в четырёх действиях, да к тому же в нём задействованы такие персонажи как пауки, списки, очереди и деревья, участвующие в странных деяниях типа перемены мест и свёрток, тем не менее, он выполняется за линейное по размеру гнезда время, и именно поэтому в результате мы получаем бесцикловую программу для прядения паутины.

Начальное состояние

Перед нами остаётся только одна задача. Нужно определить функцию $seed :: Nest \rightarrow [Bit]$, которая возвращает начальную битовую строку $a_1 a_2 \dots a_n$ для паучьего гнезда, помеченного в некотором порядке символами $[1..n]$. Мы дадим лишь наброски рассуждений, приводящих к её определению. Для представления раскрасок нам понадобится библиотека конечных отображений *Data.Map*. Эта библиотека предоставляет тип

Map k a, который реализует конечное отображение из ключей (*k*) в значения (*a*) и включает следующие четыре функции:

```
empty :: Map k a
insert :: Ord k ⇒ k → a → Map k a → Map k a
union :: Ord k ⇒ Map k a → Map k a → Map k a
elems :: Map k a → [a]
```

Значение *empty* обозначает пустое отображение, *insert* добавляет новую пару с ключом и значением, *union* объединяет два отображения, а *elems* возвращает значения в порядке возрастания ключей. Определим состояние паука (тип *State*), задав его отображением из целочисленных меток вершин паука в биты, целые числа, принимающие значения 0 или 1:

```
type State = Map.Map Int Bit
type Bit   = Int
```

Во избежание совпадения имён с функциями для типа *Queue*, называемыми похожим образом, введём следующие определения:

```
install :: Int → Bit → State → State
install = Map.insert
union  :: State → State → State
union  = Map.union
start   :: State
start   = Map.empty
```

Функция *seed* определяется через две другие функции:

$$wseed, bseed : [Leg'] \rightarrow (State, State)$$

Обе функции принимают на вход список ориентированных пауков с информацией о чётности и возвращают пары состояний; *wseed* возвращает начальное состояние, на котором действует код *wcode*, и конечное состояние, к которому он приводит. Аналогично работает функция *bseed*. Нам требуются и начальное, и конечное состояния, а также информация о чётности, поскольку последняя играет роль в определении корректности начального состояния. Определим *seed* так:

$$seed = elems \cdot fst \cdot bseed \cdot map (Dn' \cdot decorate)$$

Эта функция принимает паучье гнездо, преобразует его в список ориентированных к низу пауков с информацией о чётности, вычисляет начальное и конечное состояния, связанные с *bcode*, извлекает первый компонент и возвращает список битов, представляющих собой начальную строку $a_1 a_2 \dots a_n$.

Мы определяем функции *bseed* и *wseed*, следуя программе на рис. 30.4, заменив лишь вычисление переходов на вычисление состояний. Во-первых,

$$\begin{aligned} bseed &= foldr bsp (start, start) \cdot map bs \\ wseed &= foldr wsp (start, start) \cdot map ws \end{aligned}$$

Функция *bs* возвращает начальное и конечное состояния для переходов *bc*, аналогично действует функция *ws*. По факту, *bs* возвращает тройку, первый компонент которой это информация о чётности, необходимая для вычисления *bsp*. Функция

$$foldr bsp (start, start)$$

возвращает начальное и конечное состояния для *boxall*, тогда как функция *foldr wsp (start, start)* вычисляет то же самое для *coxall*.

Вот программа для *bs*:

$$\begin{aligned} bs(Up' (Node' (w, b) a legs)) &= (b, install\ a\ 1\ fs, install\ a\ 1\ is) \\ &\quad \text{where } (is, fs) = bseed\ legs \\ bs(Dn' (Node' (w, b) a legs)) &= (b, install\ a\ 0\ is, install\ a\ 1\ fs) \\ &\quad \text{where } is = fst(wseed\ legs) \\ &\quad fs = snd(bseed\ legs) \end{aligned}$$

Вспоминая, что $bc(Up(Node\ a\ legs)) = reverse(bcode\ legs)$, можно заметить, что начальное и конечное состояния *bseed legs* следует при вычислении *bs* обратить. Более того, так как мы рассматриваем чёрный код, значение, связанное с меткой *a*, равно 1, поэтому к состоянию добавляется именно эта информация. Информация о чётности *b* также возвращается.

Во-вторых, вспомним, что

$$bc(Dn(Node\ a\ legs)) = wcode\ legs \uparrow [a] \uparrow bcode\ legs$$

В этом случае начальное состояние, соответствующее *wcode legs*, и конечное состояние, соответствующее *bcode legs*, являются корректными начальным и конечным состояниями. Более того, метка *a* сначала соответствует нулевому биту, а затем единичному.

Похожим образом определяется *ws*. Остается рассмотреть *bsp* и *wsp*, вот их определения:

$$\begin{aligned}
 seed &= elems \cdot fst \cdot bseed \cdot map (Dn' \cdot decorate) \\
 bseed &= foldr bsp (start, start) \cdot map bs \\
 wseed &= foldr wsp (start, start) \cdot map ws \\
 bs (Up' (Node' (w, b) a legs)) &= (b, install a 1 fs, install a 1 is) \\
 &\quad \text{where } (is, fs) = bseed \text{ legs} \\
 bs (Dn' (Node' (w, b) a legs)) &= (b, install a 0 is, install a 1 fs) \\
 &\quad \text{where } is = fst (wseed \text{ legs}) \\
 &\quad fs = snd (bseed \text{ legs}) \\
 ws (Up' (Node' (w, b) a legs)) &= (w, install a 0 is, install a 1 fs) \\
 &\quad \text{where } is = fst (wseed \text{ legs}) \\
 &\quad fs = snd (bseed \text{ legs}) \\
 ws (Dn' (Node' (w, b) a legs)) &= (w, install a 0 fs, install a 0 is) \\
 &\quad \text{where } (is, fs) = wseed \text{ legs} \\
 bsp (b, ia, fa) (ib, fb) &= (union ia ib, union fa (\text{if } b \text{ then } fb \text{ else } ib)) \\
 wsp (w, ia, fa) (ib, fb) &= (union ia (\text{if } w \text{ then } ib \text{ else } fb), union fa fb)
 \end{aligned}$$
Рис. 30.8: функция *seed*

$$\begin{aligned}
 bsp (b, ia, fa) (ib, fb) &= (union ia ib, union fa (\text{if } b \text{ then } fb \text{ else } ib)) \\
 wsp (w, ia, fa) (ib, fb) &= (union ia (\text{if } w \text{ then } ib \text{ else } fb), union fa fb)
 \end{aligned}$$

Вспомним, что $as \square bs$ начинается с bs и заканчивается bs , если длина as чётная, и $reverse\ bs$ в противном случае. Следовательно, начальное состояние является объединением начальных состояний, связанных с as и bs , однако конечное состояние является объединением начального состояния, связанного с as , и либо конечного, либо начального состояний, связанных с bs , в зависимости от чётности длины as . Булево значение b в определении bsp задаёт чётность. Полный текст функции *seed* приведён на рис. 30.8.

Заключительные замечания

Бесцикловый алгоритм прядения паутины, изложенный в (Knuth and Ruskey, 2003), активно использует сопрограммы (coroutine). Бесцикловая версия Кнута также имеется на его веб-сайте (Knuth, 2001). Упрощённая задача прядения паутины древесными пауками впервые рассматривалась в (Koda and Ruskey, 1993). Бесцикловый алгоритм, основанный на продолжениях (continuation), был опубликован как жемчужина функционального программирования в (Filliâtre and Pottier, 2003).

Литература

- Filliatre, J.-C., and Pottier, F. (2003). Producing all ideals of a forest, functionally. *Journal of Functional Programming* **13** (5), 945–56.
- Knuth, D. E. (2001). SPIDERS: программа, опубликованная по адресу www-cs-faculty.stanford.edu/~knuth/programs.html.
- Knuth, D. E. and Ruskey, F. (2003). Efficient coroutine generation of constrained Gray sequences (aka deconstructing coroutines). *Object-Orientation to Formal Methods: Dedicated to The Memory of Ole-Johan Dahl*. LNCS 2635. Springer-Verlag.
- Koda, Y. and Ruskey, R. (1993). A Gray code for the ideals of a forest poset. *Journal of Algorithms* **15**, 324–40.

Предметный указатель

- ↓ — после, 130, 157
↑ — перед, 136
 \sqsubseteq — префикс, 129, 147, 157
↑ — возведение в степень, 218
 $\setminus\setminus$ — разность списков, 13, 15, 37,
79, 84, 173, 228
 \bowtie — слияние списков, 22, 44
! — индексирование массива, 39, 45,
110, 125
!! — индексирование списка, 110,
117, 125, 141
Data.Array, 13, 39, 44, 125, 142, 152
Data.Array.ST, 15
Data.Map, 321
Data.Sequence, 143
Data.Set, 90
Either, 203
Ix, 14
Queue, 143, 296
QuickCheck, 224, 233
Ratio, 217, 237
accumArray, 13, 14, 105, 152
applyUntil, 104
array, 44, 108
bounds, 39
break, 186, 198, 217
compare, 45
concatMap, 59
elems, 91
foldrn — свёртка непустых списков,
60
fork, 51, 105, 119, 146
inits, 86, 87, 145
listArray, 39, 125
minors, 206
nodups, 181
nub, 84
partition, 16
partitions, 54
reverse, 148, 289
scanl, 146, 282
scanr, 91
sort, 42, 120
sortBy, 45, 119
span, 87
subseqs, 76, 85, 190, 197
tails, 20, 102, 126, 128
transpose, 123, 221, 229
unfoldr, 241, 288
unzip, 52
zip, 52, 106
zipWith, 106
bzip2, 127
EOF — символ конца файла, 241
PSPACE-полнота, 166
S-выражение Маккарти, 262

- абелева группа, 42
абстрагирующая функция, 159, 250,
 267
адаптивное кодирование, 238
аккумулирующий параметр, 161,
 168, 170, 212, 301
алгоритм Барейса, 222
алгоритм Гарсиа—Уочса, 67
алгоритм Гассфильда (*Z*-алгоритм),
 143
алгоритм Кнута—Раски, 307
алгоритм Коды—Раски, 287
алгоритм Махаяна—Виная, 222
алгоритм Ху—Таккера, 67
алгоритм бульдозера, 233
алгоритм планирования, 166, 170
алгоритм прядения паутины Кнута,
 287
алгоритм сверху—вниз, 58
алгоритм снизу—вверх, 58
алгоритм устойчивой сортировки,
 109, 120
амортизированное время, 18, 147,
 163
аннотирование дерева, 204
арифметические выражения, 54,
 189
арифметическое декодирование, 240
асимптотическая сложность, 42
ассоциативный список, 45, 283
безопасная замена, 264
биекция, 159
бинарный поиск, 19, 23, 26, 27,
 30–33, 72
биномиальные деревья, 213
биоинформатика, 99, 113
бустрофедоновое произведение,
 291, 298, 309
быстрая сортировка (*Quicksort*), 17,
 107, 112
верхнетреугольная матрица, 221
вполне обоснованная рекурсия, 16,
 45
вращения списка, 115
- выполнимость булевой формулы,
 187
вычислительная геометрия, 224
главные подматрицы, 221
гладкие алгоритмы, 286
грани, 226
границы списка, 129
декартово произведение, 180
декартово произведение матриц,
 180
декартовы координаты, 172, 187
дерево вызовов, 202
дерево минимальной стоимости, 62
деревья, 160, 199, 294
деревья с помеченными листьями,
 58, 199, 202
диаграмма Юнга, 43
динамическое программирование,
 202
жадные алгоритмы, 59, 66, 68, 170
задача о голосующем большинстве,
 82
задача о максимальной разметке,
 99
задача поддержания порядка, 286
закон *iterate*, 124
закон для *filter*, 146, 183
закон слияния для *foldl*, 96, 159, 232
закон слияния для *foldrn*, 61
закон слияния для *foldr*, 51, 69, 81,
 293, 309, 315
закон слияния для *fork*, 52
закон слияния для *unfoldr*, 246, 251
законы *fork*, 52
инвариантцы циклов, 83, 138
инволюция, 182
индекситит, 182
индуктивный алгоритм, 59, 118, 129
индукция неподвижной точки, 244
инкрементное декодирование, 257
инкрементное кодирование, 243, 249
инкрементный алгоритм, 224, 228,
 242

- интеллектуальный анализ данных, 99
- итеративный алгоритм, 23, 104, 136, 141
- клика, 75
- клика знаменитостей, 75
- кодирование Хаффмана, 115, 236, 240
- кодирование Шеннона—Фано, 236
- кодирование движением к началу, 115
- кодирование длинами серий, 115
- комбинаторные объекты, 287
- компромисс между памятью и временем, 189
- конечный автомат, 95, 167
- крона дерева, 58
- левая обратная, 159
- левые оси, 60, 63, 212
- лексикографический порядок, 63, 71, 86, 128, 130
- лемма о просмотре, 146, 154
- ленивые вычисления, 50, 178, 221, 288
- леса, 59, 209
- линейный порядок, 61
- максимальная сумма не-сегмента, 94
- максимальная сумма сегмента, 94
- матрицы, 179, 216
- мемоизация, 196
- метод Гаусса, 215
- минимальный элемент, 71
- миноры, 216
- монады, 15, 143, 187
- мультимножества, 40, 68, 69
- наибольшая убывающая подпоследовательность, 73
- наибольшее число превосходства, 19
- наименьшая возрастающая распутька, 69
- наименьший элемент, 71
- недетерминированные функции, 60, 70
- неизменяемые массивы, 39
- нижние границы, 29, 42, 43, 85
- нормальная форма, 194
- обращение потоков, 254
- обращение функции, 24, 117
- обход графа, 213, 267
- окна текста, 148
- онлайновая разметка списка, 285
- операция изменения массива, 15, 18, 211
- оптимальная расстановка скобок, 211
- оптимизационные задачи, 67, 211
- остовное дерево, 213
- отношения, 66, 201
- очереди, 135, 167, 294, 296
- пареметричность, 82
- перестановки, 103, 113, 115, 120, 121, 215, 226, 287, 298
- переходы, 287
- подпоследовательности, 68, 84, 95, 196, 212, 287
- поиск в глубину, 168, 262, 267
- поиск в ширину, 166–168, 173, 213
- поиск с ограничениями, 187
- полный перебор, 24, 56, 76, 179, 189
- поразрядная сортировка, 120, 127
- порядок кодов Грея, 306
- потоковая обработка, 242
- правило кортежей для *foldl*, 146, 154
- правило кортежей для *foldr*, 292
- правило округления, 255
- правые оси, 212
- представляющая функция, 159, 250
- преобразование программ, 262
- префикс, 86
- префиксные деревья, 197
- префиксный обход, 290, 320
- проверка на моделях, 187
- продолжения, 324
- промежуточная структура данных, 202

- равенство Сильвестра, 222
равенство Чио, 218
развёртки, 202
разделяемые узлы, 202, 207
разделяй и властуй, 12, 15, 17, 19,
20, 30, 36–38, 43, 45, 46,
85, 103
разрешение рекурсии, 123
ранжирование списка, 101
распутка, 68
расширение интервала, 249, 250
рациональная арифметика, 215,
224, 236
рациональное деление, 217
регулярная целевая функция, 67
регулярные выражения, 95
рекуррентные соотношения, 28, 29,
46, 111
розовые кусты, 197, 290
самый длинный общий префикс,
131, 139
сбалансированные деревья, 35, 73,
277
 сборка мусора, 200
 связный список, 267
 связующие комбинаторы, 53
 свёртка Вандермонда, 30
 сегменты, 94
 седловой поиск, 27
 сжатие данных, 115, 236
 сильно ациклический орграф, 307
 симплекс, 225
 скалярное произведение, 221
 слияние, 40, 172, 191
 совершенные бинарные деревья, 206
 сопоставление строк, 139, 145, 156
 сопрограммы, 324
 сопряжение, 312
 сортировка, 22, 23, 29, 115, 181
 сортировка разбиениями, 107, 111
 сортировка слиянием, 44, 112, 206
 сортировка сравнениями, 23, 29, 42
 сортировка чисел, 12, 14
 сортирующая перестановка, 23
 стеки, 168, 263, 267
 строгая композиция, 289
 строковедение, 129
 структурные деревья, 199
 сужение, 237
 суммирующая функция, 14
 суффиксное дерево, 126
 суффиксы, 101, 125
 таблица инверсий, 23
 теорема Дилвортса, 73
 теорема о потоковой обработке, 243
 теорема об извлечении из потока,
 254
 умные конструкторы, 66, 204, 211
 упорядочивание префиксов, 131
 усечение, 202
 условие монотонности, 67, 71
 уточнение, 61, 63, 70, 103
 уточнение данных, 17, 66, 135, 142,
 159, 250
 фиктивные значения, 27, 99
 формула Лейбница, 215
 фронт, 167
 функция обработки ошибок, 163
 целевая функция, 59, 63, 71
 целочисленная арифметика, 218,
 236, 247
 целочисленное деление, 218
 цельнозерновое программирование,
 182
 цикл while, 138, 141
 циклические структуры, 163, 214
 частичное выполнение, 164
 частичный порядок, 71
 частичный предпорядок, 71

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» по электронному адресу orders@aliants-kniga.ru.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон.

Эти книги вы можете заказать и в Internet-магазине: www.dmk-press.ru.

Оптовые закупки: электронный адрес books@aliants-kniga.ru.

Ричард Бёрд

Жемчужины проектирования алгоритмов: функциональный подход

Главный редактор *Мовчан Д.А.*
dm@dmk-press.ru

Перевод с английского *Брагилевский В. Н.,*
Пеленицын А. М.

Вёрстка *Брагилевский В. Н.,*
Пеленицын А. М.

Корректор *Синяева Г. И.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 23.10.2012. Формат 60×90 1/16.
Вёрстка выполнена средствами *lhs2TeX 1.17* и *T_EXLive 2012*.

Гарнитура «Computer Modern». Печать офсетная.
Усл. печ. л. 20,5. Тираж 200 экз.

Издательство ДМК Пресс
Электронный адрес издательства: www.dmk-press.ru

Функциональное программирование

Жемчужины проектирования алгоритмов

В этой книге Ричард Бёрд представляет принципиально новый подход к проектированию алгоритмов, а именно проектирование посредством формального вывода. Основное содержание книги разделено на 30 коротких глав, называемых жемчужинами, в каждой из которых решается конкретная программистская задача. Эти задачи, некоторые из которых абсолютно новые, происходят из таких разнообразных источников, как игры и головоломки, захватывающие комбинаторные построения и более традиционные алгоритмы сжатия данных и сопоставления строк.

Каждая жемчужина начинается с постановки задачи, формулируемой на функциональном языке программирования Haskell, чрезвычайно мощном и в то же время лаконичном, позволяющем легко и просто выражать алгоритмические идеи. Новшество книги состоит в том, что каждое решение формально вычисляется из исходной постановки задачи посредством обращения к законам функционального программирования.

Издание предназначено для программистов, увлекающихся функциональным программированием, студентов, аспирантов и преподавателей, интересующихся принципами проектирования алгоритмов, а также всех, кто желает приобрести и развить навыки рассуждений в эквациональном стиле применительно к программам и алгоритмам.

Ричард Бёрд является профессором информатики Оксфордского университета и членом совета Линкольн-колледжа в Оксфорде.

Internet-магазин:

www.dmk-press.ru

Книга – почтой:

e-mail: orders@aliens-kniga.ru

Оптовая продажа:

«Альянс-книга»

Тел./факс: (499) 725-5409

e-mail: books@aliens-kniga.ru



ISBN 978-5-94074-867-0



9 785940 748670 >