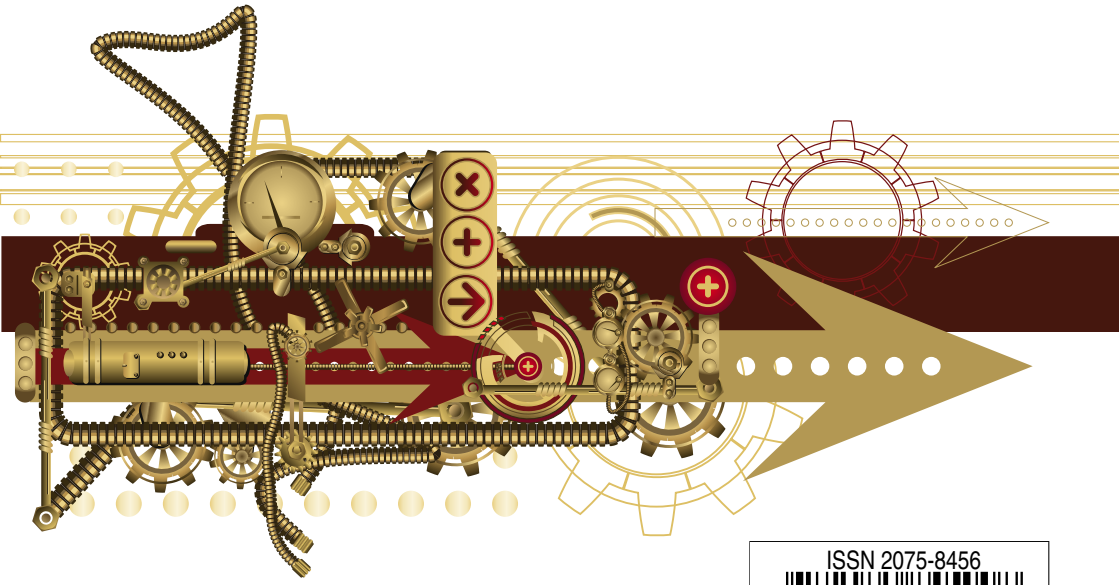
Практика
функционального программирования

Выпуск 3
Декабрь 2009



ISSN 2075-8456

Последняя ревизия этого выпуска журнала, а также другие выпуски могут быть загружены с сайта fprog.ru.

Журнал «Практика функционального программирования»

Авторы статей: Алексей Щепин
Дмитрий Астапов
Евгений Кирпичёв
Лев Валкин
Роман Душкин

Выпускающий редактор: Дмитрий Астапов

Редактор: Лев Валкин

Корректор: Алексей Махоткин

Иллюстрации: **Обложка**
© iStockPhoto/Oleksandr Bondarenko

Шрифты: **Текст**
Minion Pro © Adobe Systems Inc.
Обложка
Days © Александр Калачёв, Алексей Маслов
Cuprum © Jovanny Lemonad
Иллюстрации
GOST type A, GOST type B © ЗАО «АСКОН», используются с разрешения правообладателя

Ревизия: 2666 (2010-08-21)

Сайт журнала: http://fprog.ru/

Свидетельство о регистрации СМИ
Эл № ФС77–37373 от 03 сентября 2009 г.

Оглавление

От редактора

Конкурс!

«Практика функционального программирования» создавался и зарегистрирован как реферируемый научно-практический журнал. Научность журнала предполагает наш интерес к изучению всего, что связано с программированием. Практическая же направленность журнала позволяет производить эксперименты, в том числе — с довольно спорным материалом.

Издревле люди пытались сравнивать языки программирования: что же лучше — Basic или Pascal? Perl или Python? OCaml или Haskell? Обычно вопросы такого масштаба, заданные в курилке, FIDO или на просторах какой-нибудь новомодной социальной сети, ведут к раздору, взаимным обвинениям, обидам, и очень редко — к крупицам объективной истины. Обычно в подобных спорах эмоции бьют через край, а с аргументами дело обстоит из рук вон плохо: фактических данных, позволяющих более или менее объективно сравнивать разные языки программирования, ничтожно мало.

Чтобы исправить эту плачевную ситуацию, редакция журнала предлагает вам попробовать свои силы в решении двух достаточно простых практических задач и прислать в редакцию получившиеся программы. Все присланные решения будут оценены компетентным жюри, которое выберет три самых лучших решения для каждой задачи. Победители получат денежные и поощрительные призы, а все прочие участники соревнования и читатели журнала получат доступ к накопленному статистическому материалу и текстам всех решений.

Вы только что изучили Nemerle и мечтаете зарабатывать с его помощью деньги? Побеждайте, и получайте денежный приз.

Вы жаждете личной славы? Победа в конкурсе — прекрасный способ заявить о себе со страниц нашего журнала.

Вы давно искали повод показать, что никакие Erlang и Haskell в подметки не годятся старому доброму C++? Побеждайте — и у вас появится аргумент для будущих споров, способный сразить любого.

Подробнее о целях проведения конкурса

Мы предлагаем нашим читателям вместе с нами сравнить подходы к решению задач в рамках разных парадигм программирования и соответствующих им инструментальных средств.

Как же будут сравниваться подходы и инструментальные средства? С помощью объективного и субъективного оценивания решений нескольких практических задач.

В качестве объективных показателей мы хотим использовать не только измеримые параметры самого решения (количество строк кода, скорость работы и так далее), но и сведения о производственном процессе (например, время, затраченное вами на разработку и на тестирование).

Кроме того, хотелось бы оценить целый ряд сложно поддающихся формализации свойств: красоту и лаконичность кода, его читаемость, легкость в поддержке и сопровождении. Вместо введения каких-то формальных критериев мы хотим поручить оценку этих параметров компетентному жюри.

Естественно, мы понимаем, что нет двух одинаковых программистов, и персональные качества участников будут скрытым фактором, оказывающим решающее влияние на результат. Однако мы считаем, что невозможность измерить что-то в точности — это совсем не повод отказаться от измерений вообще.

Может быть, в результате станет очевидным преимущество функционального подхода. Может быть — императивного. Мы уверены, что это — не главное. Мы рассчитываем в первую очередь на то, что из накопленных данных можно будет извлечь интересные закономерности (например, тенденцию к потреблению памяти разными языками, скорость работы, etc), и опубликуем подробный анализ в ближайшем номере.

Мы хотим построить эксперимент так, чтобы он был лишен существенных недостатков, присущих другим подобным исследованиям (см. исторический обзор ниже): задачи будут «практические», а не «академические», они будут подобраны так, чтобы не давать очевидного преимущества той или иной парадигме программирования. Для этого мы собрали около тридцати заданий разных уровней сложности и попросили 18 программистов-профессионалов (в *разных* парадигмах программирования) ранжировать задачи по интересности и адекватности в качестве конкурсных задач. Уверены, что использование такого «разношёрстного» жюри для оценки задач — лучшее, что мы можем сделать для того, чтобы исключить «уклон» в сторону конкретного языка или парадигмы. Для того чтобы исключить влияние случайных факторов на выбор членов жюри, мы взяли для конкурса не одну, а две самых популярных задачи.

Подавляющее большинство существующих исследований подобного рода проведены в англоязычных странах Мы хотим, чтобы этот эксперимент позволил нам с вами получить сведения о навыках и подходах уникальной аудитории — русскоязычных программистов-практиков начала двадцать первого века.

Чтобы позволить другим исследователям сделать самостоятельные выводы, мы

опубликуем все корректные варианты решений и весь массив накопленных статистических данных.

Условия проведения конкурса

Конкурс стартует в момент выхода в свет этого (третьего) номера журнала и заканчивается 20 февраля 2010 года. Результаты исследования будут опубликованы в пятом номере. Любые изменения в графике будут загодя опубликованы на официальной странице конкурса на сайте журнала. Обсуждение конкурса также ведется по адресу http://community.livejournal.com/fprog/5508.html.

В конкурсе могут участвовать все желающие.

Как будут оцениваться решения? Даже с учётом предположения, что присылать вы будете только работающие программы, встаёт проблема оценки решений. Что лучше: короткий код, работающий медленно, или грузный, но работающий мгновенно? А может быть, лучше всего тот код, который структурирован для расширяемости? Конечно, все эти метрики мы попробуем проанализировать интегрально, для разных задач и для разных языков. Но задачу определения победителя это не облегчит! Мы думаем, что заставив членов разношёрстного жюри ранжировать присланные варианты по «качеству решения», а затем скомбинировав их ответы, мы сможем на основании субъективных метрик получить лучшее приближение к объективности, на которое мы только можем рассчитывать в рамках подобного конкурса.

Учтите, что жюри будет интегрально оценивать элегантность, профессиональность и читаемость решения, а скорость исполнения и использование памяти — в разумных пределах игнорировать.

Победители конкурса получат специальное упоминание на страницах журнала, а также ценные призы:

1) За первое место, по результатам голосования жюри, для каждой задачи — 8192 рубля;

2) За второе место, для каждой задачи — 4096 рублей;

3) За третье место, для каждой задачи — упоминание на страницах журнала.

Кроме того, все победители получат по экземпляру книг «Практика работы на языке Haskell» и «Справочник по языку Haskell» Р. Душкина с дарственной подписью автора.

Требования к оформлению решений

Представьте, что условие задачи — это сформулированные заказчиком требования к проекту. Или даже к первой фазе проекта — то есть, в дальнейшем ожидается развитие и модификация кода.

Заказчик ожидает получить от вас не только готовое решение, но и **исходные тексты** программы, которые он будет тщательно просматривать. Соответственно, от вас ожидается профессиональное исполнение и промышленное качество кода. В частности, заказчик наверняка будет ожидать:

- Короткое, элегантное и читаемое решение,
- Устойчивое к ошибкам,
- Содержащее тесты или тестовые примеры,
- С поясняющими комментариями в ключевых местах.

Пишите на любом языке, который вам нравится, если для него есть свободно доступный компилятор или интерпретатор для Linux или Windows (если язык очень уж экзотический, не забудьте указать адрес, с которого этот самый компилятор можно скачать). При желании можно комбинировать в решении несколько языков одновременно.

Вы можете использовать сторонние свободно доступные библиотеки для всех вспомогательных задач в рамках вашего решения. Не используйте сторонние библиотеки для реализации основных алгоритмов задачи. Если ваше решение включает в себя свободно доступный код других авторов — укажите это.

Сделайте акцент на корректности генерируемых вашим решением выходных файлов. Вы можете рассчитывать на то, что входные данные будут в точности соответствовать спецификации, но вы должны быть уверены, что в рамках спецификации вы корректно обрабатываете все возможные варианты.

Не занимайтесь избыточной оптимизацией — присылайте нам первое работающее решение. Если до окончания конкурса вы решите доработать свою программу — присылайте нам новый вариант, указав в описании время, затраченное на оптимизацию.

По возможности, **работайте сами** — результаты коллективной работы будут вносить искажения в наши измерения.

Фиксируйте время, ушедшее на разработку. Если это возможно — фиксируйте отдельно время, ушедшее на разработку, и время, потребовавшееся на «наведение красоты» (написание документации, тестов и так далее). Кроме того, постарайтесь зафиксировать, сколько всего «грязного» (календарного) времени у вас ушло на решение. Не занижайте ваши оценки — время решения не будет учитываться при выборе победителей.

Присылайте ваши решения на адрес `contest2009@fprog.ru` в архиве любого распространённого формата (zip, rar, tar.gz, ...), каждую задачу — в отдельном файле. Чтобы облегчить нам обработку результатов, назовите файл так: `<код задачи>-<ваш nick>-<номер варианта решения>.<suffix>`. Используйте номер варианта для того, чтобы указать, какое решение является более новым. В корне архива должна находится директория с именем `<код задачи>-<ваш nick>-<номер`

`варианта решения>`, и все относящиеся к решению файлы должны содержаться в ней.

Разместите в архиве файл README, описывающий решение в формате:

```
Name: <ваше имя или ник>
Task: <код задачи, см. ниже>
Level: <уровень решения, см. ниже>
Language: <название языка>
Work: <чистое время, затраченное на решение, в часах>
Duration: <грязное время, затраченное на решение, в днях>
<пустая строка>
<многострочный комментарий произвольной формы>
```

Пример архива с оформленным решением можно скачать с сайта журнала.

Условия задач

В рамках каждой задачи существуют градации сложности, называемые уровнями. Только от вас зависит, сколько задач и в каком объёме вы будете решать. Мы постарались сделать, чтобы решение обеих задач в минимальном объёме («уровень 1») заняло у вас суммарно не более 4-10 часов.

Задача 1: усечение карты

Код задачи: geo.

Краткое описание: необходимо написать утилиту, которая будет извлекать из указанной карты в формате OpenStreetMap только элементы, находящиеся внутри заданного полигона обрезки.

Уточнения и дополнения к условию (если они понадобятся), а также полный набор исходных данных будут опубликованы на сайте журнала.

Развёрнутое описание: утилита принимает три аргумента командной строки: имя xml-файла с векторной картой в формате OSM, имя файла с координатами вершин полигона обрезки и имя выходного файла в формате OSM.

Базовым элементом карты является узел (node) — это точка с указанными координатами. Узлы входят в состав путей (ways). Кроме того, в состав карты могут входить группирующие объекты, называемые отношениями (relations). В состав отношения могу входить любые другие объекты карты: пути, узлы, другие отношения. В том числе, отношения могут быть пустыми. Более подробное описание типов данных можно найти на wiki проекта OpenStreetMap.

Полигон обрезки состоит из нескольких произвольных (не обязательно выпуклых) многоугольников, которые, в частности, могут содержать «дырки» произвольной формы и иметь самопересечения.

Необходимо извлечь из указанной карты и сохранить в выходной файл только те элементы карты, которые лежат внутри полигона обрезки.

Для обработки объектов, которые попали внутрь полигона обрезки частично, необходимо предусмотреть в программе опцию с семантикой «включать частично обрезанный объект в результат целиком», далее называемую `completeObjects`.

Если эта опция отключена, то все узлы, не попавшие в полигон обрезки, удаляются из всех путей и отношений, которые на них ссылаются. Пустые пути и отношения тоже удаляются.

Если эта опция включена, то обработка путей и отношений происходит по следующим правилам: если внутрь полигона обрезки попал хотя бы один узел пути, весь путь целиком (все входящие в него узлы) должен быть включен в результат. Также в результат должны попасть все отношения, в которые входят какие-либо узлы, пути и отношения, попавшие в результат (рекурсивно). Однако пути, узлы и отношения, являющиеся членами каких-либо отношений, попавших в результат, но при этом лежащие целиком за пределами полигона обрезки, в результат не попадают.

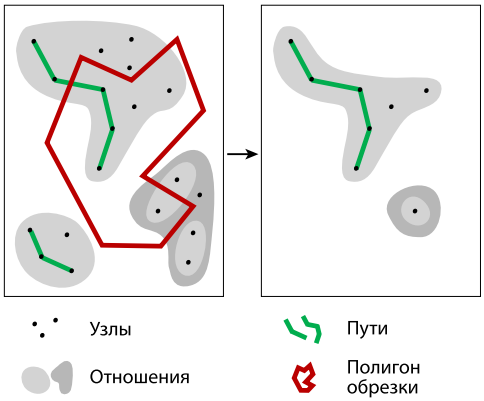


Рис. 1. Пример фильтрации с опцией `completeObjects`

Карта имеет вид XML-документа, все теги второго уровня в котором — это либо узел, либо путь, либо отношение. В теги узлов, путей и отношений могут быть заключены другие теги, которые должны быть перенесены в выходной файл без изменений.

Уровень 1: программа может выполнить обрезку по произвольному выпуклому многоугольнику, без реализации опции `completeObjects`. Пример восьмиугольного полигона, ограничивающего город Киев, находится в файле octagon.poly.

Уровень 2: программа может выполнить обрезку по произвольному многоугольнику, который может быть невыпуклым, состоять из нескольких частей и содержать «дыры», без реализации опции `completeObjects`. Пример прямоугольника, ограничивающего город Киев, и имеющего прямоугольную дыру в центре находится в файле hollow_rectangle.poly.

Уровень 3: программа может выполнить обрезку по произвольному многоуголь-

нику, который может быть невыпуклым, состоять из нескольких частей и содержать «дыры», с поддержкой опции `completeObjects`.

Уровень 4: программа способна за вменяемое время вырезать фрагмент по контуру произвольной административно-территориальной единицы из полной карты России с включенной опцией `completeObjects`. Приблизительное определение понятия «вменяемое время» таково: «порядка получаса для извлечения Московской области».

Карты всех стран мира в формате OSM можно найти на сайте CloudMade. Файлы с картами имеют суффикс `.osm.bz2`, там же находятся и полигоны обрезки, использованные для извлечения этих карт из карты мира (файлы с суффиксом `.poly`).

Для проверки своей программы можете сравнивать ее вывод с результатом работы утилиты Osmosis на небольших картах и полигонах обрезки. Чтобы Osmosis генерировал результаты, соответствующие опции `completeObjects`, необходимо запускать его так (команда разделена на несколько строк для лучшей читаемости):

```
osmosis --read-xml file=russian_federation.osm
        --bounding-polygon file=mosobl.poly
                           completeWays=true
                           completeRelations=true
                           idTrackerType=BitSet
        --write-xml file=moscow.osm
```

Задача 2: составление плана-графика

Код задачи: gantt.

Краткое описание: Построение расписания и генерация диаграммы Гантта на основе электронной таблицы со списком работ.

Развёрнутое описание

Дана электронная таблица, в которой содержится информация о списке работ. Программа должна проверить имеющуюся информацию на непротиворечивость и заполнить недостающие графы: расставить исполнителей, назначить даты начала и окончания работ и так далее.

В случае, если это сделать невозможно, программа должна объяснить причины в виде, понятном человеку.

Уточнения и дополнения к условию (если они понадобятся), а также полный набор исходных данных будут опубликованы на сайте журнала.

Формат файла

Пример электронной таблицы с исходными данными можно найти в файле source.csv.

Как видно, лист с исходными данными состоит из блоков, идущих в произвольном порядке. Начало каждого блока отмечено пустой строкой, следом за которой в первой колонке указано одно из ключевых слов, обозначающих тип блока (см. ниже).

В таблице может быть несколько блоков с одинаковыми именами, в таком случае их содержимое объединяется.

Все прочие строки можно игнорировать.

Формат блоков

Блок «маркер формата»: обозначается ключевым словом `Plan Format`. Состоит из одной строки: в колонке `B` указан номер версии формата файла: `1.0`.

Блок «даты»: обозначается ключевым словом `Dates`. Состоит из одной строки. В колонке `B` указана дата начала проекта, в колонке `C` — ориентировочная дата окончания проекта, в колонке `D` — дата последней модификации данных, которая при планировании задач трактуется как «текущая дата». Далее эта дата будет обозначаться как «время T».

Блок «задачи»: обозначается ключевым словом `Tasks`. Состоит из нескольких строк, в которых колонки имеют следующий смысл:

- `A` — идентификатор задачи (необязательное поле)
- `B` — текстовое описание задачи (обязательное поле)
- `C` — первоначальная оценка трудоемкости (в человеко-часах, обязательное поле)
- `D` — уточненная оценка трудоемкости. Обычно уточненная оценка трудоемкости появляется в процессе выполнения задачи, когда оказывается, что первоначальная оценка (из колонки `C`) была неточной. При наличии уточненной оценок необходимо перепланировать задачи с их учетом. Подробнее смотри ниже в описании алгоритма планирования (необязательное поле)
- `E` — сколько времени уже потрачено на эту задачу (необязательное поле)
- `F` — время, оставшееся на задачу (необязательное поле)
- `G` — идентификатор ресурса или исполнителя (необязательное поле)
- `H` — время начала исполнения задачи (необязательное поле)
- `I` — время окончания исполнения задачи (необязательное поле)

Блок «ресурсы»: обозначается ключевым словом `Resources`. Состоит из нескольких строк, в каждой из которых в колонке `A` указан идентификатор ресурса или исполнителя. В рамках задачи считается, что «ресурс» — это станок, человек или робот, способный выполнять любую задачу в течение 8 часов в день, 5 дней в неделю, то есть все ресурсы взаимозаменяемы.

Блок «ограничения»: обозначается ключевым словом `Constraints`. Состоит из нескольких строк, в каждой из которых в колонке `A` находится формула, задающая определенные ограничения на изменения сведений о задачах. Допустимы такие формулы:

- `A<<B` — окончание задачи A должно произойти раньше начала задачи B (тут и далее `A` и `B` — идентификаторы задач).

- `A<!B` — окончание задачи A должен произойти раньше окончания задачи B.
- `A>>B` — то же, что и `B<<A`
- `A!>B` — то же, что и `B<!A`
- `fixt(`A_1`, …,` A_n`)` — в задачах A_i нельзя перепланировать время, даты начала и окончания работ, указанные в таблице, надо воспринимать как данность.
- `fixr(`A_1`, …,` A_n`)` — в задачах A_i нельзя менять назначенные ресурсы, идентификатор ресурса надо воспринимать как данность.
- `fixrt(`A_1`, …,` A_n`)` — в задачах A_i нельзя изменять ни ресурсы, ни сроки.
- `hl(`D_1`, …,` D_n`)` — дни D_i считаются нерабочими

Планирование

Процесс планирования заключается в том, чтобы по исходному файлу с данными построить выходной файл в том же формате так, чтобы для всех задач были проставлены исполнители, даты начала и окончания работы, были соблюдены все ограничения, и все работы были завершены до даты окончания проекта. В качестве примера результата работы смотрите файл output.csv.

В ходе планирования можно выполнять такие преобразования задач:

- назначить задаче исполнителя (у каждой задачи может быть только один исполнитель);
- изменить исполнителя задачи, если для нее нет ограничений типа `fixr` или `fixrt`;
- назначить время начала исполнения задачи;
- сдвинуть время исполнения задачи, если для нее нет ограничений типа `fixt` или `fixrt`.

Не разрешается разбиение задач на подзадачи или же планирование исполнения задачи в течение нескольких разнесенных по времени интервалов.

Не разрешается назначение времени начала исполнения задачи раньше текущей даты (времени T). Не разрешается изменение времени начала исполнения, если оно находится «в прошлом» (ранее времени T). Если дата начала и окончания исполнения задачи находится в прошлом, то такую задачу вообще нельзя изменять.

Не разрешается использовать любой ресурс более чем 8 часов в день.

Уточненные оценки трудоемкости задачи, если он есть, должны использоваться вместо первоначальной оценки трудоемкости задачи. Пример использования уточненных оценок см. в файле corrected.csv.

Указанные пользователем даты и исполнители при отсутствии ограничений `fixr`, `fixt` и `fixrt` считаются «пожеланиями» и могут быть свободно изменены программой при планировании.

Полученный план работ должен иметь как можно более короткую суммарную длительность (время от начала исполнения самой ранней задачи до времени завершения самой поздней).

Уровень 1: программа способна составить план работ без учета ограничений.

Уровень 2: программа способна составить план работ с учетом ограничений.

Уровень 3: программа, дополнительно, способна считывать исходные данные из файлов формата XLS и/или ODS и записывать результат в файлы такого формата. Программа способна обрабатывать электронные таблицы, находящиеся на сервере Google Docs, обновляя данные в них или создавая там новые документы.

Уровень 4: Программа сохраняет в отдельный файл графическое изображение план-графика в виде диаграммы Гантта и, по желанию пользователя, может генерировать план, нагружающий ресурсы по возможности в одинаковой степени.

Экскурс в историю

Мы, конечно же, не первые, кто пытается сравнивать языки программирования.

В этой спорной и потенциально «взрывоопасной» области существует известное количество правильно поставленных экспериментов, которые на статистически значимом материале уверенно демонстрируют преимущество тех или иных инструментов, подходов, или языков программирования в тех или иных отношениях.

Одной из первых работ, поставившей своей целью сравнить сразу несколько принципиально различных языков, можно считать [4]. В этом классическом исследовании приведены данные, из которых следует, что скриптовые и функциональные языки имеют где-то 2-3 кратный выигрыш во времени программирования и объеме кода по сравнению с программами на C++ и Ada. С другой стороны, программы на C++ и Ada оказываются в 2-3 раза быстрее программ на других языках программирования. Впрочем, авторы справедливо посчитали, что имевшиеся в их распоряжении данные не составляли репрезентативной выборки (мало участников исследования, большая разница в степени владения языками) и ограничились в своих выводах осторожными общими фразами.

Шесть лет спустя в исследовании Лутца Прехельта ([7]) было рассмотрено семь языков (C, C++, Java, Perl, Python, Rexx и Tcl), которые использовались для написания простой программы, преобразующей номер телефона в набор слов по определенным правилам. В исследовании принимали участие добровольцы, всего было накоплено около 80 вариантов решений. В результате автор пришел, в частности, к таким выводам:

- Разработка и написание программ на Perl, Python, Rexx или Tcl требует примерно в два раза меньше времени, чем написание аналогичного кода на C, C++ или Java. Получающийся в результате программный код также вдвое короче.

- Не наблюдается существенной зависимости между выбранным языком и надежностью программы.
- Программы на скриптовых языках потребляют примерно в два раза больше памяти, чем программы на C/C++. Программы на Java потребляют в два раза больше памяти, чем программы на скриптовых языках[1]
- В группе скриптовых языков Python и Perl оказались быстрее, чем Rexx и Tcl.

В то же время, многие попытки подобных сравнений дают в результате весьма однобокий взгляд на проблему. Типичными недостатками в этом случае являются:

- сравнение всего двух-трех языков, зачастую — со сходной семантикой или синтаксисом (например, [10], [1], [6], [3], [10]);
- исследование нескольких языков в условиях, когда весь код написан одним и тем же автором, при этом степень его знакомства с языками никак не оценивается (например, [2]);
- явная предрасположенность исследователей в пользу одного из языков или группы языков (например, [8], [4]);
- исследование исключительно производительности конкретных реализаций языков (например, [9], [5], [1], [2]);
- игнорирование временны*х* и прочих аспектов процесса разработки (сколько времени ушло на разработку программы) (например, [5], [8]);
- несоотвествие результатов нынешним реалиям, так как исследование проведено 5-10 лет тому назад (все вышеупомянутые).

Мы решительно настроены не допустить подобных «проколов» и постараемся сделать все возможное, чтобы оценки были сделаны максимально объективно, а результаты анализа не были ангажированы.

Ждём ваших писем, приятного чтения и с наступающим Новым Годом!

Дмитрий Астапов, `adept@fprog.ru`

Литература

[1] Benchmarking Java against C and Fortran for Scientific Applications / J. M. Bull, L. A. Smith, L. Pottage, R. Freeman // In Proceedings of ACM Java Grande/ISCOPE Conference. — ACM Press, 2001. — Pp. 97–105.

[1] Правда, в статье не указаны параметры запуска JVM и методика измерений в случае Java.

[2] *Cowell-Shah C. W.* Nine Language Performance Round-up: Benchmarking Math & File I/O, URL: http://www.osnews.com/story/5602/Nine_Language_Performance_Round-up_Benchmarking_Math_File_I_O (дата обращения: 20 декабря 2009 г.). — 2004.

[3] *Gat E.* Lisp as an Alternative to Java // *Intelligence*. — 2000. — Vol. 11. — P. 2000.

[4] *Hudak P., Jones M. P.* vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity Available from URL: http://www.haskell.org/papers/NSWC/jfp.ps (дата обращения: 20 декабря 2009 г.): Tech. rep.: Yale University, 1994.

[5] *Kernighan B. W., Wyk C. J. V.* Timing trials, or the trials of timing: experiments with scripting and user-interface languages, URL: http://cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html (дата обращения: 20 декабря 2009 г.). — 1998.

[6] *Prechelt L., Informatik F. F.* Comparing Java vs. C/C++ efficiency differences to interpersonal differences. — 1999.

[7] *Prechelt L., Java C. C.* An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. — 2000.

[8] Ray tracer language comparison, URL: http://www.ffconsultancy.com/languages/ray_tracer/ (дата обращения: 20 декабря 2009 г.). — 2005-2007.

[9] The Computer Language Benchmarks Game, URL: http://shootout.alioth.debian.org/ (дата обращения: 20 декабря 2009 г.).

[10] *Zeigler S. F.* Comparing Development Costs of C and Ada, URL: http://www.adaic.com/whyada/ada-vs-c/cada_art.html (дата обращения: 20 декабря 2009 г.). — 1995.

Рекурсия + мемоизация = динамическое программирование

Дмитрий Астапов
adept@fprog.ru

Аннотация

Статья рассказывает о том, как ленивая модель вычислений, принятая в языке Haskell, помогает кратко и эффективно реализовывать алгоритмы с использованием метода динамического программирования.

The article shows how lazy evaluation combined with memoization leads to succinct and efficient implementations of various dynamic programming algorithms

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/4277.html.

1.1. Введение

Сложно найти программу на функциональном языке, которая не использовала бы рекурсивные или взаиморекурсивные вызовы функций. Циклические вычисления (которые в императивном языке требуют управляющих конструкций типа `for`, `while`, `foreach`) записываются либо с помощью функций высших порядков, либо рекурсивно. Сложные структуры данных часто обрабатываются рекурсивно (это называется «структурная рекурсия»). Кроме того, нередко используются и рекурсивно порождаемые структуры данных, классический учебный пример — треугольник Паскаля.

Однако при реализации и использовании рекурсивных алгоритмов легко допустить одну популярную ошибку: одни и те же вычисления могут без необходимости повторяться многократно. Рассмотрим для примера простейшую реализацию на Haskell функции, вычисляющей числа Фибоначчи:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Если мы попробуем вычислить с ее помощью 100 первых чисел Фибоначчи, то обнаружим, что с каждым следующим числом скорость вычисления ощутимо падает. Происходит это потому, что для получения каждого последующего числа выполняется вычисление всех предыдущих чисел, причем многих — по несколько раз, и общая временная сложность алгоритма получается $\Theta(\phi^n)$, где $\phi = \frac{1+\sqrt{5}}{2}$.

Данная статья (с примерами на Haskell) посвящена тому, как эффективно — со сложностью порядка $\Theta(n)$ или $\Theta(n \log n)$ — реализовывать подобные рекурсивные алгоритмы.

В качестве примеров для этой статьи взяты задачи из отборочного раунда ежегодного конкурса программистов Google Code Jam 2009.

Полные исходные тексты программ, описываемых в этой статье, можно найти на сайте журнала.

1.2. Задача 1: Подсчет числа подпоследовательностей

Дана текстовая строка. Необходимо посчитать, сколько раз в ней встречается подпоследовательность символов «welcome to code jam». Для этого необходимо найти в исходной строке одну из букв «w», правее нее — букву «e», и так далее до буквы «m». Говоря более формально, для данной строки $input$ необходимо посчитать, сколькими способами можно выбрать индексы $s_0, s_1, \ldots, s_{18}$ так, что $s_0 < s_1 < \ldots < s_{18}$ и конкатенация $input[s_0], input[s_1], \ldots, input[s_{18}]$ дает строку «welcome to code jam».[1]

[1] При наличии учетной записи в Google с оригинальным авторским условием можно ознакомиться на сайте Code Jam.

В оригинальном решении авторы просят вывести только последние четыре цифры получившегося значения, то есть расчеты можно вести по модулю 10000. В этом есть практический смысл: даже для относительно небольших строк количество вариантов может представлять 20-значное число, что затрудняет чтение и проверку результатов. Примеры, иллюстрирующие статью, также будут использовать модульную арифметику.

Подобные задачи встречаются в реальной жизни: выравнивание последовательностей (англ. sequence alignment) генов в биоинформатике (см. [7], [5, стр. 539]), компьютерный анализ и генерация текстов на естественных языках (см. [8]), метод «критического пути» (англ. critical path method) для оценки длительности проектов.

1.2.1. Наивное решение

Попробуем дать простое словесное описание алгоритма, позволяющего подсчитать, сколько раз строка-шаблон встречается как подпоследовательность в другой строке (области поиска).

Если область поиска пуста, то количество вхождений, очевидно, равно нулю. Если же пуст шаблон, то результат положим равным единице — считаем, что пустой шаблон можно «совместить» с концом строки и таким образом он входит в произвольную строку ровно один раз.

Если же и шаблон, и область поиска не пусты, то необходимо:

- найти первую букву шаблона в области поиска;
- посчитать, сколькими способами можно разместить остаток шаблона правее этой точки;
- прибавить количество способов, которыми можно разместить весь исходный шаблон правее этой точки (то есть, начиная с другой стартовой позиции).

Таким образом, имеем рекурсивное определение, которое можно практически дословно перевести в код на Haskell:

```
occurs :: String → String → Int
occurs []      str = 1
occurs pattern []  = 0
occurs (p:ps) (c:cs)
  | p == c    = ( occurs ps cs +
                  occurs (p:ps) cs ) `mod` 10000
  | otherwise = occurs (p:ps) cs
```

Чтобы получить решение изначальной задачи, необходимо реализовать чтение исходных данных из файла и вывод правильно отформатированных результатов вычислений, но этот код не имеет непосредственного отношения к теме статьи и тут не приведен. Полный текст программы можно найти в файле `naive.hs` в директории `welcome_to_code_jam` архива с исходными текстами.

1.2.2. Проблемы с производительностью

Посмотрим, как будет работать указанная функция при поиске шаблона «jam» в строке «jjaamm». Поскольку первые буквы шаблона и области поиска совпадают, результат будет состоять из суммы двух значений (взятие результата по модулю 10000 для простоты опущено):

```
1 occurs "jam" "jjaamm" =
2   occurs "am"  "jaamm"
3 + occurs "jam" "jaamm" = ...
```

Распишем следующий шаг рекурсии. Чтобы вычислить выражение в строке 2, нужно пропустить следующую букву области поиска, а чтобы вычислить выражение в строке 3, нужно снова вычислить сумму результатов рекурсивных вызовов:

```
1 ... = occurs "am" "aamm"
2   + ( occurs "am" "aamm"
3     + occurs "jam" "aamm" ) = ...
```

Значение в строке 3 будет равно нулю, так как остаток области поиска больше не содержит букв «j». Выражения в строках 1 и 2 полностью совпадают, что в будущем приведет к многократному вычислению одних и тех же значений. Посмотрим, что произойдет на следующем шаге:

```
... =
  ( occurs "m"  "amm"
  + occurs "am" "amm" )
+ ( occurs "m"  "amm"
  + occurs "am" "amm" )
```

Полностью последовательность вызовов для этого примера представлена на рисунке 1.1, при этом там изображены только значения параметров, а имя функции `occurs` опущено. Легко заметить, что по мере продвижения к концу области поиска объем повторяющихся вычислений очень быстро растет (временная сложность реализации $\Theta(C_n^m)$, где n — длина области поиска, а m — длина шаблона).

Таким образом, налицо разбиение исходной задачи на перекрывающиеся подзадачи, причем оптимальное решение задачи размера N можно вычислить на основании решений задач размера $N - 1$.

Если бы подзадачи не имели перекрытий, то использованный подход (разбить задачу на подзадачи, рекурсивно их решить, а решения — объединить) был бы оптимальным по скорости. Такой способ решения называется «разделяй и властвуй» (англ. divide and conquer). Однако в нашем случае подзадачи перекрываются, и это является прямым показанием для применения метода динамического программирования (см. [12], [2], [11]).

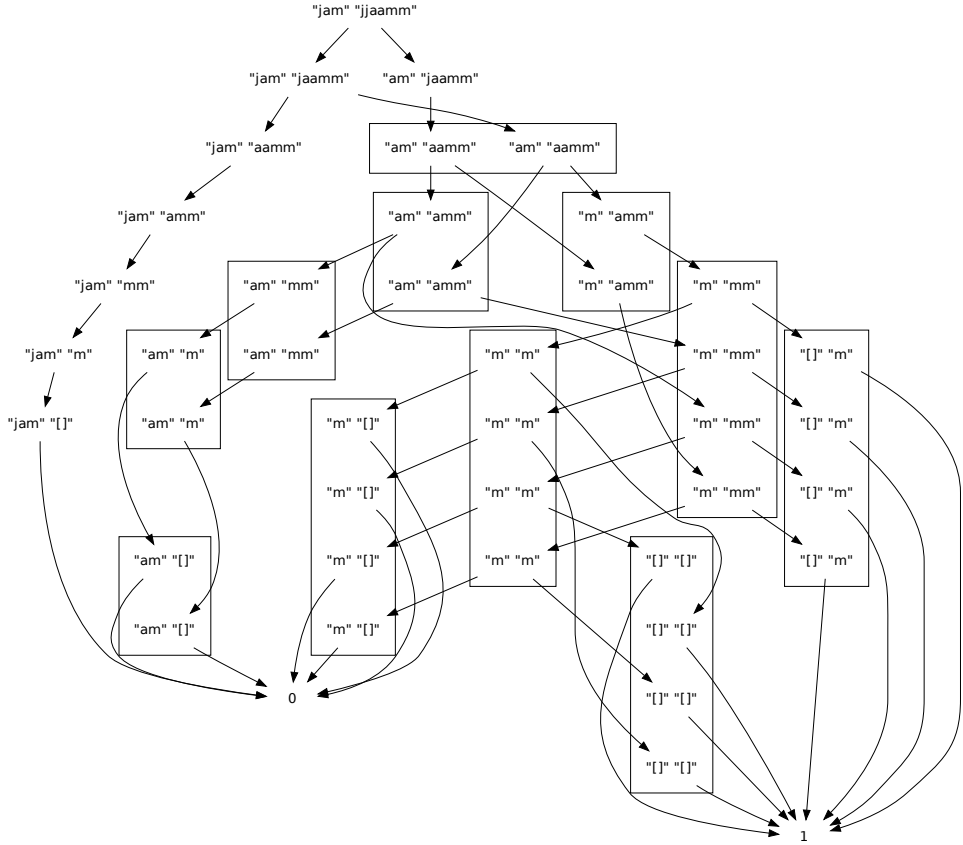


Рис. 1.1. Вычисление `occurs "jam" "jjaamm"`

1.2.3. Решение с использованием динамического программирования

Ключевой идеей динамического программирования является запоминание решений подзадач и дальнейшее их использование без дополнительных повторных вычислений. Применительно к нашей задаче это означает, что нужно запоминать значения `occurs p s` для разных значений `p` и `s`, начиная с самых коротких. Например, для эффективного вычисления `occurs "jam" "jjaamm"` наверняка необходимо запомнить результаты вычисления `occurs "m" "m"`, `occurs "m" "mm"`, `occurs "am" "amm"` и так далее.

При использовании языков с энергичной моделью вычислений (англ. eager evaluation) переход от наивного решения к решению с использованием динамического программирования почти всегда заключается в полном или существенном переписывании кода для того, чтобы он работал «от конца к началу». То есть, сначала вычисляются подзадачи минимального размера и запоминается их результат, а затем запомненные значения используются для вычисления подзадач большего размера, и так далее. В нашей задаче, например, необходимо изменить порядок сканирования шаблона и области поиска так, чтобы он происходил от последних символов к первым. Из-за этого многие начинающие программисты не могут написать решение задачи с использованием динамического программирования «с нуля» — им тяжело переиначить «в уме» интуитивно понятный наивный подход к решению.

Кроме того, далеко не всегда требуется вычислять результаты всех возможных подзадач для того, чтобы найти решение исходной задачи. Однако определение того, какие именно подзадачи можно пропустить, может быть достаточно нетривиально.

Благодаря тому, что в Haskell используется ленивая модель вычислений, можно обойтись без переписывания кода и размышлений о том, что и в каком порядке надо вычислять и запоминать. Решение общей задачи может ссылаться на сохраненные решения более мелких подзадач еще до того, как они фактически будут вычислены.

Вот как это выглядит на практике: предположим, что у нас есть ассоциативный массив `cache`, ставящий в соответствие паре строк `(p, s)` искомое количество вхождений `p` в `s`. Тогда функция `occurs` сведется к тривиальному поиску нужного значения в этом массиве:

```
import qualified Data.Map as Map

cache :: Map.Map (String, String) Int
cache = undefined

occurs pat str =
  fromJust $ Map.lookup (pat,str) cache
```

Как же сформировать массив `cache`? Если взять реализацию примитивного алгоритма из предыдущего раздела (изменив имя функции на `occurs'`), то можно построить `cache` таким образом:

```
cache :: Map.Map (String, String) Int
cache =
  Map.fromList [ ( (p,s), occurs' p s)
               | p ← tails pattern
               , s ← tails str ]
```

В `cache` будут помещены результаты вычисления `occurs'` для всех возможных суффиксов (окончаний) строк `pattern` и `str`. Благодаря тому, что модель вычисления — ленивая, такое объявление помещает в ассоциативный массив только код для отложенного вызова `occurs'`. В англоязычной литературе этот служебный код называют thunk. Когда функция `occurs` обратится за значением конкретного элемента ассоциативного массива, thunk выполнится и будет заменен на результат вычисления. Все последующие обращения к этому элементу массива будут возвращать готовый результат без дополнительных вычислений.

Раз уж `cache` содержит все возможные результаты `occurs'`, можно использовать его для того, чтобы ускорить работу самой функции `occurs'`:

```
occurs' [] str = 1
occurs' _  []  = 0
occurs' (p:ps) (c:cs)
  | p == c    = ( occurs ps cs
                + occurs (p:ps) cs ) `mod` 10000
  | otherwise = occurs (p:ps) cs
```

Таким образом, получаем косвенную рекурсию: функция `occurs` извлекает значения из `cache`, куда они попадают в результате вычисления функции `occurs'`, которая обращается к `occurs`.

Базой рекурсии являются два первых уравнения функции `occurs'` — именно к ним рано или поздно сойдутся все рекурсивные вызовы.

Полностью код, вычисляющий решение поставленной задачи, будет выглядеть так:

```
solve :: String → String
solve str = printf "%04d" $ occurs pattern str
  where
    pattern = "welcome to code jam"

    occurs :: String → String → Int
    occurs pat str =
      fromJust $ Map.lookup (pat,str) cache

    cache :: Map.Map (String, String) Int
    cache =
      Map.fromList [ ( (p,s), occurs' p s)
                   | p ← tails pattern
                   , s ← tails str ]
```

```
occurs' :: String → String → Int
occurs' [] str = 1
occurs' _  []  = 0
occurs' (p:ps) (c:cs)
  | p == c    = ( occurs ps cs
                + occurs (p:ps) cs ) `mod` 10000
  | otherwise = occurs (p:ps) cs
```

Обратите внимание, что объявление `cache` не имеет параметров, а вместо этого обращается к именам `str` и `pattern`, определенным в той же самой области видимости (блоке `where`). Это сделано для того, чтобы вызовы `solve` с любым значением параметра `str` использовали один и тот же `cache` и не создавали его заново каждый раз. Таким образом могут быть ускорены вызовы `solve` для значений `str`, содержащих одинаковые подстроки. По той же причине значение `pattern` не передается параметром в `solve`, а статически определено локально.

Получается, что значение любого элемента ассоциативного массива `cache` — это либо 0, либо 1, либо ссылка на другой элемент массива, либо сумма из каких-то двух других элементов массива. Полная схема связей между элементами `cache` при вычислении `occurs "jam" "jjaamm"` представлена на рисунке 1.2. Вычислительная сложность этой реализации — $\Theta(n^2 \log n)$, так как требуется n операций доступа к `Map`, каждая из которых имеет сложность $\Theta(n \log n)$.

Подобная техника запоминания («кэширования») результатов работы функции имеет название «мемоизация» и используется не только для реализации задач динамического программирования, но и везде, где происходят регулярные вызовы «тяжелых» функций с одними и теми же аргументами (см. [6], [12], [1]).

В качестве классического примера мемоизации можно привести способ быстрого вычисления чисел Фибоначчи, встречающийся во множестве учебных материалов по Haskell:

```
fib n = fiblist !! n
  where
    fiblist = 1:1:(zipWith (+) fiblist (tail fiblist))
```

Полный текст программы можно найти в файле `memoized.hs` в директории `welcome_to_code_jam` архива с исходными текстами.

В качестве самостоятельного упражнения по дальнейшей оптимизации программы читателю предлагается дописать в определение функции `occurs` еще одно уравнение, возвращающее `0` в случае, если первый символ шаблона не входит в область поиска.

1.3. Задача 2: Водосборы

Помимо рекурсивных вычислений (вычисление чисел Фибоначчи, вычисление факториала, решение первой задачи из этой статьи) существует целый класс задач,

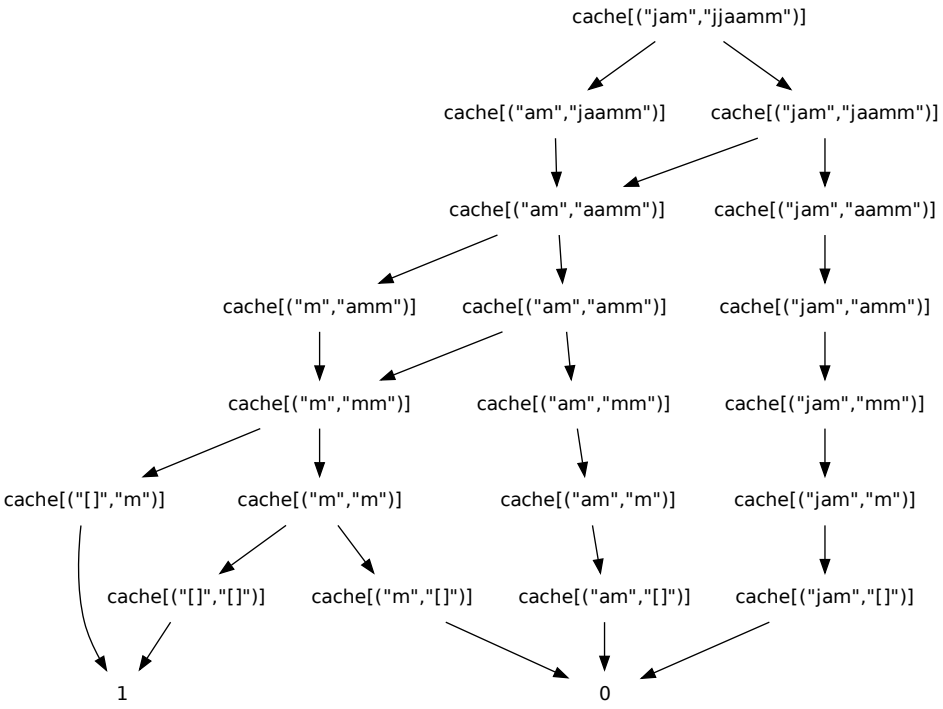


Рис. 1.2. Связи между элементами `cache` при вычислении `occurs "jam" "jjaamm"`

для решения которых требуется рекурсивное создание какой-либо сложной структуры данных. В качестве примера рассмотрим еще одну задачу из отборочного тура Code Jam 2009.

Дана прямоугольная «карта местности», каждая клетка которой имеет определенную высоту, выраженную целым числом. Необходимо разделить эти карты на «бассейны водосбора» по следующим правилам:

- Из каждой клетки карты вода стекает не более чем в одном из четырех возможных направлений («север», «юг», «запад», «восток»).
- Если у клетки нет соседей с высотой, меньшей ее собственной высоты, то эта клетка — **водосток**, и вода из нее никуда дальше не течет.
- Иначе, вода из текущей клетки стекает на соседнюю клетку с минимальной высотой.
- Если таких соседей несколько, вода стекает по первому из возможных направлений из следующего списка: «на север», «на запад», «на восток», «на юг».

Полностью текст условия доступен на сайте Code Jam.

Все клетки, вода из которых стекает в один и тот же водосток, принадлежат одному бассейну водосбора, и обозначаются одной и той же буквой латинского алфавита. Бассейны водосбора отмечаются буквами, начиная с 'a', в порядке, в котором они встречаются при просмотре карты сверху вниз, слева направо. Необходимо превратить карту высот в карту бассейнов водосбора, например:

```
1 2 3 4 5       a a a a a
2 9 3 9 6       a a b b a
3 3 0 8 7  ->   a b b b a
4 9 8 9 8       a b b b a
5 6 7 8 9       a a a a a
```

Подобные задачи встречаются в реальной жизни при программировании роботов (отыскание пути при помощи «волнового алгоритма» или карты потенциалов — см. [9]), компьютерных игр (навигация по карте), отыскании оптимального способа выбрать несколько предметов из большого набора (задачи «упаковки рюкзака» и «набора суммы минимальным количеством банкнот»).

Типичным примером подобной задачи будет отыскание пути на доске с квадратными (шестиугольными) клетками из точки A в точку B с минимальными затратами ресурса R, если для всех возможных переходов между клетками карты известно количество ресурса, которое придется израсходовать.

1.3.1. Наивное решение

Легко видеть, что решение задачи можно разбить на три простых этапа:

- Для каждой клетки карты определить, в какую сторону с нее стекает вода. В процессе идентифицировать водостоки.
- Для всех найденных водостоков определить, какие клетки (прямо или косвенно) доставляют в них воду и сгруппировать клетки по этому признаку.
- Отсортировать группы клеток по координатам самой левой верхней клетки и пометить их буквами.

Поиск клеток, прямо или косвенно доставляющих воду в выбранный водосток, можно производить с помощью одной из разновидностей «волнового алгоритма» (см. [10]). Получается, что для отыскания решения будут выполнены несколько итераций по всем клеткам карты, с просмотром и анализом непосредственных соседей каждой клетки.

На первой итерации будут идентифицированы направления стока воды (обозначенные буквами «n», «w», «e», «s»), а также клетки-водостоки, обозначенные буквой «S»:

```
1 2 3 4 5       S w w w w
2 9 3 9 6       n n s w n
3 3 0 8 7  ->   n e S w n
4 9 8 9 8       n n n n n
5 6 7 8 9       n w w w n
```

На второй — отнесены к тому или иному бассейну водосбора клетки, непосредственно соседствующие с водостоками (цифры обозначают принадлежность к тому или иному бассейну):

```
S w w w w       1 1 w w w
n n s w n       1 n 0 w n
n e S w n  ->   n 0 0 0 n
n n n n n       n n 0 n n
n w w w n       n w w w n
```

На следующей итерации к бассейнам будут присоединены клетки, отстоящие от водостоков на два шага:

```
1 1 w w w       1 1 1 w w
1 n 0 w n       1 1 0 0 n
n 0 0 0 n  ->   1 0 0 0 n
n n 0 n n       n 0 0 0 n
n w w w n       n w w w n
```

И так далее, пока не будут классифицированы все имеющиеся клетки:

```
1 1 1 w w       1 1 1 1 w       1 1 1 1 1             1 1 1 1 1
1 1 0 0 n       1 1 0 0 n       1 1 0 0 n             1 1 0 0 1
1 0 0 0 n  ->   1 0 0 0 n  ->   1 0 0 0 n  -> .. ->  1 0 0 0 1
n 0 0 0 n       1 0 0 0 n       1 0 0 0 n             1 0 0 0 1
n w w w n       n w w w n       1 w w w n             1 1 1 1 1
```

Если реализовывать «волновой алгоритм» нерекурсивно, то количество итераций будет прямо пропорционально размеру самого большого бассейна водосбора. Временная сложность такого наивного решения будет $\Theta(n^4)$.

А можно ли обойтись всего **одной** итерацией по карте для того, чтобы сгруппировать все клетки по бассейнам водосбора?

1.3.2. Решение с одной итерацией и мемоизацией

Итак, мы хотим для каждой клетки карты вычислить координаты ее водостока. Можно сформулировать простой и медленно работающий рекурсивный алгоритм: для данной клетки определить направление стока воды. Далее, определить куда стекает дальше вода на следующем шаге — и так до тех пор, пока мы не дойдем до водостока.

Естественно, что для карты большого размера такое решение будет работать недопустимо медленно. Однако легко видеть, что его можно ускорить при помощи динамического программирования. Достаточно запоминать результаты для всех клеток, которые были пройдены в ходе рекурсивного поиска.

Определим несколько простых типов данных для хранения информации о координатах клеток, карты высот и карты водостоков:

```
type Coord = (Int,Int)
type HeightMap = Array Coord Int
type SinkMap   = Array Coord Coord
```

Почему для хранения карты был взят обычный массив (`Array`), а не ассоциативный (`Map`)? У `Array` есть четко определенные границы, которые можно узнать с помощью функции `bounds`. Если же хранить карту в виде `Map`, то потребуется отдельно хранить, передавать и обрабатывать информацию о размере — например в виде пары координат левого верхнего и правого нижнего углов `((Int,Int),(Int,Int))`.

Реализация описанного алгоритма с использованием динамического программирования будет выглядеть так:

```
-- Для получения результата необходимо определить водосток
-- для каждой клетки карты. Это делается с помощью свертки
-- (foldr) списка всех координат (range $ bounds arr)
-- функцией setSink.
--
-- Эта функция сохраняет вычисленные координаты водостоков в
-- ассоциативном массиве cache, содержимое которого и будет
-- результатом свертки и результатом работы функции flow.
flow :: HeightMap → SinkMap
flow arr = foldr setSink cache (range $ bounds arr)
  where
    nothing = (-1,-1)

    cache :: SinkMap
    cache = listArray (bounds arr) (repeat nothing)
```

```
-- Если для клетки coord водосток уже определен в ходе
-- рекурсивного обхода в фукнции findSink, то пропускаем
-- эту клетку. Иначе, выполняется поиск водостока при
-- помощи findSink, которая вернет список пройденных
-- клеток и координаты найденного водостока. Этот
-- результат сразу заносится в cache.
setSink :: Coord → SinkMap → SinkMap
setSink coord cache =
  if cache!coord ≠ nothing
     then cache
     else cache//(findSink coord)

-- Если для указанной клетки уже есть закэшированный
-- результат, то он сразу возвращается. Иначе,
-- исследуется список соседних клеток (neighbors
-- coord), имеющих высоту ниже текущей.
findSink :: Coord → [(Coord,Coord)]
findSink coord
  | cache!coord ≠ nothing =
      [(coord,cache!coord)]
  | otherwise =
    let ns = neighbors coord in
      if null ns
-- Если клеток ниже текущей нет, то текущая клетка
-- является водостоком, возвращаются ее координаты
         then [(coord, coord)]

-- Если воде есть, куда течь, то выполняется вызов
-- findSink для клетки, в которую стекает вода с
-- текущей. Из результатов этого вызова извлекается
-- информация о найденном водостоке и подставляется
-- в качестве координат водостока для текущей клетки.
         else let (next:rest) =
                findSink (snd (minimum ns))
                  in (coord, snd next):rest

-- Функция neighbors возвращает список пар (высота,
-- координата) для всех клеток, соседних с указанной.
neighbors :: Coord → [(Int,Coord)]
neighbors (x,y) =
  [ (arr!n, n)
  | n ← [(x+1, y), (x-1, y), (x, y-1), (x, y+1)]
  , inRange (bounds arr) n
  , arr!n < arr!(x, y) ]
```

Казалось бы, вычислительная сложность этой реализации должна быть $\Theta(n^2)$.

Но сложность функции `(//)` для стандартного типа `Array` составляет $\Theta(m)$, где m — длина массива. Соответственно, общая временная сложность получается по-прежнему $\Theta(n^4)$. В Haskell есть изменяемые (англ. mutable) массивы с обновлением элементов за $\Theta(1)$, но сравнение различных подходов к реализации массивов на функциональных языках выходит за рамки этой статьи.

В следующем разделе будет показано, как можно упростить приведенное решение и избавиться от необходимости обновлять `cache`.

Полный текст этой программы можно найти в файле `single-pass.hs` в директории `watersheds` архива с исходными текстами.

1.3.3. Решение без использования `cache`

Видно, что значения элементов `cache` вычисляются на основании значений других элементов `cache` — наподобие того, как это происходило в задаче «Welcome To Code Jam». Однако взаимные рекурсивные обращения друг другу функций `setSink`, `findSink` и `cache` достаточно сложно проследить без детального анализа кода. Можно ли сократить и упростить программу, избавившись от явного упоминания `cache` — так, чтобы рекурсивная природа `setSink` и `findSink` была более очевидна?

Вспомним, что результат `flow arr` — это свертка исходного массива функцией `setSink`, при этом суть свертки заключается в том, чтобы породить искомый массив водостоков путем заполнения элементов `cache`.

Предположим, что мы можем непосредственно вычислить все элементы искомого массива и таким образом исключить создание `cache` и свертку `arr` для заполнения `cache` значениями. Тогда функция `flow` будет выглядеть так:

```
flow arr =
  let result =
    listArray (bounds arr)
      [ sink | coord ← range (bounds arr)
             , let sink = f coord result ]
    in result
```

Функция, которой тут дано имя `f`, должна заменить собой и `setSink`, и `findSink`.

Как именно выглядит `f`, пока не ясно, но можно с уверенностью сказать, что она будет использовать в своих вычислениях `result` — так как и `setSink`, и `findSink` нужны результаты поиска водостоков для соседних с обрабатываемой клеток.

Выясняется, что если использовать данное ранее определение `neighbors`, то функцию `f` можно реализовать с помощью одного условного выражения:

```
flow :: HeightMap → SinkMap
flow arr =

  -- Результатом работы функции flow будет создаваемый из
  -- списка значений массив (listArray (bounds arr) [...]),
```

```
    -- в котором каждой клетке исходной карты высот
    -- поставлены в соответствие координаты ее водостока.
    let result =
          listArray (bounds arr)
          [ sink | coord ← range (bounds arr)
                 ,  let ns = neighbors coord

          -- Координаты водостока - это либо координаты самой
          -- клетки, если у нее нет соседей, ниже ее самой,
          -- либо координаты водостока той клетки, куда
          -- стекает вода с текущей.
                 , let sink =
                     if null ns
                        then coord
                        else result ! snd (minimum ns)
          ]
          in result
    where
    neighbors = ...
```

Подобный подход используется достаточно часто, и для упрощения написания кода в таком стиле в модуле `Data.Function` даже определена специальная функция `fix` с определением:

```
fix f = let x = f x in x
```

С ее использованием функция `flow` может быть переписана так:

```
import Data.Function (fix)

flow :: HeightMap → SinkMap
flow arr = fix attachToSink
  where
    attachToSink :: SinkMap → SinkMap
    attachToSink result =
      listArray (bounds arr)
      [ sink |
        coord ← range (bounds arr),
        let ns = neighbors coord,
        let sink = if null ns
                   then coord
                   else result ! snd (minimum ns)
      ]

    neighbors (x,y) = ...
```

Временная сложность решения составляет $\Theta(n^2)$.

Посмотрим, как работает это определение `flow` для простой карты высот из двух элементов: `arr = array ((1,1),(1,2)) [20,10]`:

```
-- По определению flow:
flow arr = fix attachToSink

-- По определению fix:
flow arr =
  let x = attachToSink x in x

-- По определению attachToSink:
flow arr =
  let x =
    listArray ((1,1),(1,2)) [ ... ] in x

-- Поскольку 20 > 10, вода из клетки (1,1) будет
-- стекать в клетку (1,2):
flow arr =
  let x =
    listArray ((1,1),(1,2)) [ (x!1,2): ... ] in x

-- Поскольку клетка x!(1,2) — водосток:
flow arr =
  let x =
    listArray ((1,1),(1,2)) [ (x!(1,2)):(1,2) ] in x

-- По определению x:
flow arr =
  listArray ((1,1),(2,2)) [ (1,2):(1,2) ]
```

Странное на первый взгляд имя функции `fix` объясняется тем, что она является реализацией на Haskell так называемого **комбинатора неподвижной точки** (см. [4], [3]). Полезные свойства этого комбинатора не исчерпываются упрощением записи рекурсивного кода, однако, всестороннее его описание требует серьезного экскурса в теорию и выходит за рамки данной статьи.

Полный текст программы можно найти в файле `with-fix.hs` в директории `watersheds` архива с исходными текстами.

В качестве самостоятельного упражнения читателю предлагается переписать решение первой задачи с использованием `fix`.

1.4. Заключение

Статья на примерах продемонстрировала, как мемоизация позволяет просто и эффективно превращать «наивные» реализации алгоритмов на Haskell в эффективные. Кроме того, как показывает пример второй задачи, эти эффективные реализации зачастую еще и более компактные и легкие для понимания.

Литература

[1] *Crochemore M., Hancart C., Lecroq T.* Algorithms on Strings. — New York, NY, USA: Cambridge University Press, 2007.

[2] Divide and conquer. — Статья в Wikipedia (en), URL: http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm (дата обращения: 20 декабря 2009 г.).

[3] Fix and recursion. — Раздел в Wiki-книге Haskell, URL: http://en.wikibooks.org/wiki/Haskell/Fix_and_recursion (дата обращения: 20 декабря 2009 г.).

[4] Fixed point combinator. — Статья в Wikipedia (en), URL: http://en.wikipedia.org/wiki/Fixed_point_combinator (дата обращения: 20 декабря 2009 г.).

[5] *Krawetz S. A., Womble D. D.* Introduction to Bioinformatics: A Theoretical and Practical Approach. — Humana Press, 2003.

[6] Memoization. — Статья в Haskell Wiki, URL: http://www.haskell.org/haskellwiki/Memoization (дата обращения: 20 декабря 2009 г.).

[7] Sequence alignment. — Статья в Wikipedia (en), URL: http://en.wikipedia.org/wiki/Sequence_alignment (дата обращения: 20 декабря 2009 г.).

[8] Sequence learning - paradigms, algorithms, and applications. — 2001.

[9] *Siegwart R., Nourbakhsh I. R.* Introduction to Autonomous Mobile Robots. — Bradford Book, 2004.

[10] Волновой алгоритм. — Статья на сайте algolist.manual.ru, URL: http://algolist.manual.ru/maths/graphs/shortpath/wave.php (дата обращения: 20 декабря 2009 г.).

[11] Динамическое программирование. — Статья в Wikipedia (ru), URL: http://ru.wikipedia.org/?oldid=18970272 (дата обращения: 20 декабря 2009 г.).

[12] *Т. Кормен, Ч. Лейзерсон, Р. Ривест*. Алгоритмы. Построение и анализ. Издание 2-е. — Вильямс, 2007. — 1272 с.

Проектирование Erlang-клиента к memcached

Лев Валкин
vlm@fprog.ru

Аннотация

memcached — сервис кэширования данных в оперативной памяти компьютера, широко использующийся в популярных интернет-проектах (LiveJournal, Wikipedia, Twitter).

В статье рассказывается о проектировании библиотеки клиентского доступа к экземплярам и кластерам memcached. Производится сравнение с альтернативными реализациями.

Код библиотеки доступен под лицензией BSD.

memcached *is an in-memory data cache widely used in popular Internet projects, such as LiveJournal, Wikipedia and Twitter.*

The article describes an Erlang client library that facilitates access to memcached instances and clusters. Design and implementation issues of the library are discussed in detail, and a comparison to other open source implementations is provided.

The library source is available under the BSD license.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/4486.html.

2.1. Коротко о memcached

Главу 2.1 можно пропустить тем, кто уже знаком с memcached, DHT и consistent hashing.

memcached (**mem**ory **cache** **d**aemon, [5]) — это простой кэш-сервер, хранящий произвольные данные исключительно в памяти. Отказ от работы с диском обеспечивает значительную скорость работы и предсказуемость времени отклика. memcached обычно используется в качестве дополнительного уровня кэширования перед какой-либо базой данных, либо в качестве базы данных для простейших задач, не требующих высокой надёжности хранения информации. Memcached широко используется в web-проектах с высокой нагрузкой: LiveJournal, Wikipedia, Twitter и множестве других. Библиотеки доступа к сервису memcached есть для всех распространённых языков программирования.

Запустив программу memcached в фоновом режиме (`memcached -d`), мы получаем сервис на известном TCP (или UDP: [6]) порту, способный сохранять и отдавать двоичные данные по текстовому ключу. По умолчанию memcached принимает соединения на порту 11211. На рисунке 2.1 приведён сеанс общения с memcached-сервером с использованием текстового протокола. В примере мы сохранили значение **hello world**, занимающее одиннадцать байт, под именем **someKey** и успешно получили обратно это значение, предъявив ключ **someKey**.

```
$> memcached -d
$> telnet localhost 11211
Connected to localhost.
get someKey
END
set someKey 0 0 11
hello world
STORED
get someKey
VALUE someKey 0 11
hello world
END
quit
Connection closed by foreign host.
```

Рис. 2.1. Пример сеанса работы с memcached в ручном режиме

2.1.1. Работа с фермой memcached-серверов

В проектах с высокой нагрузкой практически всегда используют не один, а множество memcached сервисов, работающих на разных машинах. Это позволяет исполь-

зовать под кэш больше оперативной памяти, чем может быть доступно на одной машине, а также повысить отказоустойчивость. Такой набор похожих серверов на сленге называется «фермой» (server farm).

При работе с множеством memcached-серверов для каждой проводимой операции возникает задача выбора сервера, который должен эту операцию обслужить.

Случайный сервер

Если требуется только обновлять значения каких-то счётчиков (например, счётчика числа посещений страницы), то можно просто использовать случайный доступный сервер для каждой операции. На этапе считывания информации мы можем пройтись по всем memcached-серверам, просуммировав значения счётчиков для известных ключей. Заметим, что стандартная реализация memcached не поддерживает операцию «дай мне список всех ключей, которые у тебя есть»).

Хеширование

Случайный выбор сервера подходит для относительно небольшого числа применений. Например, если нам нужно сохранять результаты выполнения «тяжёлых» SQL запросов, то желательно сохранять данные на тот memcached-сервер, где они смогут быть прочитаны следующим же запросом на чтение.

Для поддержки такого режима доступа к набору серверов используется хеширование. На основании ключа запроса высчитывается некое число, с помощью которого выбирается нужный нам сервер. В простейшем варианте используется формула $\mathrm{hash}(K) \bmod N$, где $\mathrm{hash}(K)$ — это функция хеширования, создающая число из произвольного (строкового) ключа K, а N — количество доступных серверов memcached. В качестве hash-функции используют алгоритмы SHA-1, MD5 и даже CRC, так как устойчивости к коллизиям от неё не требуется.

В рамках данной схемы возможны два варианта реакции на выход из строя серверов memcached.

В простейшем случае, если сервер, на который указывает вышеупомянутая функция, становится недоступен, то библиотека будет возвращать ошибку на все запросы для данного ключа. Недостатки этого варианта реакции понятны: при недоступности части memcached-серверов пропорциональная часть операций с ними будет вынужденно создавать дополнительную нагрузку на следующий уровень доступа к данным (базу данных и т. п.).

В случае необходимости обеспечения бóльшей доступности фермы серверов, мы можем предложить динамически менять N в зависимости от количества доступных в данный момент серверов. Данный вариант работы хорош ровно до тех пор, пока все memcached-серверы работают бесперебойно. Но как только из-за сетевой ошибки, проблем с питанием или технологических работ на ферме исчезает или появляется один из серверов, функция $\mathrm{hash}(K) \bmod N$ мгновенно начинает перенаправлять

запросы на другие наборы серверов. Операции с каждым ключом K, которые с помощью формулы $f(K) \bmod N$ перенаправлялись на сервер S_x, при изменении количества доступных серверов N теперь уже будут перенаправляться на сервер S_y. То есть, с высокой вероятностью все виды запросов вдруг станут посылаться не на те серверы, на которые эти запросы посылались ранее. Возникает ситуация, практически эквивалентная перезагрузке всех машин с memcached: система начинает работать с чистого листа, начиная отвечать «нет данных» практически на каждый посланный ей запрос. Для проектов с высокой нагрузкой такая ситуация недопустима, так как она имеет шанс мгновенно «обвалить» резким повышением нагрузки следующие уровни доступа к данным, например, SQL сервер или сторонний веб-сервис.

Устойчивое хеширование

Для того чтобы выход из строя серверов или плановое расширение фермы не приводили к каскадным перегрузкам следующих уровней доступа к данным, применяется схема устойчивого хеширования (consistent hashing, также см. [4]). При использовании устойчивого хеширования область значений hash-функции разбивается на сегменты, каждый из которых ассоциируется с каким-то сервером memcached. Если соответствующий сервер memcached выходит из строя, то все запросы, которые должны были быть адресованы ему, перенаправляются на сервер, ассоциированный с соседним сегментом. Таким образом, выход из строя одного сервера затронет расположение только небольшой части ключей, пропорциональной размеру сегмента (сегментов) области определения хеш-функции, ассоциированной с вышедшим из строя сервером.

Запас прочности фермы memcached-серверов при перераспределении нагрузки обеспечивается как дизайном программы memcached (memcached написан на C и использует достаточно эффективные алгоритмы), так и предварительным планированием нагрузки при построении фермы. На практике нагрузочная устойчивость memcached-ферм редко является узким местом.

Наиболее развитые библиотеки клиентского доступа к memcached работают с фермами memcached-серверов с использованием именно алгоритмов устойчивого хеширования.

Стоит особо отметить, что сам механизм устойчивого хеширования в библиотеках memcached-клиентов не лишен недостатков. Допустим, мы привыкли сохранять значение для ключа K = ПОГОДА на сервере №13. В какой-то момент этот сервер временно выходит из игры (перезагружается), и сегмент области значений hash-функции, которая ранее была ассоциирована с сервером №13, начинает обслуживаться сервером №42. На этот сервер №42 начинают записываться данные вроде ПОГОДА ⇒ ДОЖДИ. Затем сервер №13 возвращается в строй, и на него опять посылаются данные для ключа ПОГОДА, например, ПОГОДА ⇒ СОЛНЕЧНО. Потребители, периодически спрашивающие значение для ключа ПОГОДА, получают ответ СОЛНЕЧНО, возвращаемый с сервера №13, и счастливы. Теперь, допустим, сервер №13 уходит в астрал

второй раз. Потребители данных опять начинают ходить на сервер №42 и видят старые данные, ПОГОДА ⇒ ДОЖДИ. Возникает проблема, которую универсально решить не так уж просто.

На практике эту проблему обычно игнорируют, стремясь свести вероятность ее проявления к минимуму путем обеспечения максимальной возможной доступности серверов memcached.

2.2. Проектирование клиентской библиотеки к memcached

Эта часть статьи описывает дизайн и некоторые детали реализации клиентской библиотеки к memcached, реализованной командой стартапа JS-Kit в 2007 году.

Для обслуживания клиентов мы развивали систему на Эрланге. Так как в системе использовались десятки машин, возникала необходимость распределённого кэширования данных.

В то время свободно доступных клиентских средств для общения с memcached сервисом на Эрланге не было, несмотря на то, что в Эрланг-сообществе то и дело анонсировались новые интернет-проекты с использованием этого языка, а также всевозможные инструменты и библиотеки для веб-разработки.

Нам было необходимо, как минимум, повысить эффективность использования памяти на машинах. Вместо одного «большого» сервера memcached лучше использовать несколько мелких серверов, получая пропорциональное увеличение доступной памяти для хранения временных данных. Тридцать машин, с выделенными четырьмя гигабайтами памяти под memcached-процесс на каждой, дают 120 гигабайт распределённого кэша. Использовать ферму серверов оказывается выгоднее, чем купить или арендовать одну машину с таким количеством памяти.

Однако, при использовании десятков серверов вступает в действие закон больших чисел: перезагрузка или выход из строя серверов становится ежемесячным, если не еженедельным явлением. Соответственно, требовался способ организации кэширования, устойчивый к случайным сбоям серверов.

Рассматривалась возможность использования встроенной в Эрланг распределённой системы управления базами данных Mnesia, но этот вариант в итоге был отброшен.

▷ Первая проблема с Mnesia заключалась в не слишком высокой надёжности кластера из экземпляров Mnesia при высоких нагрузках. При некоторых условиях (например, не слишком длинная, но существенная загрузка машины) кластер может необратимо распасться и рассинхронизироваться. Чтобы избежать необходимости ручного восстановления кластера, требуется организация системы автоматического отслеживания и реконфигурации.

▷ Вторая проблема с Mnesia состояла в том, что её использование для организации большого кэша требует специальной конфигурации. Одна ets-таблица в Mnesia не мо-

жет превышать размера доступной оперативной памяти. Это значит, что необходимо создать не одну таблицу для кэширования, а несколько, а также обеспечить механизмы распределения обращений к данным в разных таблицах и процедуры расширения пула таблиц при добавлении серверов в кластер.

▷ Третья проблема с Mnesia — в том, что в ней нет встроенных механизмов замещения наименее нужных данных. Такие механизмы необходимы для того, чтобы не выходить за рамки выделенной памяти. Пришлось бы организовывать механизм LRU самостоятельно, то есть брать на себя кодирование того, что уже сделано в memcached.

Обеспечение автоматизации решения этих проблем с Mnesia наверняка превзошло бы по сложности реализацию memcached клиента, рассматриваемого в этой статье.

Принимая решение о реализации клиента для memcached-протокола, мы также получали дополнительные преимущества, не связанные с задачами кэширования данных. Например, стало возможным осуществлять соединение с различными альтернативными базами данных, а также другими сервисами, использующими протокол memcached.

2.2.1. Требования к библиотеке и предварительные решения

При проектировании библиотеки важно правильно выбрать уровень абстракции интерфейса, который библиотека будет предоставлять приложению.

▷ **Доступ к одному или множеству серверов.** Должна ли библиотека работы с memcached просто организовывать взаимодействие с указанным сервером или обслуживать сразу ферму memcached-серверов, используя алгоритм устойчивого хеширования, описанный в предыдущей части статьи? С точки зрения предоставления максимально абстрактного API, мы можем требовать от библиотеки максимум функциональности. То есть, библиотека должна скрывать детали реализации общения с фермой серверов. С другой стороны, идеология отделения механизма от политики подсказывает нам, что системы надо собирать из как можно более ортогональных частей. Одна часть решения может заниматься транспортом данных и управлением соединением с произвольным сервером, а другая — дирижировать множеством таких однонаправленных транспортных сервисов. Каждая часть при этом решает одну задачу, но решает её хорошо. Это разделение хорошо перекликается с функциональным подходом, в котором декомпозиция задачи на простые, ортогональные подзадачи всячески приветствуется. Поэтому уже на этапе проектирования API мы внесли коррективы в наши планы: мы не стали разрабатывать универсальную библиотеку, работающую с memcached фермой, а разбили функциональность на три Эрланг-модуля:

mcd.erl: модуль, реализующий интерфейс к одному-единственному серверу memcached;

dht_ring.erl: модуль, реализующий алгоритм устойчивого хеширования для произвольных учётных единиц (в данной статье мы не будем заострять на нём внимание);

mcd_cluster.erl: модуль для организации устойчивой к сбоям фермы из многих memcached-серверов, соединяющий **mcd** и **dht_ring** вместе.

Каждый из компонентов можно отлаживать по-отдельности, а первые два — ещё и использовать независимо друг от друга в разных проектах. При необходимости, в решении можно независимо заменить на альтернативные реализации как транспортный механизм **mcd**, так и алгоритм устойчивого кэширования **dht_ring** или способ организации работы множества серверов **mcd_cluster**.

▷ **Использование постоянного соединения.** Для успешного применения в проектах с высокими нагрузками библиотека работы с memcached по TCP должна уметь устанавливать и использовать постоянное соединение с сервером, избегая затрат на установку и разрыв TCP-соединения для каждого запроса. Обеспечением обслуживания постоянного соединения будет заниматься модуль **mcd.erl**.

▷ **Полноценный игрок на поле OTP.** Open Telecom Platform (OTP) — это коллекция библиотек, *поведений* (*behavior*) и сопутствующей идиоматики. OTP — интегральная часть дистрибутива Erlang.

Библиотека работы с memcached должна быть подключаема в какой-либо OTP-супервизор — процесс, управляющий запуском дочерних процессов и их перезапуском в случае аварийного завершения. Некоторые библиотеки для доступа к memcached выбирают альтернативный вариант: они оформлены как самостоятельное OTP-*приложение* (application).

Выбор между этими двумя способами оформления библиотеки является делом вкуса, но сам факт оформленности кода в сущности OTP позволяет обеспечивать отказоустойчивость решения, упрощает отладку, развёртывание кода, обновление версий кода в «горячем» режиме, да и просто является хорошим тоном.

▷ **Поддержка именованных наборов memcached-серверов.** Использование функциональности одновременной работы с *разными* фермами memcached может понадобиться в крупных проектах, где существует какая-то иерархия кэшей или требуется изоляция данных в разных «банках».

Библиотеки, предоставляющие интерфейс типа `foobar:get(Key)` и `foobar:set(Key, Value)`, не позволят в одном приложении жонглировать данными между разными наборами memcached-серверов. Гораздо лучше, если можно явно указывать, какой сервер или ферму использовать. Отсюда возникает требование явного указания фермы в базовом API: `mcd:get(Farm, Key)`, `mcd:set(Farm, Key, Value)`.

В Эрланге вся работа делается набором процессов, адресуемых по их идентификаторам (Pid). Аргумент `Farm` из приведённых примеров — это идентификатор процесса, организующего общение с данной фермой или отдельным memcached-сервером. Эрланг в существенной степени облегчает использование подобных API, давая возможность *регистрировать* составные идентификаторы процессов, типа `<0.7772.267>`, под более мнемоничными именами, такими как `localCache`. Данная возможность приводит использование API к более приличному виду:

`mcd:get(localCache, Key)`, `mcd:set(remoteFarm1, Key, Value)`. С точки зрения пользователя API нам не важно, где находится `localCache` или из каких узлов состоит `remoteFarm1`. Достаточно знать, что процессы под названием `localCache` и `remoteFarm1` были созданы и зарегистрированы при старте системы.

▷ **Конструктор из модулей с идентичным API.** Как можно заметить из предыдущих примеров, существует некоторый конфликт между заявленной для модуля `mcd.erl` функциональностью (общение с единственным memcached-сервером) и тем, как применяется API в примерах выше (возможность использовать ферму в вызовах функций, например `mcd:set(remoteFarm1, Key, Value)`). Этот конфликт неслучаен. Мы хотим, чтобы с точки зрения API не существовало разницы между вызовом операции с единственным сервером и вызовом операции с фермой серверов. Это различие несущественно, чтобы поднимать его на уровень API: сегодня пользователю хочется использовать один memcached-сервер, а завтра — десять, но код должен оставаться идентичным, с точностью до имени процесса, ответственного за результат.

Потому мы сразу проектируем систему так, чтобы **mcd_cluster** являлся простым транслятором сообщений в нужный экземпляр **mcd**. С точки зрения пользователя API, **mcd_cluster** вызывается только один раз из супервизора для организации фермы:

```
mcd_cluster:start_link(remoteFarm1,
        [["server1.startup.tld", 11211],
         ["server2.startup.tld", 11211],
         ["server3.startup.tld", 11211]]).
```

и в дальнейшем общение с фермой идёт через API модуля **mcd**.

Почему это хорошо? Наш get/set API не зависит от конфигурации фермы и использующейся функциональности. Программист прикладного уровня имеет только один API. Мы можем использовать **mcd** отдельно от **mcd_cluster**. Мы можем сменить конфигурацию кэша, основанного на **mcd**, на более тяжеловесную (с точки зрения количества взаимодействующих частей) конфигурацию фермы, просто изменив способ инициализации с `mcd:start_link(Name, Address)` на `mcd_cluster:start_link(Name, [Address])`.

Почему это плохо? С точки зрения первоначального изучения кода, да и последующей отладки, использование API модуля **mcd** для посылки сообщений процессу, порождённому **mcd_cluster** (стрелка 1 на рисунке 2.2), может показаться неочевидным. Ведь этот процесс просто обеспечивает диспетчеризацию сообщений процессу, стартовавшему как экземпляр функциональности **mcd** (стрелка 6).

Мы постараемся сделать слой абстракции, предоставляемый модулем **mcd_cluster**, очень тонким, чтобы этот пинг-понг между модулями не мешал даже беглому изучению кода. Рисунок 2.2 поясняет, как именно происходит переброс сообщений:

- API, предоставляемый модулем **mcd.erl** (например, mcd:get/2 mcd:set/3), формирует запрос к процессу, указанному в первом аргументе

(`mcd:get(remoteFarm1, ...)`, стрелка 1). В случае организации общения с набором memcached-серверов с помощью **mcd_cluster**, этим процессом будет процесс, обслуживаемый кодом **mcd_cluster.erl**.

- Код **mcd_cluster.erl**, пользуясь функциональностью библиотеки устойчивого хеширования **dht_ring** (2…5), пересылает запрос одному из процессов **mcd**, связанному с конкретным memcached-сервером (6).
- Обработав запрос, **mcd**-процесс возвращает результат (7), завершая выполнение исходного вызова API.

▷ **Поддержка нативной модели данных.** API, предоставляемый библиотекой, должен обеспечивать возможность работы с произвольными данными. Такой подход типичен для библиотек на Эрланге. Так, мы должны иметь возможность сохранить произвольную структуру данных Erlang в memcached и вытащить её в неизменном виде. Библиотеки, предоставляющие интерфейс только к сохранению бинарных объектов, вынуждают пользователя использовать `term_to_binary/1` и `binary_to_term/1` на уровне приложения, навязывая недостаточно абстрактный, на наш взгляд, интерфейс. Сравним подобный интерфейс с гибкостью API, позволяющего использовать Erlang API на всю катушку (демонстрируется использование составных ключей, автоматическую сериализацию объектов и сохранение замыканий в memcached):

```
1> mcd:set(myCache, [42, "string"],
                    {"weather", fun() -> "normal" end}).
{ok,{"weather",#Fun<erl_eval.20.67289768>}}
2> {ok, {_, Fun}} = mcd:get(myCache, [42, "string"]).
{ok,{"weather",#Fun<erl_eval.20.67289768>}}
3> Fun.
#Fun<erl_eval.20.67289768>
4> Fun().
"normal"
5>
```

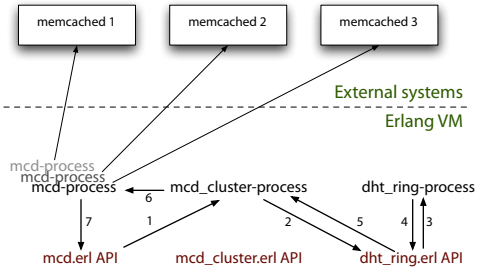


Рис. 2.2. Передача сообщений одному из серверов фермы, адресуемой через **mcd_cluster**

▷ **Асинхронность обработки запросов.** Библиотека **mcd** должна максимально использовать асинхронность. Она должна иметь возможность одновременно передавать сразу несколько независимых запросов на сервер memcached. Даже на задержках локальной сети (менее 1 миллисекунды) имеет смысл не дожидаться ответа на предыдущий запрос перед посылкой следующего. Должен существовать способ сопоставить клиенту, запросившему у **mcd** ту или иную операцию, предназначенный именно ему ответ от memcached.

В разделе 2.3.4 мы покажем, как это требование влияет на производительность библиотеки в реальных условиях.

2.2.2. Реализация mcd.erl

mcd.erl — модуль, который организует запуск Erlang-процессов, умеющих общаться с memcached-сервером по указанному при запуске адресу.

От рекурсивного обработчика к OTP-«поведению»

Одно из перечисленных выше требований к клиентской библиотеке — использование постоянного соединения с memcached. Мы должны организовать процесс так, чтобы он постоянно имел доступ к дескриптору TCP-соединения с memcached. В нижеприведенном фрагменте псевдокода показано, как можно организовать простой долгоживущий процесс, управляющий каким-то TCP-каналом.

```
-module(mcd_example).
-export([get/2, start_link/2]).

get(ServerRef, Key) ->
    Ref = make_ref(),
    ServerRef ! {get, self(), Ref, Key},
    receive
        {response, Ref, Resp} -> Resp
    after 5000 ->
        {error, timeout}
    end.

start_link(Address, Port) ->
    {ok, TcpSocket} = gen_tcp:connect(Address, Port),
    {ok, spawn_link(?MODULE, memcached_client, [TcpSocket])}.

memcached_client(TcpSocket) ->
    receive
        {get, From, Ref, Key} ->
            From ! {response, Ref, askServerGet(TcpSocket, Key)}
            memcached_client(TcpSocket);
        {tcp_closed, TcpSocket} ->
```

```
            ok.
    end.
```

Этот псевдокод почти рабочий — не хватает только функции `askServerGet`.

При вызове `spawn_link/3` запускается параллельный процесс, выполняющий функцию `memcached_client/1`. Последняя имеет обязательный аргумент — дескриптор TCP-канала, — дающий возможность процессу в любой момент послать или принять по данному каналу сообщение.

Так как мы проектируем промышленную библиотеку, а не «наколенную» поделку, здесь необходимо уйти от ручного кодирования рекурсивных функций и перейти к использованию готовых инструментов и идиом, доступных в Erlang OTP. Для данного процесса необходимо использовать *поведение* `gen_server`.

Кроме того, что `gen_server` скрывает детали организации рекурсивного цикла, он также придаёт процессу, порождённому с его помощью, ряд свойств, делающих его полноценным членом иерархии процессов в OTP.

Вот практически эквивалентный, но более идиоматичный вариант предыдущего кода, переписанный с использованием поведения `gen_server`.

```
-module(mcd).
-behavior(gen_server).
-compile(export_all).

get(ServerRef, Key) ->
    gen_server:call(ServerRef, {get, Key}).

start_link(Address, Port) ->
    gen_server:start_link(?MODULE, [Address, Port], []).

-record(mcdState, { socket }).

init([Address, Port]) ->
    {ok, TcpSocket} = gen_tcp:connect(Address, Port),
    {ok, #mcdState{ socket = TcpSocket }}.

handle_call({get, Key}, _From, State) ->
    Socket = State#mcdState.socket,
    {reply, askServerGet(Socket, Key), State}.

handle_cast(_, State) -> {noreply, State}.
handle_info({tcp_closed, Sock},
        #mcdState{ socket = Sock } = State) ->
    {stop, normal, State}.
```

Поведение `gen_server` берёт на себя организацию процесса, исполняющего рекурсивный цикл обработки приходящих к нему сообщений. При получении сообщения, посланного функцией `gen_server:call/2,3`, этот цикл вызывает функ-

цию handle_call/3. Результат выполнения handle_call/3 будет послан назад тому, кто запросил данные. При получении сообщения, посланного функцией gen_server:cast/2, этот цикл вызывает функцию handle_cast/2. При получении сообщения, которое было послано иным способом (например, через примитив Pid !Message), рекурсивный цикл вызовет функцию handle_info/2. Кроме этого, рекурсивный цикл, организованный поведением gen_server, самостоятельно отвечает на некоторые системные сообщения, обеспечивая нашему процессу улучшенное время взаимодействия с остальными модулями в OTP.

Ещё отметим, что поведение gen_server даёт нам примитив gen_server:call/2,3, реализующий надёжный механизм IPC. Встроенная в язык возможность послать сообщение через Pid !Message по многим причинам не является надёжным механизмом для обмена данными. Даже тот трюк с make_ref(), изображённый в псевдокоде на странице 44, не лишён недостатков. Этот трюк защищает от дупликатов сообщений: если функция get/2 вернула {error, timeout}, и её вызвали заново с другим аргументом, она может вернуть *предыдущий* ответ. Налицо проблема гонок (race condition) в очерёдности событий «сообщение послали», «сообщение приняли», «наступило превышение времени ожидания».

С наивно организованным на основе конструкции Pid !Message обменом сообщениями часто бывает и другая проблема. К примеру, показанная на странице 44 реализация get/2 не сможет определить, когда процесс ServerRef уже умер, и при вызовах будет завершаться через тайм-аут, а не мгновенно, как это следовало бы делать. Использование gen_server:call/2,3 избавляет нас от этой и многих других проблем.

Если вы ещё не используете поведение gen_server в своих программах, самое время начать это делать.

Восстановление после ошибок соединения

Что происходит, когда сервер memcached прекращает своё существование или становится недоступен по иной причине? По идее, нам нужно каким-то образом реагировать на закрытие канала TcpSocket. Вариантов всего два. Либо при получении сообщения {tcp_closed, _} мы тут же завершаем работу процесса (возврат {stop, ...} в обработчике сообщения handle_* провоцирует gen_server на завершение всего процесса), либо пытаемся подключиться к серверу вновь и вновь, возвращая в это время клиентам что-нибудь вроде {error, noconn}.

Оба способа обладают своими достоинствами и недостатками.

Завершая процесс в случае сбоя сервера, мы следуем мантре Erlang'а «let it fail» («пусть падает») в расчёте на то, что процесс этот запущен в вышестоящем супервизоре, и супервизор перезапустит его. Это достаточно удобно с точки зрения логики программы: не нужно делать никаких дополнительных шагов для обеспечения надёжности и защиты от сбоев. Если наш процесс умирает, его кто-то запускает заново. С другой стороны, если сервис memcached умирает надолго, а **mcd** был запу-

щен каким-то стандартным супервизором (функциональностью стандартного модуля `supervisor`), то мы, часто перезапускаясь, рискуем убить себя, супервизор и всю связанную с ним иерархию процессов. Особенностью поведения стандартного супервизора является то, что он следит за количеством перезапусков подчинённых ему процессов в единицу времени и может завершить себя и остальные процессы в его подчинении, если поведение подчинённого процесса выглядит для супервизора похожим на бесконечный цикл.

Для того чтобы иметь возможность запускать `mcd` под стандартным супервизором и не опасаться периодической недоступности memcached-серверов, нам всё-таки придётся обеспечивать логику переподключения к сбойнувшему серверу, добавив поле `status` в наше «глобальное» состояние. Чтобы знать, к чему же именно нам нужно переподключаться, необходимо иметь адрес и порт в `#mcdState{}`:

```
-record(mcdState, {
    address = "localhost",
    port = 11211,
    socket = nosocket,
    % Connection state:
    %   disabled | ready
    %   | {connecting, Since, ConnPid}
    %   | {testing, Since}  % protocol compatibility
    %   | {wait, Since}     % waiting between reconnects
    status = disabled
 }).
```

При некоторых видах выхода из строя memcached-серверов последующие попытки подсоединения к ним будут приводить к длительному ожиданию соединения. Функция `gen_tcp:connect/3`, используемая для переподключения к TCP серверу, синхронна. Подразумевается, что асинхронность, при необходимости, должна обеспечиваться средствами языка (порождением отдельного процесса для ожидания соединения), а не расширенными средствами библиотеки `gen_tcp` (например, путём предоставления асинхронного режима работы с TCP-каналами). Соответственно, имеет смысл осуществлять ожидание соединения в отдельном процессе, а затем передавать результат ожидания процессу **mcd**. Время начала попытки соединения сохраняем в `Since`, а идентификатор процесса, инициирующего соединение, хранится в `ConnPid`. Получив сообщение от `ConnPid` об успешно завершённом соединении, мы должны начать тестирование memcached-протокола, чтобы понять, что memcached-сервер действительно доступен и готов к работе (`status` меняется на `{testing, Since}`). При неуспешном соединении или результате тестирования протокола, **mcd** переходит в режим `{wait, Since}`, ждёт от нескольких миллисекунд до нескольких десятков секунд (в зависимости от количества проведённых ранее неуспешных попыток), а затем повторяет попытку соединения.

Поддержка асинхронной работы с запросами и ответами

В разделе 2.2.1 было упомянуто, что **mcd** должен работать по возможности асинхронно, чтобы избежать ненужной зависимости от сетевых задержек. Это значит, что необходимо иметь очередь посланных на memcached-сервер запросов. Очередь должна содержать идентификаторы процессов, сделавших запросы, чтобы пришедший ответ можно было направить тому, кто его ждет.

Представим на минуту протокол memcached, разработанный специально для Эрланг-программистов. В нём была бы возможность указать произвольный идентификатор транзакции, в качестве которого разработчики клиентов к memcached использовали бы обратный адрес процесса, которому нужно отправить ответ по результатам этой транзакции. Это обеспечило бы отсутствие необходимости запоминать что, для кого и в какой именно последовательности мы послали на memcached-сервер: он бы сам сообщал, что делать с ответом.

Так как протокол доступа к memcached не содержит возможности назначить *произвольный* идентификатор транзакции для каждой операции, то нам необходима именно последовательная очередь «квитков» на транзакции, `requestQueue`.

```
-record(mcdState, {
    address = "localhost",
    port = 11211,
    socket = nosocket,
    % Queue for matching responses with requests
    requestQueue,
    status = disabled
 }).
```

Приход очередного ответа от memcached-сервера инициирует извлечение самого старого квитка из очереди и отправку этого ответа процессу, обозначенному в квитке. Проблема теперь только в том, что протокол memcached достаточно нерегулярный, и вычленить данные из ответа сервера не так-то просто. Например, на запрос типа *stats* memcached посылает набор строк, оканчивающихся строкой «END». На запрос *get* memcached посылает строку «VALUE», имеющую в своём составе длину двоичного ответа, затем сам потенциально двоичный ответ (содержащий произвольные символы, в том числе переводы строк), а затем завершает это строкой «END». Но когда мы посылаем запрос типа *incr* или *decr*, memcached отвечает просто целым числом на отдельной строке, без завершающего «END».

Чтобы правильно интерпретировать этот протокольный бардак в асинхронном режиме, необходимо в квитке хранить информацию о том, как проинтерпретировать соответствующее сообщение. «Тип» квитка таков: `{{pid(), ref()}, atom()}`. Ранее в статье было показано, почему одного только `pid()` недостаточно, чтобы надёжно вернуть результат запроса тому, кто его запросил. Тип `{pid(), ref()}` описывает пару из идентификатора процесса, который запрашивает у **mcd** данные, и уникального идентификатора самого запроса. Именно эта пара приходит в качестве аргумента `From` в функцию `handle_call(Query, From, State)` поведе-

ния gen_server Значение atom() является внутренним идентификатором, позволяющим при подготовке к разбору очередного ответа от memcached понять, какого именно формата данные следует ожидать.

Таким образом, квиток содержит две сущности: информацию о том, как интерпретировать ответ от memcached, и информацию о том, куда посылать проинтерпретированный ответ. Вот пример нескольких квитков, лежащих последовательно в очереди requestQueue:

```
{{<0.21158.282>,#Ref<0.0.164.61572>},rtVersion}.
{{<0.22486.282>,#Ref<0.0.164.62510>},rtDelete}.
{{<0.23511.282>,#Ref<0.0.164.65512>},rtFlush}.
```

Когда наступает время обработать очередной ответ от сервера, информация из квитка используется для построения *дождания*[1] которое будет способно правильно интерпретировать ответ, даже если он придёт от сервера не целиком, а в несколько приёмов. Дожидание — это просто функция, которая на основании аргумента (данных из TCP-канала) возвращает новую функцию, которая сможет продолжить интерпретацию ответа, если изначально данных для завершения интерпретации не хватило. Если сервер возвращает ответ, разбитый на множество пакетов, использование дожиданий позволяет простым образом обработать их, не блокируя основной поток процесса необходимостью синхронного дочитывания данных.

Конструирование сложных функций с помощью комбинаторов — приём функционального программирования, который делает эту часть реализации **mcd** интересной. Ниже показана функция expectationByRequestType/1, которая «собирает» дожидание для приходящих из memcached данных.

```
expectationByRequestType(rtVersion) ->
    mkExpectKeyValue("VERSION");
expectationByRequestType(rtGet) ->
    mkAny([mkExpectResponse(<<"END\r\n">>, {error, notfound}),
           mkExpectValue()]);
expectationByRequestType(rtDelete) ->
    mkAny([mkExpectResponse(<<"DELETED\r\n">>,   {ok, deleted}),
           mkExpectResponse(<<"NOT_FOUND\r\n">>, {error,
               notfound})]);
expectationByRequestType(rtGenericCmd) ->
    mkAny([mkExpectResponse(<<"STORED\r\n">>,     {ok, stored}),
           mkExpectResponse(<<"NOT_STORED\r\n">>, {error,
               notstored})]);
expectationByRequestType(rtFlush) ->
    mkExpectResponse(<<"OK\r\n">>, {ok, flushed}).
```

Функция expectationByRequestType/1 использует комбинатор mkAny/1 и ряд других функций, каждая из которых умеет принимать аргумент определённого ви-

[1] По аналогии со специальными терминами-существительными «продолжение» (continuation) и «будущее» (future).

да. Комбинатор `mkAny/1` конструирует «сложную» функцию из передаваемого ему списка примитивных действий. Возвращаемая комбинатором функция при вызове пробует вызвать каждую из примитивных функций последовательно и возвращает ответ от первой из них, завершившейся без ошибок.

```
mkAny(RespFuns) -> fun (Data) -> mkAnyF(RespFuns, Data,
    unexpected) end.
mkAnyF([], _Data, Error) -> { error, Error };
mkAnyF([RespFun|Rs], Data, _Error) ->
    case RespFun(Data) of
        { error, notfound } -> {error, notfound};
        { error, Reason } -> mkAnyF(Rs, Data, Reason);
        Other -> Other
    end.
```

Рассмотрим частный случай функции `expectationByRequestType/1`. В данном случае комбинатор `mkAny/1` используется для того, чтобы создать функцию, которая будет ожидать от memcached-сервера либо строку «DELETED», либо строку «NOT_FOUND» и возвращать, соответственно, либо `{ok, deleted}`, либо `{error, notfound}`:

```
expectationByRequestType(rtDelete) ->
  mkAny([mkExpectResponse(<<"DELETED\r\n">>,   {ok, deleted}),
         mkExpectResponse(<<"NOT_FOUND\r\n">>, {error,
             notfound})]);

mkExpectResponse(Bin, Response) -> fun
    (Data) when Bin == Data -> Response;
    (_Data) -> {error, unexpected}
    end.
```

Для понимания того, что написано ниже, примем условие, что данные из TCP-канала приходят уже разбитыми на строки. Это обеспечивается включением режима (фильтра) `{packet, line}` при установке соединения с memcached-сервером: `gen_server:connect("localhost", 11211, [{packet, line}, binary])`.

Функция `mkExpectResponse/2` конструирует очень простое дожидание, которое возвращает позитивный или негативный ответ по первому же вызову.

Более интересным конструктором дожиданий является функция `mkExpectValue()`. Дожидание, сконструированное функцией `mkExpectValue()`, работает в двух фазах. В первой фазе пришедшая от memcached строка интерпретируется в соответствии с протоколом: из строки вида «VALUE key flags valueSize» (см. рис. 2.3) выясняется размер данных, которое необходимо будет вычленить из ответа memcached.

Ответ от memcached приходит в несколько приёмов: после обработки первой строки ответа дожидание переходит во вторую фазу своей жизни — фазу накопления данных. В этой фазе дожидание, отработав, возвращает следующее дожидание, и

```
$> telnet localhost 11211
Connected to localhost.
get someKey
VALUE someKey 0 11
hello world
END
quit
Connection closed by foreign host.
```

Рис. 2.3. Пример ответа memcached одиннадцатью байтами строки «*hello world*»

т. д., раз за разом принимая очередные сегменты данных, пока размер принятых данных не станет соответствовать продекларированному memcached-сервером в первой строке ответа.

```
mkExpectValue() -> fun (MemcachedData) ->
    Tokens = string:tokens(binary_to_list(MemcachedData), "
        \r\n"),
    case Tokens of
      ["VALUE", _Key, _Flags, BytesString] ->
        Bytes = list_to_integer(BytesString),
        { more, mkExpectBody([], Bytes, Bytes+2) };
      _ -> {error, unexpected}
    end end.

mkExpectBody(DataAccumulator, Bytes, ToGo) ->
  fun(MemcachedData) ->
    DataSize = iolist_size(MemcachedData),
    if
      DataSize < ToGo ->
        { more, mkExpectBody(
            [MemcachedData | DataAccumulator],
            Bytes, ToGo - DataSize) };
      DataSize == ToGo ->
        I = lists:reverse(DataAccumulator, [Data]),
        B = iolist_to_binary(I),
        V = binary_to_term(B),
        { more, mkExpectResponse(<<"END\r\n">>, { ok, V })}
    end
  end.
```

При приёме длинного ответа основная работа происходит внутри дожидания и его наследников, рекурсивно порождаемых функцией `mkExpectBody/3`. В процессе работы дожидание накапливает принятые данные в своём первом аргументе-аккумуляторе, чтобы, приняв финальный блок данных от memcached, вернуть все накопленные данные.

Хранение дожидания в состоянии mcd процесса

Когда мы находимся в процессе дожидания большого ответа, состоящего из множества пакетов, нужно где-то хранить текущую функцию дожидания. Для этого ещё раз преобразуем структуру состояния процесса, добавив туда поле `expectation`:

```
-record(mcdState, {
    address = "localhost",
    port = 11211,
    socket = nosocket,
    requestQueue,
    expectation = no_expectation,
    status = disabled
 }).
```

Теперь обработка ответов от memcached сводится к запуску дожидания при получении очередных данных из канала `#mcdState.socket`:

```
handle_info({tcp, Socket, Data}, #state{socket = Socket} = State)
    ->
        {noreply, handleMemcachedServerResponse(Data, State)}.

handleMemcachedServerResponse(Data,
        #state{requestQueue = RequestQueue,
               expectation = OldExpectation} = State) ->
    {From, Expectation} = case OldExpectation of
      no_expectation ->
        {Requestor, RequestType} = dequeue(RequestQ),
        ExpectFun = expectationByRequestType(RequestType),
        {Requestor, ExpectFun}
      ExistingExpectation -> ExistingExpectation
    end,

    NewExpectation = case Expectation(Data) of
      {more, NewExp} -> {From, NewExp};
      Result ->
        replyBack(From, Result),
        no_expectation
    end,
    State#mcdState{expectation = {From, NewExpectation}}.
```

Сообщения о приёме новых данных запускают очередной шаг текущего дожидания. Между сообщениями о приёме новых данных наш mcd-процесс может без задержки реагировать на произвольные сообщения, например, отвечать на запросы о статусе соединения с сервером memcached, количестве проведённых через mcd запросов и ответов, и других метриках, которые не требуют общения с memcached-сервером через TCP-канал. Этот положительный артефакт обработки событий в асинхронном режиме с использованием механизма дожиданий используется в коде **mcd.erl**. Так, он

позволяет быстро ответить клиенту `{error, timeout}`, если у нас в очереди по каким-либо причинам скопилось более тысячи неотвеченных сообщений.

Альтернатива дожиданиям

Описанному выше механизму дожиданий существует пара альтернатив.

Первая заключается в отказе от асинхронной обработки сообщений. Такой выбор сделан в реализации Erlang-клиента erlmc [1], за счёт чего соответствующая часть кода читается значительно проще:

```
send_recv(TcpSocket, Request) ->
    ok = send(TcpSocket, Request),
    recv(TcpSocket).
```

Эффект подобного отказа от асинхронной обработки сообщений будет рассмотрен далее в разделе 2.3.4.

Вторая альтернатива заключается разбиении mcd-процесса на два. Один процесс (назовём его *Sender*) занимается посылкой запросов memcached-серверу. При получении запроса на осуществление операции, этот процесс формирует соответствующий пакет и посылает его в канал связи с memcached-сервером. Второй процесс (*Receiver*) независимо занимается приёмом ответов от memcached-сервера.

Как было показано на странице 47, при использовании текстового протокола memcached перед приёмом данных от сервера нам необходимо знать структуру получаемого ответа. Значит, между *Sender* и *Receiver* необходима коммуникация. Так, *Sender*, посылая запрос memcached-серверу, должен сообщить процессу *Receiver* структуру сообщения, которое *Receiver* получит от memcached-сервера в ответ. С точки зрения *Receiver* это можно представить примерно следующим образом:

```
processReceiver_loop(TcpSocket) ->
    {From, RequestType} = receive
        {operation, RequestTicket} -> RequestTicket
    end,
    Response = assembleResponse(TcpSocket, MessageType),
    gen_server:reply(From, Response),
    processReceiver_loop(TcpSocket).
```

Для вычитывания нужного объема данных из TCP-канала связи с memcached-сервером в функции `assembleResponse/2` можно использовать `gen_tcp:recv/2`. Если общение с memcached происходит с помощью текстового протокола, то для облегчения синтаксического анализа ответа можно воспользоваться возможностью `gen_tcp` динамически менять канальные фильтры.

Например, если необходимо вычитать одну строку ответа (содержащую заголовочную информацию), а затем какое-то количество двоичных данных, можно делать так:

```
% Parses "VALUE someKey 0 11\r\nhello
    world\r\nEND\r\nNextAnswer..."
assembleResponse(TcpSocket, rtGet) ->
    ok = inet:setopts(TcpSocket, [{packet, line},list]),
    {ok, HeaderLine} = gen_tcp:recv(TcpSocket, 0),
    ok = inet:setopts(TcpSocket, [{packet, raw},binary]).

    case string:tokens(HeaderLine, " \r\n") of
      ["END"] -> undefined;
      ["VALUE", _Value, _Flag, DataSizeStr] ->
        Size = list_to_integer(DataSizeStr),
        {ok, Data} = gen_tcp:recv(TcpSocket, Size),
        {ok, <<"\r\nEND\r\n">>} = gen_tcp:recv(TcpSocket, 7),
        Data
    end.
```

Эффективность этой альтернативы по сравнению с использованием дожиданий выглядит существенной и заслуживает отдельных тестов. Недостатками являются невозможность запросить состояние клиента, пока он находится в процессе приёма ответа от memcached-сервера, а также некоторая потеря декларативности в описании протокола. Достаточно сравнить вышеприведённый код функции `assembleResponse(TcpSocket, rtGet)` с рассмотреным на странице 48 вариантом `expectationByRequestType(rtGet)`.

2.3. Сравнения и тесты

2.3.1. Двоичный протокол

Начиная с версии 1.3, создатели memcached-сервера внедрили в него поддержку двоичного протокола, обладающего некоторыми дополнительными возможностями. Значимость старого протокола при этом не исчезает: текстовый протокол остаётся для обеспечения обратной совместимости со старыми клиентами. Текстовый протокол memcached также используется вне связи с memcached сервером: в частности, существуют простые системы управления базами данных и системы организации очередей, использующие memcached-протокол, что позволяет им быть совместимыми с большим количеством клиентов, написанных на разных языках.

В августе 2009 года[2] двоичный протокол стало возможным применять и в асинхронном режиме.

Логично предположить, что двоичный протокол в асинхронном режиме даст большую производительность, чем текстовый протокол в асинхронном режиме. Но на момент тестирования существовала только одна библиотека, написанная целиком

[2]Исправлена ошибка № 72: http://code.google.com/p/memcached/issues/detail?id=72

на Эрланге — *erlmc* [1] — поддерживающая двоичный протокол memcached, и она работала синхронно.

За несколько дней до выхода данной статьи появилась библиотека **echou/memcached-client** [3], поддерживающая двоичный протокол в асинхронном режиме. К сожалению, библиотека вышла несколько сырая, только что «из-под пера». Запустить и протестировать её на скорость работы до выхода этого номера журнала мы не успели.

Интересно, что в конце 2009 года для Эрланга появилось сразу несколько новых клиентских библиотек работы с memcached.

2.3.2. Библиотеки работы с memcached

erlangmc Erlang-библиотека erlangmc, которая вполне может считаться использующей двоичный протокол, является простой надстройкой над C-библиотекой libmemcached. Её автор, открыв код, сделал выбор в пользу необычной для Эрланг-сообщества лицензии GPLv3.

cacherl::memcached_client Порт функциональности memcached-сервера на Erlang, cacherl, содержит реализацию функциональности memcached-клиента. Клиентский код использует текстовый протокол и работает в синхронном режиме. Клиент умеет работать с несколькими серверами memcached, но использует простую схему хеширования, неустойчивую к отказам memcached-серверов. Код **cacherl** доступен под лицензией LGPL.

joewilliams/merle Библиотека merle, написанная Joe Williams и Nick Gerakines, обеспечивает интерфейс к указанному в аргументе серверу memcached. Она использует текстовый протокол memcached и работает в синхронном режиме. Библиотека доступна по лицензии MIT.

higepon/memcached-client В декабре 2009 на GitHub появилась библиотека работы с memcached сервером higepon/memcached-client, написанная Taro Minowa. **higepon/memcached-client** использует текстовый протокол в синхронном режиме и организует работу только с одним сервером memcached. Код доступен под лицензией BSD.

echou/memcached-client Также в декабре 2009 на GitHub появилась другая библиотека, с похожим названием echou/memcached-client, написанная Zhou Li [3]. Этот очень развитый проект обладает набором полезных возможностей:

- работа с множеством именованных ферм memcached-серверов;
- автоматическое переподключение индивидуальных соединений с memcached серверами после разрыва связи;
- поддержка бинарного протокола (текстовый не поддерживается);
- поддержка асинхронной обработки команд.

Библиотека **echou/memcached-client** комбинирует код на Эрланге с с библиотеками и драйверами на C и доступна под лицензией Apache.

JacobVorreuter/erlmc Библиотека erlmc, выпущенная в октябре 2009 года, общается с memcached-сервером посредством двоичного протокола. **erlmc** позволяет работать с несколькими memcached-серверами, используя устойчивое хеширование для распределения операций между ними. Библиотека доступна под лицензией MIT.

Несмотря на то, что дизайн библиотеки **erlmc** существенно отличается от описываемого в данной статье, имело смысл сравнить **mcd** именно с ней, так как на момент написания статьи именно эти две полностью написанные на Erlang библиотеки предоставляли наиболее развитые возможности (например, устойчивое хеширование и работу с фермой memcached серверов) по сравнению с немногочисленными другими проектами.

2.3.3. Сравнение с erlmc

Jacob Vorreuter выпустил **erlmc** в октябре 2009 года.

erlmc имеет простой API с набором вызовов, отражающим набор операций, доступных в memcached-протоколе.

- `get(Key::any())→Val::binary()`
- `set(Key::any(), Val::binary())→Response::binary()`
- `replace(Key::any(), Val::binary())→Response::binary()`
- `delete(Key::any())→Response::binary()`
- …

Наборы memcached-серверов

Интерфейс **erlmc** позволяет обеспечить доступ только к одному набору memcached-серверов. Это не слишком серьёзная проблема для небольших проектов, в которых, как правило, принято использовать единственный memcached-сервер, либо одну ферму.

Интерфейс **mcd** даёт возможность использовать произвольное количество именованных наборов серверов memcached.

Поддержка встроенных типов данных

erlmc требует и возвращает двоичные данные в качестве значения для ключа.

mcd позволяет сохранять и восстанавливать произвольные структуры данных, автоматически вызывая `term_to_binary/1` и `binary_to_term/1`.

Реакция на отсутствие данных

erlmc возвращает двоичный объект нулевой длины, если memcached сервер не нашёл значения, ассоциированного с данным ключом.

mcd возвращает `{ok, any()}` при наличии данных в memcached-сервере и `{error, notfound}` при отсутствии. Это позволяет на уровне приложения отличить отсутствие данных от данных нулевой длины.

Размер и тип ключа

erlmc позволяет использовать ключи нескольких распространённых типов, но имеет ограничение на размер ключа в 64 килобайта.

mcd не имеет ограничений на размер или тип ключа, так как ключом является результат md5-хеширования двоичного представления Эрланг-структуры. Это может оказаться неудобным, так как отсутствует простая возможность запросить данные, сохранённые в memcached, посредством telnet или из программы, написанной на другом языке программирования.

Версия протокола

erlmc общается с memcached только с помощью двоичного протокола. Это позволяет использовать простой код, так как при разработке двоичного протокола создатели memcached учли ошибки проектирования его текстового эквивалента и сделали протокол достаточно регулярным. С другой стороны, по этой же причине **erlmc** не может быть использован в качестве клиента к другим программам, реализующим текстовый вариант memcached-протокола.

mcd использует более сложный в обработке, но более совместимый текстовый протокол.

Тип хеширования

erlmc умеет распределять запросы между несколькими memcached-серверами, используя устойчивое хеширование, которое мы рассмотрели на странице 37.

Интерфейс **mcd_cluster** предоставляет такую же возможность.

Способы инициализации memcached-фермы в библиотеках **erlmc** и **mcd_cluster** практически идентичны.

Реакция на ошибки соединения

При использовании **erlmc**, если в ферме memcached-серверов выходит из строя один сервер, то часть запросов, адресуемая этому серверу, начинает возвращать исключения. Это может являться адекватным решением проблемы «свежести» данных, описанной на странице 37, в разделе об устойчивом хешировании.

Реализация **mcd_cluster** перераспределяет запросы от неисправного сервера другим, в соответствии с правилами устойчивого хеширования. Таким образом, **mcd_cluster** может произвольное время работать с частично неисправной фермой, не создавая нагрузки на следующий уровень доступа к данным.

Работа над ошибками

erlmc, обнаружив потерю соединения с каким-то из memcached-серверов, не делает автоматических попыток подключиться к нему вновь. В перспективе возможна ситуация, когда перезагрузки отдельных серверов в профилактических или иных целях оставят **erlmc** совершенно без возможности совершать полезную работу.

mcd автоматически осуществляет попытки переподключения к «упавшим» memcached-серверам.

2.3.4. Скорость доступа к данным

Производительность в LAN

Все тесты, результаты которых представлены в этой части статьи были проведены между двумя одноядерными виртуализованными машинами, предоставляемыми Amazon EC2 (Small instance).

Даже в случае, когда ферма memcached-серверов находится в той же локальной сети, что и обращающийся к ней клиент, задержки на канале связи могут влиять на производительность memcached библиотеки. Так, типичные задержки в Ethernet LAN составляют 0.2 миллисекунд и более, особенно под нагрузкой. Это значит, что в случае синхронных запросов мы будем ограничены сверху величиной в 5000 запросов в секунду в идеальном случае.

И действительно, простой цикл получения небольших объёмов данных через интерфейсы **mcd** и **erlmc** показывает среднюю величину в 3567 запросов в секунду. Это соответствует средней задержке в сети 0.28 миллисекунд.

Но стоит запустить несколько независимых эрланг-процессов, которые запрашивают данные через один и тот же экземпляр процесса **mcd**, как тут же картина преображается.

В таблице 2.1 показан результат ста тысяч прогонов следующей функции:

```
1> mcd:get(web7, a).
{ok, {a,b,{[c,d,e],"some string",12324234234,<<"binary">>}}}
2>
```

В таблице 2.2 показан результат ста тысяч прогонов функции `erlmc:get(a)`. memcached-сервер в этом случае хранит и отдаёт данные того же размера, что и в примере с **mcd**.

Наглядно разница показана на рисунке 2.4. По вертикали отложены значения количества запросов в секунду. На горизонтальной оси отмечено количество параллельных тестирующих процессов, нагружающих **mcd** или **erlmc**, соответственно.

Попытка	1 поток	2 потока	4	10	100
1)	3764	4305	7622	12340	18105
2)	3668	4095	7549	12332	17279
3)	3683	3796	7275	11612	17487

Таблица 2.1. Попытка общения с **mcd** в несколько потоков

Попытка	1 поток	2 потока	4	10	100
1)	3505	3752	3774	3657	3536
2)	3218	3086	3558	3303	3633
3)	3567	3545	3556	3621	3405

Таблица 2.2. Попытка общения с **erlmcd** в несколько потоков

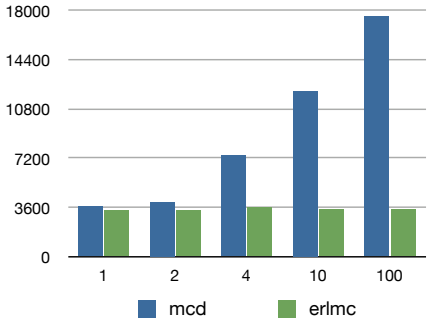


Рис. 2.4. Сравнение скорости **mcd** и **erlmc** на 64 байтах данных

Что произойдёт, если мы попробуем брать из memcached-сервера не 64 байта данных, а больше? Для 10 килобайт данных тестирование с использованием параллелизма показывает похожую разницу в результатах. Результаты десяти тысяч прогонов изображены на рисунке 2.5.

Тестирование показывает, что изначальное решение делать работу с memcached серверами асинхронной дало **mcd** существенный выигрыш в производительности по сравнению с синхронным способом работы. Даже то, что **erlmc** использует двоичный, а не текстовый протокол общения с **memcached**, не дало этой библиотеке никакого преимущества перед **mcd**.

Справедливости ради нужно отметить, что при типичном использовании **erlmc**, когда **erlmc** подключён сразу к нескольким memcached-серверам и распределяет нагрузку между ними, полученные значения скорости будут существенно выше. В идеальном случае, когда имеется достаточная энтропия в ключах, позволяющая **erlmc** разбрасывать операции с разными ключами по разным memcached-серверам, мы будем видеть производительность, кратную количеству используемых memcached-

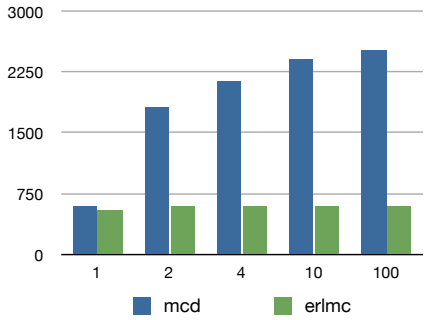


Рис. 2.5. Сравнение скорости **mcd** и **erlmc** на 10 килобайтах данных

серверов. Например, при балансировании между двумя memcached-серверами скорость **erlmc** на случайно выбранных ключах может составлять в 64-байтном тесте не 3.5 тысяч запросов в секунду (таблица 2.2), а 7 тысяч. Таким образом, при использовании пяти memcached-серверов **erlmc** может оказаться несколько быстрее, чем **mcd**, использующий один сервер.

Производительность на локальной машине

Для полноты картины приведём усреднённые результаты тестирования memcached-сервера, расположенного на локальной одноядерной машине Amazon EC2 (Small instance).

Было проведено 6 тестов **mcd** и **erlmc**, отличающихся размером получаемых от memcached-сервера данных. Каждый тест прогонялся три раза и состоял из десяти или ста тысяч итераций операции `mcd:get/2` или `erlmc:get/1`, производимых последовательно, с пятью разными степенями параллелизма в части количества инициаторов операций с memcached (1, 2, 4, 10, 100).

Результаты тестирования приведены в таблице 2.3. Как и следовало ожидать, использование разных степеней параллелизма существенно не изменяет результаты тестирования memcached, доступного на локально на одноядерной машине. Разницу в пределах 30 % можно объяснить особенностями параллельной сборки мусора для разного количества параллельных нагрузочных процессов.

Из-за отсутствия возможности устранения сетевых задержек (на локальной машине сетевых задержек как правило не бывает) ориентированный на текстовый протокол код **mcd** демонстрирует отставание от скорости работы библиотеки **erlmc**, использующей протокол бинарный. Скорость обработки запросов через **mcd** снижалась вплоть до 2.5 тысяч запросов в секунду, тогда как скорость **erlmc** практически не падала ниже 6 тысяч запросов в секунду. На малых данных разница была практически незаметна, но с ростом количества отдаваемых в ответе данных производительность **mcd** снижалась до 2.6 раз от скорости **erlmc**.

Ответ	Клиент	1 поток	2	4	10	100
пусто	mcd	7461	8452	9196	9687	9576
	erlmc	7808	8017	8255	8287	7440
<<>>	mcd	6522	7233	7822	8167	7538
	erlmc	7715	7997	8228	8303	7484
63 байта	mcd	6346	6920	7362	7443	6930
	erlmc	7648	7975	8201	8212	7352
64 байта	mcd	6422	7162	7738	7907	7138
	erlmc	7656	7959	8201	8277	7397
937 байт	mcd	2926	2977	3041	3019	2976
	erlmc	7595	7817	7967	7994	7138
10 кбайт	mcd	2656	2657	2786	2419	2454
	erlmc	6093	6279	6346	6219	5441

Таблица 2.3. Падение производительности **mcd** относительно **erlmc** с ростом размера данных

Заключение

Как было показано, создание memcached клиента — концептуально несложная задача. Интересной эту задачу делает попытка сделать решение чуть более оптимальным, используя асинхронный подход к управлению потоками операций. Результаты тестирования показывают, что на задержках, существующих в LAN, подобная оптимизация оправдана и приводит к результатам, существенно превышающим производительность более простых решений.

Интересный приём, представляющий интерес с точки зрения функционального программирования, реализован в механизме «дожиданий». Дожидания позволяют не блокировать работу клиентского процесса даже при приёме серверного ответа, разбитого на несколько пакетов с произвольными задержками между ними. В статье также рассмотрена более простая альтернатива дожиданиям, в большей степени использующая удобные механизмы Эрланга по организации параллельных процессов.

С появлением двоичного протокола появляется возможность дальнейшей оптимизации кода **mcd**, как в части его возможного упрощения, так и в части повышения его производительности на локальной машине, где сетевые задержки практически отсутствуют. Тесты, описанные в разделе «Производительность на локальной машине», показывают, что в этом отношении у **mcd** есть пространство для роста.

Полный исходный код memcached-клиента, рассмотренного в статье, размещён на GitHub под именем EchoTeam/mcd [2].

Литература

[1] Erlang-клиент для двоичного протокола memcached. — Проект Jacob Vorreuter на GitHub (en), URL: http://github.com/JacobVorreuter/erlmc/ (дата обращения: 20 декабря 2009 г.).

[2] Erlang-клиент для текстового протокола memcached. — Проект Jacknyfe на GitHub (en), URL: http://github.com/EchoTeam/mcd/ (дата обращения: 20 декабря 2009 г.).

[3] Erlang приложение для клиентского доступа к memcached. — Проект Zhou Li на GitHub (en), URL: http://github.com/echou/memcached-client/ (дата обращения: 20 декабря 2009 г.).

[4] Ketama: Consistent Hashing. — Сайт проекта (en), URL: http://www.audioscrobbler.net/development/ketama/ (дата обращения: 20 декабря 2009 г.).

[5] memcached — a distributed memory object caching system. — Страница проекта (en), URL: http://memcached.org/ (дата обращения: 20 декабря 2009 г.).

[6] Проект facebook-memcached, реализующий UDP для memcached. — Проект Marc Kwiatkowski на GitHub (en), URL: http://github.com/fbmarc/facebook-memcached (дата обращения: 20 декабря 2009 г.).

Как построить Google Wave из Erlang и Tcl при помощи OCaml

Дмитрий Астапов, Алексей Щепин
adept@fprog.ru, aleksey@fprog.ru

Аннотация

Статья рассказывает о том, как утилита camlp4, предназначенная для создания DSL, использовалась для автоматической генерации программного кода клиент-серверного приложения на Erlang (сервер) и Tcl (клиент).

Article describes the construction of the DSL for network protocol processing definition in a server-client setting and associated toolchain. Ocamlp4-based DSL translator was used to simultaneously transform DSL into Erlang code (for the server) and Tcl code (for the client).

Обсуждение статьи ведется по адресу
http://community.livejournal.com/fprog/4804.html.

3.1. Введение: что такое Google Wave и операционные преобразования

Сравнительно недавно компания Google анонсировала ограниченный доступ к первым версиям нового онлайн сервиса под названием Google Wave. Обещают, что сервис предоставит невиданные ранее возможности для совместной работы и редактирования документов с самым разнообразным содержимым — текстом, видео, изображениями и так далее, благодаря чему Google Wave уже успели окрестить рядом громких эпитетов вроде «убийца email» или «киллер сервисов мгновенных сообщений».

Впрочем, оставим в стороне маркетинговую шумиху. Что же такое Google Wave в контексте совместной работы над документами? Попросту говоря, — это серверное приложение с «тонким» клиентом (работающим в веб-браузере), которое обрабатывает команды редактирования от всех участников процесса и сводит их к единому непротиворечивому представлению документа.[1] Помимо инфраструктуры (клиента и сервера), разработанной Google, допускается использование сторонних реализаций (так называемый механизм «объединения волн», англ. «wave federation»). В частности, предполагается, что технология Google Wave может быть интегрирована в существующие службы мгновенного обмена сообщениями на базе протокола XMPP, позволив людям не только коллективно обмениваться сообщениями, но и при этом коллективно работать над документами.

Если углубиться в детали еще больше, окажется, что в основе Google Wave лежит протокол передачи изменений в XML документах. Протокол используется клиентами для того, чтобы отправлять на сервер информацию о сделанных правках, и принимать от него информацию о чужих изменениях. Любая операция по редактированию документа представляется в рамках этого протокола в виде последовательности команд: «сдвинуть курсор на n символов вправо», «вставить в позиции курсора такие-то символы», «удалить справа от курсора n символов» и т. п.

Получив информацию о «чужих» изменениях, клиент не может бездумно применить команды к своей версии документа, так как в результате может случиться рассогласование. Например, пусть клиент A вставил один символ в 8-й позиции, а клиент B в то же время удалил 15-й символ. Если клиент A буквально применит информацию о изменении, совершенном B, то в копии документа на клиенте A удаленным окажется символ на одну позицию левее нужной (см. рис. 3.1)

Существуют способы формального представления подобных операций редактирования и их обработки, которые позволяют избежать подобных проблем. Этот формализм известен под названием «операционное преобразование»[2] (см. [8], [9]). При использовании ОП клиент A сначала преобразует полученное от B изменение отно-

[1] Более подробное описание и видео-демонстрация на английском доступны на странице http://wave.google.com/help/wave/about.html.

[2] Далее «операционное преобразование» будет часто сокращаться до «ОП». Статья в русской Wikipedia использует перевод «операциональное преобразование», с чем авторы решительно не согласны.

Исходный документ:
«Казнить нельзя, помиловать»

Клиент A: `Insert 8 ','` → «Казнить, нельзя, помиловать» **Клиент A получает и буквально применяет изменения B**: `Delete 15` → «Казнить, нельз, помиловать»	**Клиент B**: `Delete 15` → «Казнить нельзя помиловать» **Клиент B получает и буквально применяет изменения A**: `Insert 8 ','` → «Казнить, нельзя помиловать»

Рис. 3.1. Конфликты, возникающие при наивном подходе к объединению правок

сительно своего документа, получив в результате команду «удалить 21-й символ» (это называется «включающее преобразование»), и только после этого применит полученную команду к своей версии документа, получив результат, идентичный имеющемуся у B.

В систему ОП, используемую в Google Wave (см. [5]), помимо набора из десяти команд для представления операций редактирования входят также две функции: включающее преобразование и композиция. Первая была рассмотрена выше, а вторая — это объединение нескольких последовательных операций редактирования в одну. Внутреннее представление документа в Google Wave включает в себя номер версии, который увеличивается после каждого изменения. Клиенты посылают свои изменения с указанием номера версии «своего» документа, а центральный сервер производит включающее преобразование изменений относительно своей версии документа, и рассылает результат остальным клиентам. Все прочие клиенты, в свою очередь, преобразовывают полученные изменения относительно своих необработанных (не отправленных) изменений и применяют к своей версии документа.

Если клиент прислал изменения к версии, которая «моложе» последней актуальной версии документа на сервере, то сервер использует композицию для объединения изменений, примененных на сервере после той версии, что указал клиент.

Подробное описание операций композиции и включающего преобразования с примерами их использования можно найти в [7].

3.2. Постановка задачи

Выбор команд и функций, входящих в систему ОП Google Wave, не является единственных возможным. Существуют другие системы ОП, обладающие теми или иными достоинствами или недостатками (см. [9]). Одной из целей ограниченного бета-тестирования Google Wave как раз и является проверка выбранной системы ОП в условиях, «приближенных к боевым». Кроме того, Google активно сотрудничает с другими компаниями для «обкатки» протокола интеграции со сторонними решениями (пресловутый federation).

В рамках этой деятельности одному из авторов потребовалось реализовать поддержку ОП из Google Wave в двух программных продуктах: Jabber-сервере ejabberd и Jabber-клиенте Tkabber. Главные требования, которые предъявлялись к реализации — это удобство поддержки, модификации и развития кода.

В рамках этой задачи необходимо реализовать со стороны клиента и со стороны сервера практически идентичную функциональность: выполнение включающего преобразования и композиции для разных возможных комбинаций команд редактирования. Если взять в качестве примера композицию, то упрощенное описание функциональности, которую требуется реализовать на Erlang (для сервера) и Tcl (для клиента) выглядит так:

- Исходными данными для операции композиции являются две последовательности команд редактирования. Каждую последовательность можно представить в виде списка с указателем на текущий элемент. В начале указатели в обоих списках установлены на первый элемент.
- Результатом композиции также является список команд редактирования.
- Композиция исходных списков заключается в циклическом применении операции «примитивной композиции» к текущим элементам списка до тех пор, пока это возможно.
- Операция примитивной композиции рассматривает текущие элементы обоих списков и, исходя из их значений, выполняет одно или несколько возможных действий: добавляет какую-то команду к списку-результату, смещает вправо один или оба указателя, модифицирует текущий элемент одного или обоих списков.

Всего существует десять различных команд редактирования — три команды для вставки текста, три команды для удаления, три команды для работы с атрибутами текста и команда для пропуска определенного количества символов без изменения (см. [6], раздел 7.2). Кроме того, любой из исходных списков может быть пустым, поэтому всего существует $11 \times 11 = 121$ различных теоретически возможных комбинаций команд примитивной композиции. В реальности, правда, имеют смысл *всего* 75 из них, и для каждой из них необходимо описать результат операции примитивной композиции.

Казалось бы, ничто не мешает взять и написать реализации на Erlang и Tcl «с нуля» вручную, но есть несколько потенциальных проблем, о которых лучше подумать заранее:

- Все 75 вариантов обработки довольно похожи друг на друга, что наводит на мысль о автоматической генерации хотя бы части из них из какого-то общего шаблона.
- В реализациях операций композиции и включающего преобразования много похожих или одинаковых мест.

- В случае изменения спецификации ОП или при нахождении ошибок потребуется вносить одинаковые изменения в оба массива кода. При этом велика вероятность ошибиться в одной из реализаций или получить реализации с поведением, отличающимся для каких-то граничных случаев

- Реализации на Erlang и Tcl требуют разных подходов к программированию из-за особенностей языков. Например, в Erlang есть сопоставление с образцом (англ. pattern matching), а в Tcl его нет. При работе со списками в Erlang эффективнее добавлять новые элементы к голове списка, а в Tcl — к хвосту, и так далее. Соответственно, не получается взять реализацию на одном языке и быстро и непринужденно мелкими правками превратить ее в реализацию на другом языке.

- С большой вероятностью список целевых языков будет в будущем расширен, и придется реализовывать эту же функциональность еще раз. Или несколько раз.

Выходит, что хорошо было бы найти такой подход к реализации, который позволит, во-первых, не делать одну и ту же работу два раза, и, во-вторых, облегчит отладку и развитие получившегося решения.

3.3. Возможный подход к решению

Попробуем создать специальный мини-язык программирования для задачи описания операций с ОП. Текст на этом языке будет транслироваться в тексты на целевых языках — Erlang и Tcl. В английской литературе подобные мини-языки традиционно называются *domain-specific languages*, и в дальнейшем создаваемый мини-язык будет сокращенно называться просто «DSL».

Трансляция программного текста, записанного на одном языке, в эквивалентный код, записанный на другом языке — это задача, традиционно решаемая компиляторами. Если пойти по пути создания компилятора DSL, то понадобится:

- Придумать и описать грамматику DSL

- Создать синтаксический анализатор текстов DSL, превращающий программы на DSL в соответствующие деревья разбора (см. [3])

- Создать транслятор дерева разбора в дерево абстрактного синтаксиса (AST). Этот транслятор переводит описание текста программы на более высокий уровень абстракции, заменяя детальную информацию о использованных синтаксических элементах информацией о конструкциях языка, которые они представляют (см. [1]).

- Создать кодогенераторы, превращающие дерево абстрактного синтаксиса в код на Erlang и Tcl.

На первый взгляд, решение этой вспомогательной задачи намного сложнее, чем решение оригинальной задачи.

Нельзя ли упростить процесс, и использовать какие-то готовые средства? Что если DSL будет синтаксически похож на какой-то существующий язык программирования (назовем его язык-носитель, англ. host language)? Тогда:

- Синтаксис DSL — это синтаксис выбранного языка программирования
- Для синтаксического анализа можно использовать готовые инструменты для синтаксического анализа языка-носителя.
- Полученное дерево разбора необходимо будет самостоятельно транслировать в дерево абстрактного синтаксиса DSL. Если код соответствующего транслятора для языка-носителя доступен — можно взять его за основу и доработать.
- После этого останется только самостоятельно реализовать кодогенераторы.

Получается эдакая задача метапрограммирования наоборот. В классическом метапрограммировании язык, специфичный для предметной области, обычно существенно отличается по синтаксису от языка-носителя. В ходе трансляции код на языке предметной области превращается в код на языке-носителе, с последующей компиляцией полученной программы как единого целого.

Мы же поступаем наоборот: наш DSL синтаксически похож на язык-носитель, и в ходе трансляции происходит его «отчуждение» от языка-носителя, с превращением кода на DSL в код на Erlang и Tcl.

Тем не менее, нельзя ли взять язык-носитель с развитыми средствами метапрограммирования и попытаться использовать их для решения нашей задачи?

Остаток статьи описывает создание транслятора DSL с использованием языка OCaml и утилиты camlp4 (см. [2]). Почему были выбраны именно эти средства? Во-первых, автор реализации хорошо с ними знаком. Во-вторых, camlp4 представляет собой практически идеальный инструмент для решения поставленной задачи. Его функциональность в области синтаксического анализа и модификации деревьев абстрактного синтаксиса можно легко расширять при помощи модулей, написанных на OCaml.

3.4. Проектирование DSL

Для начала необходимо спроектировать будущий DSL. Как было сказано выше, его синтаксис должен быть максимально похожим на синтаксис OCaml (чтобы облегчить разработку). Осталось определиться с тем, какими специфичными для задачи объектами и операциями необходимо дополнить OCaml.

Для чего будет использоваться DSL? Для описания того, как выполняется композиция или включающее преобразование отдельных команд в системе операционных преобразований Google Wave.

В каком контексте будет использоваться это описание? Из него будет получен программный код на целевом языке, который, вероятнее всего, будет оформлен в виде отдельной функции.

Аргументами этой функции будут два списка команд, которые необходимо обработать. Результатом ее работы также будет список команд. Как было сказано выше, функция должна, рассматривая оба входных списка поэлементно, генерировать значения выходного списка, при этом на каждом шаге выполняется переход к следующей команде в одном из входных списков или в обоих сразу.

В синтаксисе OCaml это можно упрощенно смоделировать следующим образом:

```
let main =
  ...
  let result = compose local_commands remote_commands in
  ...

(* Это базовые правила композиции двух списков, которые никогда не
    будут меняться *)
let compose local remote =
  match local, remote with
    | [], [] → []
    | _, _ →
        let {result = res; xs = local'; ys = remote'} =
          compose_one_cmd local remote
        in
          res :: compose local' remote'

(* Отсюда начинается часть программы, которую необходимо описывать
    с помощью DSL *)
let compose_one_cmd xs ys =
  match xs, ys with
    | Cmd1 (arg1_1, arg1_2) :: xt, Cmd1 (arg2_1, agr2_2) :: yt →
        let res = Cmd1 (arg1_1 + arg2_1, max arg2_1 arg2_2) in
        let xs' = xt in
        let ys' = Cmd1 (arg2_1 - arg1_1, arg2_2) :: yt in
          { result = res; xs = xs'; ys = ys'}
    | Cmd1 ..., Cmd2 ... → ...
    | Cmd1 ..., Cmd3 ... → ...
    | ...
    | Cmd1 ..., Cmd11 ... → ...
    | ...
    | Cmd11 ..., Cmd1 ... → ...
    | ...
    | Cmd11 ..., Cmd11 ... → ...
```

Получается, что объектами языка будут, во-первых, входные списки команд, а во-вторых — структуры, описывающие команды. Поскольку в качестве базы для DSL

берется OCaml, то списки команд будут представлены списками, а команды — конструкторами алгебраического типа данных.

В качестве операций в языке должны присуствовать, во-первых, операции над аргументами команд (как в приведенном выше примере), и, во-вторых, операции по изменению исходных списков: изменение головного элемента или переход к следующим элементам списка. Так как аргументами команд могут быть только строки, целые числа, их пары, или списки вышеперечисленных значений, для операций над аргументами можно использовать стандартные операции OCaml, не вводя новых. В приведенном выше примере головные элементы списков выделяются с помощью операций сопоставления с образцом, а в DSL для этих целей можно ввести специальные ключевые слова или переменные с зарезервированными именами.

Кроме этого, чтобы упростить трансляцию DSL в код на целевом языке, необходимо ограничить синтаксис, в котором описывается сама операция композиции: пусть там будут допустимы только условные выражения (`if`) и создание новых команд.

Читателям, заинтересовавшимся практическими аспектами проектирования DSL, рекомендуем обратиться к книге Мартина Фаулера (см. [4]).

3.5. Реализация DSL на OCaml/camlp4

Подход к реализации DSL с помощью camlp4 можно кратко описать так (см. 3.2):

- Так как синтаксис DSL — это синтаксис OCaml с добавлением собственных ключевых слов, синтаксический анализ программ на DSL выполняется camlp4 с помощью его стандартного синтаксического анализатора, без написания каких-либо дополнительных модулей.
- Для обработки полученного дерева абстрактного синтаксиса пишется программный модуль («транслятор AST»), который преобразует AST, порожденное camlp4, в AST более простой структуры (так как семантика DSL проще семантики OCaml).
- Дополнительно создаются программные модули-кодогенераторы, которые получают на вход упрощенное AST и превращают его в код программы на Erlang или Tcl.

Продемонстрируем синтаксис DSL на примере композиции двух операций сдвига курсора вправо. Интуитивно понятно, что, объединяя результаты сдвига вправо на `x_len` символов и сдвига вправо на `y_len` символов, можно сразу сдвинуть курсор вправо на `min x_len y_len` символов, а затем рассмотреть, как объединить оставшийся необработанным сдвиг на `abs (x_len - y_len)` символов и последующие команды. Соответственно, необходимо выяснить, какая из команд сдвига «короче», перенести ее в результат композиции, перейти к следующей команде в этом списке, а команду сдвига в другом списке «уменьшить» на нужное количество символов.

Код на DSL, который выполняет эти действия, выглядит так:

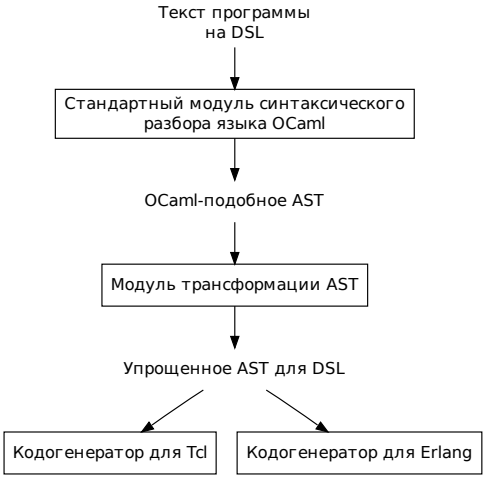


Рис. 3.2. Процесс

```
let compose_rules =
  make_compose (
    match y, x with
      | Retain y_len, Retain x_len →
          if x_len < y_len
          then {ys = Retain (y_len - x_len) :: yt;
                xs = xt;
                out = x}
          else if x_len > y_len
          then {ys = yt;
                xs = Retain (x_len - y_len) :: xt;
                out = y}
          else {ys = yt;
                xs = xt;
                out = x}
      | ...
```

Слово `make_compose`, которое хоть и выглядит как вызов функции, на самом деле является «точкой входа» для программы на DSL, и именно с нее `camlp4` начинает трансляцию AST программы (как это реализовано — см. ниже).

Как уже было сказано ранее, в результате трансляции кода на DSL порождается не самодостаточная программа, а программный модуль с заранее известным интерфейсом. Входные и выходные параметры этого модуля делаются доступными для кода на DSL с помощью переменных со специальными именами: `xs` и `ys` — это обрабатываемые списки команд редактирования, при этом `x`, `y`, `xt`, `yt` — это головы и хвосты этих списков. Результатом работы каждой ветки в `match` является запись (англ. record), в которой указываются новые значения `xs` и `ys`, и команда `out`, которую нужно доба-

вить к результату композиции.

Как выполняется трансляция программы на DSL? Утилита `camlp4` производит синтаксический анализ файла с программой на DSL, и обрабатывает полученное дерево абстрактного синтаксиса с помощью модуля «трансляции AST», который, упрощенно, выглядит так:

```
let translate_ast =
  Ast.map_expr
    (function
       | <:expr< make_compose $e$ >> → convert_compose e
       | <:expr< make_transform $e$ >> → convert_compose e
       | x → x
    )
in
  AstFilters.register_str_item_filter translate_ast#str_item
```

Этот код предписывает camlp4 вызывать для каждого узла AST, соответствующего ключевым словам `make_compose` или `make_transform`, функцию `convert_compose`, передавая ей в качестве аргумента AST выражения, «стоящего за» соответствующим ключевым словом.[3] В обоих случаях используется одна и та же функция `convert_compose`, так как в процессе разработки оказалось, что для описания операции композиции и включающего преобразования можно обойтись одним и тем же DSL и общей структурой AST.

Функция `convert_compose` выполняет рекурсивный спуск по AST, последовательно преобразовывая выражения, из которых состоит «тело» инструкции `match (y,x) with`:

```
let convert_compose =
  function
    | <:expr< match (y, x) with [ $cases$ ] >> →
    convert_compose_cases cases
    | _ → assert false

let rec convert_compose_cases =
  function
    | <:match_case@_loc< ($bpat$, $apat$) → $e$ | $rest$ >> →
      <:expr< [($convert_pat bpat$, $convert_pat apat$,
          $convert_expr e$) ::
         $convert_compose_cases rest$] >>
    | <:match_case@_loc< ($bpat$, $apat$) → $e$ >> →
      <:expr< [($convert_pat bpat$, $convert_pat apat$,
          $convert_expr e$) ::
         []] >>
    | _ → assert false
```

[3] Читателям, привыкшим к другим средствам метапрограммирования может помочь информация о том, что `<:expr <...>>` — это quotation, а `$...$` — это antiquotation.

```
let rec convert_expr =
  function
    | <:expr@_loc< if $e1$ then $e2$ else $e3$ >> →
      <:expr< PIf ($convert_expr e1$,
           $convert_expr e2$, $convert_expr e3$) >>
    | ...
```

То есть, выражение `match (y, x) with cs` заменяется на результат работы функции `convert_compose_cases`, которая, в свою очередь, последовательно преобразовывает все составные части выражения с помощью функций `convert_pat` и `convert_expr`.

В результате порождается упрощенное AST кода на DSL в виде списка вложенных друг в друга кортежей (англ. tuples). В частности, для примера, приведенного выше, будет порождено такое упрощенное AST (фрагмент):

```
let compose_rules =
  [ ((POp (PRetain (PIdent "y_len"))), (POp (PRetain (PIdent
      "x_len"))),
     (PIf
        (((PAp (PAp (PIdent "<", PIdent "x_len"), PIdent
            "y_len")),
          (PRes
             { rys = PAp (PAp (PIdent "::",
                       POp (PRetain (PAp (PAp (PIdent "-",
                                                PIdent "y_len"),
                                          PIdent "x_len")))),
                             PIdent "yt");
               rxs = PIdent "xt";
               rout = Some (PIdent "x");
               ry_out = None;
               rx_out = None;
               ry_depth = PIdent "y_depth";
               rx_depth = PIdent "x_depth";
             }),
          (PIf
             (((PAp (PAp (PIdent ">", PIdent "x_len"), PIdent
                 "y_len")),
             ...
```

Далее это упрощенное AST (являющееся результатом работы фукнции `translate_ast`) передается последовательно в функции-кодогенераторы.

Сам кодогенератор, фактически, сводится к сопоставлению с образцом очередного элемента упрощенного AST и порождению на его основе соответствующей конструкции целевого языка:

```
match ast_node with
  | PIf (e1, e2, e3) →
      Printf.sprintf "if\n%s →\n%s;\n%s\nend"
        (expr_to_string e1)
        (expr_to_string e2)
        (else_branch e3)
  ...
```

Порождаемый в результате код на Erlang выглядит так:

```
compose([{retain, YLen} = Y | Yt],
        [{retain, XLen} = X | Xt], Res) ->
  if
    XLen < YLen ->
      compose([{retain, YLen - XLen} | Yt], Xt, [X | Res]);
    XLen > YLen ->
      compose(Yt, [{retain, XLen - YLen} | Xt], [Y | Res]);
    true ->
      compose(Yt, Xt, [X | Res])
  end;
...
```

Код довольно сильно похож на оригинальную программу на DSL. Команды представляются в виде кортежей {имя_команды, параметр1, параметр2, …}, что позволяет использовать сопоставление с образцом для выбора нужного варианта композиции. Все конструкции DSL вида `if ... then ... else ...` преобразованы в Erlang-специфичный оператор `if`, допускающий наличие нескольких наборов «условие → действие». Добавление результата композиции к списку результатов реализовано с помощью хвостового рекурсивного вызова (англ. tail-recursive) функции `compose`.

Для сравнения, вот как выглядит соответствующий порожденный код на Tcl:

```
set res {}
set xi 0
set yi 0
set x ""
set y ""
while {1} {
    if {[llength $x] == 0} {set x [lindex $xs $xi]; incr xi}
    if {[llength $y] == 0} {set y [lindex $ys $yi]; incr yi}
    if {[llength $x] == 0} {set x "empty"}
    if {[llength $y] == 0} {set y "empty"}
    if {[lindex $y 0] eq "retain" && [lindex $x 0] eq "retain"} {
        set y_len [lindex $y 1]
        set x_len [lindex $x 1]
        if {$x_len < $y_len} {
            lappend res $x
            set y [list retain [expr {$y_len - $x_len}]]
            set x ""
```

```
        } elseif {$x_len > $y_len} {
            lappend res $y
            set y ""
            set x [list retain [expr {$x_len - $y_len}]]
        } else {
            lappend res $x
            set y ""
            set x ""
        }
    } elseif {...
```

Команды в коде на Tcl представлены в виде списков [«имя команды» «параметр 1» «параметр 2»]. Так как в Tcl отсутсвует возможность использовать сравнение с образцом, выбор нужного варианта композиции выполняется с помощью многоуровневой конструкции `if ... elseif ... elseif ...`. В списки Tcl эффективнее добавлять элементы справа (а не в начало списка), поэтому для формирования результата используется глобальная переменная-список `res`, к которой добавляются команды при помощи `lappend`. Кроме того, структурная рекурсия по спискам исходных команд работала бы в Tcl неэффективно, так как операция взятия «хвоста» списка является довольно дорогостоящей. В связи с этим для итерации по входным спискам используются переменные-индексы `xi` и `yi`.

Все это в сумме приводит к тому, что код на Tcl не отличается особой красотой и краткостью, т. к. манипуляции списками и индексами требуют довольно много вспомогательного кода. Однако, его не нужно писать вручную, так что на этот недостаток можно смело закрыть глаза.

3.6. Заключение

Итак, можно смело утверждать, что все поставленные в начале разработки цели были достигнуты:

Единое централизованное описание: код пишется только на DSL, нет необходимости синхронизировать реализации на Erlang и Tcl в ходе отладки или при изменении спецификаций ОП.

Унификация описания композиции и включающего преобразования: в обоих местах используется один и тот же DSL.

Абстрагирование от особенностей целевых языков: функция-кодогенератор инкапсулирует в себе все детали реализации на целевом языке, и программист, к примеру, не испытывает психологический дискомфорт от невозможности сравнивать данные с образцом в Tcl.

Возможность расширения подхода на другие целевые языки: для подключения еще одного целевого языка достаточно написать еще одну функцию-кодогенератор по образу и подобию уже существующих.

Стоила ли «овчинка выделки»? Может, объем вспомогательного кода в разы превысил объем полезного кода на Erlang и Tcl, и проще было бы не связываться с OCaml и DSL, и все-таки реализовать все вручную? Посмотрим на объемы кода программных модулей:

Программный модуль	Количество строк	Байт
Транслятор AST	175	6200
Кодогенераторы	600	11000
Программа на DSL	600	15500
Итого	**1375**	**32700**
Сгенерированный код на Erlang	500	20000
Сгенерированный код на Tcl	750	28000
Итого	**1250**	**48000**

Таблица 3.1. Объем кода программных модулей

С учетом того, какие преимущества при отладке и модификации кода дает DSL, становится понятно, что выбранный подход полностью себя оправдал несмотря на незначительную разницу в объеме вспомогательного и полезного кода. Кроме того, на момент завершения написания статьи стало известно, что ту же самую функциональность необходимо будет реализовывать на JavaScript, и тут использование DSL даст решающее преимущество.

Литература

[1] Abstract syntax tree. — Страница в Wikipedia (en), URL: http://en.wikipedia.org/wiki/Abstract_syntax_tree (дата обращения: 20 декабря 2009 г.).

[2] Camlp4 wiki. — Официальная документация по camlp4 (en), URL: http://brion.inria.fr/gallium/index.php/Camlp4 (дата обращения: 20 декабря 2009 г.).

[3] Concrete syntax tree. — Страница в Wikipedia (en), URL: http://en.wikipedia.org/wiki/Concrete_syntax_tree (дата обращения: 20 декабря 2009 г.).

[4] *Fowler M.* Domain Specific Languages, URL: http://martinfowler.com/dslwip/ (дата обращения: 20 декабря 2009 г.).

[5] Google wave operational transformation. — Официальная спецификация, URL: http://www.waveprotocol.org/whitepapers/operational-transform (дата обращения: 20 декабря 2009 г.).

[6] Google wave operational transformation, current draft. — Текущая рабочая версия, URL: http://www.waveprotocol.org/draft-protocol-specs/draft-protocol-spec (дата обращения: 20 декабря 2009 г.).

[7] Google wave: Under the hood. — Техническая презентация, URL: http://code.google.com/events/io/2009/sessions/GoogleWaveUnderTheHood.html (дата обращения: 20 декабря 2009 г.).

[8] Operational transformation. — Страница в Wikipedia (en), URL: http://en.wikipedia.org/wiki/Operational_transformation (дата обращения: 20 декабря 2009 г.).

[9] Операциональные преобразования. — Страница в Wikipedia (ru), URL: http://ru.wikipedia.org/?oldid=19384049 (дата обращения: 20 декабря 2009 г.).

Два взгляда на парадигму функционального программирования!

Овладейте практическими приёмами работы на наиболее известном языке функционального программирования!

Описание инструментальных средств разработки для языка Haskell:

- трансляторы,
- интегрированные среды разработки,
- специализированные библиотеки,
- вспомогательные утилиты.

Каждая глава посвящена отдельному классу программных средств.

Прилагается CD с инструментами для полноценной работы на языке Haskell.

288 страниц. **300 рублей**

Первая книга на русском языке о библиотеках языка Haskell.

Успешное применение Haskell на практике.

Описание синтаксиса языка.

Особые методы «правильного» программирования.

544 страницы. **300 рублей**

Интернет-магазин:
www.alians-kniga.ru

Книга почтой:
Россия, 123242, Москва, а/я 20
e-mail: books@alians-kniga.ru

Оптовая продажа:
«Альянс-книга»
Тел./факс: (495) 258-9195
e-mail: books@alians-kniga.ru

Полиморфизм в языке Haskell

Роман Душкин
darkus@fprog.ru

Аннотация

Статья предлагает к рассмотрению одно из мощнейших и перспективных средств программирования — *полиморфизм*, — на примере его использования в функциональном языке программирования Haskell. Описаны различные виды полиморфизма: *параметрический* со своими подвидами, а также *перегрузка имён* функций (так называемый «ad-hoc полиморфизм», или «специальный полиморфизм»).

Polymorphism is perceived to be one of the most powerful programming concepts. Various types of polymorphism are known: parametric, name overloading, ad-hoc *or* special, *to name a few. This article provides comprehensive description of all of them, with illustrations in Haskell.*

Обсуждение статьи ведется по адресу
http://community.livejournal.com/fprog/4987.html.

Введение

Статья продолжает цикл публикаций, посвящённых системе типов, принятой в функциональной парадигме программирования. Данный цикл начат в статье [18] во втором выпуске журнала.

Полиморфизм (от греч. πολύ — «много» и μορφή — «форма», «многообразный») в программировании — это возможность использования в одном и том же контексте различных программных сущностей (объектов, типов данных и т. д.) с одинаковым *интерфейсом*.

С самого начала применения понятия «полиморфизм» в информатике и программировании были теоретически обоснованы и разработаны различные виды полиморфизма. На рис. 4.1 приведена общая классификация видов полиморфизма, основанная на работах [4, 9, 13].

Изначально полиморфизм в языках программирования был неформально описан британским учёным К. Стрейчи в своих лекциях [11], после чего уже американский учёный области компьютерных наук Дж. Рейнольдс формально классифицировал полиморфизм на два больших типа [10]: *параметрический полиморфизм* и *ad-hoc полиморфизм* (специальный полиморфизм). Ранние работы Дж. Рейнольдса и французского логика Ж.-И. Жирара [5] ввели в научный оборот типизирован-

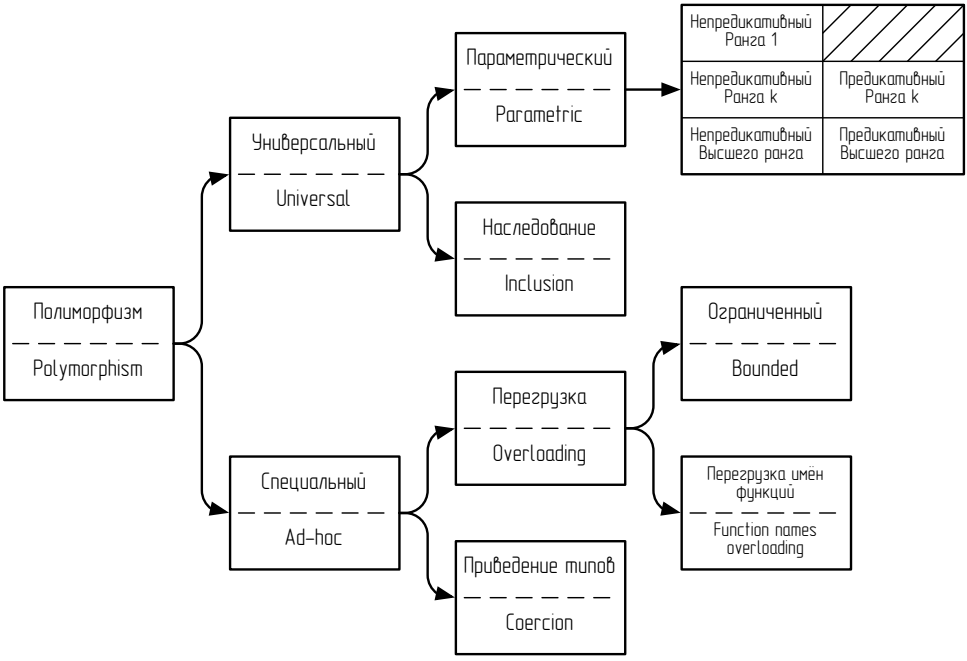


Рис. 4.1. Классификация видов полиморфизма в языках программирования

ное λ-исчисление второго порядка (так называемая «система F»). В дальнейшем формальная система F стала основой для использования параметрического полиморфизма в таких функциональных языках программирования, как Haskell и ML [9]. Наконец, голландский логик Х. П. Барендрегт, известный своими фундаментальными работами по λ-исчислению [16, 3], ввёл в научный оборот понятие λ-куба, при помощи которого структуризировал 8 систем типов, используемых как в теории, так и на практике [2].

Приведённым на диаграмме видам полиморфизма можно дать следующие упрощённые определения:

1) **Универсальный полиморфизм** (*universal polymorphism*) — в противоположность специальному полиморфизму, объединяет параметрический полиморфизм и наследование в один вид полиморфизма в соответствии с [4]. Мы вводим понятие «универсального полиморфизма» в дополнение к классификации полиморфизма, данной в лекциях [11].

   (a) **Параметрический полиморфизм** (*parametric polymorphism*) — это возможность определения обобщённых структур данных и функций, поведение которых не зависит от типов значений, которыми они оперируют. В случае типов данных (конкретнее, алгебраических типов данных, которые, как показано в [18], можно интерпретировать в качестве контейнерных типов) значения произвольных типов могут тем или иным образом использоваться внутри контейнеров (непосредственно содержаться в контейнерах, либо содержимое контейнеров будет иметь какую-либо зависимость от таких произвольных типов[1]). В случае функций именно поведение функции не зависит от типов передаваемых таким функциям значений в качестве входных параметров.

   Классификация параметрического полиморфизма основана на ограничении ранга полиморфизма и на ограничении использования *типовых переменных* (по аналогии с терминами «строковая переменная», «булевская переменная» и др.). Впрочем, параметрический полиморфизм может реализовываться и без использования типовых переменных в принципе.

      i. **Непредикативный полиморфизм** (*impredicative polymorphism*) — позволяет инстанцировать типовые переменные при конкретизации произвольными типами, в том числе и полиморфными.

      ii. **Предикативный полиморфизм** (*predicative polymorphism*) — в отличие от непредикативного полиморфизма инстанцирование типовых переменных при конкретизации типа может производиться только неполиморфными (мономорфными) типами, которые иногда называются «монотипами».

[1] В качестве примера параметризуемого алгебраического типа данных, в котором значения типа-параметра не содержатся, а используются иным образом, можно привести несколько надуманное, но имеющее смысл и право на существование определение `data Function a b = F (a → b)`.

iii. **Полиморфизм ранга** $*$ (*rank* $*$ *polymorphism*) — вместо символа подстановки $*$ могут использоваться значения «1», «k» и «N». В полиморфизме первого ранга (этот тип полиморфизма ещё называют «предварённым полиморфизмом» или «let-полиморфизмом») типовые переменные могут получать конкретные значения мономорфных типов. Полиморфизм ранга k предполагает, что в формулах, описывающих λ-термы, квантор всеобщности ($\forall$) может стоять не более чем перед k стрелками. Данный класс полиморфизма выделен потому, что при $k = 2$ проблема вывода типов разрешима, в то время как при $k > 2$ эта проблема неразрешима. Наконец, полиморфизм высшего ранга (или полиморфизм ранга N) определяется тем, что кванторы всеобщности могут стоять перед произвольным количеством стрелок.

(b) **Наследование** (*subtyping polymorphism* или *inclusion polymorphism*) — в объектно-ориентированном программировании классы (объекты) могут наследовать свойства классов-родителей так, что с точки зрения использования классы-потомки имеют те же самые наименования методов и свойств (а в случае, когда не используются перегруженные виртуальные методы, потомки имеют и те же самые реализации методов). В других парадигмах программирования под наследованием могут пониматься несколько иные средства языков программирования.

2) **Специальный (ad-hoc) полиморфизм** (*ad-hoc polymorphism*), который ещё называется полиморфизмом специального вида или «перегрузкой имён», позволяет давать одинаковые имена программным сущностям с различным поведением. Такой полиморфизм широко используется в математике, когда сходные математические операции получают одни и те же знаки (например, арифметические знаки (+), (–), (×) и (/) используются для обозначения операций сложения, вычитания, умножения и деления соответственно для произвольных чисел — целых, вещественных, комплексных и др.).

(a) **Перегрузка** (*overloading*) — объединяющее понятие, которое включает в себя ограниченный полиморфизм и перегрузку имён функций.

i. **Ограниченный полиморфизм** (*bounded polymorphism*) — регламентирует отношение «тип — подтип», когда ограниченно полиморфный тип должен быть подтипом некоторого более общего типа. В частном случае на типовые переменные накладываются ограничения, выглядящие как набор интерфейсных функций, которые должны быть определены для типов, потенциально участвующих в подстановке. Тем самым для полиморфного типа определяется набор функций, идентификаторы которых одинаковы для всех конкретных типов, которые могут быть подставлены в полиморфный тип при конкретизации. В функциональном программировании ограниченный полиморфизм часто используется совместно с параметрическим.

ii. **Перегрузка имён функций** (*function names overloading*) — перегрузка имён в смысле C++, когда разные функции с одинаковыми идентификаторами могут принимать разные наборы аргументов различных типов. Все такие функции должны быть определены до компиляции. Каждая такая функция при компиляции получает новый идентификатор, который зависит от количества и типов её аргументов.

(b) **Приведение типов** (*coercion*) — неявное приведение типов операндов при передаче их значений в функции. В языке C++ можно, например, складывать значения типов `int` и `float`, при этом значения типа `int` будут неявно преобразованы компилятором к типу `float` так, чтобы результат сложения также был этого типа.

Все эти виды полиморфизма широко используются в технологии программирования для повышения выразительности определений программных сущностей и создания обобщённых механизмов обработки данных. Тем не менее, далее в настоящей статье рассматриваются реализации отдельных видов полиморфизма в языке программирования Haskell, а именно в первом разделе изучается параметрический предикативный полиморфизм первого ранга, а во втором — ограниченный полиморфизм. Кроме того, для сравнения в третьем разделе приводится реализация некоторых видов полиморфизма на других языках программирования, в частности на языке C++.

4.1. Параметрический полиморфизм в языке Haskell

В функциональном программировании широко используется параметрический полиморфизм. Параметрический полиморфизм основан на передаче типов аргументов наряду с их значениями в виде параметров в функции и конструкторы (отсюда и атрибут «параметрический»). Реализация данного типа полиморфизма зачастую основана на типовых переменных, то есть таких переменных в сигнатурах определений функций и конструкторов типов, вместо которых можно подставлять произвольные типы. Типовые переменные повсеместно используются в практике функционального программирования в типизированных языках, к классу которых относится и рассматриваемый язык Haskell, поскольку такие переменные позволяют определять обобщённые типы и функции. Таким образом, понятно, что в функциональном программировании полиморфизм в целом относится к системе типов.

В качестве примера можно привести сигнатуры некоторых функций, работающих со списками значений:

```
reverse :: [α] → [α]
```

В этом примере функция `reverse` принимает на вход одно значение, которое имеет тип «список элементов типа α».

```
append :: [α] → [α] → [α]
```

Функция `append` принимает на вход уже два таких параметра. Здесь важно то, что оба входных параметра и результат, возвращаемый функцией, имеют один и тот же тип, а потому алгебраический тип данных «список» параметризуется одной и той же переменной.

```
zip :: [α] → [β] → [(α, β)]
```

Третья функция, `zip`, принимает на вход списки, элементы которых *могут иметь* различные типы. Результатом работы этой функции является список пар, первое значение в которых имеет тип такой же, как и у элементов первого списка, а второе значение имеет тип такой же, как и элементы второго списка. Само собой разумеется, что данные типы могут как совпадать, так и быть различны, на что указывает использование двух различных переменных типов — α и β.

Другими словами, для параметрического полиморфизма вводится формализация, которая позволяет связывать не только простые переменные, но и типовые переменные. Ранее была упомянута «система F», которая и предлагает такую формализацию. Данная система вводит дополнительную нотацию и семантику для λ-исчисления, при помощи которой вводятся типовые переменные. В этой нотации запись $\#x = \alpha$ следует читать как «значение x имеет тип α». Например, для тождественной функции $\lambda x.x$ запись с указанием типа α аргумента x выглядит следующим образом:

$$\#\Lambda\alpha.\lambda x^{\alpha}.x = \forall\alpha.\alpha \to \alpha \tag{4.1}$$

Данная запись обозначает, что в сигнатуру функции $\lambda x.x$ вводится типовая переменная α (данная переменная вводится при помощи символа (Λ), поскольку она определяет тип значений, то есть сущность более высокого порядка, нежели простые значения, переменные для которых вводятся при помощи символа (λ)). Вместо этой переменной α может быть подставлен любой конкретный тип данных при конкретизации функции, например, $(((\lambda x.x)_{Integer})1)$, где 1 — это элемент множества целых чисел. Здесь вместо типовой переменной α при вычислении результата будет подставлен тип $Integer$.

Квантор всеобщности ($\forall$) в формуле 4.1 обозначает, что тип α может быть любым, на него не накладывается никаких ограничений. Ограничения на используемые типы тесно связаны с ad-hoc полиморфизмом и будут рассмотрены в следующем разделе.

В языке Haskell используется непредикативный параметрический полиморфизм первого ранга (для стандарта Haskell-98 [6]). Специализированные расширения компилятора GHC позволяют использовать непредикативный параметрический полиморфизм высших рангов. Далее в примерах будет показан только полиморфизм, согласующийся со стандартом Haskell-98, а полиморфизм высших рангов и его реализация в языке Haskell будут рассмотрены в будущих статьях.

В качестве примеров определения полиморфных типов данных в языке Haskell можно рассмотреть определения различных структур, широко использующихся в программировании. Надо отметить, что ниже будут приводиться только определения самих типов данных, но не утилитарных функций для их обработки. Это необ-

ходимо уточнить, поскольку некоторые определения могут отличаться друг от друга только наименованиями конструкторов типов, но способы обработки этих типов определяются именно в утилитарных функциях. Это не должно смущать, поскольку в языке Haskell определение типа и определения функций для его обработки отделены друг от друга в отличие от объектно-ориентированных языков, где функции-методы классов связаны с классами непосредственно.

Простой список значений произвольного типа определяется просто (необходимо напомнить, что в стандарте Haskell-98 определена специальная синтаксическая форма для удобной работы со списком):

```
data List α = Nil
            | Cons α (List α)
```

Здесь типовая переменная α может принимать произвольное значение. При её инстанцировании она может быть замещена любым другим типом данных, но стандарт Haskell-98 предполагает, что этот тип данных уже будет мономорфным. Не должно вводить в заблуждение то, что в языке Haskell можно обрабатывать списки списков, списки деревьев, списки функций, списки действий ввода-вывода; при этом уровень вложенности может быть не ограничен, так что можно обрабатывать и списки списков списков и т. д. Такое состояние дел относится к любому полиморфному типу данных, в определении которого используются типовые переменные.

Дальнейшие примеры. Двусвязный список можно определить так:

```
data LList α = Nil
             | LCons (LList α) α (LList α)
```

Данное определение с точностью до наименования конструкторов и порядка следования элементов декартова произведения в конструкторе `LCons` совпадает с определением двоичного дерева:

```
data Tree α = Nil
            | Node α (Tree α) (Tree α)
```

Как уже упомянуто, конкретные способы обработки значений данных типов определяется утилитарными функциями. Тот же тип `Tree` можно использовать и для представления двусвязного списка, главное, чтобы функции для его создания и обработки использовали определённую семантику. Также тип `Tree` вполне подходит для представления различных видов двоичных деревьев — красно-чёрных и других видов сбалансированных деревьев и т. д. Впрочем, в целях разделения компонентов использования в различных готовых библиотеках и стандартных модулях различные виды деревьев могут реализовываться при помощи различных типов данных.

Также имеет смысл отметить, что реализация двусвязного списка в языке Haskell должна быть ленивой. Это связано с тем, что конструктор этой структуры данных вполне может замкнуть двусвязный список в «кольцо» так, что ни начала, ни конца уже будет не найти. Получится структура, аналогичная бесконечному односвязному

списку `List`, а бесконечные структуры данных всегда должны обрабатываться лениво. Детально о методиках обработки таких структур данных в языке Haskell можно ознакомиться на официальном сайте языка [12].

Ассоциативный массив может быть реализован как в виде специального вида списка, так и в виде отдельного типа данных (впрочем, отдельный тип данных в этом примере использовать нецелесообразно):

```
type AArray α β = [(α, β)]

data AArray α β = ANil
                | AArray α β (Aarray α β)
```

Дерево произвольной степени ветвления определяется примерно так же, как и двоичное:

```
data ATree α = ANil
             | ANode α [ATree α]
```

Определение дерева произвольной размерности с помеченными вершинами и дугами уже несколько сложнее, поскольку где-то необходимо хранить пометки дуг:

```
data MTree α β = MTree α [MEdge α β]

data MEdge α β = MEdge β (MTree α β)
```

Наконец, список значений двух различных типов, при этом на нечётных позициях находятся значения первого типа, а на чётных значения второго типа (так называемая «верёвка», которая используется даже в стандартной библиотеке C++):

```
data Rope α β = Nil
              | Twisted α (Rope β α)
```

Может показаться, что в языке Haskell используется полиморфизм второго ранга. Если в качестве типовых переменных подставляются такие же полиморфные типы, то, вполне вероятно, могут возникать такие примеры, как список двоичных деревьев, который определяется следующим образом:

```
type ListOfTrees α = [Tree α]
```

Либо дерево произвольной размерности, в узлах которого находятся списки:

```
type TreeOfLists α = ATree [α]
```

В этих случаях при инстанцировании типовой переменной α происходит подстановка не полиморфного, а мономорфного типа `Tree` α или `[`α`]` соответственно, при этом типовая переменная α уже связана полученным инстанцированием контейнерного полиморфного типа.

Из вышеприведенных примеров видно, что определение некоторого типа в языке Haskell может быть использовано для реализации различных *структур данных*. Семантика использования типа не связывается с его определением, что позволяет выводить сами определения типов данных на более высокий уровень абстракции, чем это

имеет место в объектно-ориентированных языках программирования при использовании механизма наследования.

Для закрепления материала осталось дать примеры определений утилитарных функций, работающих с подобными полиморфными типами данных. Далее рассматриваются определения функций, сигнатуры которых приведены в начале статьи — `reverse`, `append` и `zip`.

```
reverse :: [α] → [α]
reverse [] = []
reverse (x:xs) = append (reverse xs) [x]
```

Как видно из этого примера, функция `reverse` совершенно не принимает во внимание тип значений, которые хранятся в передаваемом ей на вход списке. Эти значений могут быть произвольного типа, функции `reverse` абсолютно всё равно, какой список обращать. Всё, что она делает, это разбивает входной список на элементы и соединяет их в обратном порядке, используя для этого функцию для конкатенации двух списков `append`, определение которой выглядит так:

```
append :: [α] → [α] → [α]
append []     l = l
append (x:xs) l = x:(append xs l)
```

Функция `append` также не обращает внимания на типы значений во входных списках. Главное, чтобы они были одинаковыми, поскольку результирующий список должен состоять из элементов обоих входных списков. Одинаковость типов значений в двух входных и результирующем списках определяется тем, что в сигнатуре функции указана одна типвая переменная α. В качестве примера использования нескольких типовых переменных можно привести определение следующей функции:

```
zip :: [α] → [β] → [(α, β)]
zip (x:xs) (y:ys) = (x, y) : (zip xs ys)
zip   _      _    = []
```

В данном определении типы значений в первом и во втором входных списках могут уже отличаться, но сама функция опять же не обращает внимания на сами эти типы. Она оперирует значениями, независимо от их типов.

Таким образом, видно, что параметрический полиморфизм позволяет определять типы и функции, которые реализуют обобщённые алгоритмы обработки структур данных, не зависящие от конкретных типов обрабатываемых значений. Данная техника позволяет разработчику программного обеспечения рассматривать код на более высоком уровне абстракции, не углубляясь в принципы обработки конкретных типов. В совокупности с техникой разработки сверху вниз [15] параметрический полиморфизм позволяет разработчику существенно повысить эффективность своей работы.

4.2. Ad-hoc полиморфизм в языке Haskell

Теперь можно рассмотреть реализацию специального полиморфизма в языке Haskell. В этом языке реализация полностью основана на понятии «связанного» или «ограниченного» полиморфизма (англ. *bounded*). В этом виде полиморфизма требуется, чтобы типы обрабатываемых значений соответствовали некоторому указанному интерфейсу, который задаётся как набор функций с сигнатурами.

Впервые на подобный вид полиморфизма указали в своей работе [4] Л. Карделли и П. Вегнер. Причина введения ограниченного полиморфизма была в том, что некоторые функции требуют от используемых в них типов значений наличия определённой семантики, реализуемой посредством специальных функций из *интерфейса*. Подобные ситуации с требованиями к типам данных возникают постоянно в различных задачах.

Например, для того чтобы понять, входит ли заданное значение в список (предикат `isElementOf`), необходимо, чтобы значения данного типа имели возможность сравнения друг с другом. Также и функция `sum`, которая складывает все значения в заданном списке, должна получать полную гарантию того, что со значениями в переданном на вход списке можно совершать арифметическую операцию сложения (`+`).

В языке Haskell для этих целей используются классы типов.

Класс типов представляет собой интерфейс, то есть набор сигнатур функций без определений.[2] В определениях алгебраических типов данных и сигнатурах функций можно указывать ограничения на используемые типовые переменные (именно об этом было упомянуто в статье [18] при рассмотрении структуры определения алгебраических типов). Такие ограничения означают, что соответствующая переменная типов должна инстанциироваться только такими типами, для которых реализованы функции класса.

Предлагается рассмотреть такой пример. Кроме обычной аристотелевой логики с двумя значениями истинности, в математике разработаны альтернативные логики; например, многозначные логики Я. Лукасевича [7] или бесконечнозначная нечёткая логика Л. А. Заде [14]. Несмотря на разнообразие построенных логических теорий, все они должны отвечать одинаковым фундаментальным требованиям. Это и есть их «интерфейс». К нему относятся базисные логические операции — обычно это операции отрицания, конъюнкции и дизъюнкции. На языке Haskell этот факт описывается следующим образом:

```
class Logic α where
  not  :: α → α
  (&&) :: α → α → α
  (||) :: α → α → α
```

[2] Здесь необходимо дополнительно отметить, что в языке Haskell можно задавать определения интерфейсных функций, используемые по умолчанию. Такие определения будут использованы тогда, когда для некоторого типа данных определения функций опущены.

Этот класс, как было уже сказано, можно использовать для описания ограничений на типы данных в сигнатурах функций. Например, вот так выглядела бы функция для проверки правила де Моргана для заданных логических значений:

```
test_deMorgan :: (Eq α, Logic α) ⇒ α → α → Bool
test_deMorgan x y = (not (x && y)) == ((not x) || (not y))
```

Теперь значения любого типа данных, который может представлять логические значения истинности и допускает сравнение значений на равенство, могут быть переданы на вход функции `test_deMorgan` для проверки. Конечно, данная функция проверяет правило де Моргана только для двух конкретных значений, а не для всей области определения логических значений истинности, принятой в конкретной логической теории, но на данном примере можно понять смысл ad-hoc полиморфизма в языке Haskell.

Итак, сигнатура функции `test_deMorgan` требует, чтобы для типовой переменной α были реализованы функции `not`, `(&&)` и `(||)` — это требование записывается как `Logic` α в *контексте* сигнатуры (другое ограничение, `Eq` α, требует наличия операций сравнения для типа α). Разработчик может использовать три упомянутые операции без всяких сомнений — их наличие для типа α гарантировано. Если эти операции не определены, то код просто не скомпилируется.

Как же использовать новый класс и утилитарную функцию `test_deMorgan` к нему? Для этого необходимо определить так называемый *экземпляр класса* для заданных типов данных, для которых требуется исполнения указанного интерфейса. Первым типом, для которого необходимо определение экземпляра класса `Logic`, является тип `Bool`. Экземпляр определяется следующим образом:

```
instance Logic Bool where
  not False = True
  not True  = False

  True && True = True
  _    && _    = False

  False || False = False
  _     || _     = True
```

Здесь видно, что для типа `Bool` определяются уже конкретные реализации интерфейсных функций, описанных в классе `Logic`. При вызове нашей функции

```
> test_deMorgan True True
True

> and $ map (uncurry · test_deMorgan) [(x, y) | x ←
[True, False], y ← [True, False]]
True
```

автоматически выбираются конкретизированные функции `(&&)` и `(||)`, реализованные для типа `Bool`. Заинтересованные читатели могут самостоятельно попытаться

реализовать экземпляры класса `Logic` для типов, представляющих другие виды значений истинности (для этого также придётся определить и сами типы данных, представляющие значения альтернативных логик).

В этом и проявляется ad-hoc полиморфизм в языке Haskell. Класс `Logic` может быть связан со многими типами данных, но для всех них будут использоваться методы классов с абсолютно одинаковыми наименованиями. Перегрузка идентификаторов функций налицо.

Абсолютно таким же образом в языке Haskell реализованы арифметические операции и предикаты сравнения величин, а также многое другое. В стандартном модуле `Prelude` описано большое число классов, которые могут использоваться в самых разнообразных задачах. В качестве дополнительного примера можно рассмотреть, каким образом определён класс типов для величин, над которыми можно совершать арифметические операции. Это сделано так:

```
class  (Eq α, Show α) ⇒ Num α  where
  (+)         :: α → α → α
  (-)         :: α → α → α
  (×)         :: α → α → α
  negate      :: α → α
  abs         :: α → α
  signum      :: α → α
  fromInteger :: Integer → α

  x - y    = x + negate y
  negate x = 0 - x
```

Как видно, ограничения на типовые переменные могут находиться не только в сигнатурах функций, но и в определениях классов и типов. Здесь приведены два ограничения — тип α должен иметь функции для сравнения величин (класс `Eq`) и функции для преобразования величин в строку (класс `Show`). Далее приводятся сигнатуры семи функций (в том числе трёх инфиксных операций), которые должны быть определены для любого типа, который будет удовлетворять требованиям ограничений на возможность производить арифметические операции. Чтобы минимизировать количество определений конкретизированных функций, можно выражать одни интерфейсные функции через другие. Так, разность выражается через сложение с отрицанием, а отрицание выражается через разность. Само собой разумеется, что при определении экземпляра необходимо реализовать либо метод `negate`, либо операцию `(-)` — что-то одно выражается через другое. Например, можно определить тип для представления комплексных чисел, над которыми определены арифместические операции, после чего определить для этого типа экземпляр класса `Num`. В этом случае к значениям типа для комплексных чисел можно будет применять все перечисленные ранее интерфейсные функции из класса `Num`. Это делается следующим образом:

```
data Num α ⇒ Complex α = Complex α α
```

```
instance Num α ⇒ Num (Complex α) where
  (Complex x1 y1) + (Complex x2 y2) = Complex (x1 + x2) (y1 + y2)
  (Complex x1 y1) - (Complex x2 y2) = Complex (x1 - x2) (y1 - y2)
  ...
```

Может показаться, что понятие класса в функциональном программировании соответствует понятию интерфейс в программировании объектно-ориентированном. Действительно, можно провести некоторые аналогии, хотя имеются и серьёзные отличия. Подробно о подобии и различии между классами типов и интерфейсами можно ознакомиться в книге [19] и статье [17], а также на официальном сайте языка Haskell [8]. Также дополнительно о проблемах, которые могут возникать при использовании специального полиморфизма, можно ознакомится в статье [13].

4.3. Полиморфизм в других языках программирования

Б. Страуструп, автор языка C++, назвал полиморфизм одним из четырёх «столпов объектно-ориентированного программирования» [21] (другие столпы — абстракция, инкапсуляция и наследование).[3] Все современные объектно-ориентированные языки программирования реализуют полиморфизм в том или ином виде. Ниже будут приведены более конкретные примеры.

В качестве примеров можно рассмотреть, как определяются полиморфные программные сущности на языке программирования C++.

Начать рассмотрение примеров можно с ad-hoc полиморфизма, как наиболее широко распространённой техники в языке C++. С одной стороны имеет место перегрузка имён функций, когда функции, получающие на вход значения различных типов, могут иметь одинаковые наименования. Этот случай не очень интересен в рамках настоящей статьи, поскольку на самом деле здесь имеет место только поверхностное, внешнее проявление ad-hoc полиморфизма — функции имеют одинаковые наименования только для разработчика. Транслятор языка преобразует такие одинаковые имена функций во внутреннее представление, в котором учитываются и типы получаемых на вход параметров.

Наследование (как один из подвидов универсального полиморфизма) в языке C++ проявляется при помощи соответствующего механизма для классов (необходимо напомнить, что в объектно-ориентированном программировании под классами понимается иная сущность, чем в функциональном программировании). Классы-потомки могут перекрывать методы классов-родителей,[4] при этом во всех классах методы имеют одинаковое наименование. В данном случае, конечно же, транслятор языка также

[3] Б. Страуструп в этом определении имел в виду именно специальный полиморфизм, который был изначально реализован в языке C++. Как уже показано, специальный полиморфизм — это не единственный вид полиморфизма, а потому создатель языка C++ говорил о более узком понятии, чем то, которое описывается в настоящей статье.

[4] Также надо отметить, что в определённых объектно-ориентированных языках программирования для некоторых видов определений перекрытие методов классов-родителей в классах-потомках обязательно.

имеет возможность различать такие методы при помощи пространства имён, но сам по себе такой полиморфизм представляет для разработчика абстракцию более высокого уровня, нежели простая перегрузка имён функций.

Приводить примеры определения иерархии классов в языке C++ смысла нет — читатели могут найти их в любом учебнике по этому языку или какому-либо подобному языку программирования. Более интересным является использование в языке C++ так называемых *шаблонов*. Вся библиотека STL для языка C++ реализована при помощи этого механизма, что и немудрено, поскольку большинство типов в этой библиотеке являются контейнерными, а от контейнерных типов естественно ожидать возможность хранить внутри себя значения произвольных типов. Шаблоны могли бы стать для языка C++ средством реализации именно параметрического полиморфизма, поскольку в них используется обычное для такого типа полиморфизма понятие *типовой переменной*. Однако, к сожалению, шаблоны стали лишь «синтаксическим сахаром» для сокращения исходного кода при определении одинаковых функций, работающих с различными типами данных, поскольку компилятором языка все шаблоны «разворачиваются» в многочисленные определения программных сущностей по одной для каждого использованного в исходном коде типа. Для более детального понимания, что это такое, можно рассмотреть несколько примеров.

Например, вот как определяется в библиотеке STL тип «двусвязный список»:

```
template<class T> class list;
```

Здесь идентификатор `T` используется в качестве типовой переменной, которая определяет тип значений, которые будут храниться внутри таких двусвязных списков. Второй параметр (типовая переменная) шаблона `Allocator` в рассмотрении настоящей статьи не важен, так как относится к модели управления памятью в языке C++. Определение методов шаблонного класса `list` не принимает во внимание действительный тип значений, которые хранятся в списке, но оперируют именно типовой переменной `T`. Это позволяет хранить в списке значения произвольного типа, абсолютно также, как это происходит и для функциональных языков — при определении такого списка в нём специфицируется конкретный тип хранимых значений (определяется двусвязный список целых чисел):

```
std::list<int> primes;
```

В качестве примера применения различных видов полиморфизма в разных парадигмах программирования для решения прикладных задач можно рассмотреть достаточно примитивную, но в целом показательную задачу получения суммы заданного списка. Пусть есть список целых чисел, определённый примерно так, как сделано в примере выше. Необходима функция, которая, получив на вход такой список, вернёт сумму его элементов. Для языка C++ задача тривиальна:

```
int sum (std::list<int> iList) {
  int result = 0;
  for (std::list<int>::iterator i = iList.begin (); i !=
      iList.end (); ++i) {
```

```
    result += *i;
  }
  return result;
}
```

А как быть, если необходима функция, которая возвращает сумму вещественных чисел? Её определение практически идентично приведённому ранее:

```
float sum (std::list<float> iList) {
  float result = 0.0;
  for (std::list<float>::iterator i = iList.begin (); i !=
      iList.end (); ++i) {
    result += *i;
  }
  return result;
}
```

Как видно, обе функции различаются только типом значений — значений элементов входного списка и возвращаемого значения. Более того, в определении этих двух функций уже используется полиморфизм операции (+) в языке C++. Это значит, что, в принципе, можно написать обобщённую функцию для сложения значений произвольных типов — главное, чтобы для них была определена операция сложения (ну и заодно должно быть определено начальное нулевое значение). На помощь приходят шаблоны:

```
template<class T> T sum (std::list<T> iList) {
  T result = 0;
  for (std::list<T>::iterator i = iList.begin (); i != iList.end
      (); ++i) {
    result += *i;
  }
  return result;
}
```

Главная проблема, которая возникает при использовании такой функции, заключается в том, что необходимо каким-то образом контролировать наличие определённой операции сложения для типа `T`. В принципе, компилятор языка поможет в этом вопросе, выдав сообщения об ошибках в случаях, когда необходимые определения отсутствуют, но тем не менее разработчик должен помнить, что необходимо реализовать требуеме операции. Но как быть в случае, например, необходимости написания функции, которая «сворачивает» заданный список в конечное значение при помощи переданной на вход бинарной операции? Здесь уже надо немного исхитриться:

```
template<class T> typedef T (*binary) (T, T);

template<class T> T foldl (list<T> iList, T zero, binary<T> op) {
  T result = zero;
```

```
  for (std::list<T>::iterator i = iList.begin (); i != iList.end
      (); ++i) {
    result = (*op) (result, *i);
  }
  return result;
}
```

Таким образом, шаблоны в языке C++ представляют собой удобное средство краткого описания многочисленных определений. И хотя, как уже сказано в языке C++ шаблоны, которые могли бы стать проявлениями параметрического полиморфизма в полной мере, компилятором преобразуются в наборы функций с перегруженными именами, для разработчика внешне такие шаблоны являются одним из способов реализации именно параметрического полиморфизма. Также в дополнение к библиотеке STL можно рекомендовать к изучению библиотеку Boost, в которой реализовано множество функциональных алгоритмов и контейнерных типов данных. Для изучения методов метапрограммирования на языке C++ при помощи шаблонов рекомендуется книга [1].

Ту же самую задачу, что представлена выше, но в более правильном свете, решают так называемые «генерики» (англ. *generics*) в языке Java или C#. Заинтересованный читатель может попытаться решить представленную задачу на языках, которые он использует в своей практике. Автор и редакция журнала будут благодарны читателям, которые пришлют свои решения. Лучшие решения будут опубликованы на официальном web-сайте журнала.

Теперь для сравнения можно рассмотреть те же самые примеры в реализации на языке Haskell. Ниже перечислены функции, которые осуществляют сложение значений из заданного списка целых чисел, вещественных чисел, произвольных значений, для которых определена операция сложения, а также функция свёртки заданного списка[5]:

```
sum_int :: [Int] → Int
sum_int l = sum_int' l 0
  where
    sum_int' []    r = r
    sum_int'(i:is) r = sum_int' is (r + i)

sum_float :: [Float] → Float
sum_float l = sum_float' l 0
  where
    sum_float' []    r = r
    sum_float'(f:fs) r = sum_float' fs (r + f)
```

[5]Необходимо отметить, что свёртка — это широко используемая идиома в функциональном программировании, причём определяемая не только для списков, но и в общем для произвольных рекурсивных типов данных. При помощи свёртки списка можно выразить очень многие функции над списком, результатом которых является одиночное значение. Например, сумма элементов списка `sum` может быть выражена через свёртку как `sum = foldl (+)0`.

```
sum :: Num α ⇒ [α] → α
sum []     = error "No elements in input list."
sum (x:xs) = sum' xs x
  where
    sum' []    r = r
    sum'(y:ys) r = sum' ys (r + y)

foldl :: (α → β → α) → α → [β] → α
foldl f z l = foldl' l z
  where
    foldl' [] r     =  r
    foldl' (x:xs) r = foldl' xs (f r x)
```

Несколько моментов требуют пояснения:

1) Во всех вышеперечисленных определениях используется идиома аккумулятора (накапливающего параметра), который реализован через определение локальной функции (часть после ключевого слова `where`). Данная технология позволяет выполнять вычисления в постоянном объёме памяти несмотря на рекурсию. Аккумулятором называется один из входных параметров локальной функции (в данных примерах — `r`), в котором накапливается результат вычислений.

2) Ограничение `Num` α в сигнатуре функции `sum` поразумевает, что для типа α определены арифметические операции, в том числе и операция (`+`) (как это было показано в предыдущем разделе). К сожалению, этот класс не определяет нулевого элемента в типе (это делает класс `Monoid`, детальное рассмотрение которого приведено в статье [20]), поэтому функции `sum` приходится использовать в качестве нулевого элемента голову списка. С этим и связано то, что для пустого списка функция не определена (при попытке такого вызова выводится сообщение об ошибке).

3) Функция `foldl` определена в стандартном модуле `Prelude` (там её определение несколько отличается, здесь форма определения видоизменена для единообразия). Операция свёртки широко используется в обработке списков. Более того, определение свёртки можно расширить для произвольного рекурсивного типа данных.

Заключение

Итак, в настоящей статье изучены реализации отдельных видов полиморфизма в языке функционального программирования Haskell, а именно параметрический предикативный полиморфизм первого ранга и ограниченный полиморфизм. Данные виды полиморфизма позволяют красиво и эффективно решать многие задачи, однако имеется целый ряд проблем, которые не могут быть решены при помощи представленных видов полиморфизма.

Одной из таких проблем является хранение в алгебраических типах данных значений произвольных типов. Например, список может содержать значения не только одного конкретного типа (список целых чисел, список строк, список деревьев с размеченными дугами и т. д.), но и произвольный набор произвольных значений. В языке Lisp такой список является естественной структурой данных, однако язык Haskell стандарта Haskell-98 с его строгой типизацией не позволяет создавать подобные структуры.

Как это можно сделать при помощи полиморфизма высших рангов, а также реализация такого вида полиморфизма в одном из расширений языка Haskell — предмет рассмотрения одной из будущих статей.

Литература

[1] *Abrahams D., Gurtovoy A.* C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). — Addison-Wesley Professional, 2004.

[2] *Barendregt H. P.* Introduction to generalized type systems // *J. Funct. Program.* — 1991. — Vol. 1, no. 2. — Pp. 125–154.

[3] *Barendregt H. P.* Lambda calculi with types. — 1992. — Pp. 117–309.

[4] *Cardelli L., Wegner P.* On understanding types, data abstraction, and polymorphism // *ACM Computing Surveys.* — 1985. — Vol. 17. — Pp. 471–522.

[5] *Girard J.-Y., Lafont Y., Taylor P.* Proofs and Types. — Cambridge University Press, 1989. — Vol. 7 of *Cambridge Tracts in Theoretical Computer Science.*

[6] *Jones S. P. et al.* Haskell 98 Language and Libraries. The Revised Report. — Academic Press, 2002.

[7] *Łukasiewicz J.* Aristotle's Syllogistic from the Standpoint of Modern Formal Logic. — Oxford University Press, 1987.

[8] Oop vs type classes. — Статья в Haskell Wiki (en), URL: http://www.haskell.org/haskellwiki/OOP_vs_type_classes (дата обращения: 20 декабря 2009 г.).

[9] *Pierce B. C.* Types and Programming Languages. — MIT Press, 2002. — (имеется перевод книги на русский язык URL: http://newstar.rinet.ru/~goga/tapl/ (дата обращения: 20 декабря 2009 г.)). http://www.cis.upenn.edu/~bcpierce/tapl.

[10] *Reynolds J. C.* Theories of programming languages. — New York, NY, USA: Cambridge University Press, 1999.

[11] *Strachey C.* Fundamental concepts in programming languages // *Higher Order Symbol. Comput.* — 2000. — Vol. 13, no. 1-2. — Pp. 11–49.

[12] Tying the knot. — Статья в Haskell Wiki (en), URL: http://www.haskell.org/haskellwiki/Tying_the_Knot (дата обращения: 20 декабря 2009 г.).

[13] *Wadler P.* Theorems for free // Functional Programming Languages and Computer Arcitecture. — ACM Press, 1989. — Pp. 347–359.

[14] *Zadeh L. A.* Fuzzy sets and systems // Fuzzy Sets Fuzzy Logic and Fuzzy Systems: Selected Papers by Lofti A. Zadeh, Advances in Fuzzy Systems-Application and Theory Vol 6, World Scientific. — 1965. — Pp. 35–43.

[15] *Астапов Д. Е.* Давно не брал я в руки шашек // «*Практика функционального программирования*». — 2009. — Т. 1, № 1. — С. 51–76.

[16] *Барендрегт Х. П.* Ламбда-исчисление. Его синтаксис и семантика. — М.: Мир, 1985. — 606 с.

[17] *Душкин Р. В.* Объектно-ориентированное и функциональное программирование // «*Потенциал*». — 2007. — Т. 26, № 2. — С. 40–50.

[18] *Душкин Р. В.* Алгебраические типы данных и их использование в программировании // «*Практика функционального программирования*». — 2009. — Т. 2, № 2. — С. 86–105.

[19] *Душкин Р. В.* Функциональное программирование на языке haskell. — 2007.

[20] *Пипони Д.* Моноиды в Haskell и их использование // «*Практика функционального программирования*». — 2009. — Т. 1, № 1. — С. 77–86. — «Haskell Monoids and their Uses», пер. с англ. К. В. Заборского.

[21] *Страуструп Б.* Язык программирования C++. — М.: Бином, 1999. — 991 с.

Элементы функциональных языков

Евгений Кирпичёв
jkff@fprog.ru

Аннотация

В этой статье описаны основные концепции, характерные для различных функциональных языков, такие как алгебраические типы данных, замыкания или бесточечный стиль. В статье приведена их история возникновения и развития, объясняется сущность и разновидности, описаны практические примеры применения, а также способы имитации в языках без встроенной поддержки соответствующей концепции.

Статья адресована в первую очередь профессиональным программистам, работающим с «общепринятыми» языками. Цель статьи — вооружить идеями из мира функционального программирования даже тех читателей, кто не планирует менять основной язык разработки.

This article provides a comprehensive description of all the basic concepts usually attributed to functional languages, such as algebraic data types, closures or point-free style. Historical roots of all concepts are explained, their essence is illustrated, typical usage is displayed along with ways of imitating the concepts in languages that do not support them natively.

The article is targeted at professional programmers using mainstream programming languages, even if they do not plan to switch to functional languages. We aim to equip them with ideas from the world of functional programming.

Обсуждение статьи ведется по адресу
http://community.livejournal.com/fprog/5223.html.

5.1. Введение

Профессиональным программистам, раздумывающим о том, стоит ли изучать функциональные языки, зачастую бывает трудно оценить количество и ценность идей, в них присутствующих. Литература о функциональных языках нередко оказывается слишком поверхностной и показывает решение простейших задач, что не производит впечатления на опытного читателя. Другая крайность — когда, напротив, иллюстрируются интересные подходы к решению нетривиальных задач, которые оказываются настолько сложными и требуют столько знаний о функциональном подходе, что неподготовленный читатель оказывается не в состоянии оценить важность описанных в такой работе идей.

Данная статья ставит себе целью построить «мостик» между этими двумя разновидностями учебных материалов. В ней будут кратко описаны наиболее важные концепции из функционального программирования, с акцентом на перспективы их практического применения или имитации в нефункциональных языках. Таким образом, неподготовленный читатель сможет окинуть взором богатство имеющихся в функциональном программировании идей, понять, насколько они могут быть ему интересны и полезны, и, возможно, продолжить изучение интересной области. Подготовленный же читатель найдет в статьях на знакомые ему темы множество отсылок к литературе о более сложных проблемах, связанных с описываемыми концепциями — а темы некоторых статей, возможно, окажутся для него новыми.

В статье намеренно не затронуты две чрезвычайно важных темы: типовый полиморфизм, а также ленивые вычисления. Обе темы настолько широки, что достойное их рассмотрение увеличило бы и без того немалый размер статьи в полтора–два раза, а рассмотреть их кратко — значило бы лишить читателя удовольствия и обмануть относительно многогранности и красоты их применений. Эти темы будут рассмотрены в ближайших номерах. Кроме того, в данном номере опубликована статья Романа Душкина о полиморфизме в языке Haskell: она прекрасно подойдет для подготовки к знакомству со всем многообразием проявлений полиморфизма в программировании.

Структура всех глав примерно одинакова и состоит из следующих пунктов:

1) **Суть**: описание концепции, изложенное в одном предложении.

2) Краткая **история**, предпосылки и контекст возникновения концепции.

3) **Интуиция**: пример, подготавливающий читателя к восприятию концепции; в идеале при прочтении этой секции читатель должен изобрести концепцию самостоятельно :)

4) Более или менее подробное **описание** сущности концепции; однако, все сложные или длинные объяснения заменены ссылками на внешние источники.

5) **Применение**: задачи, для решения которых концепция может оказаться полезной, а также указание на существующие программы, использующие данную концепцию.

6) **Реализации**: перечисление языков, инструментов, библиотек, реализующих данную концепцию.

7) **Схожие концепции** из других парадигм (императивной, логической, объектно-ориентированной) и языков, по сути своей схожие с данной, а также способы реализации или имитации данной концепции в рамках этих парадигм и языков.

5.2. Вывод типов (Type inference)

Суть

Синтаксическая структура программы позволяет составить систему уравнений относительно типов ее частей, автоматическое решение которой избавляет программиста от необходимости явно указывать типы.

История

Вывод типов также называется «неявной типизацией», или же «типизацией по Карри», в честь логика Хаскелла Карри. Роджер Хиндли в письме [52] утверждает, что алгоритм вывода типов при помощи унификации множество раз переоткрывался, в основном в период 1950—1960-х гг, или даже 1920—1930-х гг. Возможно, это объясняется чрезвычайной простотой, элегантностью и универсальностью общей схемы алгоритма: вероятно, авторы работали в разных областях математики и информатики, не знали о работах друг друга и не считали нужным публиковать данный алгоритм отдельно.

Интуиция

Рассмотрим написанную на псевдокоде программу для вычисления скалярного произведения двух двумерных векторов:

```
dot(v1,v2) = v1[0]*v2[0] + v1[1]*v2[1]
```

Очевидно, что на входе этой программы — два числовых массива, а на выходе — число. Посмотрим, какие рассуждения могут привести к такому результату:

- Выполняется обращение к v1[0], v2[0],v1[1],v2[1] — значит, v1 и v2 — массивы *чего-то*.
- Выполняется сложение v1[0]*v2[0] и v1[1]*v2[1] — значит, v1[0]*v2[0] и v1[1]*v2[1] — числа, и результат dot(v1,v2) — также число.
- Раз v1[0]*v2[0] — число, то и v1[0], и v2[0] — числа. Следовательно, v1 и v2 — массивы чисел.

В этом рассуждении были использованы типы операций «+», «*», «[]», с помощью которых была составлена система ограничений, решение которой позволило выяснить типы dot, v1, v2.

Описание

Выводом типов называется автоматическое полное или частичное вычисление типов некоторых выражений программы, основанное лишь на ее синтаксической структуре и производимое статически, без необходимости запуска программы.

Алгоритм и сама возможность вывода типов целиком зависят от используемой системы типов.

Исходная информация для алгоритма вывода типов — сама программа, способ присваивания типов литералам (таким как `42.0f`, "`hello`", и т. п.), типы некоторых функций и значений, а также правила типизации синтаксических конструкций (типы встроенных арифметических операций, типы операций доступа к структурам данных, взятые из определений этих структур, и т. п.). Правила типизации синтаксических конструкций и литералов обычно фиксированы в рамках одного алгоритма вывода типов.

Наиболее известный, простой и широко применяющийся алгоритм вывода типов — алгоритм Хиндли—Милнера (описан в Wikipedia в статье «Type inference» [159]), пригодный для таких систем, как просто типизированное лямбда-исчисление, System F (ее ограниченное подмножество) и некоторых других. System F является основой систем типов большинства современных функциональных языков, таких как OCaml и Haskell. Она описана, например, в презентации Александра Микеля «An introduction to System F» [99], учебнике Жирара, Лафонта и Тейлора «Proofs and types» [45], и в классическом учебнике Бенджамина Пирса «Types and programming languages» [123].

Мы не будем приводить этот алгоритм здесь, так как он описан в многочисленных публикациях, доступных в Интернете.

Использование

Почти в каждом языке программирования, обладающем системой типов, присутствует нечто, что можно назвать выводом типов: к примеру, программисту на Си не требуется явно указывать тип для каждого выражения в программе, а необходимо указывать его лишь при объявлении переменных и функций. Однако этого недостаточно, чтобы считать, что в языке Си есть полноценный вывод типов: компилятор Си определяет типы выражений строго снизу вверх по синтаксическому дереву программы, от литералов и ссылок на переменные к выражениям.

В качестве примера полезности вывода типов лучше подойдет Java: начиная с версии 5 в этом языке появились «дженерики» (generics: классы и методы, параметризованные типами), и в некоторых случаях компилятор способен вывести конкретные значения типовых параметров при вызове метода, например:

```
List<String> strings = Collections.emptyList();
```

вместо

```
List<String> strings = Collections.<String>emptyList();
```

Однако в сколько-нибудь нетривиальных случаях их приходится указывать явно: фактически, внутри аргументов методов-дженериков вывод типов уже не работает. Это довольно сильно ударяет по синтаксической элегантности программ, использующих дженерики — языку Java не помешало бы наличие более мощного алгоритма вывода типов.

В программах на языках с мощной системой типов, скажем, языках семейства ML, лаконичность кода во многом обусловлена именно тем, что программист может не указывать типы большинства выражений. Обычно типы указывают только для объявлений верхнего уровня, в качестве документации. Например, вот фрагмент кода из библиотеки для чтения tar-архивов [16] на Haskell:

```
correctChecksum :: ByteString → Int → Bool
correctChecksum header checksum = checksum == checksum'
  where
    -- sum of all 512 bytes in the header block,
    -- treating each byte as an 8-bit unsigned value
    checksum' = BS.Char8.foldl' (λx y → x + ord y) 0 header'
    -- treating the 8 bytes of chksum as blank characters.
    header'   = BS.concat [BS.take 148 header,
                           BS.Char8.replicate 8 ' ',
                           BS.drop 156 header]
```

В этом фрагменте кода не указаны типы `checksum'` и `header'`, типы `x` и `y` в выражении `λx y → x + ord y`, а также тип значения, возвращаемого этой анонимной функцией.

Вывод типов вкупе с классами типов (см. 5.13) зачастую позволяет добиться огромной экономии кода, которая была бы невозможна даже в языке *без* статической системы типов. Рассмотрим пример из статьи о классах типов и полюбуемся, что с ним станет, если отказаться от вывода типов; в частности, от вывода аргументов полиморфных (параметрических) типов (то, чего не умеет делать компилятор Java 6)[1]:

До:

```
data Exp = IntE Int
         | OpE  String Exp Exp

instance Binary Exp where
    put (IntE i)      = put (0 :: Word8) ≫ put i
    put (OpE s e1 e2) = put (1 :: Word8) ≫ put s ≫ put e1 ≫ put e2
    get = do tag ← getWord8
             case tag of
                 0 → liftM  IntE get
```

[1]Не стоит из примера делать вывода, что аналогичный по функциональности код на языке без хорошего алгоритма вывода типов был бы действительно настолько раздут: скорее, код пришлось бы написать как-нибудь по-другому, менее просто и ясно, чем в первом из примеров на Haskell. Хороший алгоритм вывода позволяет, по крайней мере, *не отбрасывать* некоторые красивые идиомы только из-за того, что с ними не справится вывод типов.

```
                    1 → liftM3 OpE  get get get
```

После: (это не совсем Haskell: в Haskell более сложный синтаксис явного указания типов)

```
data Exp = IntE Int
         | OpE  String Exp Exp

instance Binary Exp where
  put :: Exp → Put
  put (IntE i)          = ((≫) {Put}) (put {Word8} 0) (put {Int}
      i)
  put (OpE s e1 e2)     = ((≫) {Put}) (put {Word8} 1) ((≫) {Put}
      (put {String} s) ((≫) {Put} (put {Exp} e1) (put {Exp} e2)))
  get :: Get Exp
  get = (≫=) {Get, Word8, Exp})
          getWord8 (λ (tag::Word8) →
            case tag of
                0 → liftM {Get, Int, Exp} IntE (get {Int})
                1 → liftM3 {Get, String, Exp, Exp, Exp} OpE (get
                    {String}) (get {Exp}) (get {Exp})
```

Реализация

Большинство современных языков программирования обладают элементами вывода типов в той или иной форме.

- Все используемые на практике статически типизированные языки способны производить вывод типов выражений «снизу вверх», позволяя программисту писать выражения вида order.getCustomer().getName() (тут из типа order автоматически выводится тип выражения order.getCustomer() и компилятор убеждается, что для этого типа определена операция getName()).

- C++ может выводить типы шаблонных аргументов шаблонных функций из типов их фактических параметров («снизу вверх»). В C++0x появится ключевое слово auto, позволяющее автоматически выводить тип выражения, стоящего в правой части, при объявлении переменной.

- В C# начиная с версии 3.0 также доступен вывод типов инициализирующих выражений при объявлении переменных при помощи ключевого слова var.

- Java, аналогично, позволяет в некоторых простых случаях выводить типовые аргументы методов-дженериков, но не классов и, как ни странно, не конструкторов (до Java 7).

- В Scala присутствует механизм вывода типов выражений и типовых аргументов дженериков, однако вывод типов является *локальным*, т. е. полные типы выражений выводятся из контекста, в котором эти выражения упоминаются, без составления какой-либо системы ограничений между выражениями в разных местах программы (например, между типами двух методов класса); как следствие, явное указание типа довольно часто все же оказывается необходимым.

- В Haskell, OCaml и F# (представляющем собой подмножество OCaml, расширенное в сторону совместимости с объектной системой .NET и т. п.) вывод типов основан на алгоритме Хиндли — Милнера; в случае Haskell алгоритм дополнен обработкой специфических особенностей системы типов Haskell (классов типов (см. 5.13), «семейств типов» и т. п.)

- Coq (язык формулировки математических теорий и программирования, основанный на зависимых типах — [30], описан в книге «Coq'art: Interactive theorem proving and program development» [10], а также в презентации Евгения Кирпичёва [178]) отчасти использует алгоритм Хиндли — Милнера с определенным количеством эвристик, однако его система типов настолько сложна и мощна, что вывод типов в ней является вычислительно неразрешимой задачей, поэтому очень часто типы приходится указывать вручную.

5.3. Функция высшего порядка (ФВП) (Higher-order function (HOF))

Суть

Функция, принимающая на вход или возвращающая функцию.

История

В некоторых областях математики (например, в функциональном анализе) функции высшего порядка называются *операторами* или *функционалами*. Скажем, оператор вычисления производной является функцией высшего порядка: он действует над функциями, и к тому же, его результатом является также функция. В программировании самые ранние из объектов, схожих с функциями высшего порядка — это «комбинаторы» в комбинаторной логике и термы в нетипизированном лямбда-исчислении. Обе этих области науки на самом деле появились задолго до возникновения программирования, в районе 1920—1930-х гг., однако со временем стали его неотъемлемыми элементами. Первым языком программирования с поддержкой функций высшего порядка оказался появившийся еще в 1958 году LISP, изначально являвшийся в первую очередь реализацией лямбда-исчисления. Эта возможность оказалась настолько удобна для программирования, что стала одной из основных составляющих функциональных языков как таковых и одним из самых мощных инструментов абстракции. Тем не менее, в течение очень долгого времени в силу ряда трудностей реализации (см. секции «Имитация» и «Реализация») «промышленные»-языки предоставляли крайне ограниченную поддержку функций высшего порядка. В последние годы их полезность была осознана и оценена широким сообществом программистов по достоинству, и в настоящее время они поддерживаются в той или иной форме практически всеми языками уровня выше Си.

Интуиция

Рассмотрим два отчасти взаимосвязанных примера:

Обобщенная процедура сортировки должна быть в состоянии сортировать одни и те же данные по нескольким различным критериям, т. е. должна принимать на вход помимо набора данных также и описание способа сравнения элементов. Отметим, что сравнение элементов, как и процедура сортировки, само по себе является вычислительным процессом. Таким образом, одному вычислительному процессу необходимо передать описание другого.

С другой стороны, в движке СУБД процедура анализа секции ORDER BY должна принимать на вход ее синтаксическое дерево и возвращать способ сравнения строк таблицы согласно заданным в ней параметрам. В данном случае результатом одного вычислительного процесса является описание другого.

Отметим, что описать (задать) вычислительный процесс можно несколькими способами, различающимися по удобству и универсальности:

- Зафиксировать способ описания (задания) вычислительных процессов, подходящий для конкретной задачи. Например, в задаче сортировки рядов базы данных по полям это может быть структура данных, соответствующая конструкции ORDER BY в SQL. Очевиден недостаток такого подхода: необходимо в каждой очередной задаче организовывать подобную инфраструктуру интерпретации встроенного языка. Тем не менее, в отдельных случаях он вполне оправдан: например, может быть необходимо в целях эффективности ограничить класс возможных вычислительных процессов. Самый яркий пример такой ситуации: большинство графических библиотек ограничивают возможные преобразования изображений аффинными и перспективными, поскольку они допускают чрезвычайно эффективное в реализации и удобное для манипуляций матричное представление.

- Заставить программиста описать все возможные происходящие в программе вычислительные процессы заранее (а именно, описать все процедуры) и разрешить манипуляцию указателями на функции. Этот подход применяется в языках Си, Pascal и многих других. Недостаток такого подхода в том, что число возможных манипулируемых процессов конечно и ограничено числом процедур в программе. К примеру, трудно динамически создать процедуру сравнения двух документов согласно их релевантности относительно пользовательского запроса, поскольку на этапе разработки программы пользовательский запрос неизвестен и невозможно заранее написать процедуру сравнения, уже содержащую его. Этим недостатком обладает процедура qsort из стандартной библиотеки Си: она позволяет задавать способ сравнения всего лишь при помощи указателя на функцию. Конечно, можно сохранить пользовательский запрос в глобальной переменной, и пользоваться ей в процедуре, переданной qsort, но этот способ обладает множеством очевидных недостатков, связанных с многопоточностью и реентерабельностью (reentrancy).

- Принять описанный в предыдущем пункте подход и разрешить лишь манипуляцию указателями на функции, однако задавать вычислительные процессы парами из указателя на функцию и произвольных данных. Теперь достаточно описать процедурами лишь типы *принципиально различных* происходящих в программе процессов, а конфигурирование их вынести в данные. Такой подход общепринят в языках, не реализующих замыкания (см. 5.4), в т. ч. в Си, Pascal и т. п. Приведем пример использования такого подхода:

```
// pthread.h
int pthread_create(pthread_t *thread, const pthread_attr_t
    *attr,
 void *(*start_routine)(void*), void *arg);
```

```
// main.c
#include <pthread.h>

typedef struct our_data {
    ...
} our_data;
void *our_thread(void *arg) {
    our_data *data = (our_data*)arg;
    ...
}
...
pthread_create(..., our_thread, (void*)some_data);
...
```

Это более или менее приемлемый подход, однако его недостаток — в отсутствии синтаксической поддержки со стороны языка, ведущем к большому количеству синтаксического мусора и увеличению шансов допустить ошибку.

Несколько подробнее этот подход описан в статье о замыканиях (см. 5.4).

Поддержка «функций высшего порядка» в языке означает способность языка манипулировать описаниями произвольных вычислительных процессов как полноценными значениями, наравне с другими структурами данных.

Описание

Функция называется функцией высшего порядка, если один из её аргументов — это функция, либо её возвращаемое значение — это функция.

Например:

1) процедура сортировки принимает на вход список данных и функцию сравнения элементов;

2) в движке базы данных процедура анализа секции ORDER BY принимает на вход ее синтаксическое дерево и возвращает соответствующую процедуру сравнения строк таблицы;

3) процедура численного дифференцирования принимает на вход спецификацию требуемой точности и вещественную функцию одного аргумента f, а возвращает также вещественную функцию, отображающую x в $f'(x)$, вычисленное с указанной точностью;

4) процедура вычисления композиции функций принимает на вход две функции f и g, а возвращает функцию h, отображающую x в $f(g(x))$

Вот несколько примеров типов функций высшего порядка на Haskell:

```
sort :: ((α, α) → Bool, [α]) → [α]
analyzeOrderBy :: OrderBySection → ((Row, Row) → Bool)
differentiate :: (Float, Float → Float) → (Float → Float)
compose :: (β → γ, α → β) → (α → γ)
```

Отметим, что при каррировании (см. 5.5) всякая функция с более чем одним аргументом становится функцией высшего порядка, поскольку начинает интерпретироваться как функция от первого аргумента, возвращающая функцию от оставшихся при фиксированном значении первого; сравним:

```
isAsubstringOfB :: (String, String) → Bool
-- The following two notations are equivalent; the second
-- is just an abbreviation of the first.
isAsubstringOfB :: String → (String → Bool)
isAsubstringOfB :: String → String → Bool
```

Из-за некоторой путаницы с тем, что понимать под порядком функции при наличии каррирования (считать ли `isAsubstringOfB` функцией второго порядка из-за того, что результат ее применения к одному аргументу — функция первого порядка типа `String → Bool`?), иногда порядком функции считают глубину самой вложенной слева стрелки: ниже приведены три типа, соответствующих функциям первого, второго и третьего порядков.

```
isAsubstringOfB :: String → String → Bool
foldr :: (α → β → β) → β → [α] → β
externalSort :: ((α → α → Bool) → [α] → [α])
             → FilePath
             → Int
             → (α → α → Bool) → [α] → [α]
```

Здесь `foldr` — процедура правой списочной свертки (см. 5.11).

Процедура `externalSort` (воображаемая) выполняет внешнюю сортировку списка, используя заданный файловый путь для временного хранения данных и сортируя списки меньше заданной длины при помощи заданной «базовой» процедуры сортировки.

В статье «Functional pearl: Why would anyone ever want to use a sixth-order function?» [109] Крис Окасаки показывает, какое практическое применение может быть у функций чрезвычайно высоких порядков, вплоть до шестого.

Использование

Несколько классических примеров функций высшего порядка — сортировка, создание подпроцесса (потока, нити), композиция функций, свертка — были рассмотрены выше. Рассмотрим еще несколько простых примеров.

- Оператор отображения списка `map`:

```
map :: (α → β) → [α] → [β]
> map toUpper ["Hello", "World"]
["HELLO", "WORLD"]
```

- Оператор фильтрации списка `filter`:

```
filter :: (α → Bool) → [α] → [α]
> filter even [1..10]
[2,4,6,8,10]
```

Операторы отображения, фильтрации и списочной свертки есть не только в функциональных языках, но и, например, в Python и Ruby.

- Оператор обращения к массиву (списку) или к словарю: входом оператора является массив (список) или словарь, а выходом — функция из индекса (соответственно целого числа или ключа словаря) в соответствующий элемент коллекции. Данный оператор очень удобно использовать вместе с оператором отображения: скажем, можно «одним махом» получить из списка ключей список соответствующих им значений, преобразовав с помощью функции `map` список функцией обращения к словарю.

- Оператор построения словаря из списка ключей и функции: к каждому ключу применяется функция построения значения, и из полученных пар составляется словарь.

- Оператор построения списка при помощи порождающей функции: на входе задается размер списка n и процедура с одним аргументом, и ответ составляется из результатов применения процедуры к числам в диапазоне $0 \ldots n - 1$.

Многие функции высшего порядка оперируют на более абстрактном уровне, не над данными, но над другими функциями:

- Оператор смены базиса бинарной операции `on`:

```
on :: (β → β → γ) → (α → β) → (α → α → γ)
g `on` f = λx y → g (f x) (f y)

equalsIgnoreCase = (==) `on` toUpper
```

- Оператор смены порядка аргументов бинарной операции `flip`:

```
flip :: (α → β → γ) → (β → α → γ)
flip f = λb a → f a b
```

- Оператор вычисления в заданной точке `at`:

```
at :: α → (α → β) → β
at x = λf → f x
-- Alternative definition:
($) f x = f x
at = flip ($)
```

- Имея функцию из X в значение обнуляемого[2] типа Y, и «значение по умолчанию», можно получить функцию из X в Y, заменяющую нулевой результат первой функции на значение по умолчанию.

Существует множество полезных функций высшего порядка для работы с парами:

- Имея функцию из X в Y, можно естественным образом получить функцию из X в пару (X, Y).
- Имея функцию из X в Y, можно получить функцию из (Z, X) в (Z, Y) или из (X, Z) в (Y, Z).
- Имея функцию из X в Y и из X в Z, можно получить функцию из X в (Y, Z).
- Имея X, можно получить функцию из Y в (X, Y) или в (Y, X).
- Существуют функции взятия первого и второго компонентов пары, удобные для использования с вышеупомянутыми.

Очень часто бывает полезно выстроить целую предметную область в терминах функций высшего порядка, создав комбинаторную библиотеку (см. 5.14).

Функции высшего порядка часто используются вместе с бесточечным стилем (см. 5.6).

Функции высшего порядка позволяют уменьшить избыточность в программе и сократить код, что положительно влияет на корректность (меньше мест для совершения ошибки — меньше ошибок) и поддерживаемость (не надо делать изменения во множестве похожих мест в коде: достаточно изменить код, абстрагирующий их сходство). Особенно остро это проявляется в многопоточном программировании, где необходимо управлять несколькими процессами — скажем, «периодически выполнять заданное действие, не допуская, чтобы оно выполнялось одновременно два раза», или «параллельно применить функцию к каждому элементу списка», или «выполнить набор заданий, пользуясь не более чем 10 нитями», или «асинхронно выполнить заданное действие и в случае успеха передать его результат следующему действию» и т. п. Корректная реализация многопоточных алгоритмов — чрезвычайно трудоемкое дело со множеством нюансов. Этим объясняется тот факт, что библиотеки многопоточного программирования даже в императивных языках буквально напичканы функциями высшего порядка. Скажем, рассмотрим стандартную библиотеку Java; в ней процедуры (равно как и процедуры высшего порядка) моделируются при помощи объектов:

[2] То есть, содержащего некое «нулевое» значение, например, `null`.

- java . util . concurrent . Callable <V> — процедура без аргументов, возвращающая V.
- java . util . concurrent . ExecutorService — класс, позволяющий асинхронно выполнять задания и дожидаться их результатов. Задания задаются в виде процедур типа Callable или Runnable.
- java . util . TimerTask содержит в себе процедуру без аргументов и без возвращаемого значения.
- java . util . Timer периодически выполняет процедуры типа TimerTask.
- В Java 7 появится ParallelArray – средство для параллельной обработки массивов. API этой библиотеки целиком построен на функциях высшего порядка. Вот пример его использования:

```
ParallelArray<Student> students = new
    ParallelArray<Student>(fjPool, data);
double bestGpa = students
    .withFilter(isSenior).withMapping(selectGpa).max();

static final Ops.Predicate<Student> isSenior = new
    Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == Student.THIS_YEAR;
    }
};
static final Ops.ObjectToDouble<Student> selectGpa = new
    Ops.ObjectToDouble<Student>() {
    public double op(Student student) {
        return student.gpa;
    }
};
```

В нотации замыканий (см. 5.4), которую, однако, не планируется включать в Java 7 (планируется включить несколько иную нотацию), этот пример выглядел бы так:

```
double bestGpa = students
    .withFilter ({ Student s => (s.graduationYear == THIS_YEAR)
        })
    .withMapping({ Student s => s.gpa })
    .max();
```

Всякий, кто когда-либо программировал параллельные вычисления, согласится, что вышеприведенные фрагменты кода намного короче, более читаемы и устойчивы

к ошибкам программиста, чем типичные способы реализации таких вычислений при помощи низкоуровневых примитивов многопоточного программирования.

Конечно же, многопоточное программирование — совсем не основное применение функций высшего порядка, а лишь пример области, где контраст между удобством программирования с ними и без них особенно ярок: схожие масштабы роста корректности и сокращения кода от обращения к функциям высшего порядка наблюдаются и в других областях, однако в них эти эффекты не настолько бросаются в глаза, поскольку и код с использованием «обычных» техник программирования обладает приемлемыми характеристиками.

Реализация

Многие современные языки общего назначения (C, Pascal) позволяют манипулировать указателями на функции (многие компиляторы Fortran не позволяют даже этого).

Все функциональные языки (Haskell, OCaml, Scheme, Clean, ...) и большинство динамически типизированных языков (Python, Ruby, Javascript) позволяют манипулировать функциями как полноценными значениями, позволяют использовать замыкания (см. 5.4), анонимные функции (лямбда-выражения).

Функциональные языки с мощной системой статической типизации (Haskell, OCaml, Clean, ...) поощряют обильное использование функций высшего порядка.

Имитация

Способ имитации полноценных функций высшего порядка в языках семейства Си описан в секции «Интуиция».

В объектно-ориентированных языках имитация возможна при помощи объектов: объекты (обычно со всего одним абстрактным методом) могут моделировать функции и замыкания. Приведем интерфейс, соответствующий абстрактному понятию функции:

```
interface Function<X,Y> {
    Y apply(X x);
}
```

В стандартной библиотеке Java объекты, схожие по свойствам и назначению с функциями, и использующие их объекты, схожие, таким образом, с ФВП, встречаются достаточно часто:

- Описанные выше многопоточные средства.
- Класс TreeMap и процедура сортировки Collections . sort параметризуются процедурой сравнения, моделируемой классом Comparator.
- Метод Collections . reverseOrder (Comparator<T>) принимает на вход функцию сравнения и возвращает функцию сравнения в обратном порядке.

- Паттерны Visitor и Listener , описанные в классической книге «банды четырех» о шаблонах проектирования ([35]) имеют явное сходство с ФВП; последний используется, например, в оконной библиотеке Swing.

5.4. Замыкание (Closure)

Суть

Функция, использующая переменные из области видимости, в которой была создана.

История

Правила вычисления в лямбда-исчислении таковы, что лямбда-выражение может использовать переменные, связанные какой-нибудь из окружающих его лямбда-абстракций. Возьмем, например, $((\lambda x.(\lambda y.x))42)37 \Longrightarrow (\lambda y.42)37 \Longrightarrow 42$: здесь выражение $\lambda y.x$ «запоминает» значение определенного во внешней области видимости идентификатора x. Таким образом, идея замыканий появилась еще до появления программирования, в момент создания лямбда-исчисления — в 1930 гг. Термин «замыкание» был введен в 1964 г. Питером Ландиным. Замыкание — это пара из лямбда-выражения («кода») со свободными переменными и *окружения*, связывающего их. Внутри замыкания все переменные связаны, т. е. оно *замкнуто*. Первая полноценная реализация лексических замыканий относится к языку Scheme (диалект LISP, 1975 г.). В настоящее время они поддерживаются в той или иной форме почти всеми популярными языками общего назначения, кроме низкоуровневых, таких как C или C++ до версии C++0x (не считая эмуляции замыканий, например, при помощи boost :: function — впрочем, надо отдать должное, достаточно удобной).

В языке LISP присутствовали замыкания с динамической областью видимости, нарушавшие законы лямбда-исчисления и потому, в общем случае, неудобные в использовании[3]. Emacs LISP— практически единственный используемый на практике современный язык программирования, где динамическая область видимости используется *по умолчанию*.

Интуиция

Рассмотрим любую функцию высшего порядка из описанных в соответствующей секции (см. 5.3), скажем, `map`.

```
def map(f, list):
    for x in list:
        yield f(x)
```

Рассмотрим пример ее применения: напишем процедуру, которая принимает в качестве аргумента словарь и возвращает функцию поиска в этом словаре по ключу.

[3]Впрочем, в ряде задач переменные с динамической областью видимости бывают полезны; они поддерживаются, например, языком Common Lisp.

```
def fetcher(dict):
    def fetch(k):
        return dict[k]
    return fetch
```

Затем при помощи этой функции достанем из словаря по списку ключей список значений.

```
def values(dict, keys):
   return map(fetcher(dict), keys)
```

Как видно, функция `fetch` использует значение аргумента `dict` функции `values`, потому объявлена в теле `values`.

Переменная `dict` не является аргументом или локальной переменной `fetch` — она является свободной относительно определения `fetch`. Однако, она является аргументом `fetcher` и потому связана относительно определения `fetcher`.

Возможность объявлять локальные функции, использующие свободные переменные, связанные во внешней области видимости, жизненно важна и плохо поддается имитации. Убедимся в этом.

Представим, как выглядел бы этот код, если бы Python, как стандартный Си, разрешал объявлять функции только на верхнем уровне файла, и, таким образом, свободными переменными в определении функции могли бы быть только глобальные переменные программы.

```
def fetcher(dict):
    return fetch

def fetch(k):
    return dict[k] # Hey, how do we know which 'dict' to use?
```

Если объявлять процедуру `fetch` на верхнем уровне, то в контексте ее тела неоткуда взять `dict`, т. к. он не является и не может являться глобальной переменной.

К счастью, Python разрешает объявлять одни процедуры внутри других. В менее мощных языках можно попробовать применить «лямбда-поднятие (lambda lifting)» или «преобразование замыканий». Этот подход описан в статье об ФВП (см. 5.3). Представим n-аргументную функцию как пару из $n+1$-аргументной и значения вспомогательного $n+1$-го аргумента. При $n = 1$ подход выглядит так:

```
def call(f_and_data, x):
    f,data = f_and_data
    return f(x, data)
...
# f is a code/data pair
def map(f, list):
    for x in list:
        yield call(f, x)
...
```

```
def fetchFrom(x, data):
    # Unpack free variables from the
    # auxiliary argument
    dict = data
    return dict[k]

def fetcher(dict):
    fetch = (fetchFrom, dict)
    return fetch
...
```

Однако уже в этом коде видна проблема: способы вызова обычных и «искусственных» процедур совершенно различны. Например, возвращенную «процедуру» `fetch` нельзя вызывать как `fetch(...)`, равно как и в `map` нельзя передать в качестве `f`, скажем, процедуру `upper` из модуля `string` для перевода строк в верхний регистр, т. к. `map` ожидает уже не обычную процедуру, а пару из процедуры и ее вспомогательного аргумента. Чтобы восстановить равноправие двух появившихся разновидностей процедур, придется изменить *всю программу*, представляя *все* процедуры, потенциально используемые в качестве аргументов или результатов процедур высшего порядка, в форме со вспомогательным аргументом, что во всех случаях, кроме тривиальных, крайне неудобно и трудоемко.

Как видно, для использования ФВП необходима поддержка замыканий со стороны языка.

Описание

Итак, замыкание — это значение (обычно процедура), имеющее доступ к переменным из области видимости, в которой оно было создано. В некоторых языках (в функциональных языках, в Python, Ruby, Javascript, …) нет разницы между замыканиями и обычными процедурами. В других же языках замыкания приходится эмулировать.

Замыкания жизненно важны для использования функций высшего порядка; без замыканий становится невозможно использовать сколько-нибудь полезную ФВП (как мы видели выше на примере map, *написать* полезную ФВП возможно, однако нетривиальные варианты ее использования оказываются недостижимыми).

Как показано в секции «Интуиция», полноценная поддержка замыканий в языке и равноправие между замыканиями и процедурами важны, поскольку отсутствие такой поддержки приводит к необходимости а) дублировать каждую «обычную» процедуру ее аналогом в виде эмулированного замыкания, и б) реализовывать все функции высшего порядка исключительно в терминах эмулированных замыканий. Более того, если проблему а), по крайней мере, можно решить всегда, то проблема б) может оказаться неразрешимой, если приходится иметь дело с ФВП, находящейся в сторонней библиотеке, и реализованной в терминах обыкновенных процедур.

Эта проблема широко известна и иногда встречается в плохо спроектированных библиотеках для языков семейства Си; канонический пример — процедура сортировки qsort, принимающая в качестве функции сравнения обыкновенный указатель на

функцию. С помощью такой процедуры можно отсортировать значения лишь согласно порядку, полностью известному на момент написания программы. В языке же, обладающем поддержкой замыканий, проще простого, к примеру, отсортировать точки (банкоматы, рестораны) на карте по расстоянию от заданной пользователем точки, несмотря на то, что на момент написания неизвестно, какую именно точку пользователь задаст:

```
def closestRestaurants(userPoint, n):
    return sorted(restaurants,
        lambda r1,r2: dist(r1,userPoint)-dist(r2,userPoint))[1:n]
```

Реализация замыканий связана с двумя вопросами, называемыми вместе «проблемой функционального аргумента (фунарг-проблемой)»[4]:

- Upward funarg problem (наружняя фунарг-проблема)[5]: как реализовать возврат замыкания в качестве результата? Поскольку замыкание использует переменные, связанные в области видимости, где оно было создано (или еще выше), то цикл жизни этих переменных должен продлиться, по крайней мере, до последнего использования этого замыкания — возможно, даже после выхода из создавшей их области видимости (например, после возврата из процедуры, возвратившей это замыкание).

  Скажем, стратегия выделения переменных на стеке уже не подходит, поскольку они уничтожатся при выходе из процеруры, и замыкание, ссылавшееся на них, испортится. Таким образом, все переменные, используемые замыканиями, должны либо подвергаться глубокому копированию (копия не должна зависеть от существования исходных переменных), либо создаваться в куче и уничтожаться при помощи какого-либо механизма сборки мусора (во всяком случае, нужна какая-то договоренность относительно того, кто и в какой момент уничтожает такие переменные). Стратегия с копированием лишает замыкания некоторых интересных вариантов использования, описанных далее. Поэтому полноценная поддержка замыканий (и полноценное решение upward funarg problem) присутствует почти исключительно в языках со сборщиком мусора. Существуют, впрочем, и другие решения, основанные на статическом анализе кода: т. н. escape analysis (анализ времени жизни), region inference (вывод регионов), ownership types (типы владения) и т. п., однако они пока не получили широкого распространения.

- Downward funarg problem (внутренняя фунарг-проблема): как реализовать возможность передачи замыкания в качестве аргумента процедуре? Проблема в данном случае состоит лишь в том, как обеспечить одинаковое представление

[4]Перевод «фунарг-проблема» используется в книге «Мир Лиспа» [187].

[5]Автору не известен устоявшийся перевод этого термина на русский язык, приведенный перевод придуман специально для данной статьи.

обычных процедур и замыканий, так, чтобы процедура могла принимать в качестве аргумента другую процедуру, не заботясь о том, является ли она замыканием. В отличие от upward funarg problem, в данном случае предполагается, что передаваемое замыкание не сохраняется процедурой в глобальных переменных, и, таким образом, оно существует лишь во время ее вызова: благодаря этому исчезают проблемы управления памятью. Решения downward funarg problem существуют даже в языках без сборки мусора: так, компилятор GCC поддерживает (в качестве нестандартного расширения языка Си) передачу указателей на «вложенные функции» в другие функции при помощи техники «трамплинов» — маленьких динамически генерируемых и компонуемых фрагментов кода. Эта возможность также поддерживается языком Pascal (но, как и в случае GCC, без решения наружней фунарг-проблемы). Можно считать решением внутренней фунарг-проблемы и перегрузку оператора `()` в C++: язык позволяет в некоторых ситуациях одинаковым образом работать с функциями, указателями на функции и объектами, для чьих классов переопределен оператор `()`, а объекты, в свою очередь, позволяют эмулировать замыкания.

В языках с изменяемым состоянием встает вопрос о том, может ли замыкание изменять значения «захваченных» переменных. Разные языки подходят к этому вопросу по-разному: например, в Java это запрещено — замыкания моделируются при помощи анонимных классов, и все переменные, захватываемые анонимным классом, должны быть объявлены как `final`, и их значения при создании замыкания копируются (конечно, речь идет не о глубоком копировании объектов, а о копировании ссылок, или же о копировании значений для примитивных типов).

В Scheme разрешено изменять значения переменных из родительской области видимости. Для реализации этой функциональности используется «модель окружений»:

- Каждой существующей во время выполнения области видимости соответствует *окружение*: *кадр*, связывающий имена идентификаторов, определенных в этой области и их значения, а также ссылка на родительское окружение.

- Для получения доступа (на чтение или на запись) к значению, связанному с некоторым идентификатором, кадры просматриваются снизу вверх, начиная с текущего, пока не будет найден кадр, связывающий этот идентификатор.

- Замыкание реализуется как пара из указателя на его код и указателя на окружение, где это замыкание было создано (и содержащее в т. ч. значения используемых в нем свободных переменных).

- При применении функции-замыкания используется кадр аргументов, связывающий имена формальных параметров со значениями фактических, а также захваченное окружение. Из них составляется новое окружение, в котором и исполняется код замыкания. Таким образом, значения параметров берутся из первого кадра, а значения захваченных идентификаторов — из второго.

За подробным описанием модели окружений предлагается обратиться в один из авторитетных источников: например, в книгу «Структура и интерпретация компьютерных программ» [48] (глава 3.2).

В классической работе Стрейчи «Fundamental concepts in programming languages» [147] в секции 3.4.3 «Modes of free variables» также обсуждается вопрос изменяемости захваченных переменных.

Использование

Чаще всего замыкания появляются при использовании функций высшего порядка (см. 5.3); см. соответствующий раздел. Однако у замыканий есть два других интересных свойства, открывающих совершенно иные и очень интересные варианты использования:

- Замыкание может использовать изменяемые данные (если язык это позволяет).
- Несколько замыканий могут совместно использовать переменные из одной и той же области видимости.

Рассмотрим их по очереди.

Вот код на Scheme, реализующий счетчик при помощи замыкания:

```
> (define counter-from
    (lambda (n)
      (lambda ()
        (set! n (+ n 1))
        n)))
> (define x (counter-from 5))
> (define y (counter-from 5))
> (x)
6
> (x)
7
> (y)
6
```

В этом примере x и y — два замыкания, каждое со своим собственным состоянием.

Таким образом, замыкания позволяют реализовать объекты, аналогичные объектам в ООП, то есть, значения, обладающие инкапсулированным внутренним состоянием и собственным набором операций, манипулирующих им (как мы увидим далее, верно и обратное: объекты позволяют эмулировать замыкания. Таким образом, объекты и замыкания в определенном смысле эквивалентны[6]). В данном случае был реализован объект-счетчик со всего одним безаргументным методом; рассмотрим объект чуть посложнее, с двумя «методами»: «получить значение счетчика» и «увеличить счетчик на заданное значение».

[6]На эту тему существует дзен-коан [160].

```
> (define counter-from
    (lambda (n)
       (cons % Construct a pair
         (lambda () n)
         (lambda (k) (set! n (+ n k))))))
> (define (get-n x) ((car x))) % car extracts the first component
    of a pair
> (define (inc-n x k) ((cdr x) k)) % ... and cdr extracts the
    second one
> (define x (counter-from 5))
> (get-n x)
5
> (inc-n x 3)
> (get-n x)
8
```

Во многих динамических языках с поддержкой ООП, например в JavaScript, в сущности, объекты реализованы примерно так (объект — это словарь замыканий с общим окружением).

Значительно более интересный пример использования замыканий с изменяемым состоянием для эмуляции объектов можно найти в главах 3.3.4 и 3.3.5 книги «Структура и интерпретация компьютерных программ» [48]: в них строится симулятор цифровых схем и симулятор схем с распространением ограничений.

Заметим, что этот и похожие варианты использования замыканий невозможны при копирующем подходе к решению фунарг-проблемы, поскольку копирующий подход лишает замыкания возможности *взаимодействовать* через изменяемые переменные, над которыми они замкнуты: у каждого замыкания оказывается собственная независимая копия.

Рассмотрим еще один любопытный пример разделения областей видимости между несколькими замыканиями. Спроектируем библиотеку для работы с векторами и матрицами. Напомним, что матрицы и векторы вовсе не обязаны состоять из вещественных чисел, умножаемых и складываемых с помощью обыкновенных арифметических операций умножения и сложения. В математике алгебраическая структура со сложением, умножением и нулевым элементом (тип и реализации этих операций для него) называется полукольцом над этим типом. Перемножение матриц и умножение матрицы на вектор имеют смысл над любым полукольцом.

Например, можно взять вместо вещественных чисел логические значения, вместо умножения — операцию «И», вместо сложения — операцию «ИЛИ», а вместо нуля — `False`.

Матрицы над этим полукольцом полезны, в том числе, для поиска компонент связности графа (при возведении матрицы инцидентности в n-ю степень получится матрица, в чьем (i, j)-м компоненте стоит `True`, если и только если вершины i и j связаны путем длиной в $n - 1$ шаг). Если же рассмотреть матрицы над вещественными числами и полукольцом, где вместо сложения и умножения — операции взятия сложения и

минимума (это полукольцо называется «тропическим»), то такие матрицы можно использовать для поиска кратчайших путей в графах. У них есть и другие применения: см., например, статью Дэна Пипони «An Approach to Algorithm Parallelization» [124].

Все задачи подобного рода могут выиграть от улучшения обобщенных алгоритмов линейной алгебры (перемножение матриц, представление разреженных матриц и т. п.), способных работать с матрицами над любым полукольцом. Поэтому имеет смысл реализовать обобщенную библиотеку линейной алгебры, параметризованную полукольцом, т. е. реализацией «сложения», «умножения» и «нулем». Содержательная часть исходного кода библиотеки (на Scheme) умещается на один экран, однако это все же слишком много, чтобы поместить его целиком в данной статье. Поэтому приведем лишь фрагмент кода, а весь код целиком можно найти на сайте журнала.

```
> (define (make-semiring @+@ @*@ +0+) (list @+@ @*@ +0+))
> (define reals-semiring    (make-semiring + * 0))

> (define (make-linalg semiring)
    (define @+@ (semiring-plus semiring))
    (define @*@ (semiring-mult semiring))
    (define +0+ (semiring-plus-unit semiring))
    ...
    (define (scalar-prod u v) (vec-foldl @+@ +0+ (vec-zip @*@ u
        v)))
    (define (mat*mat m1 m2)
      (let ([m2t (mat-transpose m2)])
        (vec-map (lambda (r1)
          (vec-map (lambda (c2) (scalar-prod r1 c2)) m2t)) m1)))
    ...

    (lambda request
      (match request
        [`(make-vec ,xs)      (make-vec xs)]
        [`(build-vec ,n ,f)   (build-vec n f)]
        [`(vec-ref ,v ,i)     (vec-ref v i)]
        [`(scalar-prod ,u ,v) (scalar-prod u v)]
        ...
        [`(mat*mat ,m1 ,m2)   (mat*mat m1 m2)])))

> (define (mat*mat lib m1 m2) (lib 'mat*mat m1 m2))
> ...
> (define r-lib (make-linalg reals-semiring))

> (define u (make-vec r-lib '(1 2 3)))
> (define v (make-vec r-lib '(4 5 6)))
> (scalar-prod r-lib u v)
32
```

В данном случае make–linalg принимает на вход функцию сложения, функцию умножения и нейтральный элемент по сложению, и возвращает набор процедур линейной алгебры над этими операциями. Как видно, возвращается процедура, принимающая на вход произвольный запрос, сопоставляющая (см. 5.10) его с несколькими образцами и вызывающая одну из «рабочих» процедур, определенных внутри make–linalg. Эти процедуры являются замыканиями и используют данные из semiring и из локальных переменных @+@, @*@, +0+.

Данный пример также иллюстрирует имитацию классов типов (см. 5.13) при помощи «dictionary-passing style» (см. в статье о классах типов): имитируется класс типов «Полукольцо» примерно следующего вида:

```
class Semiring τ where
  (+) :: τ → τ → τ
  (×) :: τ → τ → τ
  zero :: τ
-- For example:
instance Semiring Bool where
  (+) = (||)
  (×) = (&&)
  zero = False
```

Реализация

Практически все появляющиеся в последнее время языки поддерживают замыкания. Замыкания реализованы в той или иной форме в большом количестве функциональных (Haskell, OCaml, Lisp, F# и др.), динамических (Python, Ruby и др.) и императивных (C# и др.) языков.

«Классический» и наиболее интуитивно понятный вариант поддержки замыканий с моделью окружений реализован в языке Scheme.

В языке Haskell отсутствуют изменяемые переменные, поэтому ряд тонкостей реализации замыканий обходит его стороной, и про них также можно сказать, что в них реализован канонический вариант поддержки замыканий.

Javascript известен своими неочевидными правилами, связанными с областями видимости (см. блог-пост «A huge gotcha with Javascript closures and loops» Кейта Ли [88]); эти же проблемы есть также и в C# (см. пост на StackOverflow «Problem with delegates in C#» [130] с примером такой проблемы, а также статью «The Beauty of Closures» [141] о различиях в стратегиях захвата переменных в C# и Java).

Замыкания можно ограниченно смоделировать при помощи анонимных классов в Java. Существуют планы включить более удобную синтаксическую поддержку замыканий в Java 7. Из-за отсутствия общей точки зрения на детали реализации и синтаксиса эта возможность была отложена, однако в ноябре 2009 г. ситуация неожиданно изменилась ([131]), и, возможно, замыкания все же будут включены в язык (впрочем, в предложенном варианте имеется множество проблем, описанных Нилом Гафтером в письме [42]).

Замыкания также планируется включить в стандарт C++0x (однако без автоматического управления памятью их полезность несколько снижается; в частности, стандарт обязывает программиста явно указывать, желает ли он копировать значения свободных переменных или же использовать их по ссылке — но в этом случае их время жизни ограничено временем жизни области видимости, создавшей их, а за ее пределами обращение к такой переменной порождает «неопределенное поведение»).

Python реализует замыкания сходным с Java образом: модифицировать захваченные переменные запрещено (хоть это и проверяется во время выполнения, а не во время компиляции).

Имитация

Часть способов имитации замыканий описана в статье о функциях высшего порядка (см. 5.3).

Процесс устранения замыканий с помощью введения вспомогательных аргументов называется *преобразованием замыканий (closure conversion)*, или *лямбда-поднятием (lambda lifting)* и применяется в качестве стадии компиляции во многих компиляторах функциональных языков. Это преобразование было предложено Томасом Джонсоном в статье «Lambda lifting: transforming programs to recursive equations».

В объектно-ориентированных языках замыкания и лямбда-выражения можно имитировать с помощью объектов, объектных литералов или анонимных классов.

Также существует техника дефункционализации — способ преобразования программы, использующей функции высшего порядка («программы высшего порядка») в программу первого порядка, использующую только функции первого порядка и дополнительные структуры данных. Дефункционализация хорошо описана в статье «Defunctionalization at work» Оливье Дэнви и Лассе Нильсена [33], а также в потрясающей диссертации Нила Митчелла «Transformation and analysis of functional programs» [100] и в его отдельной статье [101] и слайдах [102] о дефункционализаторе firstify «Losing functions without gaining data».

При дефункционализации программ высшего порядка зачастую получаются знакомые и несколько неожиданные алгоритмы: к примеру, из программы, составляющей список узлов дерева при помощи разностных списков, получается программа, использующая хвосторекурсивный алгоритм (см. 5.7) с аккумулятором. Разностные списки были предложены Джоном Хьюзом в статье «A novel representation of lists and its application to the function `reverse`».

Этот факт служит примером того, как удачный выбор средств программирования может существенно упростить понимание или написание алгоритма: некоторые эффективные, но неочевидные алгоритмы обработки данных становятся тривиальными, будучи переформулированными с помощью функций высшего порядка — ФВП позволяют освободить алгоритмы от «шелухи», связанной с низкоуровневыми деталями реализации, заслоняющими суть.

Дефункционализация также используется и для оптимизации программ высшего порядка, поскольку часто производительность дефункционализованной программы

оказывается выше за счет использования более простых, эффективных и «близких к железу» операций. Дефункционализация не решает проблемы управления памятью под свободные переменные замыканий и по-прежнему требует сборки мусора либо ручного управления. Однако существуют другие техники, позволяющие частично или полностью решить проблемы управления памятью: например, т. н. типы владения (ownership types) (описаны в статье «Ownership types for object encapsulation» [14], один из авторов которой — Барбара Лисков), вывод регионов (region inference) (описан в статье «Region-based memory management» Мэдса Тофте и Жана-Пьера Талпена [155] и в книге «Design concepts in programming languages» [157]).

5.5. Частичное применение, карринг и сечения (Partial application, currying and sections)

Суть

Фиксацией некоторых аргументов из функции получается новая функция с меньшим числом аргументов.

История

Согласно классическому труду Хенка Барендрегта «Lambda calculi with types» [7] и статье Хиндли и Кардоне «History of lambda-calculus and combinatory logic» [17], понятие карринга было введено Шейнфинкелем в 1924 г. при создании комбинáторной логики (статья «Über die Bausteine der mathematischen Logik» [138], перевод на английский язык «On the building blocks of mathematical logic» [139]) для избавления от нужды в функциях с несколькими аргументами — как один из этапов перехода от «традиционного» математического языка к языку комбинаторной логики, где понятие аргумента и функции вовсе отсутствуют. Позднее (в 1930 гг. и далее) это понятие использовалось Хаскеллом Карри (он, однако, много раз явно указывал на роль Шейнфинкеля как «первооткрывателя»), и было названо по его имени «каррингом» уже Кристофером Стрейчи в 1967 г. в его знаменитой работе «Fundamental concepts in programming languages» [147].[7]

Интуиция

Рассмотрим процедуру, определяющую, удовлетворяет ли строка регулярному выражению.

```
matchesRegexp :: String → String → Bool
matchesRegexp pattern string = ...
```

Применим ее, скажем, для валидации целых чисел:

```
isNumber s = matchesRegexp "-?[0-9]+" s
```

Получается: «Число — это такая строка `s`, что `s` удовлетворяет регулярному выражению `-?[-0-9]+`». Но почему бы не сказать покороче? «Число — это то, что удовлетворяет регулярному выражению `-?[0-9]+`»:

```
isNumber = matchesRegexp "-?[0-9]+"
```

Посмотрим на этот же пример с другой стороны. Во многих задачах первый аргумент у процедуры `matchesRegexp` меняется гораздо реже, чем второй. Это наталкивает на мысль, что неплохо бы составить следующую процедуру:

[7] Некоторые (в основном, в шутку) утверждают, что вместо термина «карринг» следует употреблять термин «шейнфинкелизация».

```
makeMatcher :: String → (String → Bool)
makeMatcher pattern = matcher
  where matcher s = matchesRegexp pattern s

isNumber = makeMatcher "-?[0-9]+"
```

Наличие процедуры `makeMatcher` позволяет не плодить вспомогательные функции и лямбда-абстракции вида `\s -> matchesRegexp "-?[0-9]+" s` или `isNumber s = matchesRegexp "-?[0-9]+" s`.[8]

Описание

Рассматриваемые термины (частичное применение, карринг) имеют смысл только в языке, где функции могут обладать несколькими аргументами (арностью более 1). Поэтому будем считать, что речь идет именно о таком языке.

Частичным применением n-арной функции f называется конструкция, значением которой является $(n-k)$-арная функция, соответствующая f при фиксированных значениях некоторых k из n аргументов. Например, в гипотетическом языке программирования частичное применение функции `drawLine` может выглядеть вот так:

```
drawLine :: (width:Double, color:Color,
             style:LineStyle, a:Point, b:Point) → void

thinSolidLine :: (color:Color, a:Point, b:Point) → void
thinSolidLine = drawLine(1.0, _, SOLID, _, _)
```

В данном случае фиксируются 2 аргумента 5-арной функции `drawLine` и получается 3-арная функция `thinSolidLine`.

У термина «карринг» есть три взаимосвязанных значения.

Первое из них — частный случай частичного применения, при котором фиксируется несколько *первых* аргументов функции:

```
thinLine :: (color:Color, style:LineStyle,
             a:Point, b:Point) → void
thinLine = drawLine 1.0
```

В этом примере `thinLine` получен фиксацией первого аргумента `drawLine`.[9]

Второе — превращение функции F над 2-местным кортежем (парой с типами компонентов X и Y) в функцию над X, возвращающую функцию над Y (такая функция называется «каррированной» версией F):

[8] А также позволяет «скомпилировать» регулярное выражение, так что многократные применения результата `matchesRegexp "-?[0-9]+"` к множеству строк $s_1, s_2, \ldots$ выполняются намного эффективнее, чем многократные вычисления `matchesRegexp "-?[0-9]+" s1`, `matchesRegexp "-?[0-9]+" s2` и т. п.

[9] Конечно же, необязательно фиксировать аргументы какими-то «конкретными» значениями (литералами).

```
matchesRegexpUncurried :: (String,String) → Bool
matchesRegexpUncurried = ...

matchesRegexpCurried :: String → (String → Bool)
matchesRegexpCurried pattern = matcher
  where matcher s = matchesRegexpUncurried (pattern,s)
```

В этом примере `matchesRegexpCurried` называется каррированной версией `matchesRegexpUncurried`, а сам процесс выражения `matchesRegexpCurried` через `matchesRegexpUncurried` называется каррингом.

Такое преобразование легко выразить в общем виде и оно оказывается достаточно полезным, чтобы быть включенным, например, в стандартную библиотеку Haskell[10]:

```
curry :: ((α, β) → γ) → (α → β → γ)
curry f = f'
  where f' x y = f (x,y)
-- Other equivalent variants:
curry f = λx y → f (x,y)
curry f x y = f (x,y)
-- Or even:
curry f x = λy → f (x,y)

uncurry :: (α → β → γ) → ((α, β) → γ)
uncurry f = f'
  where f' (x,y) = f x y
-- Or:
uncurry f = λ (x,y) → f x y
uncurry f (x,y) = f x y
```

Обратим внимание на очень важный для запоминания аспект нотации. В лямбда-исчислении вообще и в языке Haskell в частности оператор → *правоассоциативен*, а оператор применения функции — левоассоциативен. Вот несколько пар полностью эквивалентных определений, иллюстрирующих это.

```
f :: String → Int → Bool → Double
f :: String → (Int → (Bool → Double))

g = f "a" 5 True
g = ((f "a") 5) True

matchesRegexp :: String → String → Bool
matchesRegexp :: String → (String → Bool)

isNumber x = matchesRegexp "-?[0-9]+" x
```

[10] А также в теоретико-категорное определение *декартово замкнутой категории*, где при помощи `curry` фактически вводится определение оператора «применение функции к аргументу». К счастью, для понимания материала этой статьи знакомство с теорией категорий не требуется.

```
isNumber x = (matchesRegexp "-?[0-9]+") x
```

Третье значение термина «карринг» — применение каррированной функции к аргументам:

```
isNumber = matchesRegexpCurried "-?[0-9]+"
```

В таком случае говорят, что процедура `isNumber` получена каррированием процедуры `matchesRegexpCurried`.

Итак, в целом, каррингом называется явление, при котором для функции нескольких аргументов появляется возможность зафиксировать несколько первых из них: сам процесс фиксации и подготовка функции к возможности их фиксации.

Для бинарных операций примерно одинаково часто встречается необходимость зафиксировать их левый и правый аргумент, поэтому, к примеру, Haskell предоставляет одинаковый синтаксис для обоих вариантов:

```
> map (+5) [1,2,3]
[6,7,8]
> map (5-) [1,2,3]
[4,3,2]
```

Haskell также позволяет интерпретировать произвольную двухаргументную функцию как бинарный оператор, заключая ее имя в обратные кавычки, например:

```
> let elem x list = any (==x) list
> map (5 `elem`) [[5,1,2], [3,4], [6,5,1]]
[True, False, True]
> map (`elem` [5,10,15]) [2,10,3,5]
[False, True, False, True]
```

Эти варианты применения называются соответственно *левым* и *правым* сечениями функции.

Использование

При программировании на языках, поддерживающих карринг, он используется довольно часто. Приведем несколько примеров из кода библиотек языка Haskell:

Из игры «Space invaders» (hinvaders) [127]:

```
moveSprite :: Coordinate → Sprite → Sprite
moveSprite (dx, dy) (Sprite (sx,sy) ...)
     = Sprite (sx + dx, sy + dy) ...

moveSprites :: Coordinate → [Sprite] → [Sprite]
moveSprites (dx,dy) sprites
     = map (moveSprite (dx,dy)) sprites
```

В этом примере используется карринг функции `moveSprite`.

Из утилиты «Bookshelf» [4] для каталогизации документов:

```
-- Files/directories with an associated @.ignore@ file
-- are excluded from the results.
getShelfContents :: FilePath
    → IO ([FilePath], [FilePath], [FilePath], [FilePath])
getShelfContents path = do
  (ds,fs) ← getDirectoryContentsSeparated path

  let (ignores, fs') = partition
          ((".ignore"==) · takeExtension) fs      -- (1)
    ignores' = map dropExtension ignores
    files    = filter (`notElem` ignores') fs'    -- (2)
    dirs     = filter (`notElem` ignores') ds     -- (3)
```

В строках (1), (2) и (3) используются сечения: соответственно левое, правое и снова правое.

Из библиотеки для работы с zip-потоками zlib [31]:

```
-- (module Zlib.Internal)
decompress :: Stream.Format → DecompressParams
              → L.ByteString → L.ByteString
decompress = ...

-- (module Zlib)
decompressWith :: DecompressParams
              → ByteString → ByteString
decompressWith = Internal.decompress zlibFormat

decompress :: ByteString → ByteString
decompress = decompressWith defaultDecompressParams
```

Функции `decompressWith` и `decompress` определены через друг друга при помощи карринга и втроем предоставляют различные уровни настраиваемости декомпрессии.

Карринг играет очень большую роль в удобстве пользования функциями высшего порядка (см. 5.3) и бесточечным стилем (см. 5.6). Это видно, например, в приведенном выше фрагменте кода из Bookshelf.

При программировании на языках с поддержкой карринга часто располагают аргументы функций в порядке увеличения «изменчивости»: вначале идут «настройки», затем «данные». Например, у процедуры поиска ключа в словаре логично сделать словарь первым аргументом, а искомый ключ — вторым, т. к. он более изменчив: обычно в одном и том же словаре ищут несколько ключей, а не наоборот. По этой же причине и у процедуры сопоставления строки с регулярным выражением первым аргументом должно быть регулярное выражение. В данном случае есть и еще одна причина: если оно стоит первым аргументом, то можно скомпилировать его и возвратить процедуру от одного аргумента, использующую скомпилированное выражение:

```
matchesRegexp regex = λs → runCompiledRegex c s
```

```
    where c = compile regex
```

При другом порядке аргументов такая оптимизация была бы затруднена.

Реализация

Карринг реализован почти во всех языках семейства ML, в частности в Haskell, OCaml, F#, Scala (однако в Scala синтаксис объявления каррируемых и некаррируемых функций различается), Coq и других. В языке OCaml реализован более удобный вариант частичного применения за счет поддержки именованных параметров; см. блог-пост «Curried function and labeled arguments in Ocaml» Педро дель Галлего [34]. В Scala также реализован довольно удобный вариант частичного применения: неявный аргумент обозначается за _, например, goodThings.contains(_) обозначает функцию, по x возвращающую goodThings.contains(x).

С точки зрения компиляции в эффективный код поддержка карринга в языке имеет свои тонкости: см., например, статью «Making a fast curry: push/enter vs. eval/apply for higher-order languages» [93] от авторов компилятора GHC Саймона Марлоу и Саймона Пейтона-Джонса, однако существуют техники преобразования программ (дефункционализация: см. ссылки в главе о функциях высшего порядка (см. 5.3), в секции «Имитация»), позволяющие получить из программы эквивалентную ей программу, не использующую карринг и функции высшего порядка и, вследствие этого, лучше поддающуюся компиляции в эффективный код.

В языках, поддерживающих процедуры с нефиксированным числом параметров, карринг в общем случае реализовать невозможно, т. к. если `foo` — процедура с произвольным числом параметров, то непонятно, как интерпретировать, скажем, `foo a b c`: как применение `foo` к трем аргументам `a,b,c`, или же как функцию от оставшихся аргументов (по-прежнему произвольного количества)? Например, к таким языкам относятся диалекты Lisp, в частности, Scheme, а также Ruby, Python, Perl и т. п. Впрочем, в таких языках, конечно же, остается возможность использовать карринг для процедур, число аргументов которых известно. Например, вот процедура `curry` на Scheme, аналогичная приведенной выше процедуре на Haskell:

```
> (define (curry f)
    (lambda (x y)
      (f (cons x y))))
> (define (plus xy) (+ (car xy) (cdr xy)))
> (plus (cons 1 2))
3
> ((curry plus) 1 2)
3
```

Имитация

Имитация карринга возможна при помощи процедур, аналогичных определенной выше процедуре `curry`. Для этого, очевидно, необходимо, чтобы язык поддерживал замыкания (см. 5.4) (поскольку замыкание, возвращаемое `curry`, использует

аргумент `curry`, т. е. «замыкается» над каррируемой процедурой) и функции высшего порядка (см. 5.3) (т. к. процедура `curry`, принимая на вход функцию и возвращая функцию, является ФВП).

В более широком же смысле слова карринг и частичное применение как фиксация некоторых параметров алгоритма могут быть легко сэмулированы, к примеру, при помощи средств ООП или аналогичных им. Так, рассмотренный пример с регулярными выражениями эмулируется практически во всех библиотеках регулярных выражений через использование процедуры компиляции регулярного выражения; рассмотрим код на Java:

```
Pattern isNumber = Pattern.compile("-?[0-9]+");
...
if(isNumber.matcher(s).matches()) {...}
```

Существует также интересный паттерн проектирования «Curried Object» (описан в статье Джеймса Нобла «Arguments and results» [106]). Одно из его применений для облегчения многопоточной работы с изменяемым состоянием рассмотрено в статье «Изменяемое состояние: опасности и борьба с ними» Евгения Кирпичёва ([182]).

5.6. Бесточечный стиль (Point-free style)

Суть

Сборка функций из других функций при помощи комбинаторов, без явного упоминания аргументов.

История

Идея описания функций без обращения к их аргументам берет свои корни из математики. Скажем, оператор Гамильтона («набла») определяется так:

$$\nabla = \frac{\partial}{\partial x}\hat{\mathbf{x}} + \frac{\partial}{\partial y}\hat{\mathbf{y}} + \frac{\partial}{\partial z}\hat{\mathbf{z}}$$

.

В 1924 г., еще до создания лямбда-исчисления, Шейнфинкель создал комбинáторную логику — формализм, подобный лямбда-исчислению, однако не содержащий лямбда-абстракции, и, таким образом, избегающий необходимости в использовании переменных. Вместо лямбда-абстракции комбинаторная логика предоставляет набор базовых комбинаторов и правил редукции; в результате получается также Тьюринг-полный вычислительный формализм (при этом следует помнить, что комбинаторная логика создавалась до формализма Тьюринга и до появления компьютеров как таковых). Комбинаторная логика оказала огромное влияние на современные функциональные языки программирования семейства ML, такие как Haskell.

В программировании данная идея впервые появилась, вероятно, в конкатенативных языках, таких как FORTH. Однако природа бесточечности в этих языках совершенно иная, нежели в математике и в современных функциональных языках: функция в FORTH определяется не математически в виде комбинации каких-то других функций, а императивно, как последовательная композиция манипуляций со стеком.

В современном понимании этого слова бесточечный стиль был, вероятно, впервые описан Джоном Бэкусом в его знаменитой лекции «Can programming be liberated from the Von Neumann style?» ([6]), прочтенной им на церемонии вручении премии Тьюринга в 1977 году. Бесточечный стиль описывается в ней на примере манипуляций со списками, векторами и матрицами. Таким образом демонстрируется удобство формального манипулирования бесточечными определениями для доказательства свойств определенных подобным образом функций.

Практическое применение бесточечный стиль нашел в вышеупомянутых языках семейства ML, развивавшихся с начала 1970 гг.

Пожалуй, последнее из существенных событий в истории бесточечного стиля — появление в начале 1990 гг. языка J, наследника APL. О нем см. ниже в секции «Реализация».

Интуиция

Часто для задания функции не нужно описывать, как она действует на абстрактный *аргумент*, а можно выразить ее действие *в целом*, не обращаясь к *существительным, обозначающим ее аргументы*: например, «Прямые наследники — это ближайшие родственники» (в противовес «Прямые наследники *человека* — это ближайшие родственники *этого человека*»).

Описание

При бесточечном стиле функции по возможности не определяются через результат вычислений над их аргументами (следовательно, анонимные лямбда-выражения тоже не используются). Вместо этого функции выражаются через другие функции при помощи комбинаторов высшего порядка (см. главу о комбинаторных библиотеках (см. 5.14) и о функциях высшего порядка (см. 5.3)). Пожалуй, чаще всего используется оператор композиции функций, обозначаемый кружком (∘).

Такой стиль часто позволяет уменьшить количество лишней информации в определении функции, делая его более лаконичным и читаемым (см. пример в предыдущей секции), хотя и требует определенного привыкания. С другой стороны, злоупотребление бесточечным стилем влияет на читаемость крайне негативно: ср. `λf g x y → f (g x y)` против `(.).(.)`.

Использование

Бесточечный стиль в основном применяется в языках, поддерживающих каррирование (см. 5.5), функции высшего порядка (см. 5.3) и обладающих мощной системой типов — таких, как Haskell, Clean, OCaml, F#. Однако, из соображений читаемости, бесточечные определения целых функций используются очень редко. Несравнимо чаще выражения в бесточечном стиле используются как часть определения. Рассмотрим несколько реальных примеров использования бесточечного стиля в программах на Haskell (намеренно выбраны очень простые примеры).

Пример из аркады «Monadius» [153]:

```
keyProc keystate key ks _ _ =
  case (key,ks) of
    (Char 'q',_)     → exitLoop
    (Char '\ESC',_) → exitLoop
    (_,Down)→ modifyIORef keystate (nub · (++[key])) -- (1)
    (_,Up)  → modifyIORef keystate (filter (≠key)) -- (2)
```

Код, отмеченный `(1)`, `(2)`, означает: При приходе сигнала нажатия клавиши `key` произвести над ячейкой `keystate` модификацию «добавить `key` и удалить дубликаты `(nub)`»; при сигнале отпускания — модификацию «убрать `key`». Первый из отмеченных фрагментов с использованием «точечного» стиля выглядел бы как `λs → nub (s ++[key])`, второй — как `λs → filter (λk → k ≠key) s`.

Пример из комбинаторной библиотеки graphics-drawingcombinators [113]:

```
colorFunc :: (Color → Color) → Draw α → Draw α
colorFunc cf = ...

-- | @color c d@ sets the color of the drawing to exactly @c@.
color :: Color → Draw α → Draw α
color c = colorFunc (const c)
```

Процедура `colorFunc` принимает на вход функцию преобразования цвета (например, увеличение прозрачности в 2 раза выглядело бы как `λ(r,g,b,a) → (r,g,b,a/2)`) и картинку, а возвращает картинку, где цвет каждой точки изменен соответствующим образом. Процедура `color` закрашивает всю картинку одним цветом, подставляя в `colorFunc` функцию, везде равную заданному цвету. В «точечном» стиле процедура `color` выглядела бы так:

```
color c pic = colorFunc (λclr → c) pic
```

А в «еще более бесточечном» — так:

```
color = colorFunc · const
```

Пример из программы «jarfind» для поиска по jar-файлам [79]:

```
run :: Args → IO ()
run (Args dataSource searchSource searchTarget) = do
    classes ← parseDataSource dataSource
    (mapM_ (putStrLn · present)
      · concatMap (search searchSource searchTarget)) classes
```

`parseDataSource` по описанию источника данных возвращает поток разобранных class-файлов. `search` по спецификации того, в каких классах искать (задается с помощью `searchSource`), что именно искать (сами классы или же методы/поля — `searchTarget`) и class-файлу возвращает список результатов (пар из найденного элемента и пути к файлу, содержащему его). В последней строчке данного примера написано: «Выполнить поиск во всех class-файлах из `classes`, сложить результаты, отформатировать и отобразить каждый из них».

И, наконец, пример использования бесточечного стиля на Java, из практики автора: фрагмент кода обнаружения DoS-атак по логам веб-сервера (см. также статью о комбинаторных библиотеках (см. 5.14)):

```
private LogFunction<Double> maxSpeedOverSessions(double
    sessionBreakCoeff)
{
    SessionFunction<List<Double>> movingSpeed =
        moving(SPEED_WINDOW, weightedSpeed(WEIGHT_FUNCTION));
    SessionFunction<List<Double>> movingSpeedOverXml =
        over(pathContains(".xml"), movingSpeed);
    SessionFunction<Double> maxSpeed =
        aggregate(max(0.0), movingSpeedOverXml);
```

```
        return aggregate(max(0.0),
            mapValues(bySession(sessionBreakCoeff, maxSpeed)));
    }
```

«Сессией» называется удовлетворяющий особым условиям участок последовательности запросов от одного IP-адреса. `LogFunction<T>` — «функция из лога в значение типа T», аналогично SessionFunction. Данный фрагмент кода манипулирует различными примитивными функциями и комбинаторами (к примеру, комбинатор `moving` соответствует применению заданной функции к временны*м* подокнам определенной длительности) и собирает из них функцию, вычисляющую по логу скорость самой быстрой последовательности запросов.

И несколько воображаемых примеров:

- Ширина окна = наибольшая ширина составляющих:

  ```
  frameWidth = maximum · map componentWidth · windowContents
  ```

- Товар дорогой, если его цена превышает 100:

  ```
  isExpensive = (>100) · price
  ```

- Технари — те, у кого средняя оценка как по физике, так и по математике, превышает 4.5:

```
scientificMindset = all (>4.5) · map (mean · snd) · scienceMarks
  where
    scienceMarks = filter (scienceSubj · fst) · marks
    scienceSubj = (`elem` ["maths", "physics"])
```

Реализация

Как уже говорилось выше, бесточечный стиль в основном применяется в языках семейства ML: Haskell, F# и т. п. Поддержка системы типов необходима для контроля за корректностью программ с использованием ФВП; ФВП необходимы в роли комбинаторов, собирающих более сложные функции из более простых; частичное применение избавляет от нужды в имитирующих его комбинаторах (к примеру, не будь каррингa — понадобился бы комбинатор «зафиксировать первый из двух аргументов»: `bind1st f x = λy → f x y` — что и имеет место, к примеру, в C++ STL).

Также бесточечный стиль обильно используется в языках семейства APL, в т. ч. J (сайт [68]) и K ([83]). Чрезвычайно интересны и самобытны средства, предоставляемые для бесточечного программирования в языке J, где *подавляющее большинство* функций принято определять бесточечно. Язык J вводит понятия *монадных* (не путать с монадами из Haskell) и *диадных* глаголов (унарных и бинарных операций), «наречий» (модификаторов, применяемых к монаде или диаде для образования новой) а также *вилок* и *крючков* — особых синтаксических комбинаций нескольких монад или

диад, образующих вместе новую монаду или диаду. Так, классический пример использования «вилки» и вообще краткости и стиля языка J — вычисление арифметического среднего:

```
avg =. +/ % #
```

Эту фразу следует читать как «среднее есть сумма, поделенная на длину», и она образована вилкой из монады суммирования списка +/ (где / — наречие свертки (см. 5.11)), диады деления λ% и монады взятия длины λ#. Описание разнообразных видов вилок и крючков можно найти, например, в онлайн-книге Роджера Стокса «Learning J», в главе 8 «Composing verbs» ([146]).

Наконец, стековые (конкатенативные) языки, такие как FORTH или Factor, также по своей сути располагают к бесточечному программированию.

Имитация

Использование бесточечного стиля в нефункциональных языках, не поддерживающих функции высшего порядка и частичное применение, сводится к имитации функций высшего порядка (например, с помощью объектов) и к имитации частичного применения (например, при помощи паттерна «Curried Object» (описан в статье Джеймса Нобла «Arguments and results» [106]). Пример использования бесточечного стиля на Java можно найти в статье и презентации «Функциональное программирование в Java» Евгения Кирпичёва ([180], [181]). Однако такое использование оправдано лишь в специфических случаях, когда действительно необходима крайне низкая синтаксическая нагрузка на конструирование функций при помощи комбинаторов (в основном, задачи сложной обработки данных).

5.7. Хвостовой вызов (Tail call)

Суть

Применение функции, соответствующее *замене* одной вычислительной задачи на другую, вместо *сведения* одной задачи к другой.

История

Первый вычислительный формализм, в котором вообще можно говорить о «вызовах» — лямбда-исчисление. Семантика бета-редукций (аналог «вызова функции») в лямбда-исчислении основана на подстановке, а не на, скажем, операциях над стеком параметров и адресов возврата, поэтому можно сказать, что вычислители, основанные на лямбда-исчислении, поддерживают оптимизацию хвостовых вызовов в изложенном далее по тексту смысле. Несмотря на то, что язык LISP был фактически реализацией лямбда-исчисления, стандарт Common LISP не обязывает оптимизировать хвостовые вызовы. Первый язык программирования, поддерживающий оптимизацию хвостовых вызовов — Scheme (1975). Оптимизация хвостовых вызовов была впервые описана в статье «Lambda: The Ultimate GOTO» [73] из серии фундаментальных статей «Lambda Papers» Гая Стила и Джеральда Сассмана [145]. Статья описывает и способ реализации этой возможности языка, и ее применения.[11]

Интуиция

Рассмотрим две простые процедуры над двоичными деревьями: поиск в двоичном дереве поиска и свертку (см. 5.11).

Будем считать, что деревья определены как алгебраический тип (см. 5.9) `data Tree` α `= Leaf |Fork` α `(Tree` α`) (Tree` α`)`.

```
memberOf :: (Ord α) ⇒ α → Tree α → Bool
_ `memberOf` Leaf        = False
x `memberOf` Fork y l r
  | x == y = True
  | x <  y = x `memberOf` l
  | x >  y = x `memberOf` r
```

Структура решения такова: есть два «крайних» случая (пустое дерево `Leaf` и «вилка» `Fork` с искомым значением), а в каждом из двух оставшихся случаев задача поиска в дереве `t` *заменяется* на задачу поиска в другом дереве — в левой или правой ветке `t`. Если `x <y`, то отыскать `x` в `Fork y l r` — *то же самое*, что и отыскать `x` в `l`, и

[11] Статья появилась еще в то время, когда компиляторы были настолько неразвиты, что вызовы процедур *вообще* недолюбливали из-за низкой производительности. Прошло более 30 лет, однако до сих пор многие программисты и даже разработчики языков не знакомы с доводами, приведенными в этой статье.

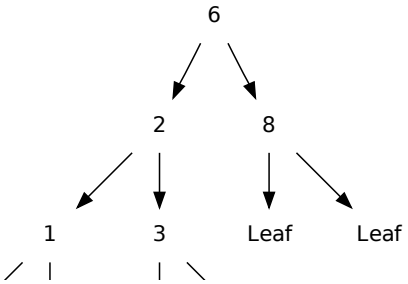


Рис. 5.1. Бинарное дерево, высота которого вычисляется в примере.

аналогично при `x >y`, то есть, ответ к исходной задаче *совпадает* с ответом к другой задаче.

Теперь рассмотрим свертку и небольшой пример ее применения — вычисление высоты дерева.

```
foldTree :: (α → β → β → β) → β → Tree α → β
foldTree fork leaf Leaf          = leaf
foldTree fork leaf (Fork x l r) = fork x vl vr
  where vl = foldTree fork leaf l
        vr = foldTree fork leaf r

height = foldTree (λx hl hr → 1 + max hl hr) 0
```

Структура решения такова: есть «крайний» случай — пустое дерево — и случай, требующий вычисления свертки в левом и правом поддереве и комбинирования результатов. Задача вычисления свертки над `Fork x l r` *сводится* к вычислению сверток над `l` и над `r`, то есть, ее ответ *выражается* через ответы других задач.

Фундаментальное отличие между этими двумя программами и является отличием между хвостовыми и не-хвостовыми вызовами, а также, соответственно, между итерацией и рекурсией.

Описание

В языке Haskell соблюдается ссылочная прозрачность, что позволяет легко манипулировать программами математически и использовать т. н. подстановочную модель вычислений, то есть, заменять произвольное выражение в программе на его результат. Однако ленивость Haskell вносит ряд тонкостей в обсуждение хвостовых вызовов, поэтому будем вести речь о мнимом строгом (не-ленивом) языке с синтаксисом Haskell. Посмотрим, как протекает выполнение двух вышерассмотренных программ. В рамку обведены редуцируемые («раскрываемые») термы.

Вычисление высоты дерева на рис. 5.1 будет протекать так:

height (Fork 6 (Fork 2 (Fork 1 Leaf Leaf) (Fork 3 Leaf Leaf)) (Fork 8 Leaf Leaf))

(Обозначим (Fork n ...) как @n для краткости)

= $1 + max($ height @2 $)(height\ @8)$

= $1 + max(1 + max($ height @1 $)\ (height\ @3))(height\ @8)$

= $1 + max(1 + max(1 + max($ height Leaf $)\ (height\ Leaf))\ (height\ @3))\ (height\ @8)$

= $1 + max(1 + max(1 + max\ 0\ ($ height Leaf $))\ (height\ @3))\ (height @8)$

= $1 + max(1 + max(1 + max\ 0\ 0)\ ($ height @3 $))\ (height\ @8)$

= ...

= 3

Запишем этот же вычислительный процесс в другой форме: будем указывать аргументы текущего вызова, а также то, как выражается окончательный результат через результат текущего вызова (то есть, «продолжение» текущего результата). Для краткости будем писать `h` вместо `height`.

$h\ @6$	$\bullet$
$h\ @2$	$1 + max\ \bullet (h\ @8)$
$h\ @1$	$1 + max\ (1 + max\ \bullet\ (h\ @3))\ (h\ @8)$
$h\ Leaf$	$1 + max\ (1 + max\ (1 + max\ \bullet\ (h\ Leaf))\ (h\ @3))\ (h\ @8)$
$h\ Leaf$	$1 + max\ (1 + max\ (1 + max\ 0\ \bullet)\ (h\ @3))\ (h\ @8)$
$h\ @3$	$1 + max\ (1 + max\ 1\ \bullet)\ (h\ @8)$
$h\ Leaf$	$1 + max\ (1 + max\ 1\ (1 + max\ \bullet\ (h\ Leaf)))\ (h\ @8)$
$h\ Leaf$	$1 + max\ (1 + max\ 1\ (1 + max\ 0\ \bullet))\ (h\ @8)$
$h\ @8$	$1 + max\ 2\ \bullet$
$h\ Leaf$	$1 + max\ 2\ (1 + max\ \bullet\ (h\ Leaf)$
$h\ Leaf$	$1 + max\ 2\ (1 + max\ 0\ \bullet)$
3	$\bullet$

Как видно, в данном процессе форма выражения окончательного результата через результат текущего вызова все усложняется по мере «погружения» рекурсивных вызовов в структуру данных. Представить такое продолжение можно в виде композиции стека «локальных» продолжений, имеющих в данной программе вид $1 + max\ \bullet\ (height\ \ldots)$ или $1 + max\ \ldots\ \bullet$ (продолжение текущего вызова относительно вызывающей процедуры).

Теперь рассмотрим поиск в дереве. Согласно системе уравнений (см. 5.10), задаю-

щих `memberOf`, можно записать следующую цепочку равенств:

$$\begin{aligned} & memberOf\ 3\ @6 \\ = & \ memberOf\ 3\ @2 \\ = & \ memberOf\ 3\ @3 \\ = & \ True \end{aligned}$$

И то же в форме с «продолжениями»:

$memberOf\ 3\ @6$	•
$memberOf\ 3\ @2$	•
$memberOf\ 3\ @3$	•
$True$	•

Как видно, в этом вычислительном процессе на каждом шаге меняется лишь *формулировка* задачи, а *ответ* остается постоянным, и ответ на исходную задачу постоянно равен ответу на текущую, поэтому выражается через него как «•».

Итак, вызов называется хвостовым, если его продолжение в контексте вызывающей процедуры — «•». Эта точная формулировка соответствует принятым туманным определениям, вроде «вызов — последнее предложение (statement) в теле функции и является непосредственным аргументом `return`» и т. п.

Очень важно обратить внимание, что понятие хвостового вызова не исчерпывается понятием хвостовой рекурсии или хвостовой взаимной рекурсии. Данные понятия — лишь частные случаи; впрочем, очень часто встречающиеся, поэтому рассмотрим их отдельно.

При хвостовой рекурсии хвостовой вызов процедуры происходит из этой же самой процедуры. Это имеет место, например, в приведенной процедуре `memberOf`.

При взаимной рекурсии известна и фиксирована система процедур $P_1, \ldots, P_n$ такая, что во всякой из них вызов каждой из остальных является хвостовым. Вот чисто иллюстративный пример взаимно рекурсивной системы функций, задающих крайне неэффективный способ проверки неотрицательных целых на четность и нечетность:

```
even x = x==0 || x≠1 && odd  (x-1)
odd  x = x==1 || x≠0 && even (x-1)

> map even [0..5]
[True,False,True,False,True,False]
> map odd  [0..5]
[False,True,False,True,False,True]
```

Наконец, отметим, что *любая* программа может быть переписана так, чтобы в ней содержались только хвостовые вызовы. Для того, чтобы сделать все вызовы функции `f` хвостовыми, достаточно передавать этой функции ее продолжение в явном

виде («реифицировать» (reify), или овеществить его). Это преобразование называется «преобразование в стиль передачи продолжений». Запишем преобразование для функции `height` в явном виде, для наглядности не выражая ее через свертку:

```
height t = height' t id
  where height' Leaf k       = k 0
        height' (Fork x l r) =
          height' l (λhl →
            height' r (λhr →
              1 + max l r))
```

Здесь функции `foldTree'` в явной форме передается продолжение для ее результата. Рассмотрим для примера вычисление высоты более простого, чем рассмотрено выше, дерева: `Fork 2 (Fork 1 Leaf Leaf) Leaf`.

Первый аргумент	Второй аргумент	Продолжение
$h\ @2$	$\bullet$	$\bullet$
$h\ @1$	$1 + max\ \bullet\ (h\ Leaf)$	$\bullet$
$h\ Leaf$	$1 + max\ (1 + max\ \bullet\ (h\ Leaf))\ (h\ Leaf)$	$\bullet$
$h\ Leaf$	$1 + max\ (1 + max\ 0\ \bullet)\ (h\ Leaf)$	$\bullet$
2		$\bullet$

Впрочем, как видно, потребление памяти этой программой ничуть не меньше, чем потребление памяти версией, не подвергшейся CPS-преобразованию: программы, по сути, оперируют одной и той же информацией, но одна явно, а другая неявно.

Добиться дальнейшего увеличения производительности или возможностей для преобразования или анализа можно, изменив представление передаваемого продолжения. Рассмотрим этот прием на примере вычисления высоты дерева. Как уже говорилось, продолжения рекурсивного вызова относительно вызывающей процедуры в данном случае имеют вид $1 + max\ \bullet\ (height\ \ldots)$ или $1 + max\ \ldots\ \bullet$, а продолжение вызова относительно вызова верхнего уровня имеет вид списка из таких элементов. Определим соответствующую структуру данных явно.

```
data Context α =
    -- heightCPS r (λhr → k (1 + max <?> hr))
  | KLeft  {r ::Tree α, k::Context α}
    -- k (1 + max hl <?>)
  | KRight {hl::Int,    k::Context α}
  | Id

height t = heightS t Id
  where
    heightS Leaf          k  = interp k 0
    heightS (Fork a  l r)  k  = heightS l (KLeft r k)

    interp  Id            x  = x
```

```
interp  (KLeft  r  k) hl = heightS r (KRight  hl k)
interp  (KRight hl k) hr = interp  k (1 + max hl hr)
```

Если внимательно проследить соответствие между этим фрагментом кода и предыдущим, то видно, что один получен из другого чисто механической трансляцией. Каждой из трех разновидностей замыканий (см. 5.4), использованных в качестве аргумента k в предыдущем фрагменте кода, сопоставлен свой конструктор типа `Context`, а аргументами являются свободные переменные этого замыкания. Использованная техника (дефункционализация) подробно описана в статье «Defunctionalization at work» Оливье Дэнви и Лассе Нильсена ([33]), а также в диссертации Нила Митчелла «Transformation and analysis of functional programs», один из разделов которой посвящен дефункционализатору firstify ([100], а также в статье «Losing functions without gaining data» [101], презентации [102]), и статье Митчелла Ванда «Continuation-based program transformation strategies» ([165]).

Использование

Эффективная поддержка хвостовых вызовов в языке позволяет реализовать некоторые структуры управления, например циклы, без соответствующей поддержки со стороны языка. Цикл заменяется на хвосто-рекурсивную процедуру, принимающую в качестве аргументов переменные состояния, используемые циклом. Например, вот императивная процедура и ее хвосто-рекурсивный аналог:

```
int sum(int a,int b) {
    for(int s = 0, i = a; i < b ; ++i)
        s += i;
    return s;
}

sum a b = sum' 0 a
  where sum' s i = if i≥b
                   then s
                   else sum' (s+i) (i+1)
```

Не следует воспринимать этот код как образец хорошего стиля: это чисто иллюстративный пример, демонстрирующий соответстве между циклами и хвостовой рекурсией. Заслуга поддержки хвостовых вызовов, конечно, заключается не в самоотверженном отречении от синтаксиса циклов — в них самих по себе нет ничего плохого — а в том, что становится можно реализовать более сложные структуры управления, такие как, например, конечные автоматы или стиль передачи продолжений (CPS).

На самом деле в функциональных программах принято избегать рекурсии, а вместо этого абстрагировать ее в рекурсивные комбинаторы общего назначения, написанные раз и навсегда. Например, идиоматический вариант данной программы на Haskell выглядел бы так: `mySum a b = sum [a..b]` (где `sum` — встроенная функция в Haskell).

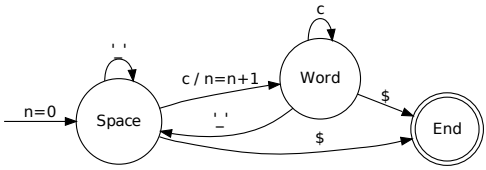


Рис. 5.2. Конечный автомат для подсчета слов

Система взаимнорекурсивных функций применяется для реализации конечных автоматов, причем реализация обычно получается значительно элегантнее, чем реализация при помощи цикла и `switch` или `goto`. Каждая функция соответствует одному состоянию. Рассмотрим, например, конечный автомат, подсчитывающий количество слов в тексте.

```
countWords s = space 0 s
  where space n [] = n
        space n (' ':rest) = space n rest
        space n (c:rest)   = word  (n+1) rest
        word  n [] = n
        word  n (' ':rest) = space n rest
        word  n (c:rest)   = word n rest
```

На рис. 5.2 приведена диаграмма состояний конечного автомата, соответствующего этой программе; сравните их.

Конечные автоматы, реализованные с помощью системы взаиморекурсивных функций, также постоянно применяются в языке многопоточного и распределенного программирования Erlang. Например, вот как может выглядеть на Erlang фрагмент контроллера телефонного аппарата. Для лучшего понимания кода рекомендуется заглянуть в статью о сопоставлении с образцом (см. 5.10).

```
off() -> receive
     on -> disconnected()
  end.
disconnected() -> receive
     {ring, Someone} ->
         ringing(Someone);
     pickupReceiver ->
          dialing([])
     off -> off()
  end.
ringing(Who) -> receive
     pickupReceiver -> talking(Who);
     {abort, Someone} -> disconnected();
     off -> off()
  end.
dialing(Digits) -> receive
     {key, backspace} ->
```

```
            dialing(lists:tail(Digits));
        {key, Digit} ->
            dialing([Digit|Digits]);
        hangUp -> disconnected();
        off -> off()
    after ?DIAL_TIMEOUT ->
        connecting(lists:reverse(Digits))
    end
connecting(Digits) ->  dialNumber(Digits), etc.
```

Возможна ситуация, когда заранее неизвестно, *какая именно* функция вызывается хвостовым вызовом (например, происходит вызов функции, хранившейся в переменной или переданной в качестве аргумента). В первую очередь речь идет о конечных автоматах с заранее неизвестным набором состояний и о стиле передачи продолжений (CPS). См. также обширную и интересную серию постов Джо Маршалла «You knew I'd say something» на тему хвостовых вызовов, их роли и отличия от обычной «хвостовой рекурсии» и циклов [94].

Кроме того, аналогичная ситуация (отсутствие информации о том, какая функция вызывается хвостовым вызовом) возникает и при вызове виртуального метода в программе на объектно-ориентированном языке! См. «Why object-oriended languages need tail calls» Гая Стила ([144]) и «Functional objects» Маттиаса Феллайзена ([37]). В этих двух источниках приводятся аргументы, почему *объектно-ориентированным* языкам особенно необходима эффективная поддержка хвостовых вызовов. На форуме Lambda the Ultimate есть обсуждение этих двух статей ([142]).

Рассмотрим иной пример: поиск самого первого элемента в дереве, удовлетворяющего предикату.

```
(define (with-first p tree
                    with-found when-notfound)
  (define (search tree when-notfound)
    (define (search-right)
      (search (right tree) when-notfound))       ; (*)
    (if (empty? tree)
        (when-notfound)                          ; (*)
        (if (p tree)
            (with-found tree)                    ; (*)
            (search (left tree) search-right))))) ; (*)
  (search tree when-notfound))
```

Спецификация процедуры такова: Возвратить результат выполнения процедуры `with-found` над самым первым (в порядке обхода в глубину слева направо) элементом в дереве `tree`, удовлетворяющим предикату `p`; если же такого элемента нет, то возвратить результат выполнения процедуры `when-notfound`.

Идея решения состоит в том, что:

- если дерево пустое — надо возвратить результат «обработчика неудачи»;

- если дерево удовлетворяет предикату — надо выполнить над ним действие удачи;

- иначе — надо посмотреть в левом поддереве, причем если в нем нужного элемента не нашлось, то *посмотреть в правом (а уж если там не нашлось, то выполнить обработчик неудачи)*. NB: это — вычисление для левого поддерева задачи, абсолютно аналогичной исходной, но всего лишь с другим «обработчиком неудачи»: оно отмечено курсивом.

Это и написано в приведенном коде на Scheme. Код написан с использованием стиля передачи продолжений: процедуре `search` передается дерево и «продолжение неудачи».

Заметим, что вызовы, отмеченные «`(×)`» — хвостовые. Посмотрим, что следует из того, что язык Scheme поддерживает полноценную оптимизацию хвостовых вызовов.

- Глубина стека вызовов при выполнении `with-first` будет постоянной.

- Потребление памяти, однако, не будет постоянным — в аргументе процедуры `when-notfound` будет выстраиваться неявный стек из заданий на обработку правых веток. Размер этого аргумента не будет превышать высоту дерева по левым веткам (т. е. максимальное количество левых поворотов по всем возможным путям в дереве).

- Если само дерево `tree` не удовлетворяет предикату, то задача сведется к вызову `(search (left tree) search-right)`, для которого уже не требуется сам объект `tree`, и он может быть собран сборщиком мусора! Это неочевидное обстоятельство превращает поддержку хвостовых вызовов из простой «оптимизации, превращающей рекурсию в цикл» в мощное средство, способное кардинальным образом уменьшить потребление памяти некоторыми типами программ и упростить разработку программ, для которых своевременное освобождение памяти (в данном случае — данных узла) в подобных ситуациях критично (описано в «Proper tail recursion and space efficiency» Виллиама Клингера [26] и в серии постов Джо Маршалла «You knew I'd say something» [94]).

Поддержка оптимизации хвостовых вызовов обладает одним недостатком: трудно сохранить отладочную информацию о стеке вызовов. Существуют различные способы борьбы с этим: см., например, «A tail-recursive machine with stack inspection» Джона Клементса и Маттиаса Феллайзена ([25]), а также по ссылкам из упомянутой серии постов Джо Маршалла [94].

Еще одно применение поддержки TCO — возможность реализации очень необычных структур управления через преобразование всего кода в CPS на этапе компиляции. В этом случае все без исключения вызовы становятся хвостовыми, однако программа очень сильно усложняется и замедляется; требуются дополнительные этапы обработки кода, чтобы восстановить производительность. Плюс этого подхода —

становятся возможными такие нестандартные структуры управления, как, например, `call/cc`, `shift/reset` («ограниченные продолжения», delimited continuations) и другие, реализовать которые без CPS-преобразования намного труднее. `call/cc` имеет множество интересных применений, многие из которых описаны в статье «Call with current continuation patterns» Даррелла Фергюссона и Деуго Дуайта [38]; огромное количество информации об этой и других разновидностях продолжений содержится на странице-библиографии «Continuations and Continuation Passing Style» [28]. CPS-преобразование кода используется в некоторых реализациях Scheme в качестве промежуточного этапа компиляции.

Реализация

Впервые полноценная поддержка оптимизации хвостовых вызовов (далее TCO) была реализована в языке Scheme; более того, это требуется стандартом языка.

Как ни удивительно, стандарт Common Lisp не требует TCO, хотя ряд реализаций ее все же предоставляют. В похожем на Lisp языке Mathematica она отсутствует.

TCO реализована в OCaml, в Prolog (насколько это понятие там вообще применимо), в Haskell (опять же, насколько это понятие там применимо).

В языке Java, точнее, в виртуальной машине JVM, поддержка TCO отсутствует, поскольку создает ряд трудностей совместимости с моделью безопасности Java, основанной на стеке вызовов.

В языках Scala (смесь Java и ML) и Clojure (диалект Lisp), компилирующихся в код для JVM, поддержка TCO также отсутствует, однако были придуманы обходные решения, пригодные в определенных ситуациях. Эти решения обсуждаются в соответствующих списках рассылки: сначала произошло обсуждение в рассылке Clojure (ветка «Trampoline for mutual recursion» [51], начатая самим Ричем Хикки), которое спровоцировало дискуссию в рассылке Scala (ветка «Tail calls via trampolining and an explicit instruction», начатая Джеймсом Айри [67]).

Существуют следующие подходы к реализации TCO в языке (первые два из них были предложены именно в статье Гая Стила «Lambda: the Ultimate GOTO» [73]):

- Изменение кода, генерируемого для вызовов, стоящих «в хвостовой позиции», т. е. таких, чье продолжение — •: при таком вызове после инструкций передачи аргументов вместо инструкции вызова (кладущей в стек адрес возврата) просто используется инструкция безусловного перехода. Простота этого подхода в низкоуровневых языках типа Си — лишь кажущаяся; на деле же имеющиеся попытки его реализации в компиляторе GCC (основная часть работы произведена Андреасом Бауэром в работе «Compilation of Functional Programming Languages using GCC — Tail Calls» [8]) до сих пор недостаточно успешны (см. баг 513007 в Debian Bug Database [70]), чтобы применять эту возможность широко и без каких-либо ограничений.

- Изменение кода, генерируемого для *всех* вызовов процедур: вместо того, чтобы класть в стек аргументы, а затем адрес возврата, поступают наоборот — сначала

кладут адрес возврата, а затем аргументы, и выполняют безусловный переход по адресу начала вызываемой процедуры.

- Трамплины. Эта техника очень похожа на стандартный способ реализации конечного автомата при помощи цикла следующего вида:

```
while(state != DONE) {
    state = state.workAndNext();
}
```

Она подразумевает, что функция, вместо того, чтобы производить хвостовой вызов другой функции, лишь возвращает, какую функцию надо вызвать и с какими аргументами. Фактически же вызовы производятся в «основном цикле» указанного вида. Пример использования этой техники для имитации TCO можно найти в блог-посте Олега Царёва о парсере заголовочных файлов Си на Python ([188]).

- «Сборка мусора» над фреймами стека вызовов. В этом случае изменяется сам способ реализации вызовов процедур: стек вызовов организуется не в виде последовательной области памяти, а в виде фреймов, содержащих связывания формальных параметров с фактическими и указатель на родительский фрейм. Таким образом получается «спагетти-стек (spaghetti stack)», поскольку один и тот же фрейм может быть родительским для нескольких других. Основное преимущество этой техники — в том, что остается возможность получить доступ к стеку вызовов для отладочных и прочих целей. Подробнее прочитать об этой технике можно в статье «Tail-recursive stack disciplines for an interpreter» Ричарда Келси ([75]) — более того, в этой статье произведен довольно-таки полный, но понятный разбор существующих техник реализации хвостовой рекурсии. Техника сборки мусора используется в SISC Scheme (реализация Scheme для JVM) и Scheme48.

Имитация

Хвостовую рекурсию и взаимную рекурсию (т. е. ситуации, когда функция, чей вызов является хвостовым, известна статически) легко преобразовать в цикл. Схемы и примеры преобразований описаны в лекции 2 «Язык Scheme. Рекурсия и итерация» курса «Функциональное программирование» Евгения Кирпичёва [183] (также там описаны и некоторые другие приемы удаления или уменьшения кратности рекурсии).

В ситуации, когда вызываемая хвостовым вызовом функция известна лишь динамически, применима техника трамплинов.

Выше была описана еще одна техника — преобразование в CPS и последующая дефункционализация. Чтобы применить эту технику в императивном языке (где CPS-форма будет слишком трудна для представления), придется, скорее всего, временно

переписать алгоритм на языке вроде Haskell, OCaml или Scheme, соптимизировать его и затем переписать результат на исходном языке.

5.8. Чисто функциональная структура данных (Purely functional data structure)

Суть

Вместо изменения структуры данных можно формировать новую структуру, немного отличающуюся от старой.

История

В истории развития неизменяемых структур данных можно отметить следующие важные вехи:

- Давно и широко известна техника Copy-on-Write («COW») (описана в Wikipedia: [29]), использующая неизменяемость для оптимизации производительности и потребления памяти.
- В языке LISP, появившемся в 1958 г., одна из основных структур данных — обычный односвязный список, реализованный при помощи т. н. изменяемых «cons-пар». Подавляющее большинство стандартных процедур обработки списков не используют возможность изменения списка и предполагают, что список, по крайней мере в процессе их работы, не изменяется.[12] В языке Scheme в стандарте версии 6 (R6RS [134]) процедуры изменения cons-пар убраны из базовых и вынесены в отдельную библиотеку. В диалекте PLT Scheme 4.0 (2007 г.) типы изменяемых и неизменяемых пар были окончательно разделены, и для изменяемых пар введен свой набор примитивов mcons, mcar, mcdr, set−mcar! и set−mcdr!.
- В 1989 г. вышла работа Дрисколла, Сарнака, Слетора и Тарьяна «Making data structures persistent» [91], посвященная тому, как превратить обычную структуру данных в чисто функциональную.
- В 1997 г. Крис Окасаки опубликовал большую работу «Purely Functional Data Structures» [110], целиком посвященную чисто функциональным структурам данных, их реализации (на ML и Haskell) и анализу производительности. Эта книга в настоящий момент является основным источником информации на данную тему.
- В 2003 г. Мартин Хольтерс продолжил дело Криса Окасаки работой «Efficient data structures in a lazy functional language» [57], где описано несколько новых структур данных и проведен более подробный, чем у Окасаки, анализ производительности некоторых известных структур.

[12] О том, что будет, если нарушить это требование, можно прочитать в посте Мэттью Флэтта «Getting rid of `set-car!` and `set-cdr!`» ([40]).

- Вышедшая в 2009 г. книга Питера Брасса «Advanced data structures» [15] в одной из глав рассказывает о превращении обычных структур данных в чисто функциональные.

Интуиция

Рассмотрим, к примеру, программу, реализующую «искусственный интеллект» для игры в шахматы. Она должна просчитывать развитие игры на много ходов вперед и выбирать вариант, приводящий к наилучшим позициям. Основной процедурой в такой программе, скорее всего, будет процедура, принимающая на вход исходную позицию (включая информацию о том, чей сейчас ход) и число ходов предпросмотра, и возвращающая оптимальный следующий ход вместе с его «рейтингом». Процедура будет перебирать все возможные ходы, вызывать себя рекурсивно в измененной позиции, и выбирать оптимальный ход из полученного списка.

Одна из сложностей в таких программах — «откат» (backtracking): после того как завершился рекурсивный вызов с позицией с учетом некоторого хода, как восстановить исходную позицию с тем, чтобы применить к ней следующий из возможных ходов? Навскидку приходят на ум следующие варианты:

- Откатить ход:

```
findBestMove(Position initial, int lookahead) {
    for(Move move : getPossibleMoves(initial)) {
        initial.makeMove(move);
        ...findBestMove(initial, lookahead-1)...
        initial.undoMove(move);
    }
}
```

- Не производить изменение в самой позиции `initial`, а создавать копию:

```
findBestMove(Position initial, int lookahead) {
    for(Move move : getPossibleMoves(initial)) {
        Position pos = initial.clone();
        pos.makeMove(move);
        ...findBestMove(pos, lookahead-1)...
    }
}
```

  В этом случае не надо отменять ход, зато надо клонировать позицию.

Второй вариант, очевидно, никуда не годится: создавать независимую изменяемую копию целой позиции — слишком дорого. Первый вариант выглядит неплохо, но не поддается параллелизации, а ведь задача перебора игровых позиций — прекрасный кандидат для этого: анализ различных позиций можно проводить совершенно независимо. Проблема в том, что логика «makeMove / рекурсивный-вызов / undoMove»

чисто последовательна по своей сути, и полагается на то, что очередной рекурсивный вызов начнется только после того, как завершится undoMove предыдущего. Если попытаться параллелизовать содержимое цикла, произойдет страшная путаница, как почти всегда происходит при попытке параллелизовать вычисления с изменяемыми структурами данных без строжайшего контроля и титанических усилий.

Решение, используемое типичной *чисто функциональной структурой данных*, ближе ко второму варианту:

```
findBestMove(Position initial, int lookahead) {
    for(Move move : getPossibleMoves(initial)) {
        ...findBestMove(initial.withMove(move),
                        lookahead-1)...
    }
}
```

Процедура withMove не копирует всю исходную позицию, но создает новую позицию, которая использует бóльшую часть внутренних структур данных первой позиции и отличается лишь небольшим количеством данных. Благодаря тому, что при этом подходе вообще не используются изменяемые данные, разделение содержимого структур становится возможным и безопасным. Приемы реализации подобных структур данных будут обсуждаться далее.

Описание

Итак, чисто функциональная структура данных — это структура данных, которую нельзя изменить, но на основе которой можно создать новую структуру данных, немного отличающуюся от первой. Рассмотрим этот подход на паре простых примеров:

Пример: Односвязные списки (стеки)

Приведем реализацию чисто функционального стека на Java.

```
public class Stack<T> {
    private final T head;
    private final Stack<T> tail;

    Stack<T>(T h, Stack<T> t) {
        head=h; tail=t;
    }

    public static <T> Stack<T> empty() {
        return null;
    }
    public static <T> Stack<T> push(T t, Stack<T> s) {
        return new Stack<T>(t, s);
```

```
    }
    public static <T> T top(Stack<T> s) {
        if(s == null) throw new NoSuchElementException();
        return s.head;
    }
    public static <T> Stack<T> pop(Stack<T> s) {
        if(s == null) throw new NoSuchElementException();
        return s.tail;
    }
}
```

Как видно, все стековые операции выполняются за время $O(1)$ и не используют изменяемых данных. Такой стек может использоваться, к примеру, для эффективного хранения текущего стека вызовов в интерпретаторе языка или в библиотеке журналирования: при входе в процедуру он будет заменяться новым стеком, который длиннее на один элемент, а при выходе — на предыдущий стек. Во время процедуры можно получить текущее значение стека вызовов и куда-нибудь его при необходимости сохранить (оно останется в первозданном виде и не «испортится»). Например, это нужно затем, чтобы вывести в асинхронный лог отладочное сообщение (из-за асинхронности лога использование изменяемого стека недопустимо), или даже затем, чтобы вернуться в предыдущую точку программы. В качестве примера реального использования такого стека см. исходный код профайлера antro [177] для сборочных скриптов Ant. Данная структура данных также называется «спагетти-стек (spaghetti stack)»

На рис. 5.3 приведена диаграмма связей между объектами, образующихся в результате выполнения следующего кода:

```
Stack<String> s1 = empty();
Stack<String> s2 = push(push(s1, "a"), "b");
Stack<String> s3 = pop(s2);
Stack<String> s4 = pop(s2);
Stack<String> s5 = push(s4, "c");
```

Пример: Бинарные деревья

Большинство разновидностей бинарных деревьев поиска (используемых для представления множеств, словарей, приоритетных очередей и т. п.) также допускают чисто функциональную реализацию. В типичной ситуации результат выполнения какой-либо простой операции над деревом (скажем, вставка/удаление элемента) отличается от исходного дерева на относительно небольшое количество элементов, пропорциональное высоте дерева, т. е. «меняются» лишь элементы, расположенные неподалеку от пути к затронутой точке дерева. В качестве примера на рис. 5.4 на одной диаграмме объектов приведены два бинарных дерева поиска — a, образованное значениями $0, 1, 3, 4, 6, 8, 10, 11, 13, 14$, и b, образованное добавлением к a значения 7. Как видно, большая часть узлов (например, все левое поддерево) у них общая, однако эти

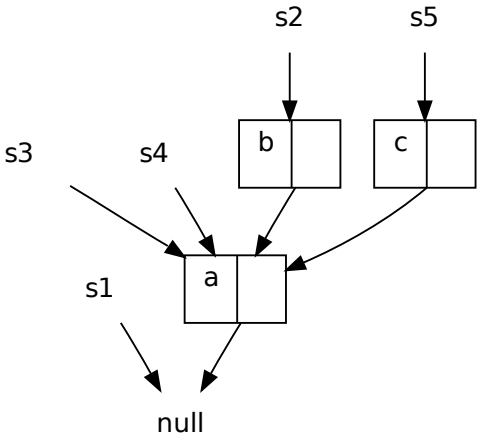


Рис. 5.3. Диаграмма объектов к коду примера со стеками

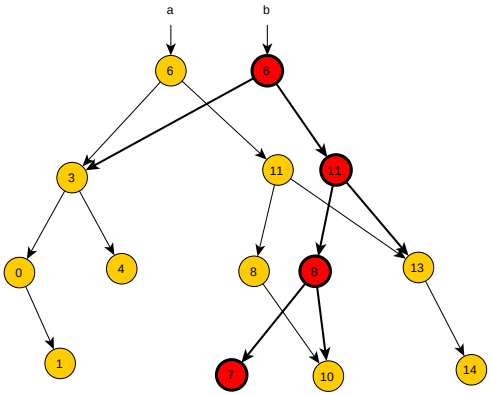


Рис. 5.4. Чисто функциональные деревья

деревья могут быть использованы независимо — опять же, благодаря неизменяемости обоих деревьев.

Существует множество различных чисто функциональных структур данных, основные из которых описаны в книге Окасаки ([110]). Нередко такие структуры используют ленивые вычисления, что затрудняет анализ производительности; в книге Окасаки акцентируется внимание на таком анализе.

Приведем, наконец, возможный интерфейс класса «чисто-функциональный словарь» в типичном императивном языке (обратите внимание на названия процедур «добавления» и «удаления»: в отличие от традиционных put(add)/remove, они названы не глаголами, обозначающими изменение, а союзами).

```
class ImmutableMap {
```

```
    ...
    int size()                          {...}
    boolean isEmpty()                   {...}
    Value get(Key k)                    {...}
    ImmutableMap with(Key k, Value v)   {...}
    ImmutableMap without(Key k)         {...}
    ...
}
```

Использование

Один из примеров использования чисто функциональных структур данных уже был показан выше: они чрезвычайно удобны для реализации комбинаторного перебора с откатами.

Чисто функциональные структуры идеально подходят для многопоточного использования, поскольку у них по определению отсутствуют проблемы с одновременными чтениями/изменениями — экземпляр структуры всегда можно использовать из нескольких потоков, и из любого потока любой экземпляр всегда виден в целостном состоянии. Таким образом, даже в задаче, где требуется изменяемый интерфейс, вместо реализации потокобезопасной изменяемой структуры разумно будет воспользоваться атомарной ссылкой на неизменяемую:

```
class ThreadSafeMutableMap {
    private AtomicReference<ImmutableMap> map;

    Value get(Key k) {
        return map.get().get(k);
    }
    void put(Key k, Value v) {
        ImmutableMap m;
        do {
            m = map.get();
        } while(!map.compareAndSet(m, m.with(k,v)));
    }
    void remove(Key k) {
        ImmutableMap m;
        do {
            m = map.get();
        } while(!map.compareAndSet(m, m.without(k)));
    }
}
```

При таком подходе, помимо гарантий целостности структуры в условиях многопоточного доступа, дополнительно получается и возможность одновременного обхода структуры данных и её изменения. Всякий итератор по структуре (реализованный

как итератор по нижележащей неизменяемой структуре) оказывается всегда целостным (он итерируется по одному из целостных *имевших место* прежних состояний), и не «портится» при изменении структуры данных, поскольку вовсе не «видит» их.

Так как при реализации чисто функциональных структур программист избавлен от проблем, связанных с изменяемым состоянием, то корректно реализовать и оттестировать такую структуру также проще, чем «обычную» (см. статью «Изменяемое состояние: опасности и борьба с ним» Евгения Кирпичёва [182]).

Наконец, возможность разделения частей между между экземплярами структуры в определенных задачах позволяет добиться существенной экономии памяти и открывает возможности оптимизации через кэширование. В задаче из практики автора, где требовалось хранить (и даже модифицировать) большое количество часто пересекающихся множеств целых чисел, переход от хэш-таблицы к неизменяемому красно-черному дереву позволил сократить расход памяти на типичном сценарии, по меньшей мере, в 10 раз. В задаче также требовалось очень частое и быстрое сравнение деревьев на равенство; неизменяемость позволила кэшировать в узлах дерева хэш-код содержимого; без этого приема производительность неизбежно была бы недопустимо низкой.

Реализация

В языке Java, C# и многих других разработчиками стандартных библиотек было принято решение сделать класс строки неизменяемым. Это позволило представлять строки как отрезки неизменяемого массива символов и, как следствие, разделять ссылку на такой массив между несколькими строками, что позволило реализовать очень эффективную операцию взятия подстроки, а также уничтожило как класс ошибки, связанные с неявным изменением строк, достаточно частые в таких языках, как C++. История показала, что это решение оказалось очень удачным и не наложило существенных ограничений на использование строк: в ситуациях, где необходима строка как изменяемая структура данных, используются другие классы (например, в Java используется StringBuilder), которые, однако, используются только для промежуточного представления текста, а не как основной класс строк. Такое же решение (отрезок разделяемого неизменяемого массива) используется в языке Erlang для представления «binaries» (байтовых массивов).

Стандартная библиотека языка Haskell по большей части содержит именно чисто функциональные структуры данных: к примеру, это стандартные списки (модуль `Data.List`), словари (`Data.Map`), множества (`Data.Set`), последовательности с быстрой конкатенацией (`Data.Sequence`) и другие.

Структуры, описанные в книге Окасаки [110], реализованы в его библиотеке «Edison» (существует на hackage [108]). Большое количество других структур данных можно найти на hackage ([154]) в разделе «Data Structures», хотя и не все из них чисто функциональные.

Стандартная библиотека языка Scala (пакет `scala.collections.immutable`) содержит несколько функциональных структур данных.

Стандартная библиотека языка Clojure содержит в основном чисто функциональные структуры данных и реализует таким образом даже хэш-словари и множества (с временем доступа порядка $O(log_{32}n)$).

Стандартные библиотеки практически всех функциональных языков содержат те или иные функциональные структуры.

В библиотеке functionaljava ([171]) реализованы некоторые функциональные структуры, в частности, красно-черные деревья.

Имитация

В языках, поддерживающих сборку мусора, никаких специальных приемов для использования идеи чисто функциональных структур данных не нужно. В языках же без оной возникают проблемы владения узлами структуры: поскольку узлы теперь могут (и должны) разделяться между экземплярами структур, то удаление одного экземпляра не должно влечь за собой удаление всех достижимых из него узлов. Однако, так как в не-ленивом языке невозможно реализовать чисто функциональную структуру данных, могущую содержать циклы ссылок (поскольку невозможно определять значения рекурсивно), то в качестве механизма «сборки мусора» вполне достаточно подсчета ссылок.

5.9. Алгебраический (индуктивный) тип данных (Algebraic (inductive) datatype)

Суть

Тип данных, состоящий из нескольких различимых разновидностей (возможно, составных) термов (значений).

История

Среди языков программирования первым поддерживать простейший аналог алгебраических типов стал, судя по всему, Algol 68, в форме «united modes». Затем в форме «variant records» поддержка схожей и достаточно примитивной возможности появилась в Pascal и других языках. В конце 1970 гг. появился язык ML, поддерживавший современную форму алгебраических типов, включая рекурсивные и полиморфные алгебраические типы; эта же форма (с другим синтаксисом и с небольшими, но важными расширениями) используется в стандарте Haskell'98. ML положил начало другим статически типизированным функциональным языкам. Практически все они сейчас поддерживают алгебраические типы.

В большинстве языков процедурного и объектно-ориентированного программирования (наследниках Си и Smalltalk) их поддержка так и не появилась, и пользователи этих языков довольствуются различными имитациями или стараются избежать их. В некоторых динамических языках с богатыми возможностями метапрограммирования (Lisp, Ruby и т. п.) можно реализовать абстрактные типы данных, схожие по возможностям с алгебраическими типами, как, например, сделано в языке Scheme (пакет «struct.ss» [176]).

В 1992 г. появился язык Coq [30], а с ним и «исчисление индуктивных конструкций» (Calculus of Inductive Constructions, CIC) и чрезвычайно мощное и общее понятие индуктивного типа. Сама концепция алгебраических типов не развивалась существенным образом с момента появления индуктивных типов в CIC, однако находились новые применения некоторым их частным случаям. Так, в 2003 г. в статье «Guarded recursive datatype constructors» [169] была предложена несколько менее общая конструкция, ныне известная под названием «обобщенный алгебраический тип» (Generalized Algebraic Datatype (GADT)). Первым языком программирования общего назначения, в котором появилась полноценная поддержка обобщенных алгебраических типов, был Haskell (начиная с компилятора GHC 6.4, выпущенного в 2005 г.).

Интуиция

Посмотрим, как можно (скажем, для задачи определения столкновений в физическом симуляторе) определить тип «двумерная геометрическая фигура», а именно — прямоугольник, круг или треугольник.

Итак, существует 3 типа фигур:

```
enum ShapeType {RECTANGLE, CIRCLE, TRIANGLE};
```

Прямоугольник определяется левым верхним и правым нижним углом:

```
struct Rectangle {
    Point topLeft, bottomRight;
};
```

Круг — центром и радиусом:

```
struct Circle {
    Point center;
    double radius;
};
```

Треугольник — тремя точками.

```
struct Triangle {
    Point a,b,c;
};
```

Никакого «общего интерфейса», как принято иллюстрировать в учебниках по ООП, у геометрических фигур нет и быть не может: геометрическая фигура — это всего лишь набор координат и длин, ведь речь идет о предметной области «геометрия». Если бы речь шла о «рисовании», то в этом случае, возможно, у фигур были бы методы «нарисовать себя в заданном месте заданной поверхности» и «измерить собственные габариты по отношению к заданной поверхности».

Но как же описать тип данных, совмещающий в себе указание формы фигуры и параметров, необходимых для задания данной конкретной формы?

Первый способ, приходящий на ум программисту, имеющему опыт разработки на Си, Java и т. п. таков: воспользоваться составной структурой данных, хранящей и форму, и параметры. К сожалению, почти все наиболее распространенные современные языки программирования допускают, в сущности, только один способ комбинирования двух элементов данных в одной сущности — помещение их в поля одной и той же структуры или класса. Но при таком способе нельзя учесть, что для формы CIRCLE осмыслен только набор параметров типа Circle, и т. п.

Другой способ заключается в том, чтобы отказаться от разделения на «форму» и «параметры», и описать тип данных явно и безызбыточно: «либо прямоугольник (задаваемый двумя точками), либо круг (задаваемый точкой и числом), либо треугольник (задаваемый тремя точками)». Вот код на Haskell:

```
data Shape = Rectangle {topLeft::Point, bottomRight::Point}
           | Circle    {center::Point, radius::Double}
           | Triangle  {a::Point, b::Point, c::Point}
```

Далее в статье мы будем в основном пользоваться несколько иной нотацией. В этой нотации тип Shape определяется не как набор способов сконструировать значение типа Shape, а как набор заявлений «Терм такого-то вида имеет тип Shape»; отличие довольно тонкое и в данном примере вовсе не проявляется, но оно станет ясно позднее.

```
data Shape where
    Rectangle {topLeft::Point, bottomRight::Point} :: Shape
    Circle    {center::Point, radius::Double} :: Shape
    Triangle  {a::Point, b::Point, c::Point} :: Shape
```

Например:

```
Circle {center=Point{x=1,y=2}, radius=0.5}::Shape
-- Or, more briefly:
Circle (Point 1 2) 0.5 :: Shape
```

Описание

Рассмотрим различные аспекты понятия алгебраического типа по отдельности.

Алгебраические типы позволяют определять типы-произведения (кортежи).

```
data FileInfo where
     FileInfo { name::String, accessRights::Int,
                lastModified::Date } :: FileInfo
```

Как видно, тип `FileInfo` задается утверждением «Если `n` — строка, `ar` — целое число, `lm` — дата, то `FileInfo { name=n, accessRights=ar, lastModified=lm }` (сокращенно `FileInfo n ar lm`) имеет тип `FileInfo`». Такой тип соответствует обыкновенной «структуре» в языках семейства Си.

Под термином «тип-произведение» в данном случае имеется в виду, что тип `FileInfo` соответствует декартову произведению множеств, соответствующих его компонентам.

Алгебраические типы позволяют определять типы-суммы.

```
data Color where
    Red :: Color
    Orange :: Color
    Yellow :: Color
    Green :: Color
    LightBlue :: Color
    Blue :: Color
    Violet :: Color
```

Как видно, тип `Color` задается утверждениями «`Red` — цвет», «`Orange` — цвет», «`Yellow` — цвет» и т. п. Такой тип соответствует «перечислению» (enumeration) в языках семейства Си.

Под термином «тип-сумма» в данном случае имеется в виду, что множество термов, имеющих тип `Color`, *складывается* из термов вида «`Red`», «`Green`», «`Blue`», «`Yellow`».

Алгебраические типы позволяют определять типы вида «сумма произведений».

```
data Shape where
    Rectangle {topLeft::Point, bottomRight::Point} :: Shape
    Circle    {center::Point, radius::Double} :: Shape
    Triangle  {a::Point, b::Point, c::Point} :: Shape
```

Этот пример сочетает в себе две предыдущих возможности: тип `Shape` *складывается* из термов вида `Rectangle ... ...` и т. п.

Важно понимать, что `Rectangle`, `Circle`, `Triangle` сами по себе не являются типами в языке Haskell и в большинстве других языков с поддержкой алгебраических типов — это лишь подмножества значений типа `Shape`, так же как, скажем, чётные числа — подмножество значений типа `int` в Java. Фундаментальных теоретических препятствий для этого нет, однако если бы `Rectangle` был сам по себе типом, то значение `Rectangle p1 p2` имело бы сразу два типа: `Rectangle` и `Shape`, что существенно затрудняло бы вывод типов (см. 5.2) и усложняло язык в целом.

Определения алгебраических типов могут быть рекурсивны и взаимно рекурсивны.

```
data Document where
    Paragraph {text::String} :: Document
    Picture {picture::Image} :: Document
    Sequence {elements::[Document]} :: Document
    Table {rowHeadings::[String], colHeadings::[String],
           cells::[[Document]]} :: Document
```

Как видно, в этом примере тип `Document` задан так:

- Если `s::String`, то `Paragraph s` — документ;
- Если `p::Image`, то `Picture p` — документ;
- Если `ds::[Document]`, то `Sequence ds` — документ;
- Если `rh::[String]`, `ch::[String]`, `c::[[Document]]`, то `Table rh ch c` — документ.

Таким образом, например, следующий терм — документ:

```
Sequence [Paragraph "hello",
          Picture (imageFromFile "earth.jpg")]
```

В этом примере тип `Document` определен через самого себя, т. е. рекурсивен.

```
data Picture where
    PictureFromFile {fileName::String} :: Picture
    AutoFigure {figure::Shape} :: Picture
    Overlay {layers::[Picture]} :: Picture
    EmbeddedDocument {document::Document} :: Picture
```

Картинка определена как «картинка из файла», «автофигура», «композиция нескольких картинок» или же «встроенный документ» (например, MS Office позволяет использовать большинство средств форматирования текста внутри текстовых элементов картинок). Как видно, типы `Document` и `Picture` определены друг через друга, т. е., взаимно рекурсивны.

Алгебраические типы могут быть параметризованы другими типами.

```
data Tree α where
    Leaf :: Tree α
    Fork {value::α, left::Tree α, right::Tree α} :: Tree α
```

В этом примере задан тип «Бинарное дерево со значениями типа α в узлах (сокращенно: дерево α)». Он задан так: «`Leaf` — дерево α (заметьте: независимо от α)», «если `x::`α, `t1::Tree` α, `t2::Tree` α, то `Fork x t1 t2` — дерево α». Например, `Leaf::Tree Int` (пустое дерево является деревом со значениями типа `Int` (как, впрочем, и любого другого) в узлах), а также:

```
Fork "a" (Fork "b"  Leaf Leaf) Leaf :: Tree String

data Hash k h v where
    Hash {hash::k→h, equal::k→k→Bool,
          table::Array h [(k,v)]}
```

В этом примере задан тип «Хэш-таблица из `k` в `v` с хэшами типа `h`». Если `k,h,v` — типы, `f::k→ h, e::k→ k→ Bool, t::Array h [(k,v)]` (что означает: массив с индексами типа `h` и значениями типа «список пар `(k,v)`»), то `Hash f e t` — хэш-таблица из `k` в `v` с хэшами типа `h`.

Интуитивно хочется избавиться от упоминания типа `h` в типе `Hash k h v`: он не имеет отношения к интерфейсу хэш-таблицы как отображения из ключа в значение. Вскоре мы увидим, что это возможно.

Алгебраические типы допускают неоднородную рекурсию: аргумент рекурсивного применения конструктора типа может быть сложным.

```
data RAList α where
    Nil      :: RAList α
    ZeroCons :: RAList (α,α) → RAList α
    OneCons  :: α → RAList (α,α) → RAList α
```

Здесь определен хитроумный способ задания структуры данных «список с произвольным доступом (random-access list)» при помощи так называемых «вложенных», или «полиморфно рекурсивных» типов. За алгоритмическими подробностями предлагается обратиться в книгу Криса Окасаки «Purely functional data structures» ([110]) и презентацию Евгения Кирпичёва по теме книги ([179], слайды 37—39).

Как видно, конструктор `OneCons`, конструирующий терм типа `RAList` α, имеет одним из аргументов значение типа `RAList (`α`,`α`)`. Например, `OneCons 5 (ZeroCons (OneCons ((6,2),(4,3)) Nil)) :: RAList Int`. Применения этой разновидности алгебраических типов см. в секции «Применение».

Конструкторы алгебраического типа T могут упоминать не только те типовые переменные, по которым параметризован тип T.

```
data Hash k v where
    Hash {hash::k→h, equal::k→k→Bool,
          table::Array h [(k,v)]}
```

В этом примере задан тип «Хэш-таблица из `k` в `v`». Если `k,h,v` — типы, `f::k→ h`, `e::k→ k→ Bool`, `t::Array h [(k,v)]`, то `Hash f e t` — хэш-таблица из `k` в `v`. Обратите внимание на разницу между этим объяснением и объяснением, данным выше: она заключается в том и только том, что из типа хэш-таблицы убрано упоминание о внутренне используемом ею типе хэшей. Таким образом, имея значение типа `Hash k v`, уже невозможно узнать, какой именно тип эта таблица использует для хэшей, однако же известно, что *какой-то* — использует, и если обозначить этот тип за `h`, то поле `hash` этой таблицы будет иметь тип `k → h` и т. п. Это позволяет реализовать все операции над хэш-таблицей без знания конкретного типа `h`.

Такой тип называется «экзистенциальным (existential)», поскольку множество его значений составляют термы, квантифицированные *квантором существования*:

$$Hash\ k\ v\ =\ \{Hash\ f\ e\ t \parallel \exists h:\ f :: k \to h,\ e :: k \to k\ toBool,\ t :: Array\ h\ [(k, v)]\}\,.$$

Экзистенциальные типы играют огромную роль при реализации абстрактных типов данных, поскольку, в общем-то, и являются воплощением абстракции как процесса «забывания» чего-то конкретного (в данном случае — конкретного типа `h`). В этом смысле они тесно связаны с понятием инкапсуляции в ООП. Хороший обзор применения экзистенциальных типов для реализации абстрактных типов можно найти в первой половине статьи Мартина Одерски и Константина Лойфера «Polymorphic type inference and abstract data types » ([86]). В особенности интересными и полезными экзистенциальные типы становятся в комбинации с классами типов (см. 5.13); см. статью Лойфера «Type classes with existential types» [85]. Также см. страницы «Existential type» [185] и «OOP vs type classes» [186] в Haskell wiki.

Типы результатов веток в определении параметризованного алгебраического типа могут различаться.

Попробуем спроектировать тип `Expr`, соответствующий синтаксическому дереву очень простого мини-языка для задания выражений, и мини-интерпретатор этого языка. Параметризуем тип в соответствии со здравым смыслом — т. е. так, чтобы в результате интерпретации `Expr` α получалось значение типа α.

```
data Expr α where
    Const :: α → Expr α
    Add :: Expr Int → Expr Int → Expr Int
    Equal :: (Eq α) ⇒ Expr α → Expr α → Expr Bool
    IfThenElse :: Expr Bool → Expr α → Expr α → Expr α
```

Например:

```
IfThenElse (Equal (Const 1) (Add (Const 2) (Const 3)))
           (Const False)
           (Const True) :: Expr Bool
```

Определим теперь (при помощи сопоставления с образцом (см. 5.10)) функцию интерпретации таких выражений (сами по себе они, разумеется, ничего не означают, и представляют собой лишь структуры данных).

```
interp :: Expr α → α
interp (Const x)          = x
interp (Add e1 e2)        = interp e1 + interp e2
interp (Equal e1 e2)      = interp e1 == interp e2
interp (IfThenElse c t e) = if interp c
                            then interp t
                            else interp e
```

Тогда вычисление описанного выражения `interp (IfThenElse ...)` даст `True`.

Специфика данного примера, как видно, в том, что типовый параметр α конструируемого терма у разных конструкторов был разным: у `Add` — `Int`, у `Equal` — `Bool`, у `Const` он был полиморфен, а у `Equal` — даже ограниченно полиморфен по типам, принадлежащим к классу (см. 5.13) `Eq`.

Большинство примеров, использующих данную возможность алгебраических типов, также реализуют тот или иной «язык» (с другой стороны, любая структура данных — своего рода язык). Это существенно увеличивает возможности системы типов по обеспечению корректности (иначе очень трудно определить алгебраический тип `Expr`, гарантируя, что значения типа `Expr a` не имеют ошибок типизации, например `IfThenElse (Const 1) ...`.

Довольно часто (в т. ч. и в стандарте Haskell'98) под термином «алгебраический тип» понимается несколько более узкое понятие: а именно, из рассмотренных возможностей исключаются две последних, поскольку они существенно усложняют реализацию языка. В этом случае становится возможным записать алгебраический тип в упрощенном синтаксисе (именно он используется в Haskell'98). Кроме того, поля конструкторов не обязаны быть именованными:

```
data Tree α = Leaf
            | Fork {value::α, left::Tree α, right::Tree α}
data Tree α = Leaf
            | Fork α (Tree α) (Tree α)
```

Типы же, использующие эти возможности, называются обобщенными алгебраическими (Generalized Algebraic Data Type, GADT).

Возможностями GADT разнообразие и мощь концепции алгебраического типа не исчерпывается: в языке Coq, поддерживающем т. н. зависимые типы, алгебраические типы называются «индуктивными» (используется система типов «Calculus of Inductive Constructions» — она хорошо описана в книге Ива Берто и Пьера Кастеррана о системе Coq «Coq'art: Interactive theorem proving and program development» [10], а формально задана в статье Кристины Паулин-Моринг «Inductive definitions in the system Coq — Rules and properties» [114]) и обладают фактически неограниченной выразительной мощью, позволяя в т. ч. выражать сколь угодно сложные классы структур данных с инвариантами. Мы не будем подробно останавливаться на них в этой статье — краткий обзор приведен в презентации Евгения Кирпичёва [178], а более полный можно найти в источниках, указанных в ней.

Для всякого алгебраического типа определена чрезвычайно общая операция свертки (см. 5.11), абстрагирующая способ вычисления «по индукции» (снизу вверх) над значениями такого типа. Предлагается обратиться за разъяснениями к соответствующей подстатье. Для обычных алгебраических типов составление процедуры свертки тривиально. Кроме того, существуют обобщения понятия свертки и на вложенные и обобщенные алгебраические типы; их автоматическое порождение реализовано в языке Coq под названием «принципов индукции» (induction principles).

У алгебраических типов есть и недостатки:

- Расширить алгебраический тип можно только путем редактирования его определения. Положительная сторона этого ограничения — то, что компилятор всегда может проверить, все ли случаи учитываются при сопоставлении с образцом.

- Алгебраические типы иногда раскрывают слишком много «частных» подробностей реализации структуры данных. Смена представления алгебраического типа, переименование конструктора, добавление или удаление конструкторов обычно ведут к тому, что клиентский код перестает компилироваться (отчасти тут помогают экзистенциальные типы). Поэтому довольно часто при программировании на Haskell вместо алгебраических типов используются абстрактные типы (типы, с которыми клиент может работать исключительно в терминах вспомогательных функций, ничего не зная об их реализации). Разработчик библиотеки *не экспортирует* из модуля конструкторы алгебраического типа, а вместо этого предоставляет различные вспомогательные функции. Примеры см. в секции «Использование».

Использование

«Обычные» (не обобщенные) алгебраические типы используются в поддерживающих их языках (Haskell, OCaml, F#, Scala и т. п.) повсюду: например, в Haskell *все*

определяемые пользователем типы данных — алгебраические; вообще говоря, алгебраические типы намного лучше подходят для описания структур данных и позволяют намного естественнее записывать алгоритмы их обработки (при помощи сопоставления с образцом (см. 5.10)), чем структуры (записи) или объекты. Хотя чаще других используются «тривиальные» алгебраические типы, которые могли бы быть записаны при помощи структуры или перечисления, но примеры, когда тривиальным типом не обойтись, имеются в изобилии. Приведем несколько совершенно произвольных отрывков из библиотек языка Haskell, опуская тривиальные типы.

Определение типа JSON-выражений из библиотеки json [39]:

```
data JSValue = JSNull
             | JSBool Bool
             | JSRational Bool Rational
             | JSString JSString
             | JSArray [JSValue]
             | JSObject (JSObject JSValue)
```

Определение типа SQL-значений из библиотеки HDBC [46]:

```
data SqlValue = SqlString String
              | SqlByteString ByteString
              | SqlWord32 Word32
              ...
              | SqlDouble Double
              | SqlLocalTimeOfDay TimeOfDay
              | SqlZonedLocalTimeOfDay TimeOfDay TimeZone
              ...
              | SqlTimeDiff Integer
              | SqlNull
```

Определение типа «Игровой объект» из игры Monadius [153] (фрагмент):

```
data GameObject = VicViper{ -- player's fighter.
 ... speed :: Double, powerUpPointer :: Int, ...}
                -- missile that fly along the terrain
                | StandardMissile{
 ... position :: Complex Double,hitDisp :: Shape,
       probe :: GameObject,... }
                | PowerUpCapsule {
 ... position :: Complex Double,hitDisp :: Shape,
       hp :: Int,age :: Int }
                | ...
                | ScrambleHatch{
 ... position :: Complex Double,gateAngle :: Double,
     gravity :: Complex Double,hitDisp :: Shape,hp :: Int,
     age :: Int,launchProgram :: [[GameObject]]}
                | ...
```

Абстрактные типы используются вместо алгебраических очень часто: например, в стандартном модуле словарей языка Haskell Data.Map (документация: [174]), в модуле приоритетных очередей heap [49], в библиотеке «идеального хэширования» PerfectHash [116], обильно используются в модуле-привязке к физическому движку «Hipmunk» [56], в библиотеке для построения графиков Chart [19] (например, в модуле Grid) и т. д.

Один из плюсов использования абстрактных типов — возможность создания «умных» конструкторов и хранения рядом со значениями дополнительных данных. Например, в Chart тип Grid (табличная укладка экранных элементов), оператор «обертывание в единичную ячейку» и оператор «один над другим» определены так:

```
type Size = (Int,Int)
type SpaceWeight = (Double,Double)
data Grid α = Value (α,Span,SpaceWeight)
            | Above (Grid α) (Grid α) Size
            | Beside (Grid α) (Grid α) Size
            | Overlay (Grid α) (Grid α) Size
            | Empty
            | Null
-- There are also functions 'width','height' that extract
-- the corresponding field from a Grid's Size.
tval a = Value (a,(1,1),(0,0))

above Null t = t
above t Null = t
above t1 t2 = Above t1 t2 size
    where size = (max (width t1) (width t2),
                  height t1 + height t2)
```

Таким образом, в значении типа `Grid` «закэширован» его размер в ячейках, что позволяет упростить и сделать более эффективными реализации некоторых операций. Пользователю библиотеки об этом ничего не известно, так как экспортированные вспомогательные функции сами заботятся о кэшировании и непротиворечивости дополнительных данных.

Вложенные типы данных и полиморфная рекурсия в основном применяются для задания структур данных со сложными инвариантами. Они используются во многих структурах данных в фундаментальном труде Криса Окасаки «Purely functional data structures» [110]. Одна из таких структур описана в статье того же автора «Binomial queues as a nested type» ([107]). Еще один экзотический, но интересный пример — циклические структуры данных, позволяющие при реализации операций над ними учитывать зацикленность в явном виде (статья «Representing cyclic structures as nested datatypes» [132]).

Что касается применений обобщенных алгебраических типов, то вы можете ознакомиться со следующими статьями.

- Ralph Hinze, «Fun with Phantom Types» ([54]): превосходный обзор, описыва-

ющий: язык выражений, подобный рассмотренному выше; обобщенные функции, в определенном смысле эмулирующие динамическую типизацию; улучшение класса Show; нормализацию типизированных лямбда-термов; типизированный аналог printf.

- Ralf Hinze, Johan Jeuring, и Andres Löh, «Typed Contracts for Functional Programming» ([55]): интереснейшая статья, предлагающая реализованную с помощью GADT библиотеку, позволяющую упростить поиск первопричин нарушения «контрактов» функций в программе.
- Ganesh Sittampalam, «Darcs and GADTs»: в статье рассказано, как применение GADT для представления «патчей» в системе контроля версий darcs позволило упростить код и обнаружить ошибку в старом коде.
- Andrew Kennedy, Claudio V. Russo, «Generalized Algebraic Datatypes and Object-Oriented Programming» ([77]): эта обширная статья демонстрирует параллели между GADT и некоторыми паттернами ООП.

Реализация

Алгебраические типы реализованы в различных формах во множестве языков:

- Haskell реализует их в представленном в данной статье объеме.
- OCaml реализует обычные (не обобщенные) алгебраические типы, но не поддерживает ни вложенные типы, ни необходимую для обращения с ними полиморфную рекурсию (впрочем, существует синтаксическое расширение, реализующее полиморфную рекурсию).
- Scala реализует обычные и обобщенные алгебраические типы.
- Для Scheme существуют пакеты макросов, позволяющие использовать для задания «типов» и для сопоставления с образцом синтаксис, схожий с синтаксисом алгебраических типов и сопоставления с образцом в других языках; впрочем, за отсутствием у Scheme статической системы типов нельзя говорить о поддержке алгебраических типов в этом языке.
- F# как надмножество OCaml реализует обычные алгебраические типы.
- Coq, Agda, Epigram реализуют индуктивные типы (индуктивные типы Coq описаны в книге «Coq'art: Interactive theorem proving and program development» [10] и в презентации Евгения Кирпичёва [178]).

Имитация

Задачи, решаемые алгебраическими типами, возникают на практике постоянно. В программистском фольклоре существует множество (неудобных) способов их решения без поддержки АТД. Вот некоторые из них.

Использование одной структуры/класса с обнуляемыми полями.

Например:

```
enum exp_type {
    et_const_int, et_const_bool, et_binop, et_unop
};
enum op_type {
    op_plus, op_minus, op_mul, op_div, op_unaryminus, op_unarynot
};
struct expression {
    exp_type type;
    op_type operation;      // If type = binop or unop.
                            // If type = unop, only op_unary_* is
                                allowed
    void *const_value;      // If type = et_const_int or
        et_const_bool
    struct expression *op1; // If type = binop or unop
    struct expression *op2; // If type = binop
};
```

Этот способ особенно часто используется в языках типа Java или C#, не поддерживающих конструкцию union. Недостатки очевидны:

- Код, оперирующий такого рода структурами, уродлив (сравните функцию интерпретации схожего типа на Haskell выше, и представьте себе реализацию такой функции для этого типа на Си).
- Код хрупок и подвержен ошибкам. Например, написав et_unop вместо et_binop в одной из веток switch (exp→type), программист обрекает себя на ошибку доступа к полю op2.
- При заполнении и изменении подобных структур очень легко забыть проинициализировать какое-либо поле.
- Хранение таких структур неэффективно в плане потребляемой памяти: например, при хранении значения типа et_const_bool зря занимают место в памяти три лишних поля: operation, op1, op2.

В целом, структура напичкана неявными инвариантами, с поддержанием которых компилятор никак не может помочь, и неэффективна.

Использование одной структуры/класса с union.

```
struct expression {
   exp_type type;
   union {
```

```
      struct { // If type = unop
        op_type operation;
        struct expression *operand;
      } unop;
      struct { // If type = binop
        op_type operation;
        struct expression *operand1;
        struct expression *operand2;
      } binop;
      int int_value; // If type = const_int
      bool bool_value; // If type = const_bool
    } data;
};
```

Инвариантов в этом коде меньше, чем в предыдущем, поэтому он несколько лучше и безопаснее. Кроме того, такое представление требует меньше памяти, хоть идеала и не достигает. Впрочем, инициализация таких структур и оперирование с ними по-прежнему чрезвычайно уродливо. Несмотря на все недостатки, этот вариант используется достаточно часто.

Неявный union.

```
struct expression {
  exp_type type;
};
struct unop {
  op_type operation;
  struct expression *operand;
};
struct binop {
  op_type operation;
  struct expression *operand1;
  struct expression *operand2;
};
int interpret(struct expression *e) {
  switch(e->type) {
    case et_const_int:
      return *(int*)(e+1);
    case et_unop:
      struct expression *operand = ((struct unop*)(e+1))->operand;
      ...
  }
}
```

Здесь подразумевается, что если type==et_unop, то в памяти непосредственно после type расположена структура типа unop, и т. п.

Этот подход довольно часто используется, например, в WinAPI, в частности, в Security API ([3]). Из его достоинств можно назвать оптимальное использование памяти, из недостатков — по уродливости кода, трудоемкости и подверженности ошибкам он многократно превосходит оба предыдущих способа вместе взятых.

Объектно-ориентированная имитация при помощи наследования и диспетчеризации по типу

```
enum ExpressionType {...}
abstract class Expression {
  ExpressionType type;
  Expression(ExpressionType t) {type=t;}
}
class Unop extends Expression {
  UnaryOperation op;
  Expression operand;
  Unop(UnaryOperation op,Expression operand) {
    super(ExpressionType.UNOP);
    this.op=op; this.operand=operand;
  }
}
class Binop extends Expression {...}
...

class ExpressionUtils {
  Object eval(Expression exp) {
    switch(exp.type) {
      case ExpressionType.UNOP:
        ...
      ..
    }
  }
}
```

При этом подходе для алгебраического типа создается базовый абстрактный класс, а для каждого конструктора создается по наследнику. Обработка значений такого типа осуществляется либо через операторы проверки и приведения типов (expr instanceof Unop, (Unop)expr), либо при помощи приема, описанного далее. Минусы этого подхода: а) по-прежнему отсутствует проверка полноты разбора случаев; б) код, написанный в стиле ExpressionUtils . eval и выполняющий диспетчеризацию по типу или полю, имитирующему тип, в ООП считается антипаттерном и его рекомендуется заменять на использование виртуальных функций, паттерна Visitor и т. п. Использование виртуальных функций в классе Expression в такой ситуации не всегда оправдано, т. к. на этапе его проектирования может быть неизвестно, какие именно операции понадобятся. Обильное использование такой техники может легко привести к замусориванию класса разнородным кодом. Отчасти решает эту проблему

паттерн Visitor, описанный в классической книге о паттернах проектирования ([35]), предназначенный для реализации двойной диспетчеризации.

Кодировка Чёрча и паттерн Visitor

```
interface ExpressionVisitor<T> {
  T visitConstInt(Integer value);
  T visitConstBool(Boolean value);
  T visitUnop(UnaryOperation op, Expression operand);
  // Or: T visitUnop(Unop expr);
  T visitBinop(BinaryOperation op, Expression rand1, Expression
      rand2);
  // Or: T visitBinop(Binop expr);
}
abstract class Expression {
  public abstract <T> T accept(ExpressionVisitor<T> v);
}
class Unop {
  UnaryOperation op;
  Expression operand;
  public <T> T accept(ExpressionVisitor<T> v) {
    return v.visitUnop(op, operand);
  }
}
...
```

Паттерн Visitor подробно описан в книге о шаблонах проектирования ([35]) и в огромном количестве других источников, в основном в варианте, приведенном в комментариях («Or:...»).

Раскомментированный вариант `visitUnop` и `visitBinop` — частный случай кодировки Чёрча (см. статью о ней в Wikipedia [21], и главы об алгебраических типах в книге «Design concepts in programming languages» [157], см. также статью Янсена, Коопмана и Пласмейера «Efficient interpretation by transforming data types and patterns to functions» [69], где реализуется небольшой язык программирования с поддержкой алгебраических типов и сопоставления с образцом при помощи кодировки Чёрча, и эта реализация оказывается чрезвычайно эффективной). Этот прием позволяет представлять структуры данных и сопоставление с образцом при помощи одних лишь функций. В блог-посте «Structural pattern matching in Java» Оли Рунара [112] можно найти пример кода на Java, иллюстрирующего представление структур данных в кодировке Чёрча. Помимо имитации алгебраических типов кодировка Чёрча иногда используется и для их реализации, в качестве промежуточной стадии компиляции или оптимизации (такое применение описано в книге «Design concepts in programming languages»).

Использование абстрактного типа данных

И, наконец, можно, *реализовав* алгебраический тип любым из описанных способов, *предоставить интерфейс* к нему каким-либо другим способом. Интерфейс можно использовать независимо от того, как реализован сам алгебраический тип, поэтому клиентский код продолжит работать при изменении его представления. Вот несколько вариантов такого интерфейса:

- Интерфейс, подобный классу `ExpressionVisitor` и функции `match` выше. Например, вот как можно реализовать `match` при представлении с помощью конструкторов:

```
module Trees (
  Tree(),  -- Do not export constructors!
  match
) where
data Tree α = Leaf | Fork α (Tree α) (Tree α)
match lf fk Leaf = lf
match lf fk (Fork a t1 t2) = fk t1 t2
```

- Интерфейс, основанный на т. н. абстракции конструкторов, селекторов и предикатов. Конструкторы — это функции, позволяющие создать значение данного типа. Селекторы обеспечивают доступ к индивидуальным компонентам значений. Предикаты используются для различения разновидностей значений (например, чтобы отличить дерево-«лист» от дерева-«вилки»). Иногда к этой классификации добавляют еще «мутаторы» (процедуры, изменяющие структуру данных) и «запросы» (нетривиальные процедуры, вычисляющие определенные свойства структуры данных, например, высоту дерева или принадлежность ключа к словарю).

```
module Trees (
  Tree(),
  leaf,fork,
  value,left,right,
  isLeaf,isFork
)
-- Constructors
leaf = Leaf
fork a t1 t2 = Fork a t1 t2
-- Selectors (none for Leaf in this case)
value (Fork a l r) = a
left  (Fork a l r) = l
right (Fork a l r) = r
-- Predicates
isLeaf Leaf = True ; isLeaf _ = False
isFork Leaf = False ; isFork _ = True
```

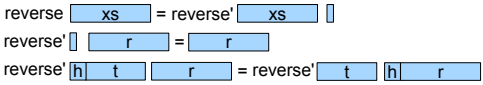


Рис. 5.5. Обращение списка

5.10. Сопоставление с образцом (Pattern matching)

Суть

Сопоставление формы структуры данных с формой образца и заполнение переменных-«дырок» в образце значениями в соответствующих местах структуры данных.

История

Первые языки с поддержкой сопоставления с образцом появились в 1960—1970-е гг; самым первым из них был SNOBOL — язык для обработки текста. Он предоставлял чрезвычайно богатые возможности по манипулированию образцами, а также позволял обращаться с ними как с объектами первого класса. Однако SNOBOL — очень экзотический язык. Согласно книге Саймона Пейтона-Джонса «Implementation of functional programming languages» [121], из более современных языков «первопроходцами» были ISWIM (абстрактный язык без реализации, представленный Петром Ландиным в его знаменитой статье «The Next 700 Programming Languages» [84]), SASL, NPL (далее эволюционировавший в Hope — предшественника Miranda и Haskell). В развитие технологии сопоставления с образцом внесли свой вклад такие языки, как РЕФАЛ (1968), Prolog (1972) и Mathematica (1988).

В конце 1980-х появились различные вариации на тему «представлений» (views), позволяющих решить некоторые проблемы расширяемости кода с использованием алгебраических типов, повысить его уровень абстракции и близости к предметной области, и т.д (см. конец секции «Описание»). Так, в 1987 г. вышла статья Филипа Вадлера на эту тему «Views: a way for pattern matching to cohabit with data abstraction» ([162]). Более сложная, но и более удобная в использовании (на взгляд автора) идея «активных образцов» нашла реализацию в работе «F# active patterns» Дона Сайма ([152], 2007 г). Возможность первоклассного манипулирования образцами в языке без встроенной поддержки такой возможности была предложена в 2000 г. Марком Таллсеном в статье «First class patterns» ([156]) и развита до удобного в использовании уровня совсем недавно — в 2008 году (Мортен Райгер, статья «Type-safe pattern combinators» [135]).

Интуиция

На рис. 5.5 графически декларативно записан алгоритм вычисления обращения списка, сведенный к задаче «приписать обращение списка xs к списку r» (изначально полагается r=[]). Алгоритм reverse ' последовательно «откусывает» от списка голову и приписывает ее к результату. В алгоритме разбираются два случая, которые вместе описывают все множество входных списков:

- xs — пустой список. В этом случае результат — r.
- xs — список с головой h и хвостом t. В этом случае задача заменяется на приписывание обращения t к списку, составленному из головы h и хвоста r.

Обратим внимание, что оба случая сопоставляют фактическую форму (структуру) списка xs с шаблоном: «пустой список» и «список с какой-то головой и каким-то хвостом», вместо того, чтобы явно вычислять голову и хвост списка при помощи соответствующих операций доступа.

На Haskell описанная функция будет выглядеть так:

```
reverse xs = reverse' xs []
  where
    reverse' xs r = case xs of
      []    → r
      (h:t) → reverse' t (h:r)
```

Такой декларативный стиль разбора значений при помощи оператора `case..of` и его аналогов называется *сопоставлением с образцом*, и довольно большое количество языков предоставляют его синтаксическую поддержку.

Для сравнения, вот реализация, не использующая в явной форме сопоставление с образцом:

```
reverse xs = reverse' xs []
    where reverse' xs r = if (null xs)
                          then r
                          else reverse' (tail xs) (head xs:r)
```

Описание

Оператор сопоставления с образцом, с точностью до синтаксиса конкретного языка, выглядит так:

```
case EXPRESSION of
  PATTERN1 → VALUE1
  PATTERN2 → VALUE2
  ...
```

Здесь `EXPRESSION` — произвольное выражение, обладающее значением (обычно принадлежащим к алгебраическому типу (см. 5.9)), `VALUE` — выражения или операторы (statement), а `PATTERN` — собственно образцы. Пары `PATTERN → VALUE` называются уравнениями (clause), иногда — «клозами».

Образец — это описание «формы» ожидаемой структуры данных: образец сам по себе похож на литерал структуры данных (т. е. он состоит из конструкторов алгебраических типов (см. 5.9) и литералов примитивных типов: целых, строковых и т. п.), однако может содержать метапеременные — «дырки», обозначающие: «значение, которое встретится в данном месте, назовем данным именем».

На рис. 5.6 изображена спецификация операции правого поворота для двоичных деревьев поиска, применяемая в сбалансированных деревьях, а также показан процесс применения этой операции к конкретному дереву. Форма шаблона сопоставляется с формой дерева, метапеременные заполняются соответствующими значениями, и, наконец, вычисляется правая часть уравнения с учетом значений метапеременных. Вот соответствующий код на Haskell:

```
data Tree α = Leaf α | Fork α (Tree α) (Tree α)

rotateR tree = case tree of
  Fork q (Fork p a b) c → Fork p a (Fork q b c)
```

Неформальная семантика case-выражения такова: «Значением выражения `case E of P1 → V1; ...; Pn → Vn` является значение правой части V_i первого из уравнений $P_i \rightarrow V_i$, такого, что E сопоставимо с P_i».

В некоторых языках case-оператор — не единственная форма сопоставления с образцом. Например, Haskell позволяет непосредственно определять функции в таком стиле:

```
rotateR (Fork q (Fork p a b) c) = Fork p a (Fork q b c)
```

Или (перепишем пример `reverse`):

```
reverse xs = reverse' xs []
  where
    reverse' []    r = r
    reverse' (h:t) r = reverse' t (h:r)
```

Обычно допускается одновременное сопоставление с образцом по нескольким аргументам.

```
zip xs [] = []
zip [] xs = []
zip (x:xs) (y:ys) = (x,y):zip xs ys

> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
```

В этом случае каждый подобный элемент определения функции называется уравнением.

Во многих языках существует специальная метапеременная «λ_», отличающаяся тем, что сопоставленное с ней значение не запоминается, и, как следствие, она не может использоваться в правой части уравнения. Она означает «В данном месте допустимо любое значение, а какое именно — неважно».

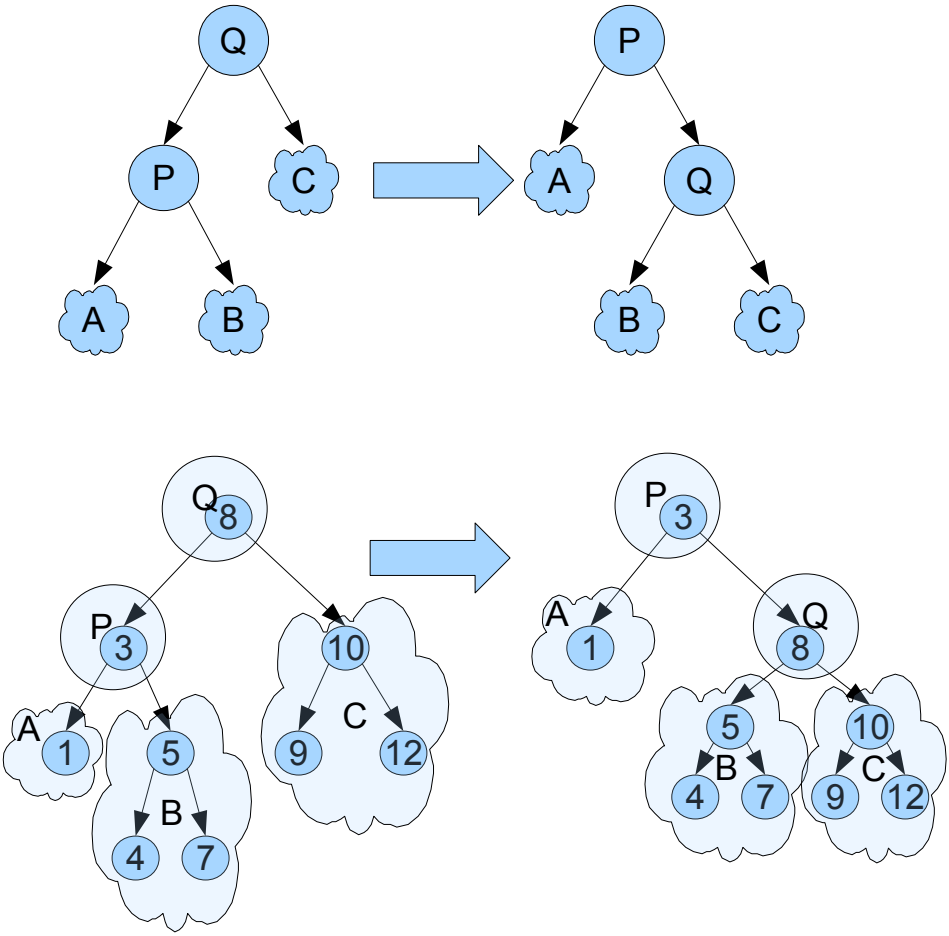


Рис. 5.6. Правый поворот дерева

Некоторые языки, например, Haskell, допускают наложение дополнительных охранных условий в уравнении, помимо совпадения формы структуры данных с формой образца.

```
filter p [] = []
filter p (x:xs) | p x  =  x : filter p xs
                | True =  filter p xs
```

В некоторых языках — обычно речь идет о языках, манипулирующих символьными выражениями, например, Mathematica — шаблоны могут упоминать одну и ту же метапеременную в образце несколько раз (что подразумевает, что соответствующие части сопоставляемого выражения должны быть эквивалентны), а также исполь-

зовать одни метапеременные в качестве частей сложных (например, условных) шаблонов других метапеременных. Например, вот так можно на Mathematica определить функцию, проверяющую, содержится ли элемент в списке:

```
> Elem[x_, {}] := False;
> Elem[x_, {x_, ys___}] := True;
> Elem[x_, {_, ys___}] := Elem[x, {ys}];
> Elem[1, {2,2,1,4}]
True
```

Такие образцы называются нелинейными (а остальные — т. е. упоминающие каждую метапеременную не более 1 раза — соответственно, линейными)

В языках с иной парадигмой нелинейные образцы обычно запрещены, поскольку более трудны для реализации и понимания.

Язык Mathematica, пожалуй, можно считать образцовой реализацией техники сопоставления с образцом. В Mathematica само понятие вычисления основано на переписывании термов в соответствии с определеннами правилами в форме уравнений. Mathematica предоставляет большое количество разновидностей образцов. Рассмотрим некоторые из них.

- «обычные» образцы: заменим все фрукты красного цвета на символ «красный фрукт». «`foo_`» — обычная метапеременная, обозначающая «произвольное выражение в данном месте образца назовем именем `foo`».

  ```
  > {apple[red], apple[green], blueberry, orange[red]}
      /. fruit_[red] :> redFruit
  {redFruit, apple[green], blueberry, redFruit}
  ```

- Нелинейные образцы: заменим выражения вида $\frac{d(a^n)}{da}$ на na^{n-1}.

  ```
  > d[x^3, x] /. d[a_^n_, a_] :> n a^(n-1)
  3 x^2
  ```

- Повторения (последовательность из выражений, подходящих под образец): обернем последовательности яблок символом «яблоки». «`P..`» обозначает «последовательность выражений, подходящих под образец `P`».

  ```
  > {blueberry, {apple[red], apple[green]},
      orange[red]} /. x:{apple[_]..} :> apples[x]
  {blueberry, apples[{apple[red], apple[green]}],
     orange[red]}
  ```

- Образцы-последовательности, сопоставляемые с подсписками: определим функцию принадлежности элемента к списку. «`___`» обозначает «произвольный подотрезок списка».

```
> Elem[x_, {___, x_, ___}] := True;
> Elem[_, _] := False;
> Elem[1, {2,2,1,4}]
True
```

- Образцы с условием: заменим числа меньше 3 на символ `tooSmall`, а числа больше 5 на `tooBig`.

```
> {1,2,3,4,5,6,7}
    /. {x_/;x<3 :> tooSmall, x_/;x>5 :> tooBig}
{tooSmall,tooSmall,3,4,5,tooBig,tooBig}
```

- Отрицание образца: выберем из списка все значения, кроме тех, что больше 4.

```
> Cases[{1,2,5,3,4,2,6,3,1}, Except[x_/;x>4]]
{1,2,3,4,2,3,1}
```

- Строковые образцы: заменим подстроку «`ab`» на «`X`».

```
> StringReplace["abc abcb abdc", "ab" ~~ _ → "X"]
"X Xb Xc"
```

- Определение функций при помощи образцов: определим детерминант матрицы 2×2 как $det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$.

```
> det[{{a_,b_}, {c_,d_}}] := a*d - b*c;
> det[{{1,2}, {3,4}}]
-2
```

Абстрактные типы данных обеспечивают легкость поддержки и развития кода, а также модульность, за счет того, что доступ к данным определен исключительно в терминах поддерживаемых операций и их свойств. Например, абстрактный тип данных «Двоичное дерево» может быть определен в терминах операций «Получить данные узла, получить левое поддерево, получить правое поддерево». При таком интерфейсе возможно, к примеру, поменять представление дерева с «обычного» (`data Tree a = Leaf a |Fork a (Tree a) (Tree a)`) на компактное представление в массиве (где левым ребенком k-го узла является $2k + 1$-й, а правым — $2k + 2$-й), ничего не меняя в клиентском коде.

К сожалению, сопоставление с образцом не позволяет работать с абстрактными типами данных, потому что данный способ доступа к данным напрямую связан с конкретной реализацией структуры, т. е. с тем, какие конструкторы с какими сигнатурами составляют ее тип (см. 5.9). Если клиентский код был написан при помощи сопоставления деревьев с образцами, состоящими из конструкторов `Leaf`,`Fork`, то при переходе к представлению при помощи массива код придется переписать.

У этой проблемы существует несколько похожих решений. Первое из них появилось под названием «views (view patterns)» («представления»; описаны в статье Вадлера «Views: a way for pattern matching to cohabit with data abstraction» [162]); со временем в Haskell стало использоваться под тем же названием несколько другое решение; через несколько лет похожие техники появились в F# («active patterns») и Scala («extractors»). Оказалось, что они не только решают проблему смены представления данных, но и позволяют писать очень красивый и компактный код во многих других случаях.

Сущность этих решений состоит в том, чтобы выполнять сопоставление с образцом не над самой структурой данных, а над некоторым ее *образом*.

Рассмотрим пример: будем сопоставлять строки с форматом дат (приведен псевдокод в предположении, что существует функция `parseDate`).

```
data Date = Date {year::Int, month::Month, day::Int,
                  hours::Int, minutes::Int, seconds::Int}

toUnixMillis :: String → Integer
toUnixMillis s = case (parseDate "YYYY/MM/DD hh:mi:ss" s) of
  Date yyyy mm dd hh mi ss → ss + 60*mi + 3600*hh + ...
```

В этом коде написано: «Если `parseDate ... s` является датой с полями `yyyy,mm,dd,hh,mi,ss`, то ответ такой-то».

Почему бы не сказать «Если `str` является представлением даты с полями `yyyy,mm,dd,hh,mi,ss` относительно формата `YYYY/MM/DD hh:mi:ss`, то ответ такой-то»? Вот как такая переформулировка будет выглядеть на Haskell с использованием расширения языка «ViewPatterns»:

```
toUnixMillis (parseDate "YYYY/MM/DD hh:mi:ss"
                  → Date yyyy mm dd hh mi ss) = ...
```

Аналогично можно, к примеру, разбирать строки при помощи регулярных выражений:

```
> let swap (regex "([0-9]+)-([0-9]+)" → [a,b])
      = show b ++ "-" ++ show a
> swap "123-4567"
"4567-123"
```

В этих двух случаях нет большой разницы между кодом с представлениями и его аналогом с явным `case`, однако она становится очень явной, когда речь идет о сопоставлении одновременно нескольких значений с несколькими образцами. Например, вот гипотетический код для слияния двух приоритетных очередей (предполагается, что `uncons q` возвращает минимальный элемент `q` и остаток `q`). Можно представить себе, во что этот код превратится, если избавиться от «view patterns» и сделать `case` явным.

```
merge (isEmpty → True) q = toList q
merge q (isEmpty → True) = toList q
```

```
merge p@(uncons → (x,xs)) q@(uncons → (y,ys))
  | x < y = x : merge xs q
  | True  = y : merge p ys
```

Очень поучительное и красивое применение алгоритмов с использованием очередей и с использованием представлений именно для абстрагирования от конкретной реализации структуры (сопоставление с образцом-представлением «Очередь с головой h и остатком t») можно найти в статье Криса Окасаки «Breadth-first numbering: lessons from a small exercise in algorithm design» ([111]).

Реализация представлений в Haskell интересна именно своей неожиданной простотой и понятностью. Однако, средства F# синтаксически чуть удобнее. Множество красивых и убедительных примеров представлений в F# приведены в статье («F# active patterns» Дона Сайма [152]).

Использование

В языках, поддерживающих сопоставление с образцом, оно применяется повсеместно; очень многие функции определяются с его помощью. Покажем парочку «повседневных» и «продвинутых» примеров его применения.

Объединение сортированных списков:

```
union xxs@(x:xs) yys@(y:ys) = case compare x y of
  LT → x : mergeUnion xs yys
  EQ → x : mergeUnion xs  ys
  GT → y : mergeUnion xxs ys
union xs [] = xs
union [] ys = ys
```

Посылка команды в протоколе POP3 (библиотека HaskellNet [104]; отрывок):

```
sendCommand (POP3C conn _) (RETR msg) =
  bsPutCrLf conn (BS.pack ("RETR " ++ show msg))
     ≫ responseML conn
sendCommand (POP3C conn _) (TOP msg n) =
  bsPutCrLf conn (BS.pack ("TOP " ++ show msg
                           ++ " " ++ show n))
     ≫ responseML conn
sendCommand (POP3C conn _) (AUTH LOGIN user pass) =
  do bsPutCrLf conn (BS.pack "AUTH LOGIN")
     bsGetLine conn
     bsPutCrLf conn (BS.pack userB64)
     bsGetLine conn
     bsPutCrLf conn (BS.pack passB64)
     response conn
  where (userB64, passB64) = A.login user pass
```

Балансировка красно-черного дерева (читатели, когда-либо видевшие или писавшие реализацию сбалансированных деревьев на языке без сопоставления с образцом,

оценят неприличную краткость и выразительность этого отрывка по сравнению с типичной реализацией):

```
balance :: Color → Tree α β → (α,β) → Tree α β → Tree α β
balance B (T R (T R a x b) y c) z d
  = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
  = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
  = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
  = T R (T B a x b) y (T B c z d)
balance color a x b = T color a x b
```

В языке Haskell сопоставление с образцом — не просто форма определения функций или записи условного выражения `case`, а основная форма связывания значений с именами, употребляемая во всех контекстах, где требуется такое связывание. Например, в генераторах списков и в do-блоках (первый пример учебный, второй взят из библиотеки HSH для Haskell [47], реализующей встроенный язык, схожий с конвейером shell):

```
presentValues map keys = [v | k ← keys,
                              Just v ← [lookup map k]]

fdInvoke (PipeCommand cmd1 cmd2) env ichan =
    do (chan1, res1) ← fdInvoke cmd1 env ichan
       (chan2, res2) ← fdInvoke cmd2 env chan1
       return (chan2, res1 ++ res2)
```

В Mathematica многие функции определяются с помощью сопоставления с образцом. Вообще, большая часть всей обработки данных там происходит через переписывание согласно образцам. Рассмотрим несколько примеров: вычисление детерминанта матрицы 2×2, упрощение выражения согласно правилам $log(xy) = log\,x + log\,y$ и $log(x^k) = k\,log\,x$ (Mathematica не производит такое упрощение самостоятельно, поскольку эти правила верны лишь для положительных аргументов), а также «устранение карринга (см. 5.5)».

```
> det[{{a_,b_}, {c_,d_}}] := a*d - b*c;
> det[{{1,2}, {3,4}}]
-2
> rules = {Log[x_ y_] :> Log[x] + Log[y], Log[x_^k_] :> k Log[x]};
> Log[Sqrt[a (b c^d)^e]] //. rules
1/2 (Log[a] + e (Log[b] + d Log[c]))
> f[a][b][c][d] //. g_[x_][y__] :> g[x, y]
f[a,b,c,d]
```

Реализация

Сопоставление с образцом реализовано во многих языках. В первую очередь это языки семейства ML: Haskell, OCaml, F# и т. п.

В Mathematica вся концепция вычислений основана на сопоставлении с образцом; фактически, движок Mathematica — лишь чрезвычайно эффективная машина для выполнения поиска и замены на основе образцов.

Сопоставление с образцом может быть реализовано и в качестве «макроса» при отсутствии непосредственной поддержки в языке. Так, например, сделано в реализации PLT Scheme (модуль «match.ss» [172]):

```
(require (lib "match.ss"))

(define-struct plus (a b))
(define-struct times (a b))
(define add make-plus)
(define mul make-times)

(define expand
  (match-lambda
   (($ times a ($ plus b c))
    (add (simplify (mul a b)) (simplify (mul a c))))
   ...)
```

Здесь определен абстрактный тип данных «выражение» — две структуры «сумма» и «произведение», и приведен отрывок функции, выполняющей «раскрытие скобок» по правилу $a(b + c) = ab + ac$.

В языке логического программирования Prolog также реализовано сопоставление с образцом, и, в определенном смысле, весь процесс вычисления программы на Prolog и состоит в сопоставлении входного запроса с уравнениями (парами образец/ответ), заданными в программе. Однако, в Prolog сопоставление происходит не до первого совпадения, а целиком по всем уравнениям (либо до достижения оператора отсечения, но это не имеет отношения к делу). Тем самым достигается поиск всех возможных значений свободных переменных терма, удовлетворяющих заданной в программе системе определений. Кроме того, в Prolog нет функций, а есть лишь отношения (поэтому нет понятия аргумента и результата); благодаря этому программы на Prolog как бы работают в обе стороны: позволяют вычислить результат на основе аргументов, но позволяют вычислить и аргументы на основе результата.

```
app([], Ys, Ys).
app([X|Xs], Ys, [X|Zs]) :- app(Xs, Ys, Zs).

?- app([1,2],[3,4], Zs).
Zs = [1,2,3,4].
?- app(Xs,Ys,[1,2,3]).
Xs = [],
```

```
Ys = [1, 2, 3] ;
Xs = [1],
Ys = [2, 3] ;
Xs = [1, 2],
Ys = [3] ;
Xs = [1, 2, 3],
Ys = [] ;
false.
```

В первом случае выражение app ([1,2],[3,4], Zs) было сопоставлено с уравнениями app и обнаружилось, что это выражение подходит только под второй из них: X=1,Xs=[2],Ys =[3,4], Zs=[1|Zs’], при условии, что верно app ([2],[3,4], Zs’). Это выражение также сопоставилось со вторым уравнением, а оставшееся выражение app ([],[3,4], Zs’’) сопоставилось с первым уравнением подстановкой Ys =[3,4], Zs’’=Ys. Таким образом, получилось, что Zs =[1,2,3,4] .

Во втором случае процесс протекал наоборот: выражение app(Xs,Ys ,[1,2,3]) сопоставилось с обоими уравнениями; в результате сопоставления с первым было порождено решение Xs=[], Ys =[1,2,3] , и т. д.

В языке многопоточного и распределенного программирования Erlang сопоставление с образцом обладает следующими интересными особенностями:

- Поддерживается сопоставление префиксов строк:

```
protocol("HTTP/"++Version) -> Version;
protocol(_) -> undefined.
```

- Поддерживается сопоставление на уровне битов, что позволяет крайне компактно и читаемо записывать разбор бинарных протоколов:

```
decode(Segment) ->
    case Segment of
    << SourcePort:16, DestinationPort:16,
       SequenceNumber:32,
       AckNumber:32,
       DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
       Checksum:16, UrgentPointer:16,
       Payload/binary>> when DataOffset > 4
    ->
        OptSize = (DataOffset - 5)*32,
        << Options:OptSize, Message/binary >> = Payload,
        << CWR:1, ECE:1, URG:1, ACK:1,
           PSH:1, RST:1, SYN:1, FIN:1 >> = << Flags:8 >>,
        %% Can now process the Message according to the
        %% Options (if any) and the flags CWR, ..., FIN.
        binary_to_list(Message)
    end.
```

Еще один язык, основанный целиком на сопоставлении с образцом — язык РЕФАЛ (Refal). Отличительная особенность его — в том, что основная структура данных в нем — двусвязный список, что позволяет выполнять сопоставление с более сложными образцами и делает РЕФАЛ удобным для сложных задач обработки текста или деревьев, например, XML. Вот пример программы на РЕФАЛе, проверяющей, является ли строка палиндромом, по правилам: строка вида aSa — палиндром, если S — палиндром; строка из одного символа — палиндром; пустая строка — палиндром; остальные строки — не палиндромы.

```
Palindrome {
   s.1 e.2 s.1 = <Palindrome e.2> ;
   s.1 = True ;
   = True;
   e.1 = False ;
}
```

Имитация

Сопоставление с образцом всегда можно заменить на ручной разбор случаев с использованием операций доступа к данным; при этом придется проделать работу компилятора по поиску оптимального дерева проверок и код, скорее всего, увеличится в несколько раз. Кроме того, компиляция сопоставления с образцом далеко не тривиальна, и реализованные вручную проверки, скорее всего, будут неоптимальны. Например, при компиляции сопоставления с образцом для типа с 20 конструкторами компилятор, скорее всего, сгенерирует вовсе не последовательность из двадцати if .. else , а бинарное дерево проверок логарифмической высоты.

В объектно-ориентированном языке сопоставление с образцом по одному аргументу алгебраического типа может быть частично реализовано при помощи введения виртуальных функций в классе, имитирующем этот тип. Для сопоставления с образцом по двум аргументам (двойная диспетчеризация) используется, например, паттерн Visitor. См. также секцию «Имитация» в статье об алгебраических типах (см. 5.9).

В языках с макросами зачастую можно реализовать сопоставление с образцом без «родной» поддержки языка, как сделано, к примеру, в языке PLT Scheme (см. выше).

Сравнительно недавно появились техники («First class patterns» Марка Таллсена [156] и «Type-safe pattern combinators» Мортена Райгера [135]), позволяющие манипулировать образцами как первоклассными объектами, и, как следствие, позволяющие в том числе выражать образцы в виде обычных функций высшего порядка (см. 5.3) и писать определения в стиле сопоставления с образцом без помощи синтаксической поддержки языка.

Также недавно появилась удобная и мощная библиотека для работы с первоклассными образцами на Haskell [128] (см. также серию блог-постов Райнера Поупа об этой библиотеке, в блоге по тегу «pattern combinators» [129]). Приведем краткий пример кода с ее использованием:

```
test1 :: Either Int (Int, Int) → Int
test1 a = match a (
     left (cst 4)          →> 0
 ||| left var              →> id
 ||| right (pair var var) →> (+))
```

Этот код эквивалентен следующему:

```
test1 :: Either Int (Int, Int) → Int
test1 a = case a of
    Left 4       → 0
    Left x       → x
    Right (x,y) → x+y
```

Основные преимущества первоклассных образцов — возможность отслеживать несовпадение значения с образцом во время выполнения (в этой ситуации при обычном сопоставлении произошла бы ошибка выполнения), возможность «дополнять» существующий образец новыми уравнениями и возможность определять принципиально новые разновидности образцов. При помощи первоклассных образцов можно, к примеру, реализовать «selective receive» (выборку сообщений по образцу из очереди процесса) в стиле Erlang. Это одна из мощных и уникальных возможностей Erlang.

5.11. Свёртка (Fold)

Суть

Вычисление снизу вверх «по индукции», применяющее в каждом узле структуры данных оператор, соответствующий данному типу узла, к содержимому узла и результатам для его подузлов.

История

Как утверждается в статье Грэма Хаттона «A tutorial on the universality and expressiveness of fold» ([65]), понятие свертки появилось в теории рекурсии в 1952 г. Свертки были впервые использованы в языке программирования APL (1962) для обработки списков. В 1978 г. свертки были упомянуты в работе Джона Хьюза «Why Functional Programming Matters» [61].

К произвольным структурам данных свертку впервые применил Г. Малкольм в 1990 г. в статье «Algebraic data types and program transformation» ([92]), обобщив идею свертки над списками. Эта идея получила развитие в очень известной статье Эрика Мейера, Маартена Фоккинги и Росса Патерсона «Functional programming with bananas, lenses, envelopes and barbed wire» ([98]) и многих других работах, где используется т. н. «универсальное свойство» свертки (описанное в вышеупомянутой статье Грэма Хаттона [65]). В 1998 г. в статье Ричарда Бёрда и Ламберта Меертенса «Nested datatypes» ([11]) было описано обобщение свертки на «вложенные» (полиморфно рекурсивные) алгебраические типы, а в статье Ральфа Хинзе «Efficient generalized folds» ([53]) и статье Джереми Гиббонса «Disciplined, efficient, generalized folds for nested datatypes» ([24]) это обобщение было улучшено.

Однако, шестью годами раньше, в 1992 г. появился язык Coq, исчисление индуктивных конструкций. В нём было разработано обобщение понятия свертки в качестве принципа индукции на значительно более сложный класс индуктивных типов (см. книгу «Coq'art: Interactive theorem proving and program development» [10] и статью «Inductive definitions in the system Coq: Rules and properties» [114])! Удивительно, что этот результат оставался незамеченным целых 6 лет: на первый взгляд некоторые определения сверток, предложенные, скажем, в «Nested datatypes» Бёрда и Меертенса [11], практически совпадают с теми, что автоматически генерирует Coq.

Интуиция

Рассмотрим несколько возможных операций над документами (возьмем для примера HTML):

- Подсчет количества слов или множества различных слов

- Вырезание картинок или javascript-сценариев
- Поиск абзаца, содержащего больше всего ключевых слов из заданного списка

Пусть документ представлен в форме синтаксического дерева, где каждый узел описывается типом узла (тег/текст/комментарий), содержимым, именем узла, атрибутами и списком подузлов. Все эти операции реализованы по одной и той же схеме:

- Подсчет слов: Результат алгоритма — целое число; вычисление происходит рекурсивно снизу вверх. Для текстового узла — обычный алгоритм; для комментария — 0; для иного узла — сумма результатов в его подузлах.
- Посчет множества различных слов: Результат алгоритма — множество строк; вычисление происходит рекурсивно снизу вверх. Для текстового узла — обычный алгоритм; для комментария — пустое множество; для иного узла — объединение результатов для подузлов.
- Вырезание картинок или javascript-сценариев: Результат алгоритма — узел (документ) или специальное значение «пустой узел» (на случай, если весь документ, скажем, состоял из одной-единственной картинки). Для текстового узла или комментария — сам узел; для тега IMG или SCRIPT — пустой узел; для иного узла — узел с теми же тегом и атрибутами, чей список подузлов составлен из непустых результатов применения алгоритма к подузлам исходного узла.
- Рендеринг в формат PostScript (например, последовательность команд для принтера): результат алгоритма — последовательность команд; вычисление происходит рекурсивно снизу вверх. Для каждого из типов узлов результаты подузлов конкатенируются, возможно, перемежаясь дополнительными командами.

Во всех четырех случаях вычисление происходило снизу вверх, и к каждой разновидности узлов применялась своя процедура, оперирующая над самим узлом и результатами алгоритма на его подузлах.

Описание

Пусть `T` — рекурсивно определенный алгебраический тип (см. 5.9), обладающий конструкторами $K_1 \ldots K_n$.

Определим алгебраический тип `F` τ с конструкторами $F_1 \ldots F_n$, где сигнатура F_i получается из сигнатуры K_j заменой аргументов типа `T` на аргумент типа τ. Будем называть `F` *схемой рекурсии* для `T`.

Рассмотрим пример — дерево следующего вида:

```
data IntTree = Leaf | Fork Int IntTree IntTree
```

Для такого типа $n = 2$, $K_1 = Leaf$, $K_2 = Fork$.

Схема рекурсии же для него такова:

```
data IntTree' τ = Leaf' | Fork' Int τ τ
```

Особенность такого типа в том, что он, будучи примененным к τ=`IntTree`, дает тип, эквивалентный `IntTree`; а при вычислении какого-либо значения снизу вверх этот тип соответствует типу «контекста» такого вычисления (известны данные текущего узла и результаты рекурсивных вызовов). Значение такого типа и является аргументом для процедуры свертки.

Итак, свёртка (вычисление снизу вверх) над типом `T` определяется функцией из соответствующего ему типа `F` τ в значение типа τ. Чтобы задать вычисление снизу вверх над типом деревьев, нужно задать функцию из `IntTree'` τ в τ. Например, функция для подсчета количества листьев в дереве будет выглядеть так:

```
leafCountFold :: IntTree' Int → Int
leafCountFold Leaf' _       = 1
leafCountFold (Fork' _ p q) = p + q
```

Еще раз обратим внимание, что тип `IntTree'` сам по себе не рекурсивен, а аргументами `Fork'` вместо значений типа `IntTree'` τ являются просто значения типа τ, т. е. результаты вычисления снизу вверх в непосредственных подструктурах.

Опишем функцию, выполняющую свертку (вычисление снизу вверх) согласно заданной процедуре:

```
foldTree :: (IntTree' τ → τ) → IntTree → τ
foldTree f Leaf           = f Leaf'
foldTree f (Fork a t1 t2) = f (Fork' a (foldTree f t1)
                                       (foldTree f t2))
```

Тогда функция, выполняющая подсчет количества листьев в дереве, будет задаваться как `countLeaves = foldTree leafCountFold`.

Показательно рассмотреть процедуру копирования дерева, чей тип получается, если подставить `IntTree` в качестве τ: в этом случае сверточная процедура имеет тип `Tree' IntTree → IntTree`. С практической точки зрения процедура бесполезна, однако она иллюстрирует связь между исходным типом, его схемой рекурсии и операцией свертки.

```
copyTree = foldTree copyFold
    where copyFold  Leaf'          = Leaf
          copyFold (Fork' a t1 t2) = Fork a t1 t2
```

Полностью аналогично определяется свёртка для параметрических алгебраических типов, например:

```
data Tree α = Leaf | Fork α (Tree α) (Tree α)

data Tree' α τ = Leaf' | Fork' α τ τ

foldTree :: (Tree' α τ → τ) → Tree α → τ
foldTree f Leaf           = f Leaf'
foldTree f (Fork a t1 t2) = f (Fork' a (foldTree f t1)
                                       (foldTree f t2))
```

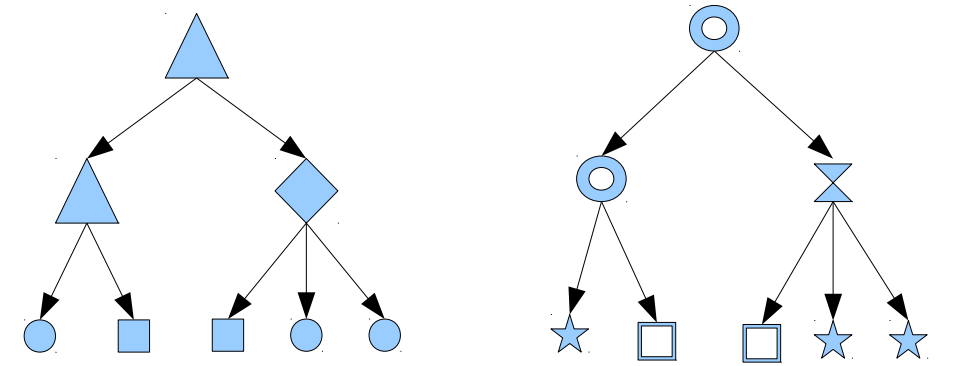


Рис. 5.7. Свёртка как подмена конструкторов

Свертку можно также интерпретировать как «подмену» каждого конструктора в структуре данных на какую-то другую функцию или значение соответствующей арности (арность — число аргументов). Поэтому, например, при подмене конструктора на него самого получается процедура копирования структуры: см. рис. 5.7.

Теперь рассмотрим, во что вырождается понятие свертки в применении к спискам, задаваемым как `data List` α `= Nil |Cons` α `(List` α`)`: схема рекурсии этого типа — тип `data List'` α τ `= Nil' |Cons'` α τ, поэтому тип сверточной процедуры — `List'` α τ $\rightarrow$ τ; можно вместо процедуры такого типа задать две «процедуры», обрабатывающие каждый из двух возможных конструкторов, по отдельности: значение типа τ для конструктора `Nil'` и $\alpha \rightarrow \tau \rightarrow \tau$ для конструктора `Cons'`. Получающаяся процедура называется правой сверткой:

```
foldr :: (α → τ → τ) → τ → List α → τ
foldr withCons whenNil Nil          = whenNil
foldr withCons whenNil (Cons a xs)
  = withCons a (foldr withCons whenNil xs)
```

Похожая процедура, однако образованная по другой схеме, называется левой сверткой, и не имеет очевидных аналогов для других структур, кроме списков. Она как бы представляет список не как сумму головы и хвоста, а как сумму начальной части и последнего элемента. Это позволяет использовать нерекурсивный алгоритм вычисления (см. хвостовой вызов (см. 5.7)):

```
foldl :: (τ → α → τ) → τ → List α → τ
foldl update init Nil          = init
foldl update init (Cons x xs) = foldl update (update init x) xs
```

Формулы для левой и правой свертки выглядят так:

```
foldr (#) u [a1,a2,...,an] = a1 # (a2 # ... # (an # u))
foldl (#) u [a1,a2,...,an] = ((u # a1) # a2) # ... # an
```

В ленивом языке левая и правая свёртка принципиально различаются. Для правой свертки, как видно, результат, независимо от элементов $a_2 \ldots$, имеет вид $a_1 \# \ldots$, и слабая заголовочная нормальная форма (СЗНФ, WHNF) этого выражения зачастую может быть вычислена и без вычисления свертки по всему остатку списка; это позволяет в т. ч. использовать правую свертку на бесконечных списках.

Различие для ленивого языка можно увидеть на примере реализации процедуры `filter`:

```
filter1 p xs = foldr (λx ys →
  if p x then x:ys else ys) [] xs
filter2 p xs = foldl (λys x →
  if p x then x:ys else ys) [] (reverse xs)

> take 5 (filter1 even [1..]) -- Turns out 2:(4:(6:(...)))
[2,4,6,8,10]
> take 5 (filter2 even [1..]) -- Infinite loop, needs
                              -- (reverse [1..]) first
^C
```

Проиллюстрируем ленивость `foldr` вычислением чуть более простого примера:

```
head (filter1 even [1..])
= head (foldr (λx ys →
    if even x then x:ys else ys) [] [1..])
= head (if even 1 then 1:ys else ys
    where ys = foldr (λx ys →
      if even x then x:ys else ys) [] [2..])
= head (foldr (λx ys →
    if even x then x:ys else ys) [] [2..])
= head (if even 2 then 2:ys else ys
    where ys = foldr (λx ys →
      if even x then x:ys else ys) [] [3..])
= head (2:foldr (λx ys →
    if even x then x:ys else ys) [] [3..])
= 2
```

Каждое из равенств в цепочке соответствует всего лишь выполнению редукции — т. е. подстановке тела той или иной функции и фактических параметров ее вызова; не происходит ничего «волшебного» и никаких нетривиальных шагов упрощения: именно в таком порядке программа на Haskell и будет вычисляться.

У левой свертки есть более экзотический, но значительно более полезный аналог — строгая левая свёртка, отличающаяся лишь строгостью (энергичностью) (см. статью «Laziness» в Haskell wikibooks [87]) по «аккумулятору»:

```
foldl' :: (τ → α → τ) → τ → List α → τ
```

```
foldl' update !init Nil         = init
foldl' update !init (Cons x xs) = foldl (update init x) xs
```

В качестве примера алгоритма, требующего строгой левой свертки, можно привести вычисление суммы списка:

```
> foldr (+) 0 [0..1000000]
Exception: stack overflow
> take 2 (foldl (λys x → x:ys) [] [0..1000000])
[1000000,999999]
> foldl (+) 0 [0..1000000]
Exception: stack overflow
> foldl' (+) 0 [0..1000000]
500000500000
```

В первом случае глубокая рекурсия внутри `foldr` приводит к переполнению стека.

Во втором случае результат `foldl` вычисляется итеративно (в смысле, указанном в статье о хвостовых вызовах (см. 5.7)) и успешно (в чем можно убедиться на третьем примере), однако результатом оказывается невычисленный терм `(((0+1)+2)+...)+1000000`, и переполнение стека происходит уже при попытке форсировать *его* вычисление для распечатки.

В третьем случае вычисление проходит успешно, поскольку из-за строгости (энергичности) `foldl'` по аккумулятору промежуточный результат всякий раз оказывается полностью вычисленным числом.

Итак:

- Правая списочная свёртка используется, когда известно, как составить результат для всего списка из его головы и результата для хвоста, причем часть результата «известна» и без полного результата для хвоста (т. е. вычисляется лишь на основе головы — например, в случае `map` или `filter`).
- Строгая левая свёртка используется, когда алгоритм выражается в виде императивного прямого цикла по списку с аккумулятором.
- Обычная левая свёртка используется, когда алгоритм выражался бы в виде правой свертки, будь список перевернут. Эта разновидность свертки нужна исключительно редко.

Подробности, касающиеся различий левых и правых списочных сверток в отношении ленивости и строгости, описаны в статье «Foldr, Foldl, Foldl'» в Haskellwiki ([41]).

В случае, когда тип результата свертки совпадает с типом элементов сворачиваемого списка, можно говорить о коммутативности (`f x y == f y x`) и ассоциативности (`f (f x y) z == f x (f y z)`) сверточной операции. Если операция ассоциативна, то такая операция называется списочным гомоморфизмом, левая и правая свертки по ней совпадают, и свертку можно вычислить с помощью «дерева» (на

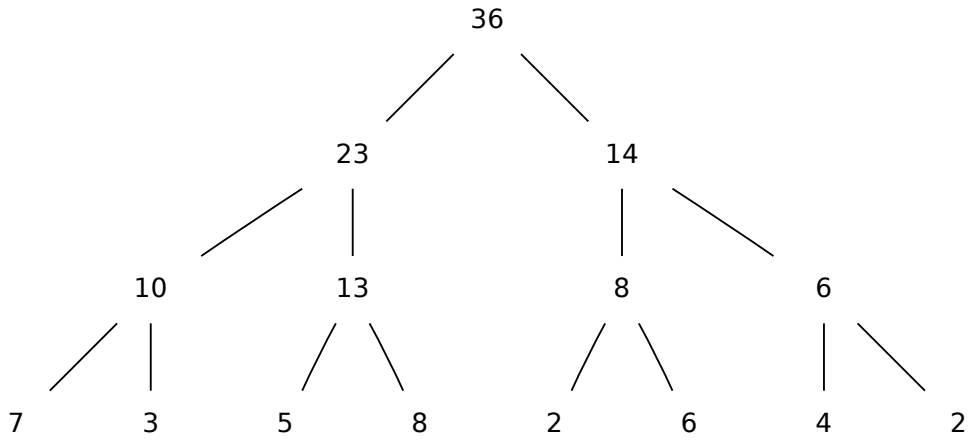


Рис. 5.8. Дерево вычисления ассоциативной свертки по сложению

рис. 5.8 изображено дерево, образующееся в процессе вычисления суммы списка), что позволяет распараллелить вычисление (вычисляя дерево «по слоям» снизу вверх и распараллеливая вычисление каждого слоя) или сделать его инкрементальным (при изменении элемента списка в дереве затрагивается лишь $O(log\ n)$ узлов, лежащих на пути от корня к нему; возможны также вставка/удаление/конкатенация за логарифмическое время). Многие параллельные алгоритмы основаны на таком подходе; узнать о нем больше можно в следующих источниках:

- В книге «Vector models for data-parallel computing» Гая Блеллоха ([12]): впервые предложен и описан подход к разработке параллельных алгоритмов на основе сверток и пробегов (префиксных сумм), и приведено множество очень красивых и несложных алгоритмов.
- Статья того же автора «Prefix sums and their applications» ([13]) обрисовывает основные положения этого подхода и некоторые их применения.
- В блог-посте Хайнриха Апфельмуза «Monoids and finger trees» описана структура данных «дерево с указателем» (finger tree), позволяющая компактно и просто сформулировать, параллелизовать и инкрементализовать множество алгоритмов, и даны ссылки на дополнительную литературу.
- В блог-посте Дэна Пипони «Fast incremental regular expression matching»[13] описано применение деревьев с указателем для реализации инкрементального (и параллельного) поиска по регулярным выражениям.
- В блог-посте Дэна Пипони «An approach to algorithm parallelisation» префиксные суммы применены для параллелизации хорошо известного и на первый взгляд

[13] По субъективному мнению автора, этот пост можно смело причислить к шедеврам программирования.

вовсе не параллелизуемого решения задачи о подотрезке с максимальной суммой.

Если операция к тому же коммутативна, то результат свертки не зависит от порядка элементов в списке и можно еще увеличить степень параллелизма: фактически, достаточно в произвольном порядке объединять элементы и промежуточные результаты объединения по сверточной операции, пока не останется всего одно значение, которое и будет окончательным результатом. Все агрегатные функции SQL (`MIN`,`MAX`,`SUM`, `DISTINCT`,...) являются коммутативными свёртками.

Существует примечательная и очень полезная теорема — «третья теорема о гомоморфизмах» (описана в статье «The third homomorphism theorem» Джереми Гиббонса [44]). Она гласит, что если алгоритм можно реализовать как в виде левой свертки, так и в виде правой свертки с тем же начальным значением, то его можно реализовать и в виде ассоциативной свертки (которая поддается распараллеливанию и инкрементализации).

Использование

Чаще всего в функциональном программировании применяются списочные свертки и их аналоги. Приведем пару примеров:

В модуле `Data.ByteString.Lazy` байтовые потоки представляются в виде ленивого списка из байтовых массивов. Там же определена функция `foldlChunks` (код этого и следующего примера слегка модифицирован для лучшей понимаемости):

```
data LazyByteString = Empty | Chunk ByteString LazyByteString

-- | Consume the chunks of a lazy ByteString with a
-- strict, tail-recursive, accumulating left fold.
foldlChunks :: (α → ByteString → α) → α
               → LazyByteString → α
foldlChunks f z = go z
  where go a _ | a `seq` False = undefined
        go a Empty        = a
        go a (Chunk c cs) = go (f a c) cs
```

Эта функция применяется, например, в библиотеке `digest` [78] (привязка к процедурам вычисления CRC32 и Adler32 из zlib):

```
crc32_l_update :: Word32 → LazyByteString → Word32
crc32_l_update n = foldlChunks updateCRC n
  where
    updateCRC crc bs = crc32_c crc (ptr `plusPtr` offset) len
      where (ptr, offset, len) = toForeignPtr bs
```

То есть, при вычислении CRC32 от ленивого байтового потока функция `crc32_c` (обновление CRC32 заданным байтовым массивов) вызывается процедурой `foldlChunks` для каждого байтового массива, входящего в поток.

В сервере `HAppS-Server` [90] определена процедура для кодирования URL-адресов:

```
urlEncodeVars :: [(String,String)] → String
urlEncodeVars ((n,v):t) =
    let (same,diff) = partition ((==n) · fst) t
    in urlEncode n ++ '=' : folding ++ urlEncodeRest diff
    where
      folding = foldl (λx y → x ++ ',' : urlEncode y)
                      (urlEncode v) (map snd same)
```

Эта процедура группирует переданные параметры при помощи `partition` и для каждой последовательности параметров с одинаковым ключом при помощи `foldl` порождает отрезок строки вида `foo=bar,baz,qux`.

Свертки абстрагируют идею вычисления снизу вверх и позволяют компактно записать многие алгоритмы над различными структурами данных, в особенности над списками: почти все стандартные функции обработки списков можно выразить с помощью свертки. Выражая операцию сверткой, программист абстрагируется от деталей обхода структуры данных и задает лишь самые существенные свойства операции: то, как она действует на различные типы узлов структуры, и как комбинирует результаты для подструктур. Почти всегда гораздо легче корректно реализовать операцию при помощи свертки, нежели без нее, особенно если речь идет о сложной древовидной структуре данных (такой как, например, HTML-документы). К примеру, в практике автора при переписывании с использованием сверток программы, отделявшей навигационные элементы HTML-страницы от «важного» содержимого, сразу же пропали все присутствовавшие ошибки в коде обхода, и исчез дублирующийся код.

Одинаковый принцип действия сверток над любой структурой данных позволяет им обладать особыми алгебраическими свойствами, полезными для оптимизации программ: см. статью «Functional programming with bananas, lenses, envelopes and barbed wire» [98].

В языке Haskell концепция структур, поддающихся списочной свертке (даже если сама структура — не список, а, например, упорядоченное множество, реализованное с помощью дерева), выражена в классе типов (см. 5.13) `Foldable`, а более общая концепция, связанная со схемами рекурсии — в библиотеке «fixpoint» [89], реализующей (буквально в несколько строк) идеи из статьи «Functional programming with bananas …» [98], и аналогичной, но более современной и значительно более сложной библиотеке «multirec» [105].

В качестве примера применения сверток в нефункциональном языке можно привести API компилятора javac для работы с синтакическим деревом кода [173]. Класс `TreeScanner` представляет собой сверточную операцию почти в чистом виде: в нем есть функции для отображения листовых узлов и функция для комбинирования промежуточных результатов.

Одно из важнейших применений ассоциативных и коммутативных сверток — параллельное программирование. В статье «Prefix sums and their applications» [13] и кни-

ге «Vector models for data-parallel computing» [12] обсуждаются применения сверток и родственной им концепции «пробега» (англ. scan) к параллельному программированию. В настоящее время описанные в этих источниках алгоритмы применяются посвеместно, к примеру, при программировании графических процессоров.

Реализация

Почти все функциональные и динамические языки (Haskell, OCaml, Scala, Ruby, Python, Scheme, Perl и прочие) содержат в стандартной библиотеке аналог операции свертки над списками. Он есть даже в PHP, однако по неизвестным автору причинам поддерживаются только свертки над целыми числами (изучение исходного кода не показало каких-либо препятствий для снятия этого ограничения).

Очень интересно обобщение сверток на структуры данных с зависимыми типами, используемое в системе автоматизированного доказательства теорем Coq (сайт [30]). В Coq при определении индуктивного (алгебраического) типа автоматически генерируется соответствующий ему *принцип индукции*. Например, для типа списков, сортированных согласно заданному отношению порядка, получается приблизительно такой принцип индукции по отсортированным спискам:

«Если свойство P выполняется для пустого списка, для всякого одноэлементного списка, и из $x \le y$ и сортированности списка $y :: rest$, удовлетворяющего P, можно вывести P для $x :: y :: rest$, то свойство P выполняется для любого сортированного списка».

Ознакомиться с Coq и с идеологией программирования с зависимыми типами можно, например, в книге «Coq'Art: Interactive theorem proving and program development» ([10]) и в презентации ([178]).

5.12. Структурная / вполне обоснованная рекурсия (индукция) (Structural / well-founded recursion (induction))

Суть

Рекурсия, при которой аргумент рекурсивного вызова в каком-то смысле строго меньше аргумента исходного.

Интуиция

Рассмотрим функцию поиска элемента в бинарном дереве поиска (binary search tree, BST)

```
find x Leaf = False
find x (Fork a l r)
  | x == a  = True
  | x <  a  = find x l
  | x >  a  = find x r
```

Почему мы можем быть уверены, что на всяком конечном дереве вызов этой функции завершается? Потому, что рекурсивный вызов `find` производится от левой или правой ветви, но нет такого конечного дерева, многократное взятие левой или правой ветви от которого никогда не приводит к листу. Поэтому рано или поздно дерево «закончится», а вместе с ним закончится и дерево рекурсивных вызовов.

Формально объяснить этот факт можно, например, следующими двумя способами:

- Всякое конечное дерево имеет конечную неотрицательную высоту. Ветка непустого дерева *имеет строго меньшую высоту*, чем само дерево. Поскольку не существует бесконечной строго убывающей последовательности неотрицательных чисел, то не существует и бесконечной последовательности деревьев, каждое из которых является веткой предыдущего. Поэтому нельзя и бесконечно переходить к ветке дерева, так и не придя к листу.

- Ветка непустого дерева *является поддеревом* самого дерева. Поскольку не существует бесконечной последовательности конечных деревьев, каждое из которых является поддеревом предыдущего, то нельзя и бесконечно переходить к ветке дерева, так и не придя к листу.

Рассмотрим интерпретатор какого-нибудь простого языка программирования, обладающего оператором вызова функции. Процедура интерпретации, встретив такой оператор в синтаксическом дереве, находит тело соответствующей функции в таблице символов программы, выполняет подстановку фактических параметров в

формальные, и производит рекурсивный вызов от синтаксического дерева тела этой функции. Про тело функции нельзя сказать, что оно в каком-нибудь смысле «меньше» тела вызывавшей его функции или является его частью, поэтому доказать с помощью вышеописанного аргумента завершаемость интерпретатора нельзя. И действительно - обычно такой интерпретатор вполне может не завершаться на некоторых программах (что неизбежно, если интерпретируемый язык Тьюринг-полон[14]).

Описание

Будем говорить, что отношение $\prec$ на множестве M является вполне обоснованным (понятие описано также в статье в Wikipedia «Well-founded relation» [168]), если не существует бесконечной последовательности $a_1 \prec a_2 \prec \ldots$. Можно переформулировать это свойство еще двумя способами: «в графе отношения $\prec$ нет циклов», или «в любом подмножестве M есть хотя бы один минимальный элемент относительно $\prec$».

Для конечных рекурсивно определенных структур данных (таких как списки, деревья и т. п.) отношение «$a \prec b$, если a — подструктура b», очевидно, является вполне обоснованным.

Процедура `f a b .. x ..` называется структурно рекурсивной (см. также статью в Wikipedia «Structural induction» [148]) по аргументу `x`, где `x` имеет алгебраический тип (см. 5.9) `T`, если она определена для всех нерекурсивных конструкторов `T` (т. е. для минимальных элементов этого типа по отношению непосредственного включения) и ее определение для таких конструкторов не содержит рекурсивных вызовов, и при этом в уравнении для каждого из рекурсивных конструкторов `T` всякий рекурсивный вызов производятся от непосредственных аргументов этого конструктора. Если убрать из этого определения требование «процедура определена для всех конструкторов» и оставить лишь ограничение на аргументы рекурсивных вызовов, то полученное свойство будет гарантировать лишь завершаемость, но необязательно тотальность (т. е. наличие результата на всех возможных аргументах)[15].

Например, вышерассмотренная процедура поиска значения в дереве структурно рекурсивна по дереву, чей тип обладает одним нерекурсивным конструктором `Leaf` и одним рекурсивным конструктором `Fork`. Вот этот тип:

```
data Tree α = Leaf | Fork α (Tree α) (Tree α)
```

Процедура была определена двумя уравнениями, первое из которых соответствует случаю `Leaf` и не содержит рекурсивных вызовов, а второе соответствует случаю, когда аргумент имеет форму `Fork a l r`, и рекурсивные вызовы производятся лишь от аргументов этого конструктора — `l` или `r`.

Из вышесказанного следует, что структурно рекурсивная процедура обязательно завершается на всех конечных аргументах. Условие конечности очень важно. Haskell,

[14] Заметим, что это не исключает возможности написания всегда завершающегося *компилятора* одного Тьюринг-полного языка в другой.

[15] Конечно, речь здесь и далее идет о тотальности процедуры при условии тотальности и завершаемости используемых ею других процедур.

благодаря ленивым вычислениям, позволяет манипулировать бесконечными структурами данных, и свойство завершаемости, разумеется, выводимо из структурной рекурсивности лишь в случае конечных структур. Кроме того, в ленивых языках используется более тонкое понятие завершаемости, поэтому в рамках данной статьи мы будем предполагать, что речь идет о строгом (энергичном) языке с синтаксисом Haskell.

Завершимость, конечно, присутствует и в случае, когда вместо отношения «являться подструктурой» используется другое вполне обоснованное отношение между аргументами исходного и рекурсивного вызова. В этом случае говорят, что процедура определена при помощи вполне обоснованной индукции (см. также презентацию Ива Берто о вполне обоснованной индукции в Coq «Well-founded induction» [9]). В статье Лоуренса Полсона «Constructing recursion operators in intuitionistic type theory» [115] описан примененный в системе Coq способ конструктивной реализации вполне обоснованной индукции в системе типов, допускающей только структурную рекурсию.

Заметим, что натуральные числа также образуют алгебраический тип с двумя конструкторами: «ноль» и «n+1»: `data Nat = Zero |Next Nat`, поэтому можно считать структурно рекурсивными арифметические процедуры, определенные для n через результат для $n - 1$.

Структурная рекурсия абстрагируется оператором свертки (см. 5.11).

Использование

Завершимость — самое важное свойство структурно рекурсивных процедур. Некоторые языки программирования, гарантирующие завершаемость программ, обеспечивают ее именно через запрещение всех форм рекурсии, кроме структурной. Кроме того, о структурно-рекурсивных процедурах легко рассуждать математически и доказывать их свойства, поскольку к ним непосредственно применимо доказательство свойства при помощи индукции по аргументу. В функциональных языках, где особенно часто используется рекурсия, считается хорошим тоном делать процедуры по возможности структурно рекурсивными.[16] Конечно же, это возможно только если процедура действительно завершается на всех конечных аргументах.

Иногда бывает полезно даже ввести в процедуру дополнительный аргумент, и сделать ее структурно рекурсивной по нему, чтобы убедиться, что она всегда завершается. Например, такой аргумент может быть числом, отражающим оставшееся количество операций (если при начальном вызове вообще можно заранее подсчитать, сколько операций потребуется). В случае работы с языком программирования, допускающим только структурную рекурсию, этот прием очень полезен для «убеждения» компилятора в завершаемости вашей процедуры (однако доказать корректность такой процедуры по отношению к спецификации становится сложнее).

[16] Еще более хорошим тоном считается по возможности избегать явного использования рекурсии и использовать ее лишь в комбинаторах (см. 5.14) общего назначения, таких как `map`, `fold` и т. п. Речь идет о структурной рекурсивности этих комбинаторов.

Рассмотрим пару примеров функций, определенных при помощи структурной рекурсии, или, напротив, не являющихся структурно рекурсивными, а затем — функций, определенных при помощи вполне обоснованной индукции или не являющихся таковыми.

Функция вычисления длины списка структурно рекурсивна по списку:

```
length [] = 0
length (x:xs) = 1 + length xs
```

Рекурсивный вызов производится от терма `xs`, являющегося непосредственным аргументом конструктора `(:)` в терме `(x:xs)`.

Функция вычисления факториала структурно рекурсивна по своему аргументу-натуральному числу.

```
factorial 0 = 1
factorial n = n × factorial (n-1)
```

Функция вставки в бинарное дерево (для простоты рассмотрим несбалансированное дерево) структурно рекурсивна по аргументу-дереву:

```
insert x Leaf = Fork x Leaf Leaf
insert x (Fork a l r)
  | x == a = Fork a l r
  | x <  a = Fork a (insert x l) r
  | x >  a = Fork a l (insert x r)
```

Рекурсивные вызовы производятся от термов `l` или `r`, являющихся непосредственными аргументами конструктора `Fork` в терме `(Fork a l r)`.

Идиоматическая реализация быстрой сортировки списка[17] *не является* структурно рекурсивной:

```
qsort [] = []
qsort (x:xs) = qsort (filter (≤x) xs) ++ [x]
               ++ qsort (filter (>x) xs)
```

Как видно, здесь аргументы рекурсивных вызовов — термы вида `filter (>x) xs`, не являющиеся непосредственными аргументами конструктора `(:)` в терме `(x:xs)`.

Впрочем, эта реализация является, например, *вполне обоснованно рекурсивной*, поскольку рекурсивные вызовы производятся от списков, чья длина строго меньше длины исходного списка (это рассуждение основано на таком свойстве `filter`, как `length (filter p xs) ≤length xs`).

Процедура объединения двух сортированных списков является вполне обоснованно рекурсивной:

[17] Она идиоматична исключительно с точки зрения своей распространенности. Разумеется, ее практическая полезность близка к нулю из-за низкой эффективности, и сортировку подобным образом на функциональных языках не реализуют.

```
merge [] xs = xs
merge xs [] = xs
merge (x:xs) (y:ys)
  | x ≤ y = x : merge xs (y:ys) -- (A)
  | x > y = y : merge (x:xs) ys -- (B)
```

В данном случае имеет место вполне обоснованная индукция: аргументы рекурсивных вызовов `merge` (обозначим их `xs', ys'`) связаны с аргументом исходного (обозначим их `xs, ys`) следующим вполне обоснованным отношением:

```
length xs' + length ys' < length xs + length ys
```

То есть, при рекурсивных вызовах строго убывает сумма длин сливаемых списков.

Однако эта процедура не является структурно рекурсивной ни по одному из двух своих аргументов: в самом деле, в вызове `(A)` нет структурной рекурсии по второму аргументу, а в вызове `(B)` — по первому. Впрочем, ее можно сделать структурно рекурсивной по *фиктивному* аргументу, обозначающему сумму длин списков.

```
merge xs ys = merge' xs ys (length xs + length ys)

merge' _      _      0        = [] -- (X)

merge' []     (x:xs) n = x : merge' [] xs (n-1)
merge' (x:xs) []     n = x : merge' xs [] (n-1)
merge' (x:xs) (y:ys) n
  | x ≤ y = x : merge xs (y:ys) (n-1)
  | x > y = y : merge (x:xs) ys (n-1)
```

Обратим внимание, как было изменено определение `merge`, чтобы удовлетворить свойству структурной рекурсивности: были в явной форме выделены уравнения для каждого из конструкторов, составляющих алгебраический тип `[Int]` третьего аргумента.

Теперь явно имеет место структурная рекурсия по третьему аргументу (заметим, что без уравнения `(X)` функция не была бы структурно рекурсивной, т. к. не был бы рассмотрен базовый (минимальный) случай относительно аргумента структурной рекурсии; была бы гарантирована лишь *завершимость*, но не *тотальность*).

Рассмотрим еще один пример: вычисление наибольшего общего делителя.

```
gcd 0 x = x
gcd x 0 = x
gcd a b | a ≤ b = gcd a (b-a)
        | a > b = gcd (a-b) b
```

Эта процедура также не является структурно рекурсивной, однако является вполне обоснованно рекурсивной по отношению между аргументами рекурсивного (`a', b'`) и исходного (`a, b`) вызовов «a'+b' < a+b», поэтому она гарантированно завершается. Вполне обоснованная индукция становится несколько более явной, если ввести дополнительный аргумент, как и в случае `mergesort`:

```
gcd x y = gcd' x y (x+y)
gcd' 0 x _ = x
gcd' x 0 _ = x
gcd' a b s | a ≤ b = gcd a (b-a) b
           | a > b = gcd (a-b) b a
```

Теперь стало ясно видно, что для любого вызова `gcd' a b s` верно `a+b == s`, следовательно при `a≠0 && b≠0` верно `a<s && b<s`, следовательно при рекурсивных вызовах аргумент `s` уменьшается, но становится нулем только если `a==0 && b==0`, а в этом случае «сработает» какое-либо из первых двух уравнений. Поэтому функция `gcd` является и завершающейся, и тотальной.

Примеры `mergesort` и `gcd` взяты из презентации Ива Берто «Well-founded induction» [9].

Еще одно часто применяемое в специфических областях вполне обоснованное отношение — т. н. «гомеоморфное вложение». Это — отношение над формулами, и его свойство вполне обоснованности применяется в таких областях, как системы переписывания термов (например, при упрощении выражений), суперкомпиляция и т. п. Это отношение и его приложение к суперкомпиляции описаны, например, в статье Мортена Соренсена и Роберта Глюка «An algorithm of generalization in positive supercompilation» ([143]).

Реализация

Языки Coq, Epigram, Agda допускают только структурную рекурсию, причем в Coq при введении структурно рекурсивного определения необходимо явно указывать аргумент, по которому производится структурная рекурсия; как следствие, все процедуры на этих языках завершаются (если исключить из рассмотрения присутствующую в Coq коиндукцию). В стандартной библиотеке языка Coq есть модуль «Coq.Init.Wf» ([184]), реализующий вполне обоснованную индукцию. В последних версиях реализована также экспериментальная команда `Function`, позволяющая непосредственно определять не-структурно рекурсивные функции и отдельно доказывать свойство вполне обоснованности отношения между аргументами исходного и рекурсивных вызовов. Кроме того, Coq при определении индуктивного типа данных (см. 5.9) генерирует определения, соответствующие операторам структурной рекурсии над этим типом.

5.13. Класс типов (Type class)

Суть

Реализации интерфейсов для типов и их комбинаций указываются в произвольном месте программы и отдельно от определений самих типов.

История

Классы типов были предложены Филипом Вадлером в 1989 году в статье «How to make ad-hoc polymorphism less ad hoc» ([164]) в качестве варианта специального полиморфизма. Изначально предполагалось применить их для реализации перегрузки арифметических операторов и операторов сравнения. Авторы работы были недовольны сложившейся ситуацией: на тот момент среди разработчиков языков программирования отсутствовало общее мнение о том, как следует решать связанные с этим проблемы. К примеру, язык ML вводил специальное понятие «типов, сравнимых на равенство» (eqtypes), а стандарт этого языка хоть и упоминал возможность перегрузки арифметических операторов, но не уточнял, что это такое.

Классы типов оказались удачным решением и вошли в стандарт языка Haskell. С тех пор было найдено огромное множество различных применений классов типов, они стали одним из основных элементов системы типов Haskell, его «визитной карточкой», а также стали проникать в некоторые другие языки (Mercury, Coq). Кроме того, был предложен ряд расширений, увеличивающих выразительную силу и удобство пользования классами типов; вот некоторые из них:

- Многопараметрические классы типов (Саймон Пейтон-Джонс, Марк Джонс, Эрик Мейер, «Type classes: an exploration of the design space», 1997 г. [120]);
- Функциональные зависимости (Марк Джонс, «Type classes with functional dependencies», 2000 г. [71]);
- Семейства типов (две статьи Чакраварти, Келлера и Пейтона-Джонса: «Associated types with class» ([2], 2005), «Associated type synonyms» ([18], 2005), а также «Type checking with open type functions» Чакраварти, Пейтона-Джонса и других ([158], 2008)).

Интуиция

Основная цель интерфейсов в ООП — абстрагирование конкретного типа некоторого значения от клиента, производящего над значением лишь ограниченный и заранее известный набор операций (в данной статье здесь и далее «клиент» — код, использующий данные операции).

Интерфейсы позволяют:

- Применять клиентский код ко множеству различных типов, коль скоро эти типы реализуют нужные операции;
- Применять клиентский код к типам, недоступным клиенту на момент написания и компиляции.

Например, разработчик библиотеки хэш-таблиц может писать ее в расчете на произвольный тип ключей, поддерживающих операцию хэширования и операцию сравнения на равенство. Пользователям библиотеки достаточно реализовать соответствующий интерфейс для своих типов ключей, а разработчик ее, может быть, никогда даже не узнает об их существовании.

Однако, интерфейсы не обеспечивают третьей возможности: возможности применять клиентский код к типам, написанным *без знания о существовании такого клиента* и о требующемся ему наборе операций.

К примеру, рассмотрим ситуацию, когда программисту-генетику требуется хранить последовательности генов в хэш-таблице, причем «генетическая» библиотека предоставлена одним поставщиком, а библиотека хэш-таблиц — другим. Разработчик генетической библиотеки ничего не знал о существовании библиотеки хэш-таблиц, и не реализовывал для типа «последовательность генов» операцию хэширования, а операцию сравнения на равенство назвал иначе, чем требует библиотека хэшей. Программист не имеет доступа к исходному коду генетической библиотеки и не может изменить тип «последовательность генов». Ему придется реализовать тип-обертку, хранящий в себе объект типа «последовательность генов» и реализующий как его интерфейс (через делегирование), так и интерфейс хэширования. К сожалению, может оказаться, что такой оберткой уже по тем или иным причинам невозможно пользоваться внутри генной библиотеки — скажем, все ее операции реализованы для одного конкретного типа «последовательность генов», закрытого для наследования. Эта трудность проистекает от того, что системы типов практически всех ОО-языков требуют при проектировании типа заранее знать все интерфейсы, которые от него могут понадобиться.

Классы типов снимают это ограничение и позволяют в любой момент разработки указывать способ реализации тех или иных операций для того или иного типа или даже целых абстрактных наборов типов (скажем, они позволяют указать способ вывода реализации хэширования для типа «коллекция T» из реализации для типа T) и их комбинаций.

Описание

Классом типов называется именованный набор сигнатур операций, упоминающих этот тип. Например: (все примеры в данной статье написаны на языке Haskell)

```
class Hashable τ where
    hash  :: τ → Int
    equal :: τ → τ → Bool
```

В этом примере объявлен класс `Hashable`: «тип, годный в качестве ключа хэш-таблицы». Класс состоит из двух сигнатур операций: операции хэширования и операции сравнения на равенство (обратите внимание на отличие сигнатуры `equal` от сигнатуры `equals` в таких ОО-языках, как Java или C#).

Клиент такого класса типов может указывать в сигнатурах полиморфных функций, что для того или иного *т*ипового аргумента обязана существовать реализация операций этого класса, например:

```
makeHashtable :: (Hashable key) ⇒ [(key, val)] → (key →
Maybe val)
```

Такая форма полиморфизма называется «ограниченным параметрическим полиморфизмом».

В этом примере функция `makeHashtable`, требует, чтобы тип `key` принадлежал классу типов `Hashable`. Она принимает на вход список пар `(key,val)`, а возвращает функцию поиска, отображающую ключ типа `key` в значение типа `val` или в сигнал об отсутствии значения для этого ключа.

Для того, чтобы указать реализацию класса типов для конкретного типа, необходимо объявить экземпляр этого класса типов. Например, объявим экземпляр `Hashable` для типа целочисленных пар.

```
instance Hashable (Int,Int) where
    hash  (x,y)       = x `xor` y
    equal (a,b) (c,d) = (a==c) && (b==d)
```

Очень часто можно использовать при реализации класса типов реализацию других классов типов. Например, при реализации `Hashable` для пар типа (α,β) можно использовать реализацию `Hashable` для α и для β:

```
instance Hashable α, Hashable β ⇒ Hashable (α,β) where
    hash  (a,b)           = (33*hash a) + hash b
    equal (a1,b1) (a2,b2) = equal a1 a2 && equal b1 b2
```

Если обычные классы типов позволяют указывать способ реализации тех или иных операций для некоторого типа, то многопараметрические классы типов позволяют делать это для комбинаций типов. Типичный пример такой ситуации — операция преобразования между различными типами:

```
class Convertible α β where
    convert :: α → Maybe β

instance Convertible Double Int32 where
    convert x
      | x < -2147483648.0 = Nothing
      | x > 2147483647.0  = Nothing
      | otherwise         = Just (round x)
```

Понятие класса типов также применяется к параметрическим типам с одним или несколькими аргументами, например, `Tree` α; такие типы также называются «*т*иповыми конструкторами».[18] В этом случае сигнатуры операций класса могут включать в себя применение этого конструктора. В нижеприведенном примере записан класс типов, соответствующий понятию коллекции с операциями «размер», «преобразование в список и обратно», «проверка вхождения», «добавление и удаление», и определена его реализация для конструктора типа списков `[`α`]`.

```
class Collection κ α where
    size     :: κ α → Int
    fromList :: [α] → κ α
    toList   :: κ α → [α]
    elem     :: α → κ α → Bool
    add      :: α → κ α → κ α
    remove   :: α → κ α → κ α

instance (Eq α) ⇒ Collection [α] α where
    size = length
    fromList    = id
    toList      = id
    elem        = Data.List.elem
    add x xs | elem x xs = xs
             | otherwise = x:xs
    remove x xs = filter (≠ x) xs
```

Использование

Выше уже был приведен один пример использования классов типов: библиотека хэш-таблиц. Приведем еще несколько примеров их использований из библиотек языка Haskell.

- Haskell определяет несколько стандартных классов типов:
  - Арифметические классы `Num`, `Integral`, `Floating` и т. п., соответствующие понятию чисел с определенными над ними различными операциями;
  - `Eq` и `Ord` — классы значений, сравнимых на равенство и на упорядоченность;
  - `Show` и `Read` — классы значений, обладающих строковым представлением и способом воссоздать значение из строки;
  - `IArray`, `MArray` — многопараметрические классы читаемых и изменяемых массивов (см. статью в Haskellwiki «Arrays» [1]).

  А также более «абстрактные» классы:

[18] Не следует путать понятие *типового конструктора* с понятием *конструктора алгебраического типа (см. 5.9)*.

  - `Monad` — класс монад;
  - `Monoid` — класс типов, чьи значения образуют алгебраическую структуру «моноид» (см. статью «Моноиды в Haskell и их использование» (Дэн Пипони, перевод Кирилла Заборского) [125], а также презентацию Эдварда Кметта «Introduction to monoids» [80]), т. е. на значениях определена бинарная операция «комбинирования»:

    ```
    mappend :: (Monoid m) ⇒ m → m → m
    ```

    А также задан «нейтральный элемент» `mempty :: (Monoid m) ⇒ m`, являющийся единицей для `mappend`); а также похожий класс `MonadPlus`;
  - `Foldable` и `Traversable` — классы «Структура данных, поддающаяся свертке (см. 5.11)» и «Структура данных, поддающаяся обходу»;

  Существует прекрасная и очень подробная статья Брента Йорджи обо всех основных классах типов в Haskell, под названием «Typeclassopedia» [170].

- Популярная комбинаторная библиотека (см. 5.14) случайного тестирования QuickCheck Коэна Клэссена ([23]) основана на классе `Arbitrary`, соответствующему (упрощенно) понятию «тип, для которого известна процедура порождения случайных значений этого типа». В библиотеке определяются экземпляры этого класса для некоторых основных типов и вывод экземпляров для пар, списков и т. п. Пользователь библиотеки может также определить экземпляр класса `Arbitrary` для своих типов данных. Скажем, примерно так выглядит код порождения случайных бинарных деревьев: для порождения случайного дерева с вероятностью $2/3$ порождается случайное значение и обертывается «листом», либо с вероятностью $1/3$ рекурсивно порождаются два случайных дерева и обертываются «вилкой» (см. также статью об алгебраических типах (см. 5.9)).

  ```
  data Tree α = Leaf α | Fork (Tree α) (Tree α)

  instance (Arbitrary α) ⇒ Arbitary (Tree α) where
      arbitrary = frequency [
          (2, liftM Leaf arbitrary),
          (1, liftM2 Fork arbitrary arbitrary) ]
  ```

  Данная библиотека интересна также тем, что портирована на множество других языков, от Erlang до Java и C++, и эти реализации тем или иным образом имитируют типы классов.

- Библиотека двоичной сериализации/десериализации «binary» также основана на классах типов. С их помощью определены способы сериализации/десериализации для базовых типов и комбинаций типов (пар, списков и т. п.). Вот пример ее использования для сериализации/десериализации арифметических выражений, взятый с сайта библиотеки:

```
data Exp = IntE Int
         | OpE  String Exp Exp

instance Binary Exp where
    put (IntE i)      = put (0 :: Word8) ≫ put i
    put (OpE s e1 e2) = put (1 :: Word8) ≫ put s
                                ≫ put e1 ≫ put e2
    get = do tag ← getWord8
             case tag of
                 0 → liftM  IntE get
                 1 → liftM3 OpE  get get get
```

Аналогичный подход используется в известной статье Эндрю Кеннеди «Pickler combinators» ([76]) из серии «жемчужины функционального программирования (functional pearls)», однако там вместо классов типов используется передача словарей (см. раздел «Имитация»).

- Классы типов могут быть использованы для частичной эмуляции зависимых типов (типов, параметризованных значениями или высказываниями: например, векторов определенной длины с гарантированно безопасным доступом без выхода индексов за границы, или сбалансированных деревьев, чье свойство сбалансированности невозможно нарушить в рамках системы типов, и т. п.). Например, см. статью Конора Мак Брайда «Faking it: Simulating dependent types in haskell» ([95]).

- Библиотека линейной алгебры фиксированной размерности «Vec» [36] целиком основана на классах типов и использует их для проведения операций над размерностями во время компиляции, что позволяет полностью избавиться от проверок выхода за границы вектора или матрицы во время исполнения.

- В статье-«жемчужине» «Generic descrimination: sorting and partitioning unshared data in linear time» Фрица Хенгляйна ([50]) при помощи классов типов описывается способ сортировки и группировки списков за линейное время, работающий для большой части часто встречающихся типов.

- Недавно появившаяся в компиляторе GHC поддержка семейств типов основана на классах типов и предоставляет множество очень интересных возможностей (упомянутые в секции «История» статьи Пейтона-Джонса и Чакраварти [2], [18], [158]). Среди практических применений этих возможностей — «самооптимизирующиеся» контейнеры (библиотека «unboxed-containers» на Hackage [82]), образцы (см. 5.10) как объекты первого класса (библиотека «first-class-patterns» [128]), обобщенные префиксные деревья (библиотека TrieMap [166]) и т. п.

Реализация

Наиболее распространенный язык, поддерживающий классы типов — это Haskell. Именно в нем они впервые появились в 1989 г. В 1997 г. в компиляторе GHC появилась поддержка многопараметрических классов типов, однако в стандарт Haskell98 не вошла. В 2005 г. появилась поддержка семейств типов.

На странице «Research papers/Type systems» в разделе «Type classes» [133] в Haskellwiki можно найти множество исследовательских работ, посвященных различным аспектам семантики, реализации и применения классов типов.

Также классы типов поддерживаются в функционально-логическом языке Mercury, в языке Coq версии 8.2, а также некоторых других языках.

В языке OCaml есть концепция, родственная классам типов — модули. В статьях Чакраварти и других «ML modules and Haskell type classes: a constructive comparison» [167] и «Modular type classes» [103] приведено их сравнение, показаны параллели между их возможностями и стилем программирования.

«Концепты», так и не вошедшие в стандарт C++0x, также схожи по своим возможностям с классами типов. В статье «A comparison of C++ concepts and Haskell type classes» Жана-Филиппа Бернарди и других [27] приведено их сравнение. Эта статья чрезвычайно интересна и ее стоит прочитать даже тем, кто не интересуется C++, поскольку в ней само понятие класса типов «разбирается по косточкам», рассматриваются его составляющие, оси изменчивости возможностей и обсуждается роль этих возможностей в программировании.

Имитация

Нередко для одного и того же типа возможно несколько различных реализаций (экземпляров) класса. Например, для записи с тремя полями существует, как минимум, 27 различных реализаций класса `Ord`, производящих лексикографическое сравнение разных подмножеств полей в различном порядке, по возрастанию или убыванию; и это не считая даже того, что и сравнение самих полей может производиться по-разному. Не для всех типов данных имеется какая-то одна «наиболее естественная» реализация некоторого класса.

Один из способов решить подобную проблему таков: создается новый тип-обертка вокруг исходного типа, и экземпляр объявляется для типа-обертки. Например, вот так можно «перевернуть» порядок над некоторым типом:

```
newtype Desc α = Desc α
instance (Eq α) ⇒ Eq (Desc α) where
  (Desc x) == (Desc y) = x == y
instance (Ord α) ⇒ Ord (Desc α) where
  (Desc x) ≤ (Desc y) = y ≤ x

> sort [Desc 3, Desc 8, Desc 5]
[Desc 8, Desc 5, Desc 3]
```

Еще один яркий пример такого подхода — модуль `Data.Monoid` [175]. Однако данный прием все же предполагает, что все возможные реализации фиксированы и известны статически, и не позволяет создать реализацию класса для некоторого типа динамически: скажем, полиморфная процедура сортировки, сортирующая значения типа α, где `Ord` α, сгодится для сравнения элементов списка по задаваемому пользователем условию (скажем, для сортировки рядов таблицы по заданным столбцам). Поэтому процедурам сортировки приходится принимать на вход процедуру сравнения непосредственно.

Непосредственная передача функциям реализаций всех операций для типа называется стилем передачи словарей (dictionary-passing style). Например, сравним сигнатуры процедур для построения хэш-индекса с использованием классов типов и с использованием стиля передачи словарей:

```
makeHashtableTC :: (Hashable key) ⇒
   [(key, val)] → (key → Maybe val)

makeHashtableDP :: (key → Int, key → key → Bool) →
   [(key, val)] → (key → Maybe val)
```

В объектно-ориентированных языках часто используется еще один прием имитации классов типов: составление словаря «тип → реализации операций для него» в явной форме. Скажем, библиотека сериализации может быть устроена примерно так:

```
interface SerialForm<T> {
    byte[] serialize(T t);
    T deserialize(byte[] data);
}

class Serializer {
    public <T> void useSerialForm(
      Class<T> clazz, SerialForm<T> serialForm);

    byte[] serialize(Object data);
}

...
s.useSerialForm(Integer.class, new IntBigEndianSF());
s.useSerialForm(String.class,  new StringSF());
...
```

Еще один любопытный пример использования стиля передачи словарей приведен в статье о замыканиях (см. 5.4), в секции «Использование».

5.14. Комбина́торная библиотека (Combinator library)

Суть

Модель предметной области, выстроенная из небольшого количества «базовых» сущностей и абстрактных способов их комбинирования.

Интуиция

Рассмотрим задачу проверки пользовательского ввода: например, является ли заданная строка записью числа с плавающей точкой или, скажем, даты в формате YYYY/Mon/DD HH:MM:SS. Есть как минимум три различных подхода к решению такой задачи:

- написание специализированного кода для разбора строки: проверка строки символ за символом, использование indexOf, subString и т. п.;
- использование специализированной библиотеки для разбора строк заданного типа — например, функций scanf или strptime;
- использование регулярных выражений.

Код, написанный первым способом, обычно а) плохо читается (то есть, по коду сложно понять, соответствует ли он спецификации, и есть ли в нём ошибки); б) его сложно изменять (после изменения обычно требуется заново перепроверить весь алгоритм); в) написание такого кода требует высокой квалификации программиста. В то же время, этот способ можно использовать для решения любых задач, и при некоторых усилиях он позволяет добиться эффективности.

Второй способ практически идеален: он менее всего подвержен ошибкам, предельно читаем, легко изменяем и, при достаточно хорошей реализации библиотеки, обладает ненамного меньшей производительностью, чем первый. Единственный его недостаток — отсутствие универсальности: к примеру, если понадобится валидация дат в формате, не предусмотренном библиотекой, или вовсе, скажем, валидация географических адресов, то этот способ придется отбросить.

Третий же способ, обладая (в ситуации, когда неприменим второй) приемлемыми характеристиками производительности, корректности, читаемости и изменяемости, обладает также и универсальностью на довольно широком классе задач (хотя и несколько меньшем, чем у первого способа).

Именно об отличиях в характере универсальности между первым и третьим способом мы и поговорим.

Оба способа моделируют предметную область «Проверка строк на принадлежность к некоторому языку».

Первый способ моделирует ее в терминах элементарных операций над строками и конструкций языка программирования (списки, массивы, переменные, циклы, рекурсия, процедуры и т. п.). Поэтому модели взаимосвязанных или даже похожих языков будут довольно сильно различаться; *из взаимосвязанности языков не следует взаимосвязанность их моделей*. Например, довольно сложно сконструировать из кода для валидации чисел и кода для валидации дат эффективный код для валидации последовательности из даты и числа. Таким образом, первый способ не удовлетворяет свойству комбинируемости.

Второй способ моделирует предметную область *непосредственно*, хотя и более узко. Язык регулярных выражений специально заточен под комбинируемость, и из двух регулярных выражений, моделирующих языки A и B, легко собрать третье, моделирующее язык $AB = \{a\ b | a \in A \wedge b \in B\}$ и т. п. Предметная область собирается, например, из следующих элементов и способов их комбинирования:

Примитивы:

- классы символов (литералы: a, диапазоны: [0−9a−zA−Z], классы: [: space :]);
- начало и конец строки (^ и $);
- пустая последовательность ((())),

и способы их комбинирования:

- последовательная композиция ($E_1 E_2$);
- параллельная композиция ($E_1 | E_2$);
- повторение 0 или 1 раз (жадное или нет) ($E?$);
- повторение 0 или более раз (жадное или нет) ($E*$);
- повторение 1 или более раз (жадное или нет) ($E+$);
- повторение от m до n раз (жадное или нет) ($E\{m, n\}$);
- заключение в скобки (это чисто синтаксический элемент: если бы регулярные выражения задавались не последовательностью символов, а собственным синтаксическим деревом, то для целей валидации в скобках необходимости бы не было) ((E));
- значимость или незначимость регистра.

Повторимся: важен не синтаксис регулярных выражений, а набор предоставляемых ими примитивов и комбинаторов.

В случае регулярных выражений набор примитивов и комбинаторов выбран так, чтобы взаимосвязанные концепции предметной области имели взаимосвязанные модели. Это положительно сказывается на читаемости, лаконичности, очевидной корректности (возможности заметить ошибку, глядя на модель), изменяемости, универсальности.[19]

Регулярные выражения — пример очень простого и понятного, но в то же время невероятно мощного и универсального приема моделирования (не только в программировании) — комбина́торной модели предметной области.

Описание

Программные библиотеки, моделирующие предметную области при помощи комбина́торной модели, называются комбина́торными библиотеками.

Для комбинаторных библиотек характерно:

- соответствие терминологии библиотеки и терминологии предметной области;
- состав: типы, примитивы, комбинаторы первого и высшего порядка;
- свойство замыкания;
- возможность эффективной реализации.

Соответствие терминологии библиотеки и терминологии предметной области.

Модели сущностей, взаимосвязанных в предметной области, должны сами быть взаимосвязаны. Всякой концепции из предметной области должна соответствовать концепция в языке моделей. Пример этого свойства был рассмотрен выше для регулярных выражений: концепция последовательной композиции языков отражается такой же концепцией в языке регулярных выражений, и т. п.

Состав: типы, примитивы, комбинаторы первого порядка, комбинаторы высшего порядка.

Обычно комбинаторная библиотека состоит из «примитивов» (базовых, неделимых сущностей предметной области, таких как «регулярное выражение, распознающее один символ») и «комбинаторов» (способов комбинирования сущностей в более сложные).

Среди комбинаторов выделяют комбинаторы первого порядка, позволяющие собирать сложные сущности предметной области из простых (такие как, например, оператор последовательной или параллельной композиции в регулярных выражениях) и комбинаторы высшего порядка, позволяющие комбинировать действия самих комбинаторов. Поэтому комбинаторные библиотеки легче писать и использовать в языках, поддерживающих функции высшего порядка (см. 5.3).

[19] Автор не является оголтелым фанатом регулярных выражений и не предлагает использовать их в *каждой* задаче. Однако в своей области применения (проверка принадлежности и разбор строк достаточно простых языков — например, задание правил токенизации в лексических анализаторах) они превосходно иллюстрируют тему данной статьи.

Комбинаторы могут выражаться друг через друга: например, в случае регулярных выражений верно $A? = A|()$ и $A+ = AA*$.

Довольно часто комбинаторные библиотеки открыты для расширения: пользователь может определить свои собственные примитивы и комбинаторы, равноправные встроенным в библиотеку (впрочем, в случае регулярных выражений это не так).

Как и у любой другой программы, терминологическая основа комбинаторной библиотеки — это типы используемых в ней сущностей. Типы определяют, в частности, в каком контексте уместны какие примитивы и комбинаторы, и какие из них совместимы друг с другом. Так, логично, что в случае комбинаторной библиотеки для описания потоков данных (например, электронных схем) комбинатор «последовательная композиция блока с самим собой N раз» уместен только если типы входа и выхода блока совпадают.

Одно из современных «веяний» в типизации в комбинаторных библиотеках — использование GADT (обобщенных алгебраических типов) (см. 5.9). Например, см. иллюстрацию этой техники для парсеров в соответствующем разделе статьи о GADT в haskellwiki ([43]) и модель представления патчей в системе контроля версий darcs, описанную в презентации Ганеша Ситтампалама «Darcs and GADTs» ([140]).

Свойство замыкания.

Самое важное свойство, за соблюдением которого следует более всего следить при проектировании комбинаторной библиотеки — свойство замыкания. Составные сущности не должны *ничем* отличаться с точки зрения использования от атомарных, и должны быть допустимы в любом контексте, где допустима атомарная сущность.

Нарушение свойства замыкания делает язык не только сложнее (так как становится необходимо различать простые и составные сущности и помнить, в каком контексте какие из них допустимы), но и менее выразительным из-за невозможности использовать некоторые комбинаторы в некоторых контекстах. Более того, это закрывает путь к развитию концепции, навсегда ограничивая ее рамками первоначального замысла создателя. Автор данной статьи принадлежит к сторонникам мысли «Путь к выразительности языка — не добавление возможностей, а устранение искусственных ограничений» (также известна цитата: «Expressive power should be by design rather than by accident» из классической статьи Питера Ландина «The next 700 programming languages» [84]).

Пример нарушения этого свойства — типы в языке FORTRAN: FORTRAN позволяет создавать массивы из примитивных типов, однако не позволяет создавать массивы из массивов (многомерные массивы — не универсальное решение, т. к. они не позволяют использовать т. н. «зубчатые» (jagged) массивы, т. е. такие, чьи ячейки имеют разный размер).

Возможность эффективной реализации.

Часто от модели требуется возможность эффективной реализации. Например, при моделировании преобразований системы координат в компьютерной графике часто ограничиваются т. н. аффинными преобразованиями (перенос, поворот, масштабирование, отражение), поскольку они допускают чрезвычайно эффективную про-

граммную и аппаратную реализацию при помощи матричной арифметики, а также удовлетворяют свойству замыкания: последовательность аффинных преобразований сама по себе является аффинным преобразованием. Тем не менее, легко представить себе ситуацию, когда требуются более сложные преобразования координат: скажем, рябь или другие нелинейные искажения экрана.

Почти всегда имеет место компромисс между эффективностью реализации и универсальностью.[20] Например, регулярные выражения в самом простом случае можно чрезвычайно эффективно реализовать при помощи компиляции в конечный автомат (преобразование подробно описано в статье «Regular expression matching can be simple and fast» Расса Кокса [32]); в более сложных языках регулярных выражений также обычно стараются предоставить набор операторов, допускающий эффективную реализацию.

Особенно хорошо, когда примитивные сущности аналогичны составным и с точки зрения эффективности. Например, в случае аффинных преобразований это так: любое преобразование представляется умножением на матрицу, и даже композиция тысячи преобразований — это всего лишь матрица, умножение на которую ничуть не сложнее, чем умножение на матрицу поворота или отражения. Та же ситуация имеет место в случае подмножества регулярных выражений, поддающегося реализации с помощью конечных автоматов, а также для многих часто встречающихся классов грамматик более общего назначения.

Однако этого удается достичь не всегда: например, в случае нелинейных преобразований точек ничего не остается, кроме как применять композицию из нескольких преобразований путем последовательного применения каждого из них, т. е. композиция тысячи преобразований будет, по меньшей мере, в тысячу раз менее эффективна, чем каждое преобразование по отдельности. Более того, свои издержки вносит и сама «инфраструктура» библиотеки; стоит проектировать библиотеку так, чтобы эти издержки были минимальны, т. е. чтобы библиотека переносила «тяжелые» операции в примитивы. Этот подход рассмотрен в упомянутой лекции 4 курса [183] по отношению к преобразованиям координат.

Джон Хьюз читал целый курс о комбинаторных библиотеках «Designing and Using Combinators: The Essence of Functional Programming»; материалы курса свободно доступны в интернете ([60]).

Использование

Комбинаторные библиотеки — один из основных способов проектирования библиотек в функциональных языках. Вероятно, это объясняется тем, что функциональные языки располагают к составлению одних сущностей из других, за счет следующих свойств:

- удобная синтаксическая поддержка создания сложных структур данных;

[20] О том, как достичь компромисса между эффективностью и универсальностью в случае преобразований координат можно прочитать, например, в лекции номер 4 «Абстракция данных» курса «Функциональное программирование» Евгения Кирпичёва ([183]).

- поощрение неизменяемых данных, что, как рассмотрено в статье «Изменяемое состояние: опасности и борьба с ними» Евгения Кирпичёва ([182]), положительно сказывается на модульности и комбинируемости;
- обилие способов комбинирования — скажем, ФВП (см. 5.3) и замыкания (см. 5.4);
- очень мощные системы типов, позволяющие удобно моделировать терминологию предметной области.

В знаменитой статье «An experiment in software prototyping productivity» [59] Пола Хьюдэка и Марка Джонса на стр. 9 описывается, как программист на Haskell разработал для решения задачи комбинаторную библиотеку (язык представления геометрических областей), благодаря чему его код оказался в 2 раза короче, чем код ближайшего конкурента (и в 13 раз короче, чем код на C++).[21]

Аналогичный язык для представления картинок и анимаций кратко рассмотрен в очень интересной статье Пола Хьюдэка «Modular domain specific languages and tools» [58], где также рассмотрены и некоторые другие аспекты разработки комбинаторных библиотек.

Существует много библиотек для комбинаторного представления графики: [113], [19] (модуль Grid) и т. п.

Одно из самых удачных применений комбинаторных библиотек (в том числе и за пределами ФП) — это библиотеки для синтаксического анализа. Одна из лучших статей на данную тему — статья Филипа Вадлера «How to replace failure by a list of successes» [161], а также статья Грэхема Хаттона «Higher-order functions for parsing» [64] и его совместная статья с Эриком Мейером «Monadic parsing in Haskell» [66]. Данный подход оказался настолько хорош, что портированные комбинаторные библиотеки синтаксического разбора существуют для очень большого количества языков, далеко не только функциональных. Однако, пользуются ими все же почти исключительно в функциональных языках — вероятно, из-за неудобства манипулирования комбинаторами без синтаксической поддержки монад, ФВП (см. 5.3) и замыканий (см. 5.4). В презентации Эдварда Кметта «Iteratees, Parsec and monoids» [81] описан подход, позволяющий сделать комбинаторы синтаксического разбора инкрементальными, однако уровень сложности этого материала очень высокий, требуется знание некоторых других концепций, в частности, монад и т. н. iteratees.

В статье Криса Окасаки «Why would anyone ever want to use a sixth-order function?» [109] описаны, в приложении к синтаксическому разбору, примеры полезных комбинаторов чрезвычайно высоких порядков, вплоть до 6го.

В книге «Структура и интерпретация компьютерных программ» [48] в главе 2.2.4

[21] Если изучить стр. 9 статьи детально, то выясняются еще более шокирующие подробности соотношения продуктивности различных языков, заставляющие усомниться в корректности методики проведенного тестирования даже самых ярых сторонников ФП. Тем не менее, стоит ознакомиться со статьей, чтобы увидеть разницу в подходах к разработке на различных языках.

рассмотрен игрушечный «язык картинок», выстроенный в форме комбинаторной библиотеки с комбинаторами высшего порядка.

В статье «The design of a pretty-printing library» [62] (автор которой, опять же, Джон Хьюз) рассмотрена комбинаторная библиотека для представления и «красивого» форматирования текстов. Это применение стало одной из классических иллюстраций идеи комбинаторной библиотеки, а за этой статьей позднее последовало множество других на ту же тему («A prettier printer» Филипа Вадлера [163], «Pretty printing with lazy deques» Олафа Читила [20], «A pretty printer library in Haskell» Саймона Пейтона-Джонса [119], «Beyond pretty-printing: Galley concepts in document formatting combinators» Вольфрама Каля [74], «Linear, bounded, functional pretty-printing» Доатсе Свирстры [149], «Optimal pretty-printing combinators» [5] Азеро и Свирстры, и т. п.).

Знаменитая библиотека случайного тестирования свойств QuickCheck Коэна Клэссена ([22], описана в статье «Specification based testing with QuickCheck» [23]) также представляет собой пример комбинаторной библиотеки.

Язык образцов (см. 5.10) в Mathematica сходен с регулярными выражениями и также основан на примитивах и комбинаторах.

В статьях «Imperative streams — a monadic combinator library for synchronous programming» [137] и «A monad of imperative streams» [136] Энно Шольца рассматривается комбинаторная библиотека для программирования, основанного на событиях (функционального реактивного программирования), ставшая базисом библиотеки для построения пользовательского интерфейса FranTk. Библиотека FranTk так и не стала широко применяться, однако идеи функционального реактивного программирования дали начало, по меньшей мере, двум гораздо более удобным, мощным и распространенным средствам:

- В языке F# поддерживаются события как первоклассные значения, и имеется комбинаторная библиотека для манипуляций над событиями, описанная в блог-посте Маттеса Подвысоцки «F# first class events — composing events until others» [126]. Так, с ее помощью можно, имея события щелчка/отпускания/передвижения мыши, составить событие «перетаскивание мыши из одной точки в другую» и подписаться на него.

- Этот подход приходит и в C#: в .NET 4.0 планируется включить мощный фреймворк для программирования интерактивных приложений: Rx Framework Эрика Мейера, использующий идеи функционального реактивного программирования и идеологически схожий с подходом к событиям в F#. Rx Framework основан на инверсии потока управления при помощи монады продолжений (впрочем, пользователям не нужно знать об этом, чтобы им пользоваться), и использует в качестве синтаксической поддержки LINQ. Вот несколько блог-постов и видео-лекций про Rx Framework: лекции Мейера [96] (часть 1) и [97] (часть 2), «Reactive programming (I.) — First class events in F#» [117] и «Reactive programming (II.) — introducing Reactive LINQ» [118] Томаша Петричека, а также «Introducing Rx (Linq to Events» [63] Джафара Хусейна.

В презентации «Функциональное программирование в Java» Евгения Кирпичёва [181] описывается комбинаторная библиотека для асинхронного программирования на Java, также использующая монаду продолжений. F# реализует похожую библиотеку («asynchronous workflows», описаны в блог-посте основного разработчика F# Дона Сайма «Introducing F# asynchronous workflows»[150], в его же видео «What's new in F# — asynchronous workflows» [151], в статье Роберта Пикеринга «Beyond foundations of F# — asynchronous workflows» [122] и т. п.) при помощи использования монад в явной форме (workflow — термин из F#, фактически совпадающий с понятием монады).

Еще одна известная и получившая в свое время довольно широкую известность статья «Composing contracts: an adventure in financial engineering (functional pearl)» [72] описывает комбинаторную библиотеку для задания финансовых контрактов.

В статье «Функциональное программирование в Java» [180] приведено несколько примеров использования гипотетических и реальных комбинаторных библиотек на Java.

Литература

[1] Arrays. — Страница в haskellwiki, URL: http://www.haskell.org/haskellwiki/Arrays (дата обращения: 20 декабря 2009 г.).

[2] Associated types with class / M. M. T. Chakravarty, G. Keller, S. P. Jones, S. Marlow // — *ACM SIGPLAN Notices*2005. — January. — Vol. 40, no. 1. — Pp. 1–13.

[3] Authorization structures. — Страница MSDN, URL: http://msdn.microsoft.com/en-us/library/aa375780(VS.85).aspx (дата обращения: 20 декабря 2009 г.).

[4] *Axelsson E.* Bookshelf: A simple document organizer with some wiki functionality. — Пакет на hackage, URL: http://hackage.haskell.org/package/Bookshelf (дата обращения: 20 декабря 2009 г.).

[5] *Azero P., Swierstra S. D.* — Optimal pretty-printing combinators1998. — April.

[6] *Backus J.* Can programming be liberated from the von neumann style? // *Communications of the ACM*. — 1978. — Vol. 21, no. 8. — Pp. 613–641.

[7] *Barendregt H. P.* Lambda calculi with types. — 1992. — Pp. 117–309.

[8] *Bauer A.* — Compilation of Functional Programming Languages using GCC—Tail Calls. — Master's thesis, Institut für Informatik, Technische Universität München, 2003. http://home.in.tum.de/~baueran/thesis/.

[9] *Bertot Y.* Well-founded induction. — Слайды, URL: http://www-sop.inria.fr/members/Yves.Bertot/tsinghua/tsinghua-5.pdf (дата обращения: 20 декабря 2009 г.).

[10] *Bertot Y., Castéran P.* Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. — Springer Verlag, 2004. http://www.labri.fr/publications/l3a/2004/BC04.

[11] *Bird R., Meertens L.* Nested datatypes // MPC: 4th International Conference on Mathematics of Program Construction. — LNCS, Springer-Verlag, 1998.

[12] *Blelloch G.* Vector Models for Data-Parallel Computing. — Cambridge, MA: The MIT Press, 1990.

[13] *Blelloch G. E.* — Prefix sums and their applications1993. — February.

[14] *Boyapati C., Liskov B., Shrira L.* Ownership types for object encapsulation // In Principles of Programming Languages (POPL). — 2003. — Pp. 213–223. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.1440.

[15] *Brass P.* Advanced Data Structures. — Cambridge: Cambridge University Press, 2009.

[16] *Bringert B., Coutts D.* tar: Reading, writing and manipulating ".tar"archive files. — Пакет на hackage, URL: http://hackage.haskell.org/package/tar (дата обращения: 20 декабря 2009 г.). — 2007–2009.

[17] *Cardone F., Hindley J. R.* History of lambda-calculus and combinatory logic // *Handbook of the History of Logic*. — Vol. 5. http://lambda-the-ultimate.org/node/2679.

[18] *Chakravarty M. M. T., Keller G., Jones S. P.* Associated type synonyms // — *ACM SIGPLAN Notices*2005. — September. — Vol. 40, no. 9. — Pp. 241–253.

[19] Chart: A library for generating 2d charts and plots. — URL: http://hackage.haskell.org/package/Chart (дата обращения: 20 декабря 2009 г.).

[20] *Chitil O.* Pretty printing with lazy dequeues // — *ACM Trans. Program. Lang. Syst.*2005. — January. — Vol. 27, no. 1. — Pp. 163–184. http://dx.doi.org/10.1145/1053468.1053473.

[21] Church encoding. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Church_encoding (дата обращения: 20 декабря 2009 г.).

[22] *Claessen K.* Quickcheck: Automatic testing of haskell programs. — Пакет на hackage, URL: http://hackage.haskell.org/package/QuickCheck (дата обращения: 20 декабря 2009 г.).

[23] *Claessen K., Hughes J.* Specification based testing with QuickCheck // The Fun of Programming. — Palgrave, 2003. — Cornerstones of Computing. — Pp. 17–40.

[24] *Clare E. M., Gibbons J., Bayley I.* Disciplined, efficient, generalised folds for nested datatypes // *Formal Asp. Comput.* — 2004. — Vol. 16, no. 1. — Pp. 19–35. http://dx.doi.org/10.1007/s00165-003-0013-6.

[25] *Clements J., Felleisen M.* A tail-recursive machine with stack inspection // *ACM Trans. Program. Lang. Syst.* — 2004. — Vol. 26, no. 6. — Pp. 1029–1052.

[26] *Clinger W. D.* Proper tail recursion and space efficiency // Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation. — Montréal, Québec1998. — June. — Pp. 174–185.

[27] A comparison of c++ concepts and haskell type classes / J.-P. Bernardy, P. Jansson, M. Zalewski et al. // ICFP-WGP / Ed. by R. Hinze, D. Syme. — ACM, 2008. — Pp. 37–48.

[28] Continuations and continuation passing style. — URL: http://library.readscheme.org/page6.html (дата обращения: 20 декабря 2009 г.).

[29] Copy-on-write. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Copy-on-write (дата обращения: 20 декабря 2009 г.).

[30] The coq proof assistant. — URL: http://coq.inria.fr/ (дата обращения: 20 декабря 2009 г.).

[31] *Coutts D.* zlib: Compression and decompression in the gzip and zlib formats. — Пакет на hackage, URL: http://hackage.haskell.org/package/zlib (дата обращения: 20 декабря 2009 г.).

[32] *Cox R.* Regular expression matching can be simple and fast. — — URL: http://swtch.com/~rsc/regexp/regexp1.html (дата обращения: 20 декабря 2009 г.)2007. — January.

[33] *Danvy O., Nielsen L. R.* Defunctionalization at work // PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming. — New York, NY, USA: ACM, 2001. — Pp. 162–174. http://dx.doi.org/10.1145/773184.773202.

[34] *Del Gallego P.* Curried function and labeled arguments in ocaml. — — Пост в блоге, URL: http://theplana.wordpress.com/2007/04/26/curried-function-and-labeled-arguments-in-ocaml/ (дата обращения: 20 декабря 2009 г.)2007. — April.

[35] Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, J. Vlissides. — Addison-Wesley Professional, 1995.

[36] *Dillard S. E.* Vec: Fixed-length lists and low-dimensional linear algebra. — Пакет на hackage, URL: http://hackage.haskell.org/package/Vec (дата обращения: 20 декабря 2009 г.). — 2009.

[37] *Felleisen M.* Functional objects. — Слайды с выступления на ECOOP 2004, URL: http://www.ccs.neu.edu/home/matthias/Presentations/ecoop2004.pdf (дата обращения: 20 декабря 2009 г.).

[38] *Ferguson D., Deugo D.* Call with current continuation patterns // Proceedings of the 2001 Pattern Languages of Programs Conference. — 2001.

[39] *Finne S.* json: Support for serialising haskell to and from json. — Пакет на hackage, URL: http://hackage.haskell.org/package/json (дата обращения: 20 декабря 2009 г.).

[40] *Flatt M.* Getting rid of `set-car!` and `set-cdr!` — Пост в блоге PLT Scheme Blog, URL: http://blog.plt-scheme.org/2007/11/getting-rid-of-set-car-and-set-cdr.html (дата обращения: 20 декабря 2009 г.).

[41] Foldr, foldl, foldl'. — Страница в haskellwiki, URL: http://www.haskell.org/haskellwiki/Foldr_Foldl_Foldl' (дата обращения: 20 декабря 2009 г.).

[42] *Gafter N.* Comments on the straw man…. — URL: http://mail.openjdk.java.net/pipermail/lambda-dev/2009-December/000023.html (дата обращения: 20 декабря 2009 г.).

[43] Generalized algebraic datatype: Parsing example. — Статья в HaskellWiki, URL: http://www.haskell.org/haskellwiki/GADT#Parsing_Example (дата обращения: 20 декабря 2009 г.).

[44] *Gibbons J.* The Third Homomorphism Theorem // — *Journal of Functional Programming*1996. — July. — Vol. 6, no. 4. — Pp. 657–665. — Functional Pearls.

[45] *Girard J.-Y., Lafont Y., Taylor P.* Proofs and Types. — Cambridge University Press, 1989. — Vol. 7 of *Cambridge Tracts in Theoretical Computer Science*. — This is textbook on proof theory and type systems, based on lectures by Girard. It contains an appendix by Lafont on linear logic, and also treats Girard's polymorphic lambda calculus.

[46] *Goerzen J.* Hdbc: Haskell database connectivity. — Пакет на hackage, URL: http://hackage.haskell.org/package/HDBC (дата обращения: 20 декабря 2009 г.).

[47] *Goerzen J.* Hsh: Library to mix shell scripting with haskell programs. — Пакет на hackage, URL: http://hackage.haskell.org/package/HSH (дата обращения: 20 декабря 2009 г.).

[48] *Harold Abelson J. S.* Structure and Interpretation of Computer Programs, second edition. — 1996. http://mitpress.mit.edu/sicp/full-text/book/book.html.

[49] heap: Heaps in haskell. — URL: http://hackage.haskell.org/package/heap (дата обращения: 20 декабря 2009 г.).

[50] *Henglein F.* Generic discrimination: Sorting and paritioning unshared data in linear time // Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008 / Ed. by J. Hook, P. Thiemann. — ACM, 2008. — Pp. 91–102. http://doi.acm.org/10.1145/1411204.1411220.

[51] *Hickey R.* Trampoline for mutual recursion. — Обсуждение в рассылке clojure, URL: http://groups.google.com/group/clojure/browse_thread/thread/6257cbc4454bcb85/3addf875319c5c10 (дата обращения: 20 декабря 2009 г.).

[52] *Hindley R.* Сообщение об истории алгоритма вывода типов Хиндли-Милнера. — Сообщение в рассылке TYPES в MIT, URL: http://www.cis.upenn.edu/~bcpierce/types/archives/1988/msg00042.html (дата обращения: 20 декабря 2009 г.). — 1988.

[53] *Hinze R.* Efficient generalized folds // Proc. of 2nd Workshop on Generic Programming, WGP'2000 (Ponte de Lima, Portugal, 6 July 2000) / — Ed. by J. Jeuring2000. — June. — Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ. http://www.cs.ioc.ee/~tarmo/papers/wgp00.ps.gz.

[54] *Hinze R.* Fun with phantom types // The Fun of Programming / Ed. by J. Gibbons, O. de Moor. — Palgrave Macmillan, 2003. — Pp. 245–262. — ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

[55] *Hinze R., Jeuring J., Löh A.* — Typed Contracts for Functional Programming2006. — January. — Vol. 3945. — Pp. 208–225. http://dx.doi.org/10.1007/11737414_15.

[56] Hipmunk: A haskell binding for chipmunk. — URL: http://hackage.haskell.org/package/Hipmunk (дата обращения: 20 декабря 2009 г.).

[57] *Holters M.* — Efficient data structures in a lazy functional language2003. — November. http://citeseer.ist.psu.edu/706344.html;http://www.tu-harburg.de/~simh0054/Edison/EfficientDataStructures.ps.

[58] *Hudak P.* Modular domain specific languages and tools // — Proceedings of Fifth International Conference on Software Reuse1998. — June. — Pp. 134–142.

[59] *Hudak P., Jones M. P.* Haskell vs. ada vs. c++ vs awk vs ... an experiment in software prototyping productivity: Tech. rep.: 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.2902.

[60] *Hughes J.* Designing and using combinators: The essence of functional programming. — URL: http://www.math.chalmers.se/~rjmh/Combinators/ (дата обращения: 20 декабря 2009 г.).

[61] *Hughes J.* Why functional programming matters: Tech. Rep. 16: Programming Methodology Group, University of Goteborg1984. — November.

[62] *Hughes J.* The design of a pretty-printing library. — 1995. — Pp. 53–96. http://dx.doi.org/10.1007/3-540-59451-5_3.

[63] *Husain J.* Introducing rx (linq to events). — Пост в блоге, URL: http://themechanicalbride.blogspot.com/2009/07/introducing-rx-linq-to-events.html (дата обращения: 20 декабря 2009 г.).

[64] *Hutton G.* Higher-order Functions for Parsing // — *Journal of Functional Programming*1992. — July. — Vol. 2, no. 3. — Pp. 323–343.

[65] *Hutton G.* A tutorial on the universality and expressiveness of fold // *J. Funct. Program.* — 1999. — Vol. 9, no. 4. — Pp. 355–372.

[66] *Hutton G., Meijer E.* Monadic parsing in haskell // — *Journal of Functional Programming*1998. — July. — Vol. 8, no. 4. — Pp. 437–444.

[67] *Iry J.* Tail calls via trampolining and an explicit instruction. — Обсуждение в рассылке Scala, URL: http://old.nabble.com/Tail-calls-via-trampolining-and-an-explicit-instruction-td20702915.html (дата обращения: 20 декабря 2009 г.).

[68] J software. — URL: http://jsoftware.com (дата обращения: 20 декабря 2009 г.).

[69] *Jansen J. M., Koopman P. W. M., Plasmeijer R.* Efficient interpretation by transforming data types and tatterns to functions // Trends in Functional Programming. — 2006. — Pp. 73–90.

[70] *Jarno A.* gcc-4.3/alpha: `-foptimize-sibling-calls` generates wrong code. — Письмо в рассылку `debian-gcc`, URL: http://www.mail-archive.com/debian-gcc@lists.debian.org/msg31603.html (дата обращения: 20 декабря 2009 г.).

[71] *Jones M. P.* Type classes with functional dependencies // Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings / Ed. by G. Smolka. — Vol. 1782 of *Lecture Notes in Computer Science*. — Springer, 2000. — Pp. 230–244. http://link.springer.de/link/service/series/0558/bibs/1782/17820230.htm.

[72] *Jones S. P., Eber J.-M., Seward J.* Composing contracts: an adventure in financial engineering (functional pearl) // ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. — New York, NY, USA: ACM, 2000. — Pp. 280–292.

[73] *jr. G. L. S.* Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, lambda: The ultimate goto: Report A. I. MEMO 443. — Cambridge, Massachusetts: Massachusetts Institute of Technology, A.I. Lab., 1977.

[74] *Kahl W.* Beyond pretty-printing: Galley concepts in document formatting combinators // PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages. — London, UK: Springer-Verlag, 1998. — Pp. 76–90.

[75] *Kelsy R. A.* Tail-recursive stack disciplines for an interpreter: Tech. Rep. NU-CCS-93-03: College of Computer Science, Northeastern University1993. — March. ftp://ftp.cs.indiana.edu/pub/scheme-repository/txt/stack-gc.ps.gz.

[76] *Kennedy A.* Pickler combinators // *J. Funct. Program.* — 2004. — Vol. 14, no. 6. — Pp. 727–739. http://dx.doi.org/10.1017/S0956796804005209.

[77] *Kennedy A., Russo C. V.* Generalized algebraic data types and object-oriented programming // *SIGPLAN Not.* — 2005. — Vol. 40, no. 10. — Pp. 21–40.

[78] *Kirpichov E.* digest: Various cryptographic hashes for bytestrings; crc32 and adler32 for now. — Пакет на hackage, URL: http://hackage.haskell.org/package/digest (дата обращения: 20 декабря 2009 г.).

[79] *Kirpichov E.* jarfind: Tool for searching java classes, members and fields in classfiles and jar archives. — Пакет на hackage, URL: http://hackage.haskell.org/package/jarfind (дата обращения: 20 декабря 2009 г.).

[80] *Kmett E.* Introduction to monoids (slides). — Пост в блоге, URL: http://comonad.com/reader/2009/iteratees-parsec-and-monoid/ (дата обращения: 20 декабря 2009 г.). — 2009.

[81] *Kmett E.* Iteratees, parsec and monoids. — — Пост в блоге, URL: http://comonad.com/reader/2009/iteratees-parsec-and-monoid/ (дата обращения: 20 декабря 2009 г.)2009. — August.

[82] *Kmett E. A.* unboxed-containers: Self-optimizing unboxed sets using view patterns and data families. — Пакет на hackage, URL: http://hackage.haskell.org/package/unboxed-containers (дата обращения: 20 декабря 2009 г.).

[83] Kx systems - fast database for streaming and historical data. — URL: http://kx.com (дата обращения: 20 декабря 2009 г.).

[84] *Landin P. J.* The next 700 programming languages // *CACM*. — 1966. — Vol. 9. — Pp. 157–166.

[85] *Läufer K.* Type classes with existential types // — *Journal of Functional Programming*1996. — May. — Vol. 6, no. 3. — Pp. 485–517. ftp://ftp.math.luc.edu/pub/laufer/papers/haskell+extypes.ps.gz.

[86] *Läufer K., Odersky M.* Polymorphic type inference and abstract data types // *ACM Trans. Program. Lang. Syst.* — 1994. — Vol. 16, no. 5. — Pp. 1411–1430.

[87] Laziness. — Страница в Haskell Wikibooks, URL: http://en.wikibooks.org/wiki/Haskell/Laziness (дата обращения: 20 декабря 2009 г.).

[88] *Lea K.* A huge gotcha with javascript closures and loops. — Пост в блоге, URL: http://joust.kano.net/weblog/archive/2005/08/08/a-huge-gotcha-with-javascript-closures/ (дата обращения: 20 декабря 2009 г.).

[89] *Leshchinsky R.* fixpoint: Data types as fixpoints. — Пакет на hackage, URL: http://hackage.haskell.org/package/fixpoint (дата обращения: 20 декабря 2009 г.).

[90] *LLC H.* Happs-server: Web related tools and services. — Пакет на hackage, URL: http://hackage.haskell.org/package/HAppS-Server (дата обращения: 20 декабря 2009 г.).

[91] Making data structures persistent / Driscoll, Sarnak, Sleator, Tarjan // *JCSS: Journal of Computer and System Sciences*. — 1989. — Vol. 38.

[92] *Malcolm G.* Algebraic Data Types and Program Transformation: Ph.D. thesis / Groningen University. — 1990.

[93] *Marlow S., Jones S. P.* Making a fast curry: push/enter vs. eval/apply for higher-order languages // *SIGPLAN Not.* — 2004. — Vol. 39, no. 9. — Pp. 4–15.

[94] *Marshall J.* — Серия блог-постов: URL: http://funcall.blogspot.com/2009/04/you-knew-id-say-something.html (дата обращения: 20 декабря 2009 г.), URL: http://funcall.blogspot.com/2009/04/you-knew-id-say-something-part-ii.html (дата обращения: 20 декабря 2009 г.), URL: http://funcall.blogspot.com/2009/05/you-knew-id-say-something-part-iii.html (дата обращения: 20 декабря 2009 г.), URL: http://funcall.blogspot.com/2009/05/you-knew-id-say-something-part-iv.html (дата обращения: 20 декабря 2009 г.), URL: http://funcall.blogspot.com/2009/05/you-knew-id-say-something-part-v.html (дата обращения: 20 декабря 2009 г.).

[95] *McBride C.* Faking it: Simulating dependent types in haskell // *J. Funct. Program.* — 2002. — Vol. 12, no. 4&5. — Pp. 375–392.

[96] *Meijer E., Dyer W.* Reactive framework (rx) under the hood — 1 of 2. — Видео на Channel9, URL: http://channel9.msdn.com/shows/Going+Deep/E2E-Erik-Meijer-and-Wes-Dyer-Reactive-Framework-Rx-Under-the-Hood-1-of-2/ (дата обращения: 20 декабря 2009 г.).

[97] *Meijer E., Dyer W.* Reactive framework (rx) under the hood — 2 of 2. — Видео на Channel9, URL: http://channel9.msdn.com/shows/Going+Deep/E2E-Erik-Meijer-and-Wes-Dyer-Reactive-Framework-Rx-Under-the-Hood-2-of-2/ (дата обращения: 20 декабря 2009 г.).

[98] *Meijer E., Fokkinga M., Paterson R.* Functional programming with bananas, lenses, envelopes and barbed wire // *Lecture Notes in Computer Science.* — 1991. — Vol. 523. — Pp. 124–??

[99] *Miquel A.* An introduction to system f. — Слайды доступны по ссылке URL: http://www.pps.jussieu.fr/~miquel/slides/got03-1.pdf (дата обращения: 20 декабря 2009 г.).

[100] *Mitchell N.* Transformation and Analysis of Functional Programs: Ph.D. thesis / — University of York2008. — June. — P. 225. http://community.haskell.org/~ndm/downloads/paper-transformation_and_analysis_of_functional_programs-4_jun_2008.pdf.

[101] *Mitchell N., Runciman C.* Losing functions without gaining data // Haskell '09: Proceedings of the second ACM SIGPLAN symposium on Haskell. — ACM2009. — September. http://community.haskell.org/~ndm/downloads/paper-losing_functions_without_gaining_data-03_sep_2009.pdf.

[102] *Mitchell N., Runciman C.* Losing functions without gaining data. — — Слайды, URL: http://community.haskell.org/~ndm/downloads/slides-losing_functions_without_gaining_data-03_sep_2009.pdf (дата обращения: 20 декабря 2009 г.)2009. — September.

[103] Modular type classes / D. Dreyer, R. Harper, M. M. T. Chakravarty, G. Keller // POPL / Ed. by M. Hofmann, M. Felleisen. — ACM, 2007. — Pp. 63–70.

[104] *Mukai J.* Haskellnet: network related libraries such as pop3, smtp, imap. — Пакет на hackage, URL: http://hackage.haskell.org/package/HaskellNet (дата обращения: 20 декабря 2009 г.).

[105] *Rodriguez A., Holdermans S., Löh A., Jeuring J.* multirec: Generic programming for families of recursive datatypes. — Пакет на hackage, URL: http://hackage.haskell.org/package/multirec (дата обращения: 20 декабря 2009 г.).

[106] *Noble J.* Arguments and results // *Comput. J.* — 2000. — Vol. 43, no. 6. — Pp. 439–450. http://www3.oup.co.uk/computer_journal/hdb/Volume_43/Issue_06/430439.sgm.abs.html.

[107] *Okasaki C.* Binomial queues as a nested type. — Пост в блоге от 22 октября 2009 г., URL: http://okasaki.blogspot.com/2009/10/binomial-queues-as-nested-type.html (дата обращения: 20 декабря 2009 г.).

[108] *Okasaki C.* Edisoncore: A library of efficent, purely-functional data structures (core implementations). — Пакет на hackage, URL: http://hackage.haskell.org/package/EdisonCore (дата обращения: 20 декабря 2009 г.).

[109] *Okasaki C.* Functional pearl: Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function? // *Journal of Functional Programming*. — 1998. — Vol. 8, no. 2. — Pp. 195–199. http://citeseer.ist.psu.edu/okasaki99functional.html.

[110] *Okasaki C.* Purely Functional Data Structures. — Cambridge University Press, 1999.

[111] *Okasaki C.* Breadth-first numbering: Lessons from a small exercise in algorithm design // ICFP. — 2000. — Pp. 131–136. http://doi.acm.org/10.1145/351240.351253.

[112] *Oli R.* Structural pattern matching in java. — Пост в блоге, URL: http://apocalisp.wordpress.com/2009/08/21/structural-pattern-matching-in-java/ (дата обращения: 20 декабря 2009 г.).

[113] *Palmer L.* graphics-drawingcombinators: A functional interface to 2d drawing in opengl. — Пакет на hackage, URL: http://hackage.haskell.org/package/graphics-drawingcombinators (дата обращения: 20 декабря 2009 г.).

[114] *Paulin-Mohring C.* Inductive definitions in the system coq — rules and properties // TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications. — London, UK: Springer-Verlag, 1993. — Pp. 328–345.

[115] *Paulson L. C.* Constructing recursion operators in intuitionistic type theory // — *Journal of Symbolic Computation*1986. — December. — Vol. 2, no. 4. — Pp. 325–355.

[116] Perfecthash: A perfect hashing library for mapping bytestrings to values. — URL: http://hackage.haskell.org/package/PerfectHash (дата обращения: 20 декабря 2009 г.).

[117] *Petříček T.* Reactive programming (i.) — first class events in f#. — Пост в блоге, URL: http://tomasp.net/blog/reactive-i-fsevents.aspx (дата обращения: 20 декабря 2009 г.).

[118] *Petříček T.* Reactive programming (ii.) — introducing reactive linq. — Пост в блоге, URL: http://tomasp.net/articles/reactive-ii-csevents.aspx (дата обращения: 20 декабря 2009 г.).

[119] *Peyton Jones S.* A pretty printer library in haskell. — Часть стандартной библиотеки GHC, URL: http://research.microsoft.com/en-us/um/people/simonpj/downloads/pretty-printer/pretty.html (дата обращения: 20 декабря 2009 г.).

[120] *Peyton-Jones S., Jones M., Meijer E.* Type classes: an exploration of the design space // Haskell workshop / Ed. by J. Launchbury. — Amsterdam: 1997.

[121] *Peyton Jones S. L.* The Implementation of Functional Programming Languages. — Prentice–Hall, 1987.

[122] *Pickering R.* Beyond Foundations of F# — Asynchronous Workflows. — Статья на infoq, URL: http://www.infoq.com/articles/pickering-fsharp-async (дата обращения: 20 декабря 2009 г.).

[123] *Pierce B. C.* Types and Programming Languages. — Cambridge, Massachusetts: The MIT Press, 2002.

[124] *Piponi D.* An approach to algorithm parallelisation. — — Пост в блоге, URL: http://blog.sigfpe.com/2008/11/approach-to-algorithm-parallelisation.html (дата обращения: 20 декабря 2009 г.)2008. — November.

[125] *Piponi D.* Моноиды в haskell и их использование (перевод Кирилла Заборского) // — *Практика функционального программирования*2009. — July. — Т. 1.

[126] *Podwysocki M.* F# first class events — composing events until others. — Пост в блоге, URL: http://codebetter.com/blogs/matthew.podwysocki/archive/2009/10/15/f-first-class-events-composing-events-until-others.aspx (дата обращения: 20 декабря 2009 г.).

[127] *Pope B.* hinvaders: Space invaders. — Пакет на hackage, URL: http://hackage.haskell.org/package/hinvaders (дата обращения: 20 декабря 2009 г.).

[128] *Pope R.* first-class-patterns: First class patterns and pattern matching, using type families. — Пакет на hackage, URL: http://hackage.haskell.org/package/first-class-patterns (дата обращения: 20 декабря 2009 г.).

[129] *Pope R.* Pattern combinators. — Серия блог-постов, URL: http://reinerp.wordpress.com/category/pattern-combinators/ (дата обращения: 20 декабря 2009 г.).

[130] Problem with delegates in c#. — Вопрос на форуме Stack Overflow, URL: http://stackoverflow.com/questions/1660483/problem-with-delegates-in-c (дата обращения: 20 декабря 2009 г.).

[131] *Reinhold M.* Project lambda: Straw-man proposal. — URL: http://cr.openjdk.java.net/~mr/lambda/straw-man/ (дата обращения: 20 декабря 2009 г.).

[132] *Ghani N., Hamana M., Uustalu T., Vene V.* Representing cyclic structures as nested datatypes.

[133] Research papers/type systems/type classes. — Страница в haskellwiki, URL: http://haskell.org/haskellwiki/Research_papers/Type_systems#Type_classes (дата обращения: 20 декабря 2009 г.).

[134] Revised(6) report on the algorithmic language scheme. — URL: http://www.r6rs.org/ (дата обращения: 20 декабря 2009 г.).

[135] *Rhiger M.* Type-safe pattern combinators // *J. Funct. Program.* — 2009. — Vol. 19, no. 2. — Pp. 145–156. http://dx.doi.org/10.1017/S0956796808007089.

[136] *Scholz E.* A Monad of Imperative Streams / Department of Computing Science, University of Glasgow. — Ullapool, Scotland1996. — July. http://www.dcs.gla.ac.uk/fp/workshops/fpw96/Scholz.ps.gz.

[137] *Scholz E.* Imperative streams —a monadic combinator library for synchronous programming // ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming. — New York, NY, USA: ACM, 1998. — Pp. 261–272.

[138] *Schönfinkel M.* Über die bausteine der mathematischen logik // *Math. Annalen.* — 1924. — Vol. 92. — Pp. 305–316. — An English translation appears in *From Frege to Godel*, edited by Jean van Heijenoort (Harvard Univ. Press, 1967), pages 355-366.

[139] *Schönfinkel M.* On the building blocks of mathematical logic. — 1967. — Pp. 355–366. — A Source Book in Mathematical Logic, 1879–1931.

[140] *Sittampalam G.* Darcs and gadts. — Видео и слайды в PDF, URL: http://www.londonhug.net/2008/02/02/video-darcs-and-gadts/ (дата обращения: 20 декабря 2009 г.).

[141] *Skeet J.* The beauty of closures. — Глава из книги «C# in depth», URL: http://csharpindepth.com/Articles/Chapter5/Closures.aspx (дата обращения: 20 декабря 2009 г.).

[142] *Smith L. P.* Why object-oriented languages need tail calls. — Ветка на форуме Lambda the Ultimate, URL: http://lambda-the-ultimate.org/node/3702 (дата обращения: 20 декабря 2009 г.).

[143] *Sørensen M. H., Glück R.* An algorithm of generalization in positive supercompilation // Proceedings of the International Symposium on Logic Programming / Ed. by J. Lloyd. — Cambridge: MIT Press1995. — December. — Pp. 465–479.

[144] *Steele G.* Why object-oriented languages need tail calls. — Пост в блоге Project Fortress, URL: http://projectfortress.sun.com/Projects/Community/blog/ObjectOrientedTailRecursion (дата обращения: 20 декабря 2009 г.).

[145] *Steele G., Sussman G. J.* Lambda papers. — Серия статей, доступны по адресу URL: http://library.readscheme.org/page1.html (дата обращения: 20 декабря 2009 г.).

[146] *Stokes R.* Learning j: Chapter 8. composing verbs. — URL: http://www.jsoftware.com/help/learning/08.htm (дата обращения: 20 декабря 2009 г.).

[147] *Strachey C.* Fundamental concepts in programming languages // *Higher Order Symbol. Comput.* — 2000. — Vol. 13, no. 1-2. — Pp. 11–49.

[148] Structural induction. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Structural_induction (дата обращения: 20 декабря 2009 г.).

[149] *Swierstra S. D., Chitil O.* Linear, bounded, functional pretty-printing // *J. Funct. Program.* — 2009. — Vol. 19, no. 1. — Pp. 1–16.

[150] *Syme D.* Introducing f# asynchronous workflows. — Пост в блоге, URL: http://blogs.msdn.com/dsyme/archive/2007/10/11/introducing-f-asynchronous-workflows.aspx (дата обращения: 20 декабря 2009 г.).

[151] *Syme D.* What's new in f# — asynchronous workflows. — Видео на Channel9, URL: http://channel9.msdn.com/posts/Charles/Don-Syme-Whats-new-in-F-Asynchronous-Workflows-and-welcome-to-the-NET-family (дата обращения: 20 декабря 2009 г.).

[152] *Syme D.* F# active patterns. — URL: http://blogs.msdn.com/dsyme/attachment/2044281.ashx (дата обращения: 20 декабря 2009 г.). — 2007.

[153] *Tanaka H., Muranushi T.* Monadius: 2-d arcade scroller. — Пакет на hackage, URL: http://hackage.haskell.org/package/Monadius (дата обращения: 20 декабря 2009 г.).

[154] `hackageDB :: [Package]`. — URL: http://hackage.haskell.org/packages/hackage.html (дата обращения: 20 декабря 2009 г.).

[155] *Tofte M., Talpin J.-P.* Region-based memory management // *Inf. Comput.* — 1997. — Vol. 132, no. 2. — Pp. 109–176.

[156] *Tullsen M.* First class patterns // *Lecture Notes in Computer Science*. — 2000. — Vol. 1753. — Pp. 1–?? http://link.springer-ny.com/link/service/series/0558/bibs/1753/17530001.htm;http://link.springer-ny.com/link/service/series/0558/papers/1753/17530001.pdf.

[157] *Turbak F. A., Gifford D. K.* Design Concepts in Programming Languages. — The MIT Press, 2008.

[158] Type checking with open type functions / T. Schrijvers, S. L. P. Jones, M. M. T. Chakravarty, M. Sulzmann // Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008 / Ed. by J. Hook, P. Thiemann. — ACM, 2008. — Pp. 51–62. http://doi.acm.org/10.1145/1411204.1411215.

[159] Type inference: Hindley-milner type inference algorithm. — Секция статьи в Wikipedia, URL: http://en.wikipedia.org/wiki/Type_inference#Hindley-Milner_type_inference_algorithm (дата обращения: 20 декабря 2009 г.).

[160] *van Straaten A.* Коан об объектах и замыканиях. — URL: http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html (дата обращения: 20 декабря 2009 г.).

[161] *Wadler P.* How to replace failure by a list of successes // Proc. of a conference on Functional programming languages and computer architecture. — New York, NY, USA: Springer-Verlag New York, Inc., 1985. — Pp. 113–128.

[162] *Wadler P.* Views: A way for pattern matching to cohabit with data abstraction // POPL. — 1987. — Pp. 307–313.

[163] *Wadler P.* A prettier printer // *Journal of Functional Programming*. — 1999. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.635.

[164] *Wadler P., Blott S.* How to make ad-hoc polymorphism less ad hoc // 16'th ACM Symposium on Principles of Programming Languages. — Austin, Texas1989. — January.

[165] *Wand M.* Continuation-based program transformation strategies // *Journal of the ACM*. — 1980. — Vol. 27. — Pp. 164–180.

[166] *Wasserman L.* Triemap: Automatic type inference of generalized tries. — Пакет на hackage, URL: http://hackage.haskell.org/package/TrieMap (дата обращения: 20 декабря 2009 г.).

[167] *Wehr S., Chakravarty M. M. T.* Ml modules and haskell type classes: A constructive comparison // APLAS / Ed. by G. Ramalingam. — Vol. 5356 of *Lecture Notes in Computer Science*. — Springer, 2008. — Pp. 188–204.

[168] Well-founded relation. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Well-founded_relation (дата обращения: 20 декабря 2009 г.).

[169] *Xi H., Chen C., Chen G.* Guarded recursive datatype constructors // *SIGPLAN Not.* — 2003. — Vol. 38, no. 1. — Pp. 224–235.

[170] *Yorgey B.* Typeclassopedia // — *The Monad.Reader*2009. — March. — Т. 13.

[171] Библиотека functional java. — URL: http://functionaljava.org (дата обращения: 20 декабря 2009 г.).

[172] Библиотека match.ss для plt scheme. — URL: http://download.plt-scheme.org/doc/372/html/mzlib/mzlib-Z-H-27.html (дата обращения: 20 декабря 2009 г.).

[173] Документация к классу treescanner. — URL: http://java.sun.com/javase/6/docs/jdk/api/javac/tree/com/sun/source/util/TreeScanner.html (дата обращения: 20 декабря 2009 г.).

[174] Документация к модулю data.map стандартной библиотеки языка haskell. — URL: http://www.haskell.org/ghc/docs/latest/html/libraries/containers/Data-Map.html (дата обращения: 20 декабря 2009 г.).

[175] Документация к модулю data.monoid. — URL: http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Monoid.html (дата обращения: 20 декабря 2009 г.).

[176] Документация к пакету struct.ss из библиотеки языка plt scheme. — URL: http://planet.plt-scheme.org/package-source/dherman/struct.plt/2/4/doc.txt (дата обращения: 20 декабря 2009 г.).

[177] *Кирпичёв Е.* antro: a line-level profiler for ant build scripts. — Проект на sourceforge, URL: sourceforge.net/projects/antro (дата обращения: 20 декабря 2009 г.).

[178] *Кирпичёв Е.* Coq. — Доклад на собрании SPb Haskell User Group, 24 октября 2008 г., URL: http://spbhug.folding-maps.org/wiki/Coq (дата обращения: 20 декабря 2009 г.).

[179] *Кирпичёв Е.* Чисто функциональные структуры данных (ч.2). — Доклад на собрании SPb Haskell User Group, 11 ноября 2007 г., URL: http://spbhug.folding-maps.org/wiki/FunctionalDataStructures2 (дата обращения: 20 декабря 2009 г.). — 2007.

[180] *Кирпичёв Е.* Приемы программирования на java: Повышение читаемости кода и функциональное программирование. — Доступно по адресу URL: http://www.rsdn.ru/article/java/JavaFP.xml (дата обращения: 20 декабря 2009 г.). — 2008.

[181] *Кирпичёв Е.* Функциональное программирование в java. — Слайды к докладу на математическо-механическом факультете СПбГУ, URL: http://spbhug.folding-maps.org/wiki/EugeneKirpichov?action=AttachFile&do=get&target=JavaFP_ru.ppt (дата обращения: 20 декабря 2009 г.). — 2008.

[182] *Кирпичёв Е.* Изменяемое состояние: опасности и борьба с ними // — *Практика функционального программирования*2009. — July. — Т. 1.

[183] *Кирпичёв Е.* Курс лекций «Функциональное программирование» для студентов АФТУ. — URL: http://spbhug.folding-maps.org/wiki/SICP_Course (дата обращения: 20 декабря 2009 г.). — 2009.

[184] Модуль coq.init.wf, реализующий вполне обоснованную индукцию. — URL: http://coq.inria.fr/stdlib/Coq.Init.Wf.html (дата обращения: 20 декабря 2009 г.).

[185] Страница в haskellwiki «existential type». — URL: http://www.haskell.org/haskellwiki/Existential_type (дата обращения: 20 декабря 2009 г.).

[186] Страница в haskellwiki «oop vs type classes». — URL: http://www.haskell.org/haskellwiki/OOP_vs_type_classes (дата обращения: 20 декабря 2009 г.).

[187] *Хювёнен, Э. и Сеппянен, Й.* Мир Лиспа. — 1990.

[188] *Царёв О.* Парсер заголовочных файлов Си на python. — Пост в блоге, URL: http://zabivator.livejournal.com/359491.html (дата обращения: 20 декабря 2009 г.).

Предметный указатель