Практика
функционального программирования

Выпуск 4
Февраль 2010

Данный вариант форматирования журнала предназначен для максимально компактной, экономной печати. В связи с этим, редакция просит с пониманием отнестись к отдельным недочётам форматирования таблиц, иллюстраций и листингов кода.

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

Журнал «Практика функционального программирования»

Авторы статей:	Алексей Отт Виталий Брагилевский Виталий Маяцких Денис Москвин Роман Душкин
Выпускающий редактор:	Дмитрий Астапов
Редактор:	Лев Валкин
Корректоры:	Алексей Махоткин, Виктор Целов, Ольга Боброва
Иллюстрации:	**Обложка** © Mattel, Inc.
Шрифты:	**Текст** Minion Pro © Adobe Systems Inc. **Обложка** Days © Александр Калачёв, Алексей Маслов Cuprum © Jovanny Lemonad
Ревизия:	2158 (2010-05-09)
Сайт журнала:	`http://fprog.ru/`

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

Журнал «Практика функционального программирования» распространяется в соответствии с условиями Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License.

Копирование и распространение приветствуется.

© 2010 «Практика функционального программирования»

Оглавление

От редактора 5

1. Лисп — абстракции на стероидах. Виталий Маяцких 6
- 1.1. Лисп — медленный? 7
- 1.2. Факты, факты... 10
- 1.3. Заключение 11

2. Clojure, или «Вы все еще используете Java? Тогда мы идем к вам!». Алексей Отт 13
- 2.1. Что такое Clojure? 14
- 2.2. Установка и запуск 15
- 2.3. Из чего состоит язык Clojure 15
- 2.4. Конкурентное программирование 25
- 2.5. Взаимодействие с Java 27
- 2.6. Поддержка языка 29
- 2.7. Заключение 31

3. Пределы выразительности свёрток. Виталий Брагилевский 33
- 3.1. Введение 34
- 3.2. Неуловимый dropWhile 34
- 3.3. Свёртка как комбинатор неподвижной точки 39
- 3.4. Свёртка как оператор примитивной рекурсии 41
- 3.5. Заключение 44

4. Мономорфизм, полиморфизм и экзистенциальные типы. Роман Душкин 45
- 4.1. Введение 46
- 4.2. Пара нетривиальных задач для статически типизированных языков 46
- 4.3. Экзистенциальные типы и их использование в языке Haskell 48
- 4.4. Заключение 50

5. Сечения композиции как инструмент бесточечного стиля. Денис Москвин 51
- 5.1. Введение 52
- 5.2. Бесточечный стиль: сечения и композиция 52
- 5.3. Левое сечение композиции 53
- 5.4. Правое сечение композиции 55
- 5.5. Конструкции, порождаемые обоими сечениями 55
- 5.6. Дополнительные техники 58
- 5.7. Заключение 58

© 2010 «Практика функционального программирования»

От редактора

О светлом будущем

Буквально накануне выхода этого номера компания Mattel сообщила, что очередной «профессией» куклы Барби будет инженер-компьютерщик. Коллектив редакции воспринял эту новость с большим энтузиазмом: во-первых, решился вопрос с подарками на 8-е марта. Во-вторых, поднимется привлекательность профессии для представительниц прекрасного пола. Длинноногие девушки-хакеры перестанут быть героями голливудских фильмов, а станут частью повседневной реальности.

Кроме того, обратите внимание, что на кофточке Барби изображен компьютер, на экране которого изображена Барби. А на экране ноутбука — двоичный код, который можно преобразовать в последовательность ASCII-символов «BarbieBarbieBarbieBar». То есть, в созданном образе легко «читаются» рекурсивные конструкции и бесконечные списки — понятия, традиционно ассоциируемые с функциональным программированием.

Мы думаем, что вывод напрашивается сам собой: если вы хотите иметь общие темы для разговора с девушками, которые через каких-то пять-семь лет будут писать приложения на OCaml и Erlang за соседним с вами столом, читайте наш журнал!

О конкурсе

До окончания конкурса осталось несколько дней, но уже сейчас можно огласить некоторые предварительные результаты.

Во-первых, серьезность заданий явно отпугнула многих. Невзирая на довольно длительный срок и денежные призы, мы получили меньше тридцати решений. Тем не менее, цель конкурса однозначно достигнута — мы получили программы на C#, Clojure, Common Lisp (SBCL), Ada, Python/C, Haskell, Erlang и даже на Visual Basic, что позволит нам провести межъязыковые сравнения и сделать определенные выводы, пусть даже и на такой ограниченной выборке.

Победители и результаты будут объявлены в пятом выпуске журнала.

О насущном

Спасибо за благодарности и критику, которую вы присылаете на адрес редакции. Ваша моральная поддержка очень важна для нас. Материальную же поддержку, как всегда, можно оказать на сайте журнала.[1]

Кроме того, если у вас есть желание и возможность написать для нас статью о практическом использовании инструментов функционального и декларативного программирования для решения повседневных задач — мы всегда будем рады получить от вас письмо на адрес редакции.[2]

Дмитрий Астапов, `adept@fprog.ru`

[1] http://fprog.ru/donate/

[2] mailto:editor@fprog.ru

Лисп — абстракции на стероидах

Виталий Маяцких

> I know God had six days to work, So he wrote it all in Lisp.
>
> Eternal Flame (song parody)

Аннотация

Существует расхожее мнение о невысокой скорости работы языка Common Lisp. В данной статье автор попытается развенчать этот миф.

Article tries to debunk the widespread myth of Common Lisp slowness, providing numerous illustrations to the contrary.

Обсуждение статьи ведётся по адресу
`http://community.livejournal.com/fprog/7453.html`.

1.1. Лисп — медленный?

Почему Лисп **должен** быть медленным? Для этого есть несколько вполне обоснованных причин:

1) Лисп — динамический язык, для которого, как известно, нельзя сгенерировать быстрый машинный код, эквивалентный коду на C. Динамичность языка требует существенных накладных расходов при исполнении программы, как то: необходимость постоянной проверки типов аргументов функций, преобразование данных из внутреннего представления в машинное и обратно (т. н. boxing/unboxing), автоматическое управление памятью и т. п.

2) Лисп, при всей простоте синтаксиса, является очень мощным языком, для которого в принципе написать эффективный компилятор весьма сложно. Макросы, функции высших порядков и обобщённые функции, система рестартов, интроспекция, метаобъектный протокол, доступный во время исполнения компилятор с возможностью на ходу переопределять код (в некоторых случаях и код самого компилятора!) — ну как тут разогнаться?

3) Маргинальность и малая распространённость Лиспа не привлекает крупных разработчиков ПО (например, Microsoft), которые могли бы реализовать хороший компилятор и постоянно его поддерживать. Любители из числа увлечённых студентов, научных работников и профессионалов высокого класса не могут на постоянной основе уделять достаточно времени для эффективного развития компилятора.

Практическим подтверждением этих постулатов является знаменитый «The Computer Language Benchmarks Game».[1] Действительно, если сравнить какую-либо реализацию Scheme[2] с GNU/gcc, то проигрыш Лиспа практически по всем пунктам составляет десятки, а то и сотни раз. Получается, что при всей своей разрекламированной элитарности, затрачивать усилия и время на изучения Лиспа не стоит, т. к. когда дело дойдёт до реального, «боевого» применения в более-менее критичном к ресурсам проекте, то в сто раз более медленная программа заказчика не устроит, даже если она будет представлять собой образец программистского искусства?

В действительности, дела обстоят не так плохо. Отложим пока вопрос распространённости языка, наличия популярных библиотек, а также размера и качества сообщества, сформировавшегося вокруг языка (адекватность «The Computer Language Benchmarks Game» тоже вызывает многочисленные сомнения).

1.1.1. Современные реализации

Лисп — очень старый язык. Появившись на заре вычислительной техники примерно в одно время со своим антагонистом — Фортраном, Лисп за прошедшие 50 лет не только выжил, но и стал значительно мощнее и гибче первых набросков его автора, Джона МакКарти. Например, Лисп обзавёлся чрезвычайно мощной, уникальной системой макросов, а также первой в мире официально стандартизованной объектной системой — CLOS. Особая структура программ на Лиспе и требования к среде времени исполнения привели даже к тому, что были разработаны специальные Лисп-машины,[3] аппаратно ускоряющие многие специфичные примитивы языка.

Однако, время Лисп-машин ушло, рост популярности языка Си сказался на современных машинных архитектурах, и многие интересные возможности оказались за бортом мейнстрима. В условиях довольно скудной, хотя и производительной аппаратной базы, написать компилятор, эффективно преобразующий программу с языка сверхвысокого уровня в примитивный машинный код, стало ещё сложнее. Тем не менее, появились коммерческие компиляторы Лиспа, такие, как Allegro[4] и LispWorks,[5] успешно существующие уже более двух десятков лет.

Академическая среда также не осталась в стороне: в университете Карнеги—Меллона в начале 80-х годов начал развиваться весьма удачный проект CMUCL.[6] Отличительной чертой CMUCL является наличие эффективного компилятора в машинный код для множества аппаратных платформ. Впоследствии на базе CMUCL были созданы коммерческий компилятор Scieneer Common Lisp[7] и открытый компилятор Steel Bank Common Lisp.[8]

Существует множество других реализаций, например, CLISP,[9] Clozure CL,[10] GCL,[11] ABCL,[12] не говоря о многочисленных реализациях более простого языка Scheme или совсем специфичных лисподобных языков.

Многие реализации либо занимаются трансляцией кода на Lisp в код на языке C с последующей сборкой в обычной инфраструктуре, либо используют компиляцию в байт-код для встроенной виртуальной машины. Оба подхода не оптимальны с точки зрения эффективности выполнения готового кода.

В первом случае промежуточным представлением является код на другом языке, механически сгенерированный по шаблонам, и поэтому выглядящий несколько инородно и избыточно. По такому коду компилятор Си, как правило, не может сгенерировать эффективный машинный код. К тому же, полностью теряется метаинформация о структуре лисповой программы после её предварительного разбора. Эту информацию можно было бы использовать на стадии генерации машинного кода, но до сишного компилятора она не доходит, и возможность оптимизации теряется.

Второй подход (виртуальная машина) плох тем, что интерпретация виртуального кода существенно медленнее, чем исполнение программы на «родном» для процессора языке. Конечно, существуют техники вроде JIT, когда байт-код предварительно компилируется в машинный код, но возникает та же проблема, что и в первом случае: цепочка «исходный код — машинный код» оказывается разорванной, и многие преобразования и оптимизации сделать оказывается уже невозможно.

Гораздо лучше эффективность у компиляторов, осуществляющих всю работу собственными силами. На текущий момент, вероятно, самым мощным из них является, как ни стран-

[1] http://shootout.alioth.debian.org/
[2] Распространённый диалект Лиспа (не путать с Common Lisp)

[3] http://ru.wikipedia.org/w/index.php?oldid=21545124
[4] http://www.franz.com/products/allegrocl/
[5] http://www.lispworks.com/
[6] http://www.cons.org/cmucl/
[7] http://www.scieneer.com/scl/
[8] http://www.sbcl.org/
[9] http://www.gnu.org/software/clisp/
[10] http://trac.clozure.com/ccl/
[11] http://www.gnu.org/software/gcl/
[12] http://common-lisp.net/project/armedbear/

но, свободный SBCL (к слову о силе свободного сообщества). Именно о нём и пойдёт в дальнейшем речь, как о наилучшем представителе семейства Common Lisp.

1.1.2. Steel Bank Common Lisp

Начав свой славный путь как попытку более логичной реорганизации исходного кода CMUCL, SBCL перехватил пальму первенства у своего родича в плане активности разработки компилятора. На данный момент Linux-порт SBCL поддерживает процессоры Intel x86 и x86-64, PowerPC, Sparc, Alpha и MIPS. Ведутся работы по переносу компилятора на архитектуру ARM. Существуют также порты для других операционных систем, таких как Mac OS X, *BSD и Windows, но с меньшей степенью поддержки. SBCL полностью соответствует стандарту ANSI Common Lisp, обладает одной из лучших реализаций метаобъектного протокола MOP (движущая сила объектной системы CLOS), широко поддерживается библиотеками, а также имеет обширное сообщество разработчиков и пользователей. Не последним по значимости фактором является весьма детальная поддержка техник оптимизаций для процессоров x86 и, особенно, для x86-64. Например, по умолчанию используются команды условной загрузки `CMOV`, косвенная адресация относительно точки исполнения (RIP-relative), а для операций с числами в формате single и double используется векторное расширение SSE2. На машинах с большим количеством регистров SBCL минимизирует операции с памятью, размещая в них часто используемую служебную информацию.

SBCL является двухпроходным компилятором. На первом этапе исходная программа трансформируется в форму ICR (Implicit Continuation Representation), которая представляет собой граф вычислений. В полученном графе проводятся все высокоуровневые преобразования: раскрытие макросов, частичное вычисление (вместо `(+ 2 2)` компилятор вставит сразу `4`), поиск и замена знакомых компилятору паттернов на более эффективные конструкции, устранение «мёртвого» кода. Важным шагом является вывод типов, что во многом и обуславливает превосходство SBCL в скорости над другими реализациями. Конечно, не везде в динамическом языке можно вывести типы. Да и там, где можно, компилятор иногда ошибается. Но факт налицо: SBCL автоматически выводит типы и оптимизирует код в соответствии с ними. Все стадии оптимизации выполняются в цикле до тех пор, пока между соседними итерациями не перестанут появляться изменения в графе (внутреннем представлении компилируемой программы).

На втором этапе подготовленный граф конвертируется в набор виртуальных операций VOP, которые представляют собой блоки кода, написанного на смеси Лиспа и ассемблера, встроенного в SBCL. Этот код выполняет какую-либо характерную операцию: например, преобразование числа из внутреннего представления в машинный, сложение двух чисел типа `complex` и т. п. Для каждого типа процессора в SBCL реализован свой набор VOP. Компилятору доступны свойства VOP, такие, как виртуальная «стоимость» исполнения данной операции, количество и типы аргументов, задействованные машинные регистры, условия исполнения и т. д. Основываясь на этих данных, компилятор комбинирует виртуальные операции, стараясь сгенерировать максимально быстрый код. На этом этапе происходит устранение лишнего копирования значений, оптимизация загрузки регистров, а также частичная реорганизация кода для оптимальной загрузки конвейеров процессора.

Финальной стадией является непосредственно генерация машинного кода и разрешение адресов переменных, точек перехода и т. д.

Как видно, в SBCL реализованы практически все оптимизации, доступные в других компиляторах. Всё это позволяет генерировать машинный код, приближающийся по эффективности к коду на C. Однако, для многих частных случаев в SBCL не описана трансформация, либо из-за ошибки в логике выбирается не самый оптимальный вариант. Поэтому для написания оптимальных по производительности лисповых программ очень желательно хотя бы уметь читать ассемблерный код процессора, для которого пишется программа. Стоит отметить, что это одинаково верно и для всех остальных языков, т. к. профайлер — другое доступное средство для понимания программы во время её исполнения, показывает только **как** работает программа, но не **почему**. Иной раз, гораздо быстрее дизассемблировать функцию и посмотреть, что именно сделал компилятор, чем гадать с перебором множества вариантов. Конечно, достаточно хорошее знание ассемблера в наше время — умение редкое, поэтому SBCL в процессе компиляции старательно выдаёт на экран сообщения, в которых предупреждает о неоптимально сгенерированном коде, с аргументацией, почему он так сделал. Устранив недостатки в исходном коде, можно получить значительно более компактный и быстрый машинный код.

В крайнем случае, когда на каком-то участке программы нужна максимальная производительность, имеет смысл воспользоваться интерфейсом для подключения внешних модулей, написанных, например, на языке Си. Такое подключение делается действительно очень просто при помощи библиотеки CFFI. Но обычно хватает и производительности лиспового кода, в чём мы ещё убедимся.

1.1.3. Как оптимизировать программу

Рассмотрим на очень простом коротком примере, на что способен SBCL. Определим функцию `sum` для сложения двух чисел:

```
(defun sum (a b)
  (+ a b))
```

Компилятор ничего не знает о типах аргументов, потому что функция в динамическом языке может получить на вход аргументы любого типа. Если с помощью стандартной лисповой функции **`disassemble`** посмотреть на результат, то сразу становится понятно, что быстро наше сложение работать не будет:

```
; disassembly for SUM
; 02D59DFD: 488B55F8       MOV RDX, [RBP-8] ; no-arg-parsing entry point
;      E01: 488B7DF0          MOV RDI, [RBP-16]
;     E05: 4C8D1C25E0010020 LEA R11, [\#x200001E0] ; GENERIC-+
;      E0D: 41FFD3            CALL R11
;      E10: 480F42E3          CMOVB RSP, RBX
;      E14: 488BE5            MOV RSP, RBP
;      E17: F8                CLC
;      E18: 5D                POP RBP
;      E19: C3                RET
```

В первых трех инструкциях компилятор передаёт аргументы в функцию `GENERIC-+`, которая для данных аргументов определяет наиболее подходящую специализированную функцию сложения. (Последние четыре инструкции — это

восстановление предыдущего фрейма локальных переменных, и непосредственно к коду логики функции не относятся).

Допустим, мы знаем, что складывать этой функцией придётся только целые значения. Сделаем соответствующую подсказку компилятору, объявив типы:

```
(defun sum (a b)
  (declare (type fixnum a b))
  (+ a b))
```

Здесь мы указываем компилятору типы переменных `a` и `b`. Для типа `FIXNUM` по стандарту компилятор должен использовать наиболее эффективную форму представления числа для данной аппаратной архитектуры. Для процессора это, естественно, будет обычное целое число, загружаемое непосредственно в регистр. Но в Лиспе программисту такой тип данных недоступен, потому что любая сущность в динамическом языке — это объект, который нужно уметь идентифицировать и обрабатывать соответствующим образом. Поэтому в SBCL для типа `FIXNUM` несколько младших битов машинного слова отдано под служебную информацию, и при машинных операциях над `FIXNUM` нужно эти служебные биты убирать и потом восстанавливать. Тем не менее, код этого варианта функции получился гораздо лучше:

```
; disassembly for SUM
; 02DC6B43: 488BC1   MOV RAX, RCX ; no-arg-parsing entry point
;       46: 48C1F803 SAR RAX, 3
;       4A: 488BD7   MOV RDX, RDI
;       4D: 48C1FA03 SAR RDX, 3
;       51: 4801D0   ADD RAX, RDX
...
```

Компилятор «откусил» у аргументов служебные биты и использовал непосредственно машинную операцию сложения. Правда, после сложения компилятор проверяет, вместилось ли получившееся число в тип `FIXNUM`, или нет. Если не вместилось, то из функции возвращается весьма медленное для последующей обработки число в формате `BIGNUM`[13]:

```
...
;       54:     486BD008         IMUL RDX, RAX, 8
;       58:     710E             JNO L0
;       5A:     488BD0           MOV RDX, RAX
;       5D:     4C8D1C250C060020 LEA R11, [#x2000060C]
;                                ALLOC-SIGNED-BIGNUM-IN-RDX
;       65:     41FFD3           CALL R11
;       68: L0: 488BE5           MOV RSP, RBP
;       6B:     F8               CLC
;       6C:     5D               POP RBP
;       6D:     C3               RET
```

На этом коротком примере становится понятно, что если от кода требуется максимальная скорость работы, то компилятору лучше подсказывать типы входных аргументов. На практике аргументы функции — это единственное место, где компилятор не может гарантированно вывести типы.

Естественно, не стоит забывать о правиле 20/80 (или даже 10/90), которое гласит, что 20% кода тратят 80% ресурсов процессора, и бросаться декларировать типы по всей программе. Как правило, это излишне, и даже может помешать компилятору выполнить некоторые эффективные преобразования.

[13]Формат представления не ограниченных по размеру чисел.

Существуют подходы, способные ещё больше улучшить машинный код. Например, если компилятор будет уверен, что при сложении двух целых чисел не произойдёт переполнения типа `FIXNUM`, то он не будет вставлять финальные проверки результата. Положим, что в функцию будут передаваться только целые числа от нуля до тысячи:

```
(defun sum (a b)
  (declare (type (integer 0 1000) a b))
  (+ a b))
```

SBCL сгенерировал более оптимальное сложение с использованием одной единственной команды `Load Effective Address` (данная техника описана в официальном документе по оптимизации кода от Intel):

```
; disassembly for SUM
; 02EB06DB: 488D143B LEA RDX, [RBX+RDI] ; no-arg-parsing entry point
;       DF: 488BE5   MOV RSP, RBP
;       E2: F8       CLC
;       E3: 5D       POP RBP
;       E4: C3       RET
```

Компилятор не только убрал проверку типа результата, но ещё и не стал преобразовывать числа из внутреннего формата в машинный. Трюк основан на знании компилятора о том, что для целых чисел в формате FIXNUM служебные биты будут равны нулю, поэтому операцией сложения эти биты не будут искажены.

Измерим время выполнения сложения чисел тремя вариантами функций. Один вызов функции исполняется ничтожно малое время, поэтому воспользуемся сложением в цикле:

```
(defun sum1 (a b)
  (+ a b))

(defun sum2 (a b)
  (declare (type fixnum a b))
  (+ a b))

(defun sum3 (a b)
  (declare (type (integer 0 1000) a b))
  (+ a b))

(let ((a 1)
      (b 2)
      (n 100000000))

  (time
   (dotimes (i n)
     (sum1 a b)))

  (time
   (dotimes (i n)
     (sum2 a b)))

  (time
   (dotimes (i n)
     (sum3 a b))))
```

```
Evaluation took:
  0.705 seconds of real time
  0.700894 seconds of total run time (0.700894 user, 0.000000 system)
  99.43% CPU
  1,547,423,284 processor cycles
  0 bytes consed
```

```
Evaluation took:
  0.592 seconds of real time
  0.590910 seconds of total run time (0.590910 user, 0.000000 system)
  99.83% CPU
  1,298,565,939 processor cycles
  0 bytes consed

Evaluation took:
  0.704 seconds of real time
  0.700893 seconds of total run time (0.695894 user, 0.004999 system)
  99.57% CPU
  1,543,727,900 processor cycles
  0 bytes consed
```

Казалось бы, результат получился странным, т. к. третья, наиболее оптимизированная функция сложения работала со скоростью первой, в которой вообще нет оптимизаций. Такое странное поведение объясняется тем, что компилятор в начале каждой функции по умолчанию вставляет блок проверки количества аргументов и соответствия их типов, если типы в функции были жёстко заданы. Соответственно, для первой функции проверяется только количество аргументов, но не проверяется их тип, т. к. обобщённой функции «+» будут переданы аргументы любых типов. Во второй функции для аргументов делается очень быстрая проверка на тип `FIXNUM` (это действительно самая быстрая проверка). Для третьей же функции проверяется диапазон значений аргументов, что обуславливает низкую суммарную скорость работы функции.

Так как было решено, что в функцию будут передаваться числа заведомо правильного типа, то можно отключить генерацию проверочного кода, а также включить оптимизацию по скорости. Для этого достаточно определить в самом начале программы декларацию вида:

```
(declaim (optimize (speed 3) (safety 0)))
```

Документацию по любой конструкции из стандартного языка можно получить на одном из многочисленных зеркал[14] HyperSpec.[15] Стоит особо отметить, что отключение абсолютно всех проверок безопасности в реальном проекте — очень плохая идея, но в данном случае позволяет продемонстрировать, на что способен SBCL. В крайнем случае, язык позволяет задать флаги оптимизации для одной единственной формы, где эта оптимизация наиболее необходима.

Запустим тест ещё раз:

```
Evaluation took:
  0.670 seconds of real time
  0.658900 seconds of total run time (0.658900 user, 0.000000 system)
  98.36% CPU
  1,470,513,473 processor cycles
  0 bytes consed

Evaluation took:
  0.535 seconds of real time
  0.524921 seconds of total run time (0.523921 user, 0.001000 system)
  98.13% CPU
  1,172,394,883 processor cycles
  0 bytes consed

Evaluation took:
  0.411 seconds of real time
  0.404938 seconds of total run time (0.404938 user, 0.000000 system)
  98.54% CPU
  901,260,173 processor cycles
  0 bytes consed
```

Теперь гораздо лучше: самая оптимизированная функция показала лучший результат. Кроме того, выбор другой политики генерации кода привёл к изменению машинного кода. Наиболее значимое изменение видно в третьем варианте функции, где компилятор использовал одну команду сложения.

```
; disassembly for SUM3
; 02DBC24F: 4801FA ADD RDX, RDI ; no-arg-parsing entry point
;       52: 488BE5 MOV RSP, RBP
;       55: F8     CLC
;       56: 5D     POP RBP
;       57: C3     RET
```

С точки зрения процессора команда `ADD` выполняется по скорости не хуже, чем `LEA`, но занимает в памяти на 1 байт меньше.

1.1.4. Лисп — не медленный

SBCL по качеству генерируемого кода приближается к коду, выдаваемому современными компиляторами Си. Некоторое отставание от них обусловлено тем, что SBCL пока не применяет технику Static Single Assignment, позволяющую устранять избыточное копирование данных и оптимизировать загрузку регистров. Вторым слабым местом является отсутствие поддержки т. н. peephole-оптимизации — архитектурно-зависимой перестановки инструкций для более интенсивной загрузки конвейеров процессора, замены нескольких инструкций на одну, более оптимальную, и др. Впрочем, даже gcc, самый распространённый свободный компилятор Си, научился всем этим хитростям относительно недавно, поэтому стоит ожидать дальнейшего подтягивания SBCL в плане оптимальности кода.

1.2. Факты, факты...

Не всё гладко в мире функционального программирования, даже если огородить его от нападок адептов Си и производных по синтаксису. Основной фронт противоборства лежит между подходами к типизации в языке. С одной стороны стоят поклонники Haskell, OCaml и других языков со статической типизацией, с другой — Lisp, Smalltalk, Python и прочая «динамика».

Естественной преградой для проведения оптимизаций в динамических средах является отсутствие информации о типах данных и функций на этапе компиляции: на вход произвольно взятой функции может прийти аргумент любого типа, и рантайму языка нужно проверять, допустим ли данный тип в рамках этой функции, решить, как его нужно обрабатывать. Например, сложение 32-битных целых отличается от сложения `BIGNUM`-чисел, хотя с точки зрения программиста функция для них применяется одна и та же — полиморфный оператор `+`.

Но в данный момент нас интересует, скорее, сравнительный материал, наработанный в этом вечном противостоянии: насколько динамически типизируемые языки проигрывают своим статическим собратьям.

[14] http://www.lispworks.com/documentation/HyperSpec/Body/d_optimi.htm#optimize
[15] HyperSpec — гипертекстовый вариант стандарта языка.

© 2010 «Практика функционального программирования»

1.2.1. Трассировка лучей

Консалтинговая компания «Flying Frog Consultancy», продвигающая решения на OCaml, провела сравнительное тестирование C++, Haskell, диалектов ML и Lisp. Воспользуемся результатами титанических трудов лучших программистов, и проведём соревнование между постепенно отмирающим, но известным своей хорошей скоростью C++, функциональным Haskell и мультипарадигменными OCaml и Common Lisp.

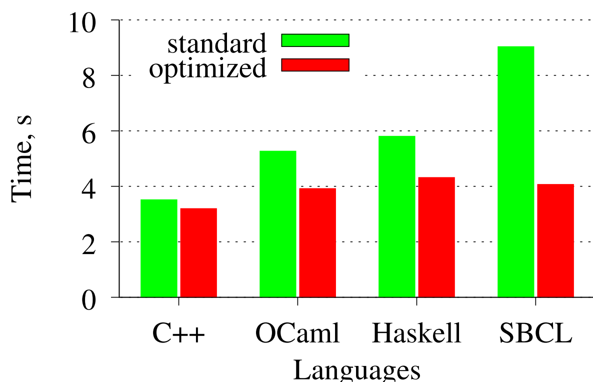


Рис. 1.1. Производительность языков на тесте трассировки лучей

График 1.1 показывает, что статические Haskell и OCaml работают всего лишь немногим медленнее, чем вариант на более низкоуровневом C++. Динамический SBCL тоже оказался не в сотни раз медленнее, как можно было бы ожидать. Оптимизированная же версия лиспового трейсера даже обогнала Haskell и вплотную приблизилась к OCaml.

Можно смело делать вывод, что для объёмных математических расчётов языки семейства ML являются неплохой альтернативой «плюсам», т. к. предоставляют программисту несравнимо больше возможностей, за которые не приходится платить чересчур высокую цену при выполнении кода.

По мнению автора, оптимизированный вариант трейсера на Лиспе потерял свою лаконичность и стал сложным для понимания. Не говоря уже о том, что такая оптимизация требует чрезмерных усилий даже хорошо подготовленного программиста. Впрочем, серебряных пуль в программировании отливать ещё не научились, и выбор правильного инструмента для решения задачи остаётся одним из самых главных принципов. В данном случае, писать максимально производительный рейтрейсер имеет смысл на другом языке.

1.2.2. В бой вступают профессионалы

Некоторое время назад к автору обратился его старый знакомый, чтобы автор помог ему со сравнительным тестированием языков. Это не такая грандиозная задача, как уже отмеченные «The Computer Language Benchmarks Game» или трассировка лучей, но интересна она тем, что «сложную» для машинного вычисления задачу сформулировал самый обычный программист, и все реализации делали самые обычные, за пределами своих компаний никому не известные программисты.

Итак, задача состоит в том, чтобы создать N потоков (threads), сгенерировать в каждом массив из тысячи случайных элементов типа Point (пары значений `x, y` типа double, 64-битное число с плавающей точкой в формате IEEE754), и в двух вложенных циклах рассчитать квадрат расстояния между точками. На псевдокоде программа описывается следующим образом:

```
for a from array.begin to array.end
    for b from array.begin to array.end
        dx = a.x - b.x
        dy = a.y - b.y
        result[next] = dx * dx + dy * dy
```

В тесте принимали участие программы на языках C, C# (Mono), Java, Common Lisp (SBCL) и Erlang. Для того, чтобы гарантированно загрузить двухъядерный процессор, в программах создавалось по 5 потоков. Исходные тексты программ можно найти на сайте журнала.[16]

Результаты тестирования приведены на графике 1.2.

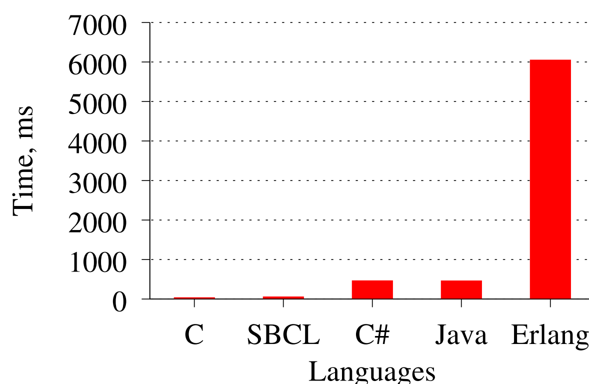


Рис. 1.2. Программистский супертест

Плохой результат у Erlang связан с не очень быстрым генератором псевдослучайных чисел. При отказе от случайных чисел обсчёт матриц происходит на порядок быстрее, но всё равно понятно, что Erlang для объёмных математических вычислений подходит плохо. Естественно, делать вывод о полной негодности Erlang во всех сферах не стоит, т. к. в том поле, для которого Erlang проектировался, его побить трудно. Да и далеко не все задачи в реальной жизни сводятся к массивной числовой обработке.

Посмотрим ещё раз на тот же график, на этот раз без Erlang (рис. 1.3). Перед тем, как делать окончательные выводы, хочется ещё раз отметить, что тесты разрабатывались обычными людьми, поэтому могут быть далеки от идеала. Но в целом, результаты получились показательными. Стоить отметить, что в SBCL для генерации псевдослучайных чисел используется весьма дорогой по тактам вихрь Мерсенна, поэтому выигрыш получился не за счёт более «лёгкого» генератора.

Видно, что языки C# и Java, популярные в софтверных компаниях, оказались в этом «тесте» на порядок медленее SBCL. Конечно, нельзя сказать, что SBCL будет всегда быстрее, чем, например, C#, но нельзя сказать и обратное.

1.3. Заключение

Надеемся, что миф о страшно медленно работающих программах, как родовом проклятии Лиспа, разрушен.

Учитывая весьма отличающийся подход к разработке программ в Лиспе — модификацию кода и данных «по-живому»,

[16] http://fprog.ru/2010/issue4/vitaly-mayatskikh-lisp-abstrakcii-na-steroidah/lisp-performance.zip

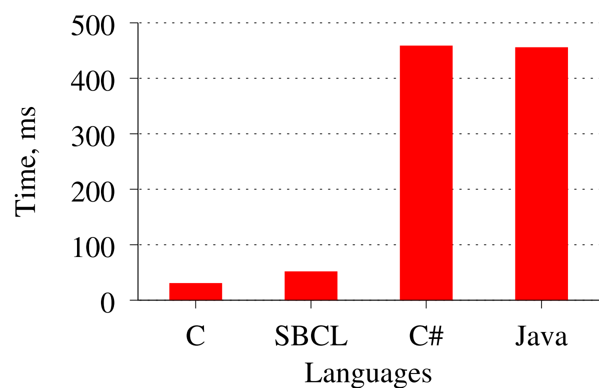


Рис. 1.3. Программистский супертест, без Erlang

без постоянной пересборки бинарного кода — время разработки значительно сокращается. В условиях жёсткой конкуренции на рынке разработки ПО это может принести немалые выгоды.

 © 2010 «Практика функционального программирования»

Clojure, или «Вы все еще используете Java? Тогда мы идем к вам!»

Алексей Отт
alexott@fprog.ru

Аннотация

В статье рассказывается о недавно появившемся функциональном языке Clojure, работающем на платформе JVM. Данный язык имеет интересные возможности, которые упрощают разработку многопоточных программ для платформы JVM и может быть интересен для программистов использующих данную платформу.[1]

This article provides a short introduction to the Clojure programming language. This language has very interesting features simplifying development of concurrent programs for JVM platform. We hope that this article is going to be useful for developers working with JVM.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/7703.html.

[1]В статье мы старались избежать требований к наличию знаний о семействе языков Lisp, но это не всегда получалось. Однако мы надеемся, что статья все-таки будет интересна для программистов на Java, которые смогут воспользоваться возможностями языка в части конкурентного программирования.

2.1. Что такое Clojure?

Clojure — Lisp'ообразный язык общего назначения, разработанный для Java Virtual Machine (JVM).[2] Автором языка является Rich Hickey,[3] который несколько лет разрабатывал язык в одиночку вплоть до выпуска первой публичной версии. В настоящее время, стабильной версией является версия 1.1,[4] выпущенная в конце декабря 2009-го года, и наш рассказ будет вестись именно о ней.

В отличие от других реализаций Lisp'а и Scheme для виртуальной машины Java, таких как ABCL, Kawa и т. д., Clojure не совместим на 100 процентов ни с Common Lisp, ни с Scheme, но позаимствовал многие идеи из этих языков, добавив новые вещи, такие как неизменяемость данных, конкурентное выполнение кода и т. п. Более подробно о том, зачем был создан новый язык, можно прочитать на сайте проекта.[5]

Несмотря на то, что Clojure — молодой язык программирования, достаточно много людей используют его в своих проектах, в том числе и коммерческих, например, FlightCaster,[6] который использует Clojure при обработке большого количества данных, решая задачи Machine Learning в распределенной среде. Существуют и другие фирмы (например, Sonian,[7] Runa,[8] Emendio[9]), использующие этот язык в своей работе — ссылки на них вы сможете найти на сайте языка.

2.1.1. Основные возможности языка

Clojure является функциональным языком программирования с поддержкой функций в качестве объектов первого класса (first class objects) и неизменяемыми (за исключением специальных случаев) данными, включая поддержку «ленивых» коллекций данных. От Lisp'а Clojure «унаследовал» макросы, мультиметоды и интерактивный стиль разработки, а JVM дает переносимость и доступ к большому набору библиотек, созданных для этой платформы.

Неизменность структур данных позволяет использовать их в разных потоках выполнения программы, что упрощает многопоточное программирование. Однако не все структуры являются неизменяемыми — в нужных случаях программист может явно использовать изменяемые структуры данных, используя Software Transactional Memory (STM), что обеспечивает надежную работу в многопоточной среде. (В качестве примера многопоточной программы, работающей с разделяемыми данными, можно привести программу «муравьи» (ants),[10] которую достаточно сложно написать на Java из-за большого количества моделируемых сущностей, но которая достаточно просто выглядит на Clojure).

За счет того, что Clojure был спроектирован для работы на базе JVM, обеспечивается доступ к большому набору библиотек, существующих для данной платформы. Взаимодействие с Java реализуется в обе стороны — как вызов кода, написанного на Java, так и реализация классов, которые доступны как для вызова из Java, так и из других языков, существующих для JVM, например, Scala. Подробнее о взаимодействии с JVM написано далее.

2.1.2. Отличия от Lisp

Несмотря на схожесть синтаксиса, Clojure отличается и от Common Lisp, и от Scheme. Некоторые отличия обусловлены тем, что язык разработан для платформы JVM, что накладывает некоторые ограничения на реализацию. Например, JVM не поддерживает оптимизацию хвостовых вызовов (tail call optimization, TCO), поэтому в язык были введены явные операторы `loop` и `recur`. Также важными определяющими факторами JVM-платформы являются:

- boxed integers — нет поддержки полного набора типов чисел (numeric tower), которые есть в Scheme и Common Lisp;
- система исключений как в Java (в Common Lisp используется сигнальный протокол);
- используется соглашение о вызовах как в Java.

Полный список отличий можно найти на отдельной странице[11] на сайте языка. Из явных отличий от Common Lisp можно отметить следующие:

- идентификаторы в Clojure регистрозависимы (case-sensitive);
- большая часть данных — неизменяемая;
- пользователь не может изменять синтаксис языка путем ввода собственных макросов в процедуре чтения кода (read macros);
- введен специальный синтаксис для литералов, векторов, отображений (maps), регулярных выражений, анонимных функций и т. д.;
- существует возможность связывания метаданных с переменными и функциями;
- можно реализовать функции с одним именем и разным набором аргументов;
- многие привычные вещи, такие как `let`, по синтаксису отличаются от их аналогов в Common Lisp и Scheme (при этом используется меньше скобок), например, `let` связывает данные последовательно, аналогично `let*` в Scheme;
- вместо функций `car` и `cdr` используются функции `first` и `rest`;
- `nil` не равен пустому списку или другому набору данных (коллекции) — он всего лишь означает отсутствующее значение (аналог `null` в Java);
- используется общее пространство имен, как в Scheme;
- сравнение на равенство производится одной функцией в отличие от Common Lisp и Scheme;
- поддержка «ленивых» коллекций.

[2] Также существует версия для платформы .Net, но в ней пока отсутствуют некоторые возможности, реализованные в Clojure для JVM.
[3] http://en.wikipedia.org/wiki/Rich_Hickey
[4] http://github.com/richhickey/clojure/blob/1.1.x/changes.txt
[5] http://clojure.org/rationale
[6] http://flightcaster.com
[7] http://sonian.com
[8] http://runa.com
[9] http://www.emendio.de/
[10] http://clojure.googlegroups.com/web/ants.clj
[11] http://clojure.org/lisps

© 2010 «Практика функционального программирования»

2.1.3. Источники информации о языке

Основной источник информации по данному языку — сайт проекта[12] и список рассылки.[13] Помимо сайта проекта, хорошим источником информации является набор видеолекций на Blip.TV,[14] а также видеолекции,[15] в которых автор языка рассказывает о Clojure и об особенностях его использования. Кроме того, следует отметить набор скринкастов,[16] созданных Sean Devlin, в которых он рассказывает о разных возможностях языка, включая новые, появившиеся в версии 1.1.

Из книг в настоящее время доступна книга «Programming Clojure» [2], выпущенная в серии Pragmatic Programmers, которая в принципе содержит всю необходимую информацию о языке, включая описание основных возможностей языка, вопросы взаимодействия с Java, основные функции, отличие языка от Common Lisp, и т. п. В мае 2010 года издательство Apress выпустит еще одну книгу по Clojure — «Practical Clojure» [4]. А на осень 2010 года в издательстве Manning запланирован выход книг «Clojure in Action» [3] и «Joy of Clojure. Thinking the Clojure Way» [1].

В свободном доступе можно найти книгу Clojure Programming,[17] работа над которой ведется в рамках проекта WikiBooks. Также существует достаточно подробный практический учебник — Clojure Scripting.[18]

Хорошее описание того, как можно использовать макросы для построения абстракций, можно найти в известной книге On Lisp[19] Пола Грэма (Paul Graham). Несмотря на то, что в ней используется Common Lisp, многие вещи будут применимы и для Clojure.[20]

Очень большое количество информации о языке, разрабатываемых библиотеках и проектах, использующих Clojure, публикуется в блогах. Для того, чтобы свести всю эту информацию воедино, существует проект Planet Clojure,[21] на который вы можете подписаться, чтобы быть в курсе новостей о языке.

2.2. Установка и запуск

Установка Clojure достаточно проста — скачайте последнюю версию с сайта языка[22] и распакуйте в нужный каталог. После этого вы можете запустить ее с помощью команды:

```
java -cp clojure.jar clojure.main
```

Эта команда приведет к запуску JVM и вы получите доступ к REPL («read-eval-print loop» — цикл ввода выражений и выдачи результатов). Стандартный REPL имеет не очень хорошие возможности по редактированию кода, так что при работе с REPL лучше использовать библиотеку `jline`, как описано в разделе Getting Started[23] официальной документации Clojure, или воспользоваться одной из сред разработки, описанных в разделе 2.6.1. Более подробные инструкции по развертыванию для разных сред разработки вы можете найти в описании проекта labrepl,[24] целью которого является упрощение начала работы с Clojure. В составе данного проекта имеется набор учебных материалов, которые будут полезны начинающим работать с языком.

Работая в REPL вы можете получать информацию о функциях, макросах и других объектах языка. Для получения информации о каком-либо символе или специальной форме вы можете использовать макрос `doc`. Например, `(doc map)` напечатает справку по функции `map`, которая была задана при объявлении этой функции. А если вы не помните точное название символа, можно провести поиск по документации с помощью функции `find-doc`, которая принимает один аргумент — строку с регулярным выражением по которому будет проводиться поиск.

2.3. Из чего состоит язык Clojure

Синтаксис языка Clojure следует стандартному для Lisp'образных языков подходу «код как данные», когда данные и код имеют общий синтаксис. Как и в других диалектах Lisp'а, код записывается в виде списков, используя префиксную нотацию и представляя собой синтаксическое дерево. Однако по сравнению с другими языками, в Clojure введены дополнительные сущности: кроме стандартных для Lisp'а символов, базовых литералов (строки, числа и т. п.) и списков, в язык введен дополнительный синтаксис для векторов, отображений (maps) и множеств (sets), являющихся объектами первого класса (first class objects).

Кроме этого, процедура чтения кода (reader) распознает специфические для Clojure конструкции: @ — для доступа к изменяемым данным и различные конструкции, начинающиеся с символа # — анонимные функции, метаданные (включая информацию о типах данных), регулярные выражения и т. д. Процедура чтения также рассматривает пробелы и запятые между элементами языка как один символ, разделяющий эти элементы.

2.3.1. Основные типы данных

Данные в Clojure можно разделить на две большие группы: базовые типы данных — числа, строки и т. д., и последовательности (коллекции), к которым относятся списки, векторы, отображения и множества. Пользователь может определять свои структуры данных с помощью `defstruct`, но они являются частным случаем отображений и введены для обеспечения более эффективной работы со сложными данными.

Все типы данных имеют общий набор характеристик: данные неизменяемы и реализуют операцию «равенство» (equality).

Базовые типы данных

К базовым типам данных Clojure относятся следующие:

логические значения: в языке определено два объекта для представления логических значений: true — для истинного значения и false — для ложного. (Все остальные значения, кроме `false` и `nil`, рассматриваются как истинные);

числа: в языке могут использоваться числа разных типов. По умолчанию для представления целых чисел используются классы, унаследованные от `java.lang.Number` —

[12] http://clojure.org/
[13] http://groups.google.com/group/clojure
[14] http://clojure.blip.tv/
[15] http://www.infoq.com/author/Rich-Hickey
[16] http://www.vimeo.com/channels/fulldisclojure
[17] http://en.wikibooks.org/wiki/Clojure_Programming
[18] http://pacific.mpi-cbg.de/wiki/index.php/Clojure_Scripting
[19] http://www.paulgraham.com/onlisp.html
[20] В интернете можно найти примеры кода из книг On Lisp и Practical Common Lisp, переписанные на Clojure.
[21] http://planet.clojure.in/
[22] http://clojure.org/
[23] http://clojure.org/getting_started
[24] http://github.com/relevance/labrepl

`Integer`, `BigInteger`, `BigDecimal`, но в Clojure реализуется специальный подход, который позволяет представлять число наиболее эффективным способом, автоматически преобразуя числа в случае необходимости — например, при переполнении числа. Если вы хотите для целого числа явно указать тип `BigDecimal`, то вы можете добавить букву `M` после значения.

Для чисел с плавающей точкой используется стандартный класс `Double`.

Кроме этих видов чисел, в Clojure определен специальный тип `Ratio`, представляющий числа в виде рациональных дробей, что позволяет избегать ошибок округления — например, при делении.

строки в Clojure являются экземплярами класса `java.lang.String` и к ним можно применять различные функции определенные в этом классе. Форма записи строк Clojure совпадает со стандартной записью строк в Java;

знаки (characters) являются экземплярами класса `java.lang.Character` и записываются либо в форме `\N`, где `N` — соответствующая буква, либо как названия для неотображаемых букв — например, как `\tab` и `\space` для символа табуляции и пробела и т. д.;

символы используются для ссылки на что-то — параметры функций, имена классов, глобальные переменные и т. д. Для представления символа как отдельного объекта, а не как значения, для которого он используется в качестве имени, используется стандартная запись `'symbol` (или специальная форма `quote`);

keywords (ключевые символы) — это специальные символы, имеющие значение самих себя[25] аналогично символам (symbols) в Lisp и Ruby. Одним из важных их свойств является очень быстрая операция проверки на равенство, поскольку происходит проверка на равенство указателей. Это свойство делает их очень удобными для использования в качестве ключей в отображениях (maps) и тому подобных вещах. Для именованных аргументов существует специальная форма записи `:keyword`.

Стоит также отметить, что символы и keywords имеют некоторую общность — в рамках интерфейса `IFn` для них создается функция `invoke()` с одним аргументом, что позволяет использовать символы и keywords в качестве функции. Например, конструкция `(:mykey my-hash-map)` или `('mysym my-hash-map)` аналогичны вызову `(get my-hash-map :mykey)` или `(get my-hash-map 'mysym)`, который приведет к извлечению значения с нужным ключом из соответствующего отображения.

В языке Clojure имеется специальное значение `nil`, которое может использоваться как значение любого типа данных, и совпадающее с null в Java. nil может использоваться в условных конструкциях наравне со значением false. Однако стоит отметить, что, в отличие от Lisp, nil и пустой список — () не являются взаимозаменяемыми и использование пустого списка в условной конструкции будет рассматриваться как значение true;

[25] Фактически, своего адреса в памяти.

Коллекции, последовательности и массивы

Кроме общих характеристик базовых типов перечисленных выше, все коллекции в Clojure имеют следующие характеристики:

- вся работа с коллекциями проводится через общий интерфейс;
- существует возможность связывания метаданных с коллекцией;
- для коллекций реализуются интерфейсы `java.lang.Iterable` и `java.util.Collection`, что позволяет работать с ними из Java;
- все коллекции рассматриваются как «последовательности» данных, вне зависимости от конкретного представления данных внутри них.

Неизменяемость коллекций означает, что результатом работы всех операций по модификации коллекций является другая, новая коллекция, в то время как исходная коллекция остается неизменной. В Clojure существует эффективный механизм, помогающий реализовывать неизменяемые коллекции. С его помощью операции, изменяющие коллекцию, могут эффективно создавать «измененную» версию данных, которая использует большую часть исходных данных, не создавая полной копии.

В текущей версии Clojure реализованы следующие основные виды коллекций:

списки (lists) записываются точно также как и в других реализациях Lisp. В Clojure списки напрямую реализуют интерфейс `ISeq`, что позволяет функциям работы с последовательностями эффективно работать с ними. (При использовании функции `conj` новые элементы списков добавляются в начало);

векторы (vectors) представляют собой последовательности, элементы которых индексируются целым числом (с последовательными значениями индекса в диапазоне `0..N`, где `N` — размер вектора). Для определения вектора необходимо заключить его элементы в квадратные скобки, например, [1 2 3]. Для преобразования других коллекций в вектор можно использовать функции `vector` или `vec`. Поскольку вектор индексируется целым числом, то операция доступа к произвольному элементу реализуется достаточно эффективно, что удобно при работе с некоторыми видами данных. (При использовании функции `conj` новые элементы векторов добавляются в конец.)

Кроме того, для вектора в Clojure создается функция одного аргумента (целого числа — индекса значения) с именем, совпадающим с именем символа, связанным с вектором. Это позволяет использовать имя вектора в качестве функции для доступа к нужному значению. Например, вызов (v 3) в данном коде:

```
user> (def v [1 2 3 4 5 "string"])
user> (v 3)
4
```

вернет значение четвертого элемента вектора.

© 2010 «Практика функционального программирования»

отображения (maps) — специальный вид последовательности, который отображает одни значения данных (ключ) в другие (значения). В Clojure существуют два вида отображений: `hash-map` и `sorted-map`, которые создаются с помощью соответствующих функций. `hash-map` обеспечивает более быстрый доступ к данным, а `sorted-map` хранит данные в отсортированном по ключу виде. Отображения записываются в виде набора значений (с четным количеством элементов), заключенных в фигурные скобки. Значения, стоящие на нечетных позициях рассматриваются как ключи, а на четных — как значения, связанные с данным ключом. В качестве ключа могут использоваться любые поддерживаемые Clojure типы данных, но очень часто в качестве ключей используют keywords, поскольку для них реализована очень быстрая проверка на равенство.

Также как и для векторов, для отображений создается функция одного аргумента (ключа), которая позволяет использовать имя символа, связанного с отображением, для доступа к элементам. Например,

```
user> (def m {:1 1 :abc 33 :2 "2" })
#'user/m
user> (m :abc)
33
```

множества (sets) представляет собой набор уникальных значений. Также как и для отображений, существует два вида множеств — `hash-set` и `sorted-set`. Определение множества имеет следующий вид #{elements ...} , а для создания множества из других коллекций может использоваться функция `set`, например, для получения множества уникальных значений вектора, можно использовать следующий код:

```
user> (set [1 2 3 2 1 2 3])
#{1 2 3}
```

В Clojure также определены дополнительные виды отображений, позволяющие в специальных случаях добиться большей производительности:

отображения-структуры (struct maps) могут использоваться для эмуляции записей (records), имеющихся в других языках программирования. В этом случае отображения имеют набор одинаковых ключей и Clojure реализует эффективное хранение информации о ключах, а также предоставляет быстрый доступ к элементам по ключу. В случае необходимости, имеется возможность генерации специализированной функции доступа с помощью функции `accessor`.

Определение отображения-структуры производится с помощью макроса `defstruct` или функции `create-struct`. Новые экземпляры отображений создаются с помощью функции `struct-map` или `struct`, которые получают список элементов для заполнения данного отображения. При этом стоит отметить, что отображение-структура может иметь большее количество ключей, чем было определено в `defstruct` — в этом отношении, отображения-структуры ведут себя точно также, как и обычные отображения.

отображения-массивы (array maps) — специальный вид отображений, в котором сохраняется порядок ключей. Такие отображения реализованы в виде обычного массива, содержащего ключи и значения. Поиск в отображении является линейной функцией от количества элементов, и поэтому, такие отображения должны использоваться только для хранения небольшого количества элементов. Новые отображения-массивы могут создаваться с помощью функции `array-map`.

Работа с коллекциями выполняется единообразно — для всех коллекций поддерживаются операции `count` для получения размера коллекции, `conj` для добавления элементов в коллекцию (реализуется по разному, в зависимости от конкретного типа) и `seq` для представления коллекции в виде последовательности — это позволяет применять к ним функции работы с последовательностями: `cons`, `first`, `map` и т. д. Функцию `seq` также можно использовать для преобразования в последовательности и коллекций Java.

Большая часть функций для работы с последовательностями[26] является «ленивой», обрабатывая данные по мере их надобности, что позволяет эффективно работать с данными большого размера, в том числе и с бесконечными последовательностями. Пользователь может создавать свои функции, которые возвращают «ленивые» последовательности, с помощью макроса `lazy-seq`. Также в версии 1.1 было введено понятие блоковых последовательностей (chunked sequence), которые позволяют создавать элементы блоками по N элементов, что в некоторых случаях позволяет улучшить производительность.

Из общего ряда выпадает работа с массивами Java, поскольку они не являются коллекциями в терминах Clojure. Для работы с массивами определен набор функций, которые позволяют определять массивы разных типов (`make-array`, `XXX-array`, где `XXX` — название типа), получения (`aget`) и установки (`aset`) значений в массиве, преобразования коллекций в массив (`into-array`) и т. д.

«Ленивые» структуры данных

Как уже упоминалось выше, большая часть структур данных в Clojure (и функций для работы с этими структурами данных) являются «ленивыми» (lazy). В языке имеется явная поддержка «ленивых» структур данных, позволяя программисту эффективно работать с ними. Одним из достоинств поддержки «ленивых» структур данных является то, что можно реализовывать очень большие или бесконечные последовательности используя конечное количество памяти. «Ленивые» последовательности и функции также могут использоваться для обхода ограничений, связанных с отсутствием оптимизации хвостовых вызовов в Clojure.

В Clojure программист может определить «ленивую» структуру данных воспользовавшись макросом `lazy-seq`. Данный макрос в качестве аргумента принимает набор выражений, которые возвращают структуру данных, реализующую интерфейс `ISeq`. Из этих выражений затем создается объект типа `Seqable`, который вызывается по мере необходимости, кешируя полученные результаты.

В качестве примера использования «ленивых» структур данных давайте рассмотрим создание бесконечной последовательности чисел Фибоначи:

[26] http://clojure.org/Sequences

```
(defn fibo
  ([] (concat [1 1] (fibo 1 1)))
  ([a b]
     (let [n (+ a b)]
       (lazy-seq (cons n (fibo b n))))))
```

В данном случае мы определяем функцию, которая при запуске без аргументов создает начальную последовательность из чисел 1 и 1 и затем вызывает сама себя, передавая эти числа в качестве аргументов. А функция двух аргументов обернута в вызов `lazy-seq`, который производит вычисление следующих чисел Фибоначи. При этом мы можем определить переменную, которая, например, будет содержать первые сто миллионов чисел Фибоначи:

```
user> (def many-fibs (take 100000000 (fibo)))
#'user/many-fibs
```

но поскольку мы работаем с «ленивыми» последовательностями, то значение будет создано мгновенно, без вычисления всех чисел Фибоначи. А само вычисление чисел будет происходить по мере надобности. Например, мы можем получить 55-е число Фибоначи с помощью следующего кода:

```
user> (nth many-fibs 55)
225851433717
```

Переходные структуры данных (transients)

При интенсивной работе с неизменяемыми коллекциями иногда возникает слишком много промежуточных объектов, что достаточно неэффективно. В версии 1.1 появилась возможность временно использовать изменяемые коллекции данных используя переходные (transient) структуры данных. Эта функциональность была специально введена в язык для оптимизации производительности.

Основная идея заключается в том, чтобы избежать ненужного копирования данных, что происходит когда вы работаете с неизменяемыми данными. Стоит отметить, что не все структуры данных поддерживают эту возможность — в версии 1.1.0 поддерживаются векторы, отображения и множества, а списки — нет, поскольку для них нет существенного выигрыша в производительности. В общем виде работа с переходными структурами данных происходит следующим образом:

- вы преобразуете стандартную структуру данных в переходную структуру (вектор, отображение и т. д.) с помощью функции `transient`, получающей один параметр — соответствующую структуру данных;
- выполняете изменение структуры по месту (inplace) с помощью специальных функций `assoc!`, `conj!` и т. п., которые аналогичны по действию соответствующим функциям, но без символа `!`, но применяются только к переходным структурам данных;
- после окончания обработки, превращаете переходную структуру данных в стандартную, неизменяемую структуру данных с помощью функции `persistent!`.

Рассмотрим простой пример использования стандартных и переходных структур данных[27]:

```
(defn vrange [n]
  (loop [i 0
         v []]
    (if (< i n)
      (recur (inc i) (conj v i))
      v)))

(defn vrange2 [n]
  (loop [i 0
         v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))))
```

Обе функции создают вектор заданного размера, состоящий из чисел в диапазоне `0..n`. В первой функции используются стандартные, неизменяемые структуры данных, а во второй — переходные структуры данных. Как видно из примера, во второй функции выполняются все требования к использованию переходных структур — сначала с помощью вызова `(transient [])` создается ссылка на вектор, который затем заполняется с помощью функции `conj!`, и в конце происходит возвращение неизменяемой структуры данных, созданной из переходной структуры с помощью вызова `(persistent! v)`. Если мы запустим обе функции с одинаковыми параметрами, то мы получим следующие результаты:

```
user> (time (def v (vrange 1000000)))
"Elapsed time: 439.037 msecs"
user> (time (def v2 (vrange2 1000000)))
"Elapsed time: 110.861 msecs"
```

Как видно из этого примера, использование переходных структур дает достаточно большой выигрыш в производительности — примерно в четыре раза. А на некоторых структурах данных выигрыш в производительности может быть больше. Копирование исходных данных и создание неизменяемой структуры — это операции со сложностью O(1), при этом происходит эффективное использование оригинальных данных. Также стоит отметить, что использование переходных структур данных приводит к принудительной изоляции потока выполнения — изменяемые данные становятся недоступными из других потоков выполнения.

Более подробно о переходных структурах данных вы можете прочитать на сайте языка.[28]

2.3.2. Базовые конструкции языка

Синтаксис языка в общем виде (за исключением специальных форм для отображений, векторов и других элементов) совпадает с синтаксисом Common Lisp и Scheme — все описывается S-выражениями.[29] Код записывается в виде списков, используя префиксную нотацию, когда первым элементом списка является функция, макрос или специальная форма,[31] а остальные элементы — аргументы, которые будут переданы в это выражение. Кроме списков, S-выражения могут состоять из атомов: чисел, строк, логических констант и т. д.

2.3.3. Объявление и связывание символов

В Lisp'е существуют объекты типа символ, которые в отличие от других языков представляют собой не просто имена пе-

[27] Примеры взяты из описания на сайте языка. Конструкции `loop` и `recur`, используемые в них, применяются для организации циклов и описаны далее.

[28] http://clojure.org/transients

[29] http://tinyurl.com/ye3bv63

[30] Специальные формы — это отдельные элементы языка, для которых не выполняются стандартные правила вычисления. Про специальные формы в Lisp вы можете прочитать в отдельной статье.[31]

© 2010 «Практика функционального программирования»

ременных, а отделенные сущности. То, что в других языках понимается как присваивание значения переменной (с определенным именем, которое по сути является просто меткой области памяти), в Lisp'е формулируется по другому — как связывание значения с символом, т. е. связывание двух объектов. Через эти же понятия формулируется и подход Lisp'а к типизации — значения имеют типы, а переменные — нет, поскольку переменная — это и есть пара символ-значение. Если вы используете имя символа без маскирования (специальная форма `quote`, или `'`), то вместо символа будет подставлено значение, связанное с этим символом. Если же вам нужно передать сам символ, то вы должны использовать запись `'имя-символа`.

Имеется два вида связывания символа со значением:

- *лексическое*, когда значение привязано к символу только внутри тела одной формы, например `let` или `fn`. Как раз тут и проявляется особенность Clojure — функциональные переменные: внутри тела `let` нельзя установить значение используя `set!`. Можно считать, что в этом случае `x` — это псевдоним для значения `2`.

- *динамическое*, когда значение привязывается к символу на время выполнения программы. Как правило, в этом случае символ делается доступным глобально (хотя в Common Lisp он может быть и локальным, но объявленным с помощью *declare special*). Суть слова *динамичность* в том, что в любой момент и в любом месте символ может быть заново связан с другим значением, и значение будет «видно» при всех обращениях к этому символу в любых формах. Такие динамические (в данном случае и глобальные в рамках пространства имен) символы определяются и связываются в Clojure формой `def`, на основе которой затем строятся макросы `defn`, `declare`, `defonce` и т. д. Таким образом, например, имя функции привязывается к самому объекту функции, что будет видно на примере раскрытия макроса `defn` ниже.

Таким образом, говорится о двух областях видимости: лексической и динамической. Нужно иметь в виду, что это не то же самое, что и локальные и глобальные переменные. Хотя динамически связанные символы, как правило, глобальны, а лексически связанные — локальны. А как быть в случае конфликта лексической и динамической привязки?

- В Common Lisp существует дуализм связывания, выраженный в специальной форме `let`, которая может связывать как лексические, так и специальные (динамические) символы. Преимущество этой формы в том, что при такой привязке динамический символ как бы приобретает свойство лексического: его новое значение видно только во время выполнения тела `let`, в то время как в других формах доступно его «глобальное» значение. Это очень мощная возможность, благодаря которой использование глобальных переменных в Lisp не приводит к тем разрушительным последствиям, которые присутствуют в других языках;

- В Clojure реализован тот же принцип за тем исключением, что разделены формы, выполняющие обычное лексическое связывание (для лексических символов, которые теперь смело можно назвать локальными) — `let`, и лексическое связывание для динамических символов — это форма `binding`. Подробней об этом можно прочитать в обсуждении «Let vs. Binding in Clojure[32]» на StackOverflow.com.

Общая форма объявления «динамического» символа выглядит как `(def имя значение?)`, при этом значение может не указываться и тогда символ будет несвязанным со значением. Существуют разные макросы для объявления символов, которые выполняют определенные задачи, например, `defonce` позволяет объявить символ только один раз, и если он уже имеет значение, то новое значение будет проигнорировано. А с помощью макроса `declare` можно выполнить опережающее определение (forward declaration) символа — это часто удобно при объявлении взаимно рекурсивных функций. При объявлении символа можно указать метаданные, связанные с данным символом (см. раздел 2.3.10).

Для объявления «лексических» символов используется форма `let`, которая выглядит следующим образом: `(let [имя1 знач1 имя2 знач2 ....] код)`. Например, выполнение следующего кода:

```
(let [x 1
      y 2]
  (+ x y))
```

выдаст значение `3`. В этом случае переменные `x` и `y` видны только внутри формы `(+ x y)`, и возможно маскируют значения переменных, объявленных на глобальном уровне, или выше по коду.

В некоторых случаях, программист может переопределить значение «динамического» символа для конкретного потока выполнения. Это делается с помощью макроса `binding`, который переопределяет значение указанного символа для данного потока выполнения и всех вызываемых из него функций. Например:

```
user> (def x 10)
user> (def y 20)
user> (defn test-fn []
        (+ x y))
user> (test-fn)
30
user> (binding [x 11
                y 22]
        (test-fn))
33
user> (let [x 11
            y 22]
        (test-fn))
30
```

В данном коде, если мы выполним `test-fn` на верхнем уровне кода, то получим значение `30`, равное сумме значений переменных `x` и `y`. А если эта функция будет вызвана из `binding`, то мы получим значение `33`, равное сумме переменных объявленных в `binding`. Данные значения изменяются только для текущего потока выполнения, и только для кода, который будет вызван из `binding`. После завершения выполнения кода в этой форме все предыдущие значения восстанавливаются. А при использовании `let` значения `x` и `y` не воздействуют на функцию `test-fn`, и в ней используются «глобальные» значения, давая в результате `30`. *Стоит быть осторожным при использовании функций работы с последовательностями внут-*

[32] http://stackoverflow.com/questions/1523240/let-vs-binding-in-clojure

ри `binding`, поскольку они возвращают «ленивые» последовательности, данные в которых будут вычисляться уже вне `binding`.

2.3.4. Управляющие конструкции

Для управления потоком выполнения программы в Clojure имеется некоторое количество специальных форм, на основе которых затем строятся остальные примитивы языка, управляющие выполнением программы.

Для организации последовательного выполнения нескольких выражений существует специальная форма `do`, которой передаются выражения, которые вычисляются последовательно, и результат вычисления последнего выражения возвращается как результат `do`. Использование `do` похоже на использование `let` без объявления переменных. `do` часто используется в ветках условных выражений, когда необходимо выполнить несколько выражений вместо одного.

Условные операторы

Для обработки простых условий используется конструкция `if`, которая является специальной формой. На основе `if` затем строятся остальные конструкции — `when`, `when-not`, `cond` и т. д. `if` выглядит стандартно для Lisp'образных языков: `(if условие t-выражение f-выражение?)`, т. е. если `условие` вычисляется в истинное значение, то выполняется выражение `t-выражение`, иначе — выражение `f-выражение` (оно может не указываться, что используется в макросе `when`). Результаты вычисления одного из этих выражений возвращаются в качестве результата `if`.

Макрос `cond` позволяет проверить сразу несколько условий. По своему синтаксису он отличается от `cond` в Common Lisp и Scheme, и в общем виде записывается как `(cond условие1 выражение1 условие2 выражение2 ... :else выражение-по-умолчанию)`. Заметьте, что дополнительных скобок вокруг пары `условиеN выражениеN` не требуется. Например:

```
(defn f1 [n]
  (cond
    (number? n) "number"
    (string? n) "string"
    :else "unknown"))
```

выражение `:else` не является ключевым словом языка и введено исключительно для удобства использования — вместо него можно использовать значение `true`.

Циклы и рекурсивные функции

Для организации циклов Clojure имеет несколько специальных форм, функций и макросов. Поскольку JVM имеет некоторые ограничения, не позволяющие реализовать оптимизацию хвостовых вызовов (Tail Call Optimization, TCO), то это накладывает ограничения на способ реализации некоторых алгоритмов, которые обычно реализуются через TCO в Scheme и других языках, поддерживающих эту оптимизацию.

Явные циклы организуются с помощью специальных форм `loop` и `recur`. Объявление `loop` похоже на `let`, но при этом имеется возможность повторного выполнения выражений, путем вызова `recur`[33] с тем же числом аргументов, которые были объявлены в списке переменных `loop` — обычно это новые значения цикла. Вот простой пример — реализация функции вычисления факториала с помощью `loop/recur` (здесь нет проверки на отрицательный аргумент):

```
(defn factorial[n]
  (loop [cnt n
         acc 1]
    (if (zero? cnt)
      acc
      (recur (dec cnt) (* acc cnt)))))
```

В данном случае объявляется цикл с двумя переменными `cnt` и `acc`, которые получают начальные значения `n` и `1`. Цикл прекращается, когда `cnt` будет равен нулю — в этом случае возвращается накопленное значение, хранящееся в переменной `acc`. Если `cnt` больше нуля, то цикл начинается снова, уменьшая значение `cnt`, и увеличивая значение `acc`.

В большинстве случаев явный цикл по элементам последовательности можно заменить на вызов функций `reduce`, `map`, `filter` или макроса раскрытия списков (list comprehension) `for`. Функция `reduce` реализует операцию «свертка» (fold), и используется при реализации многих функций, таких как функции `+`, `-`, `import` и т. д. Для примера с факториалом, код становится значительно проще, чем в предыдущем примере:

```
(defn fact-reduce [n]
  (reduce * (range 1 (inc n))))
```

Существует еще один метод оптимизации потребления стека при использовании взаимно рекурсивных функций — функция `trampoline`. Она получает в качестве аргумента функцию и аргументы для нее, и если переданная функция возвращает в качестве результата функцию, то возвращенная функция вызывается уже без аргументов. Вот пример определения четности числа, написанная для использования с `trampoline`:

```
(declare t-odd? t-even?)
(defn t-odd? [n]
  (if (= n 0)
    false
    #(t-even? (dec n))))
(defn t-even? [n]
  (if (= n 0)
    false
    #(t-odd? (dec n))))
```

Единственными отличиями от «стандартных» версий является то, что функции возвращают анонимные функции (строки 5 и 9). Если мы вызовем одну из этих функций с большим аргументом, например, вот так: `(trampoline t-even? 1000000)`, то вычисление произойдет без ошибки переполнения стека, в отличие от версии, которая не использует `trampoline`.

Стоит также отметить, что достаточно часто рекурсивные функции можно преобразовать в функции, производящие «ленивые» последовательности, как это было показано в разделе 2.3.1. Это положительно сказывается на производительности кода и потреблении памяти.

Исключения

Clojure поддерживает работу с исключениями (exceptions), которые часто используются в коде на Java. Специальная форма `throw` в качестве аргумента получает выражение, результат вычисления которого будет использован в качестве объекта-исключения.[34]

[33] Форма `recur` также может использоваться отдельно, без `loop` — тогда он выполнит переход к началу функции, в которой он используется.

[34] Результат должен быть объектом, унаследованным от `Throwable`.

Для выполнения выражений и перехвата исключений, которые могут возникнуть во время выполнения кода, используется специальная форма `try`. Форма `try` записывается следующим образом: `(try выражения* (catch имя-класса аргумент выражения*)* (finally выражения*)?)`. Блоки `catch` позволяют обрабатывать разные исключения в одном выражении `try`. А форма `finally` может использоваться для выражений, которые должны выполняться, и для случаев нормального завершения кода, и если произошел выброс исключения — например, для закрытия файлов и подобных этому задач. Если мы введем следующий код:

```
(try
  (/ 1 0)
  (println "not executed")
  (catch ArithmeticException ex
    (println (str "exception caught... " ex)))
  (finally (println "finally is called")))
```

то на экран будет выведено следующее:

```
exception caught... java.lang.ArithmeticException: Divide by zero
finally is called
```

т. е. мы перехватили исключение и вывели его на экран, а затем выполнили выражение, указанное в блоке `finally`. При этом выражения, стоящие после строки приводящей к ошибке — `(println "not executed")`, не выполняются.

2.3.5. Функции

Функции в общем случае создаются с помощью макроса `fn`. Для объявления функций на верхнем («глобальном») уровне пространства имен используются макросы `defn` или `defn-`, которые раскрываются в запись вида `(def имя-функции (fn ...))`. Второй макрос отличается от первого только тем, что функция будет видна только в текущем пространстве имен. Например, следующие объявления являются одинаковыми:

```
(def  func1 (fn [x y] (+ x y)))
(defn func2 [x y] (+ x y))
```

В общем виде объявление функции с помощью `fn` выглядит как `(fn имя? [аргументы*] условия? выражения+)`. Функция также может иметь разные наборы аргументов (разное число параметров) — тогда объявление будет выглядеть как `(fn имя? ([аргументы*] условия? выражения+)+)`. Например, объявление функции:

```
(defn func3
  ([x] "one argument")
  ([x y] "two arguments")
  ([x y z] "three arguments"))
```

позволяет вызывать ее с одним, двумя или тремя аргументами.[35] Программист также может определить функцию, имеющую переменное число параметров, если укажет знак амперсанд (`&`) перед аргументом, в который будут помещены оставшиеся параметры. Например, следующий код:

```
user> (defn func4 [x y & z] z)
user> (func4 1 2)
nil
user> (func4 1 2 3 4)
(3 4)
```

объявит функцию `func4`, имеющую два обязательных параметра — `x` и `y`, а остальные параметры будут помещены в список, который будет передан как аргумент `z`.

Для объявления небольших анонимных функций используется специальная запись `#(выражения+)`, а доступ к аргументам производится с помощью специальных переменных `%1`, `%2` и т. д., или просто `%`, если функция принимает один параметр. Например, следующие выражения эквивалентны:

```
(#(+ %1 %2) 10 20)
((fn [x y] (+ x y)) 10 20)
```

оба этих выражения возвращают одно и то же число. Специальная запись удобна, когда вам надо передать функцию в качестве аргумента, например, для функции `map` или `filter`.

Начиная с версии 1.1, при объявлении функции можно указывать пред- и постусловия, которые будут применяться к аргументам и результату. Эта функциональность реализует концепцию «контрактного программирования».[36] Пред- и постусловия задаются как метаданные `:pre` и `:post`, которые указываются после списка аргументов. Каждое из условий состоит из вектора анонимных функций, которые должны вернуть `false` в случае ошибочных данных. Например, рассмотрим следующую функцию:

```
(defn constrained-sqr [x]
  {:pre  [(pos? x)]
   :post [(> % 16), (< % 225)]}
  (* x x))
```

Данная функция принимает в качестве аргументов только положительные числа — условие `(pos? x)`, в диапазоне `5..14` — условие `(> % 16), (< % 225)`, иначе будет выдана ошибка проверки аргументов или результата.

В Clojure имеется набор функций, которые позволяют создавать новые функции на основе существующих. Функция `partial` используется для создания функций с меньшим количеством аргументов путем подстановки части параметров (каррирование), а функция `comp` создает новую функцию из нескольких функций (композиция функций):

```
user> (defn sum [x y] (+ x y))
user> (def add-5 (partial sum 5))
user> (add-5 10)
15
user> (def my-second (comp first rest))
user> (my-second [1 2 3 4 5])
2
```

В первом примере мы создаем функцию, которая будет прибавлять число 5 к переданному ей аргументу. А во втором примере, мы создаем функцию, эмулирующую функцию `second`, которая сначала применяет функцию `rest` к переданным ей аргументам, а затем применяет к результатам функцию `first`. Хороший пример использования функций `comp` и `partial` можно увидеть в скринкасте Point Free Clojure.[37]

2.3.6. Макросы

Макросы — это мощное средство уменьшения сложности кода, позволяющие строить проблемно-ориентированную среду на основе базового языка. Макросы активно используются в Clojure, и множество конструкций, составляющих

[35] Внутри Clojure функции представляются как классы, реализующие интерфейс `IFn`, с функцией `invoke`, получающей нужное количество параметров.

[36] http://tinyurl.com/kk9zqq

[37] http://vimeo.com/8665159

язык, определены как макросы на основе ограниченного количества специальных форм и функций, реализованных в ядре языка.

Макросы в Clojure смоделированы по образцу макросов в Common Lisp, и являются функциями, которые выполняются во время компиляции кода. В результате выполнения этих функций должен получиться код, который будет подставлен на место вызова макроса. Основная разница заключается в синтаксисе. В общем виде определение макроса выглядит следующим образом:

```
(defmacro name doc-string? attr-map? ([params*] body)+)
```

описание макроса (документация) — `doc-string?` и список атрибутов — `attr-map?` являются не обязательными, а список параметров и тело макроса могут указываться несколько раз, что позволяет определять макросы с переменным числом аргументов также, как и при объявлении функций (см. пример ниже).

Тело макроса должно представлять собой список выражений языка, результат выполнения которых будет подставлен на место использования макроса в виде списка Clojure, содержащего набор операций. Этот список может быть сформирован с помощью операций над списками — этот подход используется в простых случаях, так что вы можете сформировать тело макроса, используя операции `list`, `cons` и т. д. Хорошим примером этого подхода является макрос `when`, показанный ниже.

Другим подходом является маскирование (quote) всего выражения, с раскрытием только нужных частей кода. Для этого используется специальный синтаксис записи ‘ (обратная кавычка), внутри которого можно использовать ~ (тильда) для подстановки значений (аналогично операции `,` (запятая) в Common Lisp). Для подстановки списка не целиком, а поэлементно, используется синтаксис: ~@. Хорошим примером второго подхода является макрос `and`, приведенный далее.

При работе с макросами очень полезными являются функции `macroexpand-1` и `macroexpand`, которые производят раскрытие заданного макроса, что позволяет программисту проверять корректность кода, используемого в макросах. Отличие между этими функциями заключается в том, что первая функция раскрывает макрос один раз, выполняя подстановки, но возможно возвращая код, который использует другие макросы. В то время как функция `macroexpand` — раскрывает макрос рекурсивно, раскрывая все использованные макросы.

Рассмотрим подстановки имен и значений более детально. Допустим, у нас есть две переменные — `x` со значением `2`, и `y`, представляющая собой список из трех элементов — `(4 5 6)`. Если мы попытаемся раскрыть разные выражения, то мы будем получать разные результаты:

```
user> (def x 2)
user> (def y '(4 5 6))
user> `(list 1 x 3)
(list 1 user/x 3)
user> `(list 1 ~x 3)
(list 1 2 3)
user> `(list 1 ~y 3)
(list 1 (4 5 6) 3)
user> `(list 1 ~@y 3)
(list 1 4 5 6 3)
```

В первом случае мы не выполняем никакой подстановки, поэтому `x` подставляется как символ. Во втором случае мы раскрываем выражение, подставляя значение символа и получая выражение `(list 1 2 3)`. В третьем случае у нас подставляется значение символа `y` в виде списка, в отличие от четвертого выражения, когда значение списка поэлементно подставляется (spliced) в выражение в раскрытом виде.

Примеры макросов

В качестве простого примера рассмотрим макрос `when`, определенный в базовой библиотеке:

```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an
      implicit do."
  [test & body]
  (list 'if test (cons 'do body)))
```

Данный макрос принимает один обязательный аргумент — условие `test`, а остальные аргументы рассматриваются как набор выражений, которые будут выполнены, если условие вернет истинное значение. Для того, чтобы можно было указать несколько выражений в качестве тела макроса, они обертываются в конструкцию `do`. Если мы воспользуемся `macroexpand` для раскрытия макроса, то для конструкции вида:

```
(when (pos? a)
   (println "positive") (/ b a))
```

мы получим следующий код:

```
(if (pos? a)
  (do
    (println "positive")
    (/ b a)))
```

`when` — это достаточно простой макрос. Более сложные макросы могут создавать переменные, иметь разное количество аргументов, и т. д. Например, макрос `and`, определенный следующим образом:

```
(defmacro and
  "Evaluates exprs one at a time, from left to right.
  If a form returns logical false (nil or false), and
  returns that value and doesn't evaluate any of the
  other expressions, otherwise it returns the value
  of the last expr. (and) returns true."
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

имеет три разных раскрытия для разного количества аргументов, при этом, если макросу передано больше одного аргумента, то он рекурсивно вызывает сам себя. Мы можем увидеть это, раскрыв макрос для разных наборов аргументов:

```
user> (macroexpand '(and ))
true
user> (macroexpand '(and (= 1 2)))
(= 1 2)
user> (macroexpand '(and (= 1 2) (= 3 3)))
(let* [and__4457__auto__ (= 1 2)] (if and__4457__auto__
    (clojure.core/and (= 3 3)) and__4457__auto__))
```

В этом примере вызов макроса без параметров приводит к подстановке значения `true`. При использовании одного параметра-выражения подставляется само выражение. А если параметров больше одного, то формируется форма `let`, в

 © 2010 «Практика функционального программирования»

которой вычисляется первое выражение и связывается с переменной с уникальным именем (сгенерированным автоматически), а затем проверяется значение этой переменной. Если это значение истинное, то макрос вызывается еще раз, получая в качестве аргументов список параметров без первого элемента. А в том случае, если выражение не истинное, возвращается результат вычисления.

В макросе `and` для избежания конфликтов с кодом пользователя используется генерация уникальных имен переменных. Для этого используется специальный синтаксис prefix #, который создает уникальное имя, начинающееся с заданного префикса (в нашем случае имя начинается с `and`).

2.3.7. Шаблоны

В версии 1.1 была введена поддержка шаблонов (templates), которые могут использоваться с следующих случаях:

- когда нужны простые текстовые подстановки;
- когда подстановки нужны только в конкретном пространстве имен;
- когда нет возможности использовать функции высшего порядка для выполнения данной задачи.

Рассмотрим как это работает. Пространство имен `clojure.template` определяет два макроса: `apply-template` — предназначен для использования в других макросах, и `do-template` — для использования в обычном коде. Оба этих макроса имеют общую форму: `(do-template argv expr & values)`, где первым параметром указывается вектор параметров `argv`, которые будут подставляться в выражение `expr`, а в заключение идет список значений `values`, которые будут подставлены в выражение вместо соответствующих параметров. Необходимо помнить, что длина `values` должна быть кратной длине вектора `argv`, иначе остающиеся значения будут просто проигнорированы. Рассмотрим пример с генерацией кода для тестов, взятый из скринкаста про работу с шаблонами[38]

```
(do-template [input result]
             (is (= (first input) result))
             [:a :b :c] :a
             "abc" \a
             (quote (:a :b :c)) :a)
```

Это выражение будет раскрыто в следующий код (вы можете проверить это с помощью `macroexpand-1`):

```
(do
  (is (= (first [:a :b :c]) :a))
  (is (= (first "abc") \a))
  (is (= (first '(:a :b :c)) :a)))
```

В данном случае у нас имеются следующие объекты: `argv` имеет значение `[input result]`, `expr` равен `(is (= (first input) result)`, а `values` — `[:a :b :c] :a ....`. При раскрытии происходит следующее: берутся первые `N` значений из списка `values` (`N` — длина `args`), и подставляются на места соответствующих параметров в выражении `expr`, затем берутся следующие `N` значений, и т. д., до тех пор, пока список значений не будет исчерпан.

[38] http://vimeo.com/8360422

2.3.8. Мультиметоды

Так же как и Common Lisp, Clojure поддерживает использование мультиметодов, которые позволяют организовать диспетчеризацию вызовов функций в зависимости от аргументов. Синтаксис мультиметодов немного отличается, и вместо `defgeneric` используется макрос `defmulti`, а в остальном принцип работы схож с CLOS.[39]

Объявление функции, которая будет вести себя по разному в зависимости от аргументов, производится с помощью макроса `defmulti`. Данный макрос получает в качестве аргументов имя объявляемой функции, функцию диспетчеризации (которая должна вернуть значение, служащее ключом диспетчеризации) и список опций, определяющих способ диспетчеризации.

После объявления функции, пользователь может добавлять реализации с помощью макроса `defmethod`, который получает в качестве аргументов имя функции, значение по которому будет производиться диспетчеризация (часто это имя класса), список аргументов и тело функции. Например, если мы объявим следующую функцию, которая выполняет диспетчеризацию по типу переданного значения (с помощью функции `class`):

```
(defmulti foo class)
(defmethod foo java.util.Collection [c] :a-collection)
(defmethod foo String [s] :a-string)
(defmethod foo Object [u] :a-unknown)
```

и попробуем применить ее к разным аргументам, то мы получим следующие результаты:

```
user> (foo [])
:a-collection
user> (foo #{:a 1})
:a-collection
user> (foo "str")
:a-string
user> (foo 1)
:a-unknown
```

Но этот пример является достаточно простым и похож на стандартную диспетчеризацию в ОО-языках. Clojure предоставляет возможность диспетчеризации вызова в зависимости от значения аргументов, а также других признаков. Например, мы можем определить мультиметод с собственной функцией диспетчеризации, которая вызывает разные функции в зависимости от переданного значения:

```
(defn my-bar-fn [n]
  (cond
   (not (number? n)) :not-number
   (= n 2) :number-2
   (>= n 5) :number-5-ge
   :else :number-5-lt))
(defmulti bar my-bar-fn)
(defmethod bar :not-number [n] "not a number")
(defmethod bar :number-2 [n] "number is 2")
(defmethod bar :number-5-ge [n] "number is 5 or greater")
(defmethod bar :number-5-lt [n] "number is less than 5")
```

и, вызывая этот мультиметод, мы получим соответствующие значения:

```
user> (bar 2)
"number is 2"
```

[39] http://en.wikipedia.org/wiki/Common_Lisp_Object_System

```
user> (bar 5)
"number is 5 or greater"
user> (bar -1)
"number is less than 5"
user> (bar "string")
"not a number"
```

В Clojure имеется набор функций для работы с иерархиями классов: получения информации об отношениях между классами — `parents`, `ancestors`, `descendants`; проверки принадлежности одного класса к иерархии классов — `isa?` и т. д. Программист также может создавать свои иерархии классов, используя функцию `make-hierarchy`, и определять отношения между классами с помощью функции `derive`. Например, следующий код:

```
(derive java.util.Map ::collection)
(derive java.util.Collection ::collection)
```

устанавливает `::collection` в качестве родителя классов `java.util.Map` и `java.util.Collection`, что позволяет изменять существующие иерархии классов.[40]

В том случае, если имеется перекрытие аргументов, и Clojure не может выбрать соответствующую функцию, то программист может выбрать наиболее подходящий метод с помощью функции `prefer-method`. Другие примеры и дополнительную информацию о мультиметодах и иерархиях классов вы можете найти на сайте.[41]

2.3.9. Пространства имен и библиотеки

Пространства имен (namespaces) используются в Clojure для организации кода и данных. По своему характеру, пространства имен аналогичны пакетам (packages) в Common Lisp, и наиболее часто они используются при создании библиотек кода.[42] Пространства имен являются объектами первого класса (first class objects), и могут динамически изменяться — создаваться, удаляться, изменяться, их можно перечислять и т. д. Пользователь может управлять видимостью символов используя метаданные, или специальные макросы, такие как `defn-`, который определяет функцию, видимую только в текущем пространстве имен.

При работе в REPL, все символы определяемые пользователем помещаются в пространство имен `user`. Пользователь может переключиться в другое пространство имен с помощью функции `in-ns` и/или подключить символы из других пространств имен с помощью функций `use`, `require` и `import`. Имя текущего пространства имен можно всегда найти в специальной переменной `*ns*`, которая автоматически устанавливается макросом `ns` и функцией `in-ns`.

Наиболее часто используемыми функциями при работе с пространствами имен являются:

`use` — помещает в текущее пространство имен символы (все, или только указанные) из другого пространства имен, в том числе и находящихся в других библиотеках, загружая их при необходимости;

`require` — загружает заданные библиотеки, но не помещает символы, определенные в них, в текущее пространство имен;

[40]Стоит отметить, что можно добавлять только родительские объекты, а создание потомков возможно только через стандартные механизмы создания классов с помощью `gen-class`.

[41]http://clojure.org/Multimethods

[42]http://clojure.org/libs

`import` — используется для библиотек JVM и импортирует заданные классы из указанного пакета.

Каждая из этих функций имеет разное количество параметров, описание которых можно найти в документации. В качестве пример давайте рассмотрим следующий код:

```
(use 'clojure.contrib.str-utils)
(require 'clojure.contrib.lazy-xml)
(require '[clojure.contrib.str-utils2 :as str2])
(import 'org.apache.commons.io.FileUtils)
(import '(java.io File InputStream))
```

Первая строка загружает библиотеку `clojure.contrib.str-utils` и помещает все определенные в ней символы в текущее пространство имен. Вторая строка загружает библиотеку `clojure.contrib.lazy-xml`, но для доступа к ее объектам, необходимо использовать полное имя символа, включающее название пространства имен. Третья строка также загружает библиотеку, но создает псевдоним для названия пространства имен, что позволяет использовать более короткое имя символа, например, `str2/butlast`. Четвертый пример импортирует один класс (`FileUtils`) из пакета `org.apache.commons.io`, а в пятой строке мы видим как можно импортировать несколько классов из одного пакета.

При написании кода, лучше всего определять пространство имен с помощью макроса `ns`, который выполняет всю работу по созданию пространства имен, а также позволяет указать список импортируемых классов (используя `import`), используемых пространств имен (используя `use`), и т. п. операции, включая генерацию новых классов, с помощью `get-class`. В общей форме, использование макроса `ns` выглядит следующим образом:

```
(ns name
  (:require [my.cool.lib :as mcl])
  (:use my.lib2)
  (:import (java-package Class))
  .... more options)
```

Данный код определяет пространство имен `name`, импортирует в него класс `Class` из пакета `java-package`, импортирует библиотеку `my.lib2` и определяет псевдоним `mcl` для библиотеки `my.cool.lib`. Опции, указываемые в макросе `ns`, совпадают с опциями соответствующих функций. Более подробное описание вы можете найти в документации.

Описание дополнительных операций, которые можно производить с пространствами имен, вы можете найти в официальной документации.[43]

2.3.10. Метаданные

Одним из интересных свойств Clojure является возможность связывания произвольных метаданных с символами, определенными в коде. Некоторые функции и макросы позволяют указывать определенные метаданные для управления видимостью символов, указания типов данных, документации и т. п. Стоит отметить, что наличие метаданных никак не влияет на значения, связанные с символом. Например, если мы имеем два отображения, с одинаковым содержимым, но разными метаданными, то эти отображения будут эквивалентны между собой.

[43]http://clojure.org/Namespaces

Для указания метаданных используется специальный синтаксис, который распознается функцией чтения кода. Эта функция переходит в режим чтения метаданных, если она встречает строку #^. После этой строки может быть указано либо название типа, например, #^Integer, либо отображение, перечисляющее ключ метаданных и значение, связанное с данным ключом. Стоит отметить, что явное указание типов программистом помогает компилятору сгенерировать более компактный код, что в свою очередь ведет к увеличению производительности программ.

Некоторые специальные формы, такие как `def` и т. п., имеют определенный набор названий ключей метаданных, которые могут изменять поведение определяемого символа. Например, следующий код:

```
(defn
 #^{:doc "my function"
    :tag Integer
    :private true}
 my-func [#^Integer x] (+ x 10))
```

определяет функцию `my-func`, которая получает и возвращает целое число (форма #^Integer при указании аргументов функции, и атрибут `:tag` для возвращаемого значения), имеет строку описания `my function`, и видима только в текущем пространстве имен, поскольку атрибут `:private` имеет истинное значение. Если мы прочитаем метаданные данной функции:

```
user> (meta #'my-func)
{:ns #<Namespace user>, :name my-func,
  :file "NO_SOURCE_FILE", :line 1,
  :arglists ([x]), :doc "my function",
  :tag java.lang.Integer, :private true}
```

то мы увидим, что интерпретатор добавил дополнительные данные, такие как `:ns`, `:file` и т. д. Это выполняется для всех символов

Разработчик имеет возможность считывания и изменения метаданных символов с помощью функций. Функция `meta` возвращает отображение, содержащее все имеющиеся метаданные. А с помощью функции `with-meta` можно добавить или изменить метаданные заданного символа.

2.4. Конкурентное программирование

Помимо стандартных средств Java, предназначенных для выполнения кода в отдельных потоках выполнения, Clojure имеет в своем арсенале собственные средства конкурентного выполнения кода (`pmap` и `pcalls`), выполнения кода в отдельном потоке, используя механизм `future` и синхронизации между потоками с помощью `promise`.

`pmap` — это параллельный вариант функции `map`, который может использоваться в тех случаях, когда функция-параметр не имеет побочных эффектов, и требует достаточно больших затрат на вычисление. Функция `pcalls` позволяет вычислить результат нескольких функций в параллельном режиме, возвращая последовательность их результатов в качестве результата выполнения функции.

2.4.1. future & promise

Достаточно часто при разработке приложений возникает необходимость выполнения долго работающего кода одновременно с выполнением других задач. Для более простой работы с таким кодом, в версии 1.1 было введено понятие `future`.

`future` позволяет программисту выделить некоторый код в отдельный поток выполнения, который выполняется параллельно с основным кодом. Результат выполнения `future` затем сохраняется, и может быть получен с помощью операции `deref` (`@`). Эта операция может заблокировать выполнение основного кода, если работа `future` еще не завершилась — в этом `future` похож на `promise`, который описан ниже. Значение, установленное при выполнении `future` сохраняется, и при последующих обращениях к нему, возвращается сразу, без вычисления. Рассмотрим простой пример:

```
(def future-test
     (future
       (do
          (Thread/sleep 10000)
          :finished)))
```

Тут создается объект `future`, в котором выполняется задержка на 10 секунд, а затем устанавливается значение `:finished`. Если мы обратимся к объекту `future-test` до завершения операции, то мы будем ожидать завершения указанного блока кода.

Но в отличие от `promise`, `future` имеет больше возможностей — вы можете проверить, закончилось ли выполнение кода с помощью функции `future-done?`, что позволяет избежать блокирования в случае обращения к еще не закончившейся операции. Кроме того, вы можете отменить выполнение операции с помощью функции `future-cancel` и проверить, не была ли отменена операция, с помощью функции `future-cancelled?`.

Иногда возникают ситуации, когда один поток исполнения должен передать какие-то данные другому. Это может быть организовано с помощью `promise`, которые в некоторых вещах похожи на `future`. Общая схема работы следующая: в одном потоке выполнения вы создаете некоторый объект с помощью `promise`, выполняете работу и затем с помощью `deliver` устанавливаете значение объекта. Результат, сохраненный в объекте, может быть получен с помощью операции `deref` (`@`) и не может быть изменен после установки с помощью `deliver`. Но если вы попробуете обратиться к значению, сохраненному в объекте, до того, как оно будет установлено, то ваш поток выполнения будет заблокирован, и возобновит работу только после установки значения. Однако после того как значение было установлено, его получение будет производиться уже без выполнения кода, использующегося для его вычисления. Рассмотрим следующий пример:

```
(def p (promise))
(do (future (Thread/sleep 5000)
            (deliver p :fred))
  @p)
```

В первой строке мы создаем объект `p`, который затем используется для синхронизации в блоке `do`. Если мы выполним код в блоке `do`, то выполнение затормозится на 5 секунд, поскольку поток выполнения, созданный `future`, еще не установил значение. А после окончания ожидания и установки значения с помощью `deliver`, операция `@p` сможет получить установленное значение равное `:fred`. Если мы попробуем выполнить операцию `@p` еще раз, то мы сразу получим установленное значение.

2.4.2. Работа с изменяемыми данными

Хотя по умолчанию переменные в Clojure неизменяемые, язык предоставляет возможность работать с изменяемыми переменными в рамках четко определенных моделей взаимодействия — как синхронных, так и асинхронных.[44] Сочетание неизменяемых данных с механизмами обновления данных (через ссылки, атомы и агенты) создает очень удобную среду для многопотокового программирования, что становится все более актуальным, поскольку число ядер в современных процессорах продолжает расти.

Имеющиеся средства для работы с изменяемыми данными можно классифицировать по нескольким параметрам, как показано в следующей таблице:

	Синхронное	Асинхронное
Координированное	ref	
Независимое	atom	agent
Изолированное	var	

В Clojure имеется три механизма синхронного обновления данных и один — асинхронного. Наиболее часто в коде используются ссылки (`ref`), которые предоставляют возможность синхронного обновления данных в рамках транзакций, и агенты (`agent`), которые реализуют механизмы асинхронного обновления данных. Кроме этого, существуют еще атомы, рассмотренные ниже, и «переменные» (var).[45] «Переменные» имеют «глобальное» (root) значение, которое определено для всех потоков выполнения, но это значение можно переопределять для отдельных потоков выполнения, используя `binding` или `set!`.

Хорошим примером использования возможностей Clojure в части работы с изменяемыми данными в многопоточных программах является пример «муравьи» (ants),[46] который Rich Hickey продемонстрировал в видеолекции «Clojure Concurrency[47]», в котоой рассказывается о возможностях Clojure в части конкурентного программирования. Еще один хороший пример использования Clojure для таких задач можно найти в серии статей Tim Bray.[48]

Ссылки

Синхронное изменение данных производится через ссылки на объекты данных. Изменение ссылок можно проводить только в рамках явно обозначенных транзакций. Изменение нескольких ссылок в рамках транзакции[49] является атомарной операцией, обеспечивающей целостность данных и выполняемой в изоляции (atomicity, consistency, isolation) — ACI (аналогично свойствам транзакций в базах данных,[50] но без долговечности (durability)).

Изменение данных с помощью ссылок возможно благодаря использованию Software Transactional Memory, которая обеспечивает целостность данных при работе с ними из нескольких потоков выполнения. Описание принципов работы STM, вместе с подробным описанием ее реализации в Clojure, вы можете найти в статье Software Transactional Memory[51] Марка Волкманна (R. Mark Volkmann).

Чтобы обновить какой-то объект, необходимо сначала объявить его с использованием функции `ref`, а изменение затем выполняется с помощью операций `alter`, `commute` или `ref-set`, которые находятся внутри блоков `dosync` или `io!`, запускающих новую транзакцию. Доступ к данным на чтение осуществляется с помощью оператора `deref` (или специального макроса процедуры чтения — `@`). При этом операции чтения не видят результатов еще не закончившихся транзакций. Необходимо помнить о том, что транзакция может быть запущена повторно (retried), и это надо учитывать в функциях, вызываемых из функций `alter` или `ref-set`.

Рассмотрим, например, код для управления набором счетчиков (например, для сбора статистики по каким-то действиям):

```
(def counters (ref {}))

(defn add-counter [key val]
  (dosync (alter counters assoc key val)))

(defn get-counter [key]
  (@counters key 0))

(defn increment-counter [key]
  (dosync
   (alter counters assoc key (inc (@counters key 0)))))

(defn rm-counter [key]
  (let [value (@counters key)]
    (if value
      (do (dosync (alter counters dissoc key))
          value)
      0)))
```

Загрузим этот код, выполним несколько функций и посмотрим на состояние переменной `counters` после выполнения каждой из функций:

```
user> @counters
{}
user> (dosync (add-counter :a 1) (add-counter :b 2))
user> @counters
{:b 2, :a 1}
user> (dosync (increment-counter :a) (increment-counter :b))
user> @counters
{:a 2, :b 3}
```

Это простой пример, показывающий координированное изменение данных разных счетчиков, но эти функции можно использовать в разных потоках выполнения без страха потерять или получить неправильные данные.

Для обеспечения корректности данных, сохраняемых по ссылке, программист может установить функцию-валидатор. Это выполняется с помощью функции `set-validator!` (или сразу, при создании ссылки), которая получает два аргумента — ссылку и функцию-валидатор для данной ссылки. В том случае, если программист устанавливает некорректное значение, функция-валидатор должна вернуть ложное значение или выбросить исключение. Например, чтобы запретить отрицательные значения счетчиков, мы можем использовать следующую функцию-валидатор:

---

[44] Стоит отметить, что изменяются не данные, а ссылки на данные. В статье мы будем говорить об «изменяемых данных», понимая под этим использование соответствующих механизмов изменения.

[45] http://clojure.org/vars

[46] http://clojure.googlegroups.com/web/ants.clj

[47] http://blip.tv/play/AbKALwI

[48] http://www.tbray.org/ongoing/When/200x/2009/09/27/Concur-dot-next

[49] Транзакции нужны не только для изменения данных, но и для координированного чтения данных из нескольких ссылок.

[50] http://en.wikipedia.org/wiki/ACID

[51] http://java.ociweb.com/mark/stm/article.html

© 2010 «Практика функционального программирования»

```
(set-validator! counters
  (fn [st] (or
            (empty? st)
            (not-any? #(neg? (second %)) st))))
```

и если пользователь попробует установить отрицательное значение счетчика, то Clojure выдаст ошибку.

Кроме этого, при работе с ссылками вы можете использовать так называемые функции-наблюдатели, которые позволяют получать информацию об изменениях состояния. Для добавления функции-наблюдателя вы можете воспользоваться функцией `add-watch`, которая принимает в качестве аргумента функцию, которая будет вызвана при изменении состояния, и ей будут переданы предыдущее и новое значение ссылки.

Более подробную информацию о работе с ссылками вы можете найти на сайте языка.[52]

Агенты

Агенты (agents) позволяют осуществлять асинхронное обновление данных. Работа с агентами похожа на работу со ссылками (только вы должны использовать `agent` вместо `ref`), но обновление данных может произойти в любой момент (и программист не может на это влиять) в зависимости от количества заданий. Эти задания выполняются в отдельном пуле потоков выполнения, размер которого ограничен. В отличие от ссылок, вам нет необходимости явно создавать транзакцию с помощью функции `dosync` — вы просто посылаете «сообщение», состоящее из функции, которая установит новое значение агента, и аргументов для этой функции.

Пример со счетчиками, переписанный на использование агентов, будет выглядеть следующим образом:

```
(def acounters (agent {}))

(defn add-acounter [key val]
  (send acounters assoc key val))

(defn increment-acounter [key]
  (send acounters assoc key (inc (@counters key 0))))

(defn get-acounter [key]
  (@acounters key 0))

(defn rm-acounter [key]
  (let [value (@acounters key)]
    (send acounters dissoc key)
    value))
```

Функции `send` и `send-off` получают в качестве аргументов имя агента, функцию, которую надо выполнить, и дополнительные параметры, которые будут переданы вызываемой функции. Вызываемая функция получает в качестве аргумента текущее состояние агента, и должна вернуть новое значение, которое получит агент.[53] Во время своего выполнения функция «видит» актуальное значение агента.

Разница между `send` и `send-off` заключается в том, что они используют разные по размеру пулы нитей выполнения. `send` рекомендуется применять для действий, которые ограничены по времени выполнения, такие как `conj` и т. д. А `send-off` лучше использовать для длительно выполняемых задач и задач, которые могут зависеть от ввода/вывода и других блокируемых операций.

В некоторых случаях вам может понадобиться, чтобы задания, посланные агенту, были завершены. Для этого в язык введены две функции, которые позволяют остановить выполнение текущего потока выполнения до завершения задач переданных агенту (или агентам). Функция `await` блокирует выполнение текущего кода до завершения всех задач, а функция `await-for` блокирует выполнение на заданное количество миллисекунд и возвращает контроль текущему потоку, даже если выполнение всех задач не было завершено.

Так же как и при использовании ссылок, при работе с агентами вы можете использовать функции-валидаторы и функции-наблюдатели. Остальную информацию об агентах вы можете найти на отдельной странице[54] сайта языка.

Атомы

Атомы (atoms) предоставляют возможность синхронного изменения независимых данных, для которых не требуется синхронизация в рамках транзакции. Работа с атомами похожа на работу со ссылками, только производится без координации: вы объявляете переменную с помощью функции `atom`, можете получить доступ к значению используя `deref` (или `@`) и установить новое значение с помощью функции `swap!` (или низкоуровневой функции `compare-and-set!`).

Изменения осуществляются с помощью функции `swap!`, которая в качестве аргументов принимает функцию и аргументы для этой функции (если необходимо). Переданная функция применяется к текущему значению атома для получения нового значения, и затем делается попытка атомарного изменения с помощью `compare-and-set!`. В том случае, если другой поток выполнения уже изменил значение атома, то вызов пользовательской функции повторяется для вычисления нового значения, и опять делается попытка изменения значения атома и т. д., пока попытка изменения не будет успешной.[55]

Вот простой пример кода, который использует атомы:

```
(def atom-counter (atom 0))
(defn increase-counter []
  (swap! atom-counter inc))
```

При использовании данного кода мы можем быть уверены, что значение счетчика будет увеличиваться всегда, независимо от того, сколько потоков выполнения вызывают эту функцию. Подробней об атомах вы можете прочитать на сайте языка.[56]

2.5. Взаимодействие с Java

Clojure реализует двухстороннее взаимодействие с библиотеками, работающими на базе JVM — код на Clojure может использовать существующие библиотеки и вызываться из других библиотек, реализовывать классы и т. п. Отдельно стоит отметить поддержку работы с массивами объектов Java — поскольку они не являются коллекциями, то Clojure имеет отдельные операции для работы с массивами: создание, работа с индивидуальными элементами, конвертация из коллекций в массивы и т. д. Подробную информацию о взаимодействии с Java вы можете найти на сайте языка.[57]

[52] http://clojure.org/refs

[53] Т. е. новое значение агента равно результату выполнения `(apply функция состояние-агента аргументы)`.

[54] http://clojure.org/agents

[55] В текущей версии Clojure количество попыток изменения ограничено значением 10000.

[56] http://clojure.org/atoms

[57] http://clojure.org/java_interop

Вы также можете встроить Clojure в ваш проект на Java и использовать его в качестве языка расширения. Дополнительную информацию об этом вы можете найти в учебнике о Clojure.[58]

2.5.1. Работа с библиотеками Java

Код, написанный на Clojure, может без особых проблем использовать библиотеки, написанные для JVM. По умолчанию в текущее пространство имен импортируются классы из пакета `java.lang`, что дает доступ к основным типам данных и их методам. А остальные пакеты и классы должны импортироваться явно, как это описано в разделе 2.3.9, иначе вам необходимо будет использовать полные имена классов с указанием названий пакетов.

Создание экземпляра класса производится с помощью специальной формы `new`, которая принимает в качестве аргументов имя нужного класса (записываемое с заглавной буквы) и аргументы, которые будут переданы конструктору класса. Кроме этого, определен макрос, который позволяет записывать создание новых экземпляров класса в более компактной форме. Для этого необходимо добавить знак `.` (точка) после имени класса и указать нужные аргументы для конструктора класса. Например, следующие виды записи эквивалентны:

```
(new String "Hello")
(String. "Hello")
```

Для обращения к членам классов существует несколько форм:

- для доступа к не статическим членам класса используется форма `(.имяЧленаКласса объект аргументы*)` или `(.имяЧленаКласса ИмяКласса аргументы*)`. Например, `(.toUpperCase "Hello")` в результате вернет `"HELLO"`;
- для доступа к статическим членам класса используется запись вида `(ИмяКласса/имяМетода аргументы*)` — для вызова методов, или `ИмяКласса/имяПеременной` — для переменных. Например, `(Math/sin 1)` или `Math/PI`.

Данные формы являются макросами, которые раскрываются в вызов специальной формы `.` (точка). В общем виде эта форма выглядит следующим образом: `(. объект имяЧленаКласса аргументы*)` или `(. ИмяКласса имяЧлена аргументы*)`. Такчто вызов `(Math/sin 1)` раскрывается в `(. Math sin 1)`, вызов `(.toUpperCase "Hello")` в `(. "Hello-toUpperCase)` и т. д.

Существует еще один макрос, который позволяет организовывать связанные вызовы вида `System.getProperties().get("os.name")`, которые очень часто встречаются в коде на Java. Этот макрос называется `..` (две точки) и записывается в виде `(.. объектИлиИмяКласса выражение+)`. Например, код на Java, приведенный выше, в Clojure будет выглядеть следующим образом:

```
(.. System (getProperties) (get "os.name"))
```

В том случае, если нет необходимости передавать аргументы, можно использовать запись выражения без скобок:

```
(.. System getProperties (get "os.name"))
```

Есть еще одна форма, которая позволяет выполнить вызовы нескольких методов, примененных к одному объекту — это макрос `doto`, который в качестве аргументов получает объект и выражения, и в качестве результата возвращает объект. Например, следующий код:

```
(doto
  (new java.util.HashMap)
  (.put "a" 1) (.put "b" 2))
```

создаст новое отображение и поместит в него два объекта.

Поскольку объекты Java в отличие от объектов Clojure изменяемы, то программист имеет возможность установки значений полей класса. Это выполняется с помощью специальной формы `set!`, которая имеет следующий вид: `(set! (. объектИлиИмяКласса имяЧленаКласса) выражение)`. Однако помните, что вы можете применять эту форму только к классам Java.

Вы также можете использовать методы классов в качестве функций первого порядка. Для этого определен макрос `memfn`, который принимает имя метода и список аргументов этой функции, и создает соответствующую функцию Clojure. Например, код:

```
(map (memfn toUpperCase) ["aa" "bb" "cc"])
```

применит метод `toUpperCase` из класса `String` к каждой из строк вектора. В простых случаях этот код можно заменить на анонимную функцию вида:

```
(map #(.toUpperCase %) ["aa" "bb" "cc"])
```

но в некоторых случаях `memfn` просто удобнее.

2.5.2. Вызов кода на Clojure из Java

Существует несколько причин, по которым вам может понадобится вызвать код, написанный на Clojure из Java. Первая причина — вам необходимо реализовать так называемые обратные вызовы (callbacks), которые будут реализовывать обработку каких-то событий, например, при обработке XML файла или при реализации GUI. Вторая причина — вы хотите реализовать некоторую функциональность на Clojure, и позволить классам Java пользоваться этой функциональностью.

В Clojure существуют разные способы выполнения этих задач — вы можете создавать анонимные классы, полезные при реализации callbacks, с помощью макроса `proxy`, или создавать именованные классы с помощью макроса `gen-class`. Обе эти возможности описываются более подробно в следующих разделах.

Реализация обратных вызовов (callback) с помощью proxy

Макрос `proxy` используется для создания анонимных классов, которые реализуют указанные интерфейсы и/или расширяют существующие классы. В общем виде вызов этого макроса выглядит следующим образом: `(proxy [списокКлассовИлиИнтерфейсов] [аргументыКонструктораКласса] РеализуемыеМетоды+)`.

Например, если вы хотите обрабатывать XML с помощью парсера SAX, то вы можете создать свой класс, который будет обрабатывать определенные события:

```
(import '(org.xml.sax InputSource)
        '(org.xml.sax.helpers DefaultHandler)
        '(java.io StringReader)
        '(javax.xml.parsers SAXParserFactory))
```

[58] http://en.wikibooks.org/wiki/Clojure_Programming/Tutorials_and_Tips#Invoking_Clojure_from_Java

 © 2010 «Практика функционального программирования»

```
(def print-element-handler
  (proxy [DefaultHandler] []
    (startElement
      [uri local qname atts]
      (println (format "Saw element: %s" qname)))))

(defn demo-sax-parse [source handler]
  (.. SAXParserFactory newInstance newSAXParser
      (parse (InputSource. (StringReader. source))
             handler)))

(demo-sax-parse "<foo><bar>body</bar></foo>"
    print-element-handler)
```

и после выполнения этого кода на стандартный вывод будут выданы названия элементов, составляющих данный XML документ.

Создание классов с помощью gen-class

Как отмечалось выше, макрос `gen-class` используется для создания именованных классов, которые будут доступны для кода на Java, только если вы откомпилируете исходный код в байт-код. В отличие от `proxy`, `gen-class`[59] имеет значительно больше опций, которые управляют его поведением, но при этом он предоставляет и большую функциональность.

В общем виде вызов макроса выглядит следующим образом: `(gen-class опции+)`. Полный список опций можно найти в официальной документации[60] или в записи[61] в блоге Meikel Brandmeyer, а здесь мы приведем небольшой пример реализации класса и рассмотрим, из чего он состоит:

```
(ns myclass
  (:import
    (org.apache.tika.parser Parser
                            AutoDetectParser
                            ParseContext)))

(gen-class
 :name MyClass
 :implements [org.apache.tika.parser.Parser])

(defn -parse [this stream handler metadata context]
  '())
```

В данном примере мы создаем класс `MyClass`, который реализует интерфейс `org.apache.tika.parser.Parser` и определяет метод `parse`, принимающий четыре аргумента. После компиляции этого кода, мы можем использовать его из кода на Java как самый обычный класс.

Отметьте, что методы не указываются в объявлении класса, а реализуются в текущем пространстве имен. Но это относится только к тем методы, которые уже объявлены в родительском классе или интерфейсе. Также заметьте, что имя реализуемого метода начинается со знака `-` (минус) — это префикс, который используется по умолчанию, чтобы отличать методы-члены класса от обычных функций. Разработчик может выбрать другой префикс с помощью опции `:prefix`.

В том случае, если вы хотите расширить существующий класс, вам необходимо использовать опцию `:extends`, вместо или наравне с опцией `:implements`, которая приведена в нашем примере.

Для инициализации класса вы можете использовать функцию, которая указывается в опции `:init`, и которой будут переданы аргументы конструктора класса. Кроме этого, существует опция `:constructors`, которая может использоваться, если вы хотите создать конструкторы класса, не совпадающие по количеству аргументов с конструкторами родительского класса. А новые методы могут быть добавлены к классу с помощью опции `:methods`.

По умолчанию, сгенерированный класс не имеет доступа к защищенным переменным родительского класса. Однако, к ним можно получить доступ, если использовать опцию `:exposes`. А с помощью опции `:exposes-methods` можно указать псевдонимы для методов родительского класса, если вам необходимо вызывать их из вашего класса.

Еще одной полезной опцией является опция `:state`, которая указывает имя переменной, в которой будет храниться внутреннее состояние вашего класса. Обычно в качестве значения используется `ref` или `atom`, которые могут быть изменены в процессе выполнения методов класса. Стоит отметить, что данное состояние должно быть установлено функцией, указанной в опции `:init`.

2.6. Поддержка языка

Эффективное использование языка невозможно без наличия инфраструктуры для работы с ним — редакторов кода, средств сборки, библиотек и т. п. вещей. Для Clojure имеется достаточное количество таких средств — как адаптированных утилит (Maven, Eclipse, Netbeans и т. п.), так и разработанных специально для этого языка — например, системы сборки кода Leiningen. Отладку приложений, написанных на Clojure, поддерживают почти все среды разработки, перечисленные ниже, а для профилирования можно использовать существующие средства для Java.

Число библиотек для Clojure постоянно увеличивается. Некоторые из них — лишь обертки для библиотек написанных на Java, а некоторые — специально разработанные для Clojure. Вместе с Clojure часто используют набор библиотек `clojure-contrib`,[62] который содержит различные полезные библиотеки, не вошедшие в состав стандартной библиотеки языка: функции для работы со строками и потоками ввода/вывода, дополнительные функции для работы с коллекциями, монады и т. д. Среди других библиотек можно отметить `Compojure` — для создания веб-сервисов; `ClojureQL` — для работы с базами данных; `Incanter` — для статистической обработки данных; `crane`, `cascading-clojure` и `clojure-hadoop` — для распределенной обработки данных. Это лишь малая часть существующих библиотек, многие из которых перечислены на сайте языка.[63]

2.6.1. Среды разработки

В настоящее время для работы с Clojure разработано достаточно много средств — поддержка Clojure имеется в следующих редакторах и IDE:

Emacs: Подсветка синтаксиса и расстановка отступов выполняются с помощью пакета `clojure-mode`. Для выполнения кода можно использовать `inferior-lisp-mode`,

[59] Очень часто `gen-class` используется в объявлении пространства имен с помощью макроса `ns`.

[60] http://richhickey.github.com/clojure/clojure.core-api.html#clojure.core/gen-class

[61] http://kotka.de/blog/2010/02/gen-class_how_it_works_and_how_to_use_it.html

[62] http://richhickey.github.com/clojure-contrib/index.html

[63] http://clojure.org/libraries

но лучше воспользоваться пакетом SLIME, для которого существует адаптер для Clojure — swank-clojure.[64] SLIME разработан для работы с разными реализациями Lisp и предоставляет возможности интерактивного выполнения и отладки кода, анализа ошибок, просмотра документации и т. д. Судя по последнему опросу среди программистов на Clojure, Emacs и SLIME являются самым популярным средством разработки.

Установка обоих пакетов может быть выполнена (и это рекомендуется авторами пакетов) через Emacs Lisp Package Archive.[65] Небольшое описание того, как установить и настроить `clojure-mode` и SLIME, вы можете найти в записи[66] в блоге Романа Захарова.

Если вы используете Windows, то вы можете воспользоваться Clojure Box[67] — пакетом, в котором поставляется уже настроенный Emacs, SLIME, Clojure и библиотека `clojure-contrib`. Использование этого пакета позволяет немного упростить процесс освоения языка.

Vim: Поддержка Clojure в Vim реализуется с помощью модуля VimClojure,[68] который реализует следующую функциональность:

- подсветку синтаксиса языка;
- правильную расстановку отступов;
- выполнение кода;
- раскрытие макросов;
- дополнение символов (omni completion);
- поиск в документации, как для самого кода на Clojure, так и в документации Java (javadoc).

На домашней странице проекта вы можете найти необходимую информацию по установке плагина, а также скринкаст, демонстрирующий возможности VimClojure.

Eclipse: Для Eclipse существует плагин Counterclockwise,[69] который обеспечивает выполнение следующих задач:

- подсветка, расстановка отступов и форматирование исходного кода;
- навигация по исходному коду;
- базовая функциональность по дополнению имен функций и переменных, включая функции библиотек написанных на Java;
- выполнение кода в REPL;
- отладка на уровне исходного кода.

Информацию по установке вы можете найти на странице проекта.

Netbeans: В Netbeans поддержка Clojure осуществляется плагином Enclojure[70] со следующей функциональностью:

- подсветка и расстановка отступов в исходном коде, а также работа с S-выражениями;
- выполнение кода в REPL, включая работу с REPL на удаленных серверах, историю команд, тесную интеграцию с редактором кода;
- навигация по исходному коду, включая навигацию для мультиметодов;
- дополнение имен для функций Clojure и Java;
- отладка на уровне исходного кода, с установкой точек останова, показом значений переменных и пошаговым выполнением кода.

IntelliJ IDEA: Для этой IDE создан плагин La Clojure,[71] реализующий следующие функции:

- подсветка и форматирование исходного кода с возможностью настройки пользователем;
- навигация по исходному коду;
- свертывание определений функций и переменных;
- дополнение имен для функций, переменных и пространств имен Clojure, а также поддержка дополнений для имен классов и функций в библиотеках написанных на Java;
- выполнение кода в REPL;
- отладка кода, в том числе и для кода в REPL;
- рефакторинг кода на Clojure;
- компиляция исходного кода в Java classes.

Процесс установки некоторых из этих средств можно найти в наборе скринкастов,[72] созданных Sean Devlin.

2.6.2. Компиляция и сборка кода на Clojure

Сборку кода, написанного на Clojure, можно осуществлять разными способами — начиная с компиляции, используя Clojure в командной строке, и заканчивая использованием высокоуровневых утилит для сборки кода, таких как Maven и Leiningen.

В принципе, компиляция кода — необязательный этап, поскольку Clojure автоматически откомпилирует загружаемый код, и многие проекты пользуются этим, распространяясь в виде исходных кодов. Однако предварительная компиляция (ahead-of-time, AOT) позволяет ускорить загрузку вашего кода, сгенерировать код, который будет использоваться из Java, а также позволяет не предоставлять исходный код, что важно для коммерческих проектов.

Компиляция кода на Clojure осуществляется в соответствии со следующими принципами:

- единицей компиляции является пространство имен;
- для каждого файла, функции и `gen-class` создаются отдельные файлы `.class`;
- также для каждого файла создается класс-загрузчик, вида `имя-файла__init.class`;
- файл, содержащий пространство имен, использующий имя со знаком `-` (минус), должен иметь имя, в котором `-` заменены на знак `_` (подчеркивание).

[64] http://github.com/technomancy/swank-clojure
[65] http://tromey.com/elpa/install.html
[66] http://zahardzhan.blogspot.com/2010/02/emacs.html
[67] http://clojure.bighugh.com/
[68] http://kotka.de/projects/clojure/vimclojure.html
[69] http://code.google.com/p/counterclockwise/
[70] http://www.enclojure.org/
[71] http://plugins.intellij.net/plugin/?id=4050
[72] http://vimeo.com/channels/fulldisclojure

© 2010 «Практика функционального программирования»

Компиляция кода с помощью Clojure

Для компиляции из REPL имеется функция `compile`, которая в качестве аргумента получает символ, определяющий пространство имен, например:

```
(compile 'my-class)
```

что приведет к компиляции файла `my_class.clj`. Стоит отметить, что `CLASSPATH` также должен содержать в себе каталог `class`, находящийся в том же каталоге, что и исходный файл. В этот каталог будут помещены сгенерированные файлы `.class`.

Провести компиляцию исходного текста можно и не запуская REPL, для этого можно воспользоваться следующей командой:

```
java -cp clojure.jar:'pwd'/class
    -Dclojure.compile.path=class clojure.lang.Compile
    my-class
```

которая выполняет компиляцию пространства имен `my-class`, находящегося в файле `my_class.clj`. Заметьте, что в `CLASSPATH` явно добавлен подкаталог `class`, указанный с помощью свойства `clojure.compile.path`. Команды такого вида можно использовать в других системах сборки, таких как Ant.

Ant

Чтобы не изобретать код для компиляции файлов Clojure для каждого нового проекта сборки, был создан проект clojure-ant-tasks,[73] который определяет стандартные задачи (tasks) для компиляции и тестирования кода, написанного на Clojure. Подробное описание использования пакета задач вы можете найти на странице проекта.

Использование Maven с Clojure

Система сборки кода Maven[74] достаточно популярна среди разработчиков на Java, поскольку она позволяет декларативно описывать процесс сборки, тестирования и деплоймента, а выполнение конкретных задач ложится на плечи конкретных модулей (plugins).

Для Maven написан модуль clojure-maven-plugin,[75] который позволяет компилировать и тестировать код, написанный на Clojure. Этот модуль позволяет прозрачно интегрировать Clojure в существующую систему сборки на основе Maven. Кроме компиляции и тестирования, данный модуль определяет дополнительные задачи, такие как запуск собранного пакета, запуск REPL с загрузкой собранного пакета, а также запуск серверов SWANK или Nailgun, что позволяет использовать SLIME и VimClojure для интерактивной работы с собранным пакетом.

Подробное описание того, как использовать этот модуль вместе с Maven, вы можете найти на странице проекта, а в качестве примера использования Maven для сборки проекта на Clojure, состоящего из нескольких подпроектов, вы можете обратиться к исходным текстам проекта Incanter.[76]

Leiningen

Для Clojure также существует своя собственная система сборки — Leiningen,[77] описывающая проекты и процесс сборки, используя язык Clojure. В последнее время эта система становится все более популярной — она имеет возможности расширения с помощью дополнительных модулей, например, для компиляции кода на Java и т. п.

Из коробки Leiningen позволяет выполнять базовые задачи — компиляцию кода, тестирование, упаковку кода в jar-архив, сборку jar-архива со всеми зависимостями и т. д. Кроме того, имеется базовая поддержка работы с Maven, что позволяет использовать собранный код в других проектах.

Установка Leiningen достаточно проста и описана на странице проекта.[78] После установки вы можете начать его использовать в своем проекте, добавив файл `project.clj`, содержащий что-то вроде следующего кода:

```
(defproject test-project "1.0-SNAPSHOT"
  :description "A test project."
  :url "http://my-cool-project.com"
  :dependencies [[org.clojure/clojure "1.1.0"]
                 [org.clojure/clojure-contrib "1.1.0"]]
  :dev-dependencies [[org.clojure/swank-clojure "1.0"]])
```

который определяет новый проект `test-project` с зависимостями от Clojure и набора библиотек `clojure-contrib`, а также зависимостью, которая используется в процессе разработки — `swank-clojure`. `defproject` — это макрос Clojure, который раскрывается в набор инструкций по сборке, и является единственной обязательной конструкцией, которая должна быть указана в файле `project.clj`. Кроме этого, `project.clj` может содержать и произвольный код на Clojure, выполняемый в процессе сборки. Более подробную информацию о Leiningen можно найти на странице проекта.

Репозитории кода

Некоторые системы сборки, такие как Maven и Leiningen, поддерживают автоматическую загрузку зависимостей из центральных репозиториев кода. Для Clojure также имеются отдельные репозитории, совместимые с этими системами.

В первую очередь это build.clojure.org,[79] который содержит сборки как самой Clojure, так и набора библиотек `clojure-contrib`. Например, для Maven вы можете добавить Clojure в зависимости с помощью следующего кода, добавленного в файл проекта `pom.xml`:

```
<repositories>
  <repository>
    <id>clojure-releases</id>
    <url>http://build.clojure.org/releases</url>
  </repository>
</repositories>
```

Кроме того, для распространения библиотек написанных на Clojure, был создан проект clojars.org,[80] который поддерживает работу с Maven и Leiningen, и на котором можно найти достаточно большое количество полезных библиотек.

2.7. Заключение

Мы надеемся, что данная статья помогла вам познакомиться с этим интересным языком. Количество проектов (в том числе и коммерческих) на Clojure постоянно увеличивается, и, может быть, вы также сможете использовать данный язык для

[73] http://github.com/jmcconnell/clojure-ant-tasks
[74] http://maven.apache.org/
[75] http://github.com/talios/clojure-maven-plugin
[76] http://github.com/liebke/incanter
[77] http://github.com/technomancy/leiningen
[78] http://github.com/technomancy/leiningen/
[79] http://build.clojure.org/
[80] http://clojars.org

написания программ, которые будут работать на платформе JVM.

Литература

[1] *Fogus M., Houser C.* Joy of Clojure. Thinking the Clojure Way. — Manning, 2010.

[2] *Halloway S.* Programming Clojure. — Pragmatic Bookshelf, 2009.

[3] *Rathore A.* Clojure in Action. — Manning, 2010.

[4] *VanderHart L.* Practical Clojure. — Apress, 2010.

 © 2010 «Практика функционального программирования»

Пределы выразительности свёрток

Виталий Брагилевский
bravit@fprog.ru

Аннотация

Свёртки — удобное средство обработки списков и других структур данных. Они позволяют записывать многие функции без явной рекурсии, а также упрощают формальное доказательство их корректности. В настоящей статье рассматривается вопрос о пределах того, что вообще можно записать (вычислить) с помощью свёрток. В частности, показывается, что свёртки могут вести себя как комбинаторы неподвижной точки при моделировании рекурсивных вызовов. С помощью свёрток также реализуется оператор примитивной рекурсии на списках, что позволяет выразить в соответствующих терминах любые примитивно рекурсивные функции.

Folding is a convenient tool for processing lists and other data structures. Folds allow to avoid explicit use of recursion when defining functions and simplify formal correctness proofs. This article discusses the expressive power of folds. It is shown that a fold can be used as a fixed-point combinator to simulate recursion and to express the primitive recursion operator on lists, thus being capable of implementing arbitrary primitive recursive functions.

Обсуждение статьи ведется по адресу
http://community.livejournal.com/fprog/8096.html.

3.1. Введение

В центре современной информатики (*computer science*) находится теория алгоритмов, в рамках которой изучаются формальные вычислительные модели. Пожалуй, самыми известными моделями являются машина Тьюринга и λ-исчисление Чёрча. Именно они служат теоретическим основанием для императивного и функционального программирования.

Помимо λ-исчисления и машины Тьюринга в XX веке было сформулировано множество других формализаций понятия алгоритма: машины Поста и Минского, нормальные алгорифмы Маркова, теория рекурсивных функций. Все эти формализации являются эквивалентными в следующем смысле: всё, что можно вычислить в рамках одной формализации, можно вычислить и в другой.

В этой статье демонстрируется связь между таким важным инструментом функционального программирования как свёртки и некоторыми разделами теории алгоритмов. В частности, во втором разделе рассматривается несколько искусственная, но от того не менее интересная задача выражения функции **dropWhile** из стандартной библиотеки языка Haskell в терминах свёрток. Здесь показаны полезные на практике приёмы передачи контекстной информации в процессе свёртки списка и использования там же функций высшего порядка. Третий раздел посвящён связи между свёртками и *комбинаторами неподвижной точки*, используемыми в λ-исчислении для реализации рекурсивных вызовов. Там показывается, что свёртку можно использовать вместо таких комбинаторов, моделируя с её помощью рекурсию. Наконец, в четвёртом разделе рассматривается построение достаточно широкого класса *примитивно рекурсивных функций*, который в свою очередь является подклассом класса *частично рекурсивных функций*, исчерпывающего все вычислимые функции. Это достигается выражением *оператора примитивной рекурсии* в терминах свёрток.

Необходимо заметить, что данная статья носит теоретический характер, её полезность для практического программирования невелика. В основе статьи лежат две работы: статьи Берни Поупа [5] и Грэма Хаттона [3].

Для чтения статьи необходимо понимать определение и порядок работы функций **foldl** и **foldr** ([8], п. 5.11), а также уметь читать код на языке Haskell, включая определение функций, сопоставление с образцом ([8], п. 5.10), функции высшего порядка ([8], п. 5.3), каррирование ([8], п. 5.5) и бесточечный стиль ([8], п. 5.6). Словом, лучше предварительно прочитать статью Евгения Кирпичёва «Элементы функционального программирования» [8]. Кроме того, мы будем активно использовать различные термины из λ-исчисления, в том числе понятие *комбинатора неподвижной точки*. Лучший способ познакомиться с ними — прочитать вторую и третью главы курса Джона Харрисона «Введение в функциональное программирование» [9], переведённого на русский язык силами энтузиастов.

3.2. Неуловимый dropWhile

Функция **dropWhile** из модуля **Prelude** стандартной библиотеки языка Haskell предназначена для удаления самого длинного префикса (начальной части) заданного списка, состоящего из элементов, удовлетворяющих некоторому предикату. В качестве аргументов она принимает предикат и исходный список, а возвращает новый список. Приведём её определение и примеры использования:

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile predicate [] = []
dropWhile predicate list@(x:xs)
            | predicate x =  dropWhile predicate xs
            | otherwise   =  list

> dropWhile (<5) [1..10]
[5,6,7,8,9,10]
> dropWhile (>0) [1,2,−3,2,1]
[−3,2,1]
> dropWhile even [2,4,1,2,3]
[1,2,3]
> dropWhile (==1) [2,2,1,1]
[2,2,1,1]
> dropWhile (≠0) [1..5]
[]
> take 3 (dropWhile (<5) [1..])
[5,6,7]
```

Из определения видно, что рекурсия завершается при обнаружении первого элемента, не удовлетворяющего предикату, при этом возвращается весь остаток списка. Если ни один из элементов списка не удовлетворяет предикату, то возвращается пустой список. Последний пример демонстрирует ленивость функции **dropWhile**, благодаря которой её можно применять к бесконечным спискам. Поскольку из всего списка (`dropWhile (<5) [1..]`) для вывода результата требуются только три первых элемента (`take 3`), все остальные не вычисляются.

Ричард Бёрд (Richard Bird) в своей классической книге «Introduction to Functional Programming using Haskell» поставил вопрос о том, можно ли реализовать стандартную функцию **dropWhile**, пользуясь свёртками ([1], с. 126, упр. 4.5.2). Эта задача оказалась неожиданно сложной, причём было получено множество совершенно разных решений с отличающимися свойствами. Демонстрируя некоторые из этих решений, мы будем следовать преимущественно, но не исключительно, работе Берни Поупа [5].

3.2.1. Готовим сцену

Напомним, что различают правую и левую свёртки. Левая свёртка определяется следующим образом:

```
foldl :: (a → b → a) → a → [b] → a
foldl combine base [] = base
foldl combine base (x:xs)
    = foldl combine (combine base x) xs
```

При обработке списка `[x1,x2,...,xn]` левая свёртка трансформируется в последовательность вызовов:

```
combine (...(combine (combine base x1) x2)...) xn
```

Правая свёртка определяется так:

```
foldr :: (a → b → b) → b → [a] → b
foldr combine base [] = base
foldr combine base (x:xs)
    = combine x (foldr combine base xs)
```

Соответственно, правая свёртка списка `[x1,x2,...,xn]` преобразуется в вызовы:

```
combine x1 (combine x2 (...(combine xn base)...))
```

Использование свёрток предполагает единую схему реализуемой функции, которой мы будем всюду придерживаться. Например, для правой свёртки эта схема выглядит так:

```
dwImpl :: (a → Bool) → [a] → [a]
dwImpl predicate list
  = foldr combine base list
    where
      base = ...
      combine x xs = ...
```

Для удобства тестирования различных реализаций функции **dropWhile** подготовим две вспомогательные функции. Функция `evalDW` вычисляет результаты тестируемой реализации на каждом задании из списка `tasks`, а функция `testDW` сравнивает результаты, полученные тестируемой реализацией, с результатами **dropWhile**. Список `tasks` состоит из пар (*предикат*, *исходный список*).

```
evalDW dwImpl
 = let tasks = [((<5), [1..10]),
       ((>0), [1,2,−3,2,1]),
       (even, [2,4,1,2,3]),
       ((==1), [2,2,1,1]),
       ((≠0), [1..5])] in
   map (uncurry dwImpl) tasks

testDW dwImpl = evalDW dwImpl == evalDW dropWhile

> evalDW dropWhile
[[5,6,7,8,9,10],[−3,2,1],[1,2,3],[2,2,1,1],[]]
> testDW dropWhile
True
```

Таким образом, если `testDW dwImpl == True`, то на указанных примерах функция `dwImpl` ведёт себя так же, как **dropWhile**, а значение **False** будет сигнализировать об ошибке. Использованная в функции `evalDW` стандартная функция **uncurry** позволяет вызвать функцию `dwImpl` с двумя аргументами, передав ей только один — пару (*предикат*, *список*).

Чтобы понять, в чём состоит основная сложность поставленной задачи, попробуем привести неправильное, хотя и очень близкое к правильному, решение:

```
dropWhileWrong predicate list
     = foldr combine base list
       where
        base = []
        combine x xs
             | predicate x = xs
             | otherwise = x:xs

> testDW dropWhileWrong
False
> evalDW dropWhileWrong
[[5,6,7,8,9,10],[−3],[1,3],[2,2],[]]
```

Из примеров видно, что эта реализация удаляет не только начальные элементы списка, удовлетворяющие предикату, но и все остальные, реализуя тем самым стандартную функцию **filter**. Действительно, свёртка проходит по всему списку, а у нас нет способа остановить её в нужном месте. Проблема здесь в нехватке контекстной информации: удалять нужно не просто элементы, удовлетворяющие предикату, а элементы, удовлетворяющие предикату *и* стоящие в начале списка. Как раз вторая часть этого условия в нашей неправильной реализации не учитывается, а извлечь её из аргументов функции `combine` невозможно.

3.2.2. Добавляем контекст

Итак, первоочередная задача — добавление контекста. Это можно сделать, например, так: результатом свёртки становится пара (*сформированный список*, *контекстная информация*). Самый простой вариант достаточной контекстной информации — признак начала списка (логическое значение). Тогда при свёртке списка слева направо (с использованием левой свёртки **foldl**) этот флаг учитывается, и удаление элемента происходит, только если он имеет истинное значение. Значение флага меняется на **False** при обнаружении первого элемента, не удовлетворяющего предикату, после чего все оставшиеся элементы переносятся в новый список без изменения. В самом конце из пары выделяется первый элемент — список, который и является результатом:

```
dropWhileWithContext predicate list =
       fst (foldl combine base list)
       where
         base = ([], True)
         combine res@(xs, lead) x
              | lead && predicate x = res
              | otherwise = (xs++[x], False)

> testDW dropWhileWithContext
True
```

Рассмотрим работу функции на конкретном примере, предположив, что выполняется следующий вызов:

```
> dropWhileWithContext (<3) [1,2,3,4,1,5]
```

Согласно определению левой свёртки, этот вызов трансформируется в последовательность вызовов `combine`:

```
combine
    (combine
        (combine
            (combine
                (combine
                    (combine ([], True) 1)
                    2)
                3)
            4)
        1)
    5
```

Соберём в таблицу аргументы и результаты всех вызовов функции `combine`, начиная с самого внутреннего (звёздочкой помечены результаты, полученные по ветви **otherwise**):

Шаг	Аргументы		Результат
	`res@(xs, lead)`	`x`	
1	`([], True)`	1	`([], True)`
2	`([], True)`	2	`([], True)`
3	`([], True)`	3	`([3], False)*`
4	`([3], False)`	4	`([3,4], False)*`
5	`([3,4], False)`	1	`([3,4,1], False)*`
6	`([3,4,1], False)`	5	`([3,4,1,5], False)*`

Окончательно, получаем:

```
> dropWhileWithContext (<3) [1,2,3,4,1,5]
[3,4,1,5]
```

Фактически, обработка элементов списка при левой свёртке ведётся от начала к концу, поэтому каждый следующий элемент должен добавляться к концу формируемого списка с помощью относительно неэффективной операции конкатенации `(++)`. Если бы элементы списка обрабатывались в обратном порядке, то очередной элемент можно было бы добавлять в начало формируемого списка с помощью эффективного конструктора списка `(:)`. Для достижения соответствующего результата придётся прибегнуть к правой свёртке. К сожалению, в этом случае контекстная информация в виде признака начала списка перестаёт работать: определить, в какой момент «начинается» начальная часть списка при движении с конца невозможно. К счастью, можно придумать другой контекст (предложенный Грэмом Хаттоном в статье [3]) — будем формировать пару, состоящую из двух списков:

1) Все просмотренные к настоящему моменту элементы.

2) Все просмотренные к настоящему моменту элементы, за исключением начальных, удовлетворяющих предикату.

Если в какой-то момент в ходе работы оказывается, что встречен элемент, не удовлетворяющий предикату, второй список отбрасывается, а его место занимает первый. Результатом является второй список, причём первый список — копия исходного.

```
dropWhileWithContext2 predicate list =
       snd (foldr combine base list)
       where
         base = ([], [])
         combine x (xs, ys)
               | predicate x = (x:xs, ys)
               | otherwise = (x:xs, x:xs)

> testDW dropWhileWithContext2
True
```

Теперь попробуем выполнить вызов:

```
> dropWhileWithContext2 (<3) [1,2,3,4,1,5]
```

Правая свёртка преобразуется в последовательность вызовов:

```
combine
    1
    (combine
        2
        (combine
            3
            (combine
                4
                (combine
                    1
                    (combine 5 base)))))
```

Видно, что при правой свёртке самому внутреннему вызову функции `combine` передаётся последний элемент списка. Посмотрим на аргументы и результаты всех вызовов функции `combine` (звёздочка снова обозначает результат, полученный по ветви **`otherwise`**):

Шаг	Аргументы		Результат
	x	(xs, ys)	
1	5	([],[])	([5],[5]
2	1	([5],[5])	([1,5],[5
3	4	([1,5],[5])	([4,1,5],[4,
4	3	([4,1,5],[4,1,5])	([3,4,1,5],[3,
5	2	([3,4,1,5],[3,4,1,5])	([2,3,4,1,5],[3
6	1	([2,3,4,1,5],[3,4,1,5])	([1,2,3,4,1,5],[

Итак, в результате получаем:

```
> dropWhileWithContext2 (<3) [1,2,3,4,1,5]
[3,4,1,5]
```

К сожалению, обе разработанные функции имеют слишком строгое ограничение — они не работают с бесконечными списками. Например:

```
> take 3 (dropWhileWithContext2 (<5) [1..])

ERROR - ...
```

В следующих реализациях мы попытаемся обойти это ограничение.

3.2.3. Пишем функцию, возвращающую функцию

Очередная идея основывается на использовании функций высшего порядка: результатом свёртки будет являться функция, которая, будучи применённой к списку, вернёт требуемый результат.

Эта идея исходит из того, что функция **`dropWhile`** фактически эквивалентна композиции нескольких функций **`tail`** (напомним, что функция **`tail`** возвращает хвост списка, т. е. весь список без первого элемента):

```
> dropWhile (<5) [1..10]
[5,6,7,8,9,10]
> tail (tail (tail (tail [1..10])))
[5,6,7,8,9,10]
```

Ту же композицию можно записать явно, пользуясь операцией `(.)`:

```
> (tail.tail.tail.tail) [1..10]
[5,6,7,8,9,10]
```


© 2010 «Практика функционального программирования»

Теперь при свёртке исходного списка можно строить композицию нужного количества функций **tail**, а затем применять её снова к исходному списку:

```
dropWhileHigherOrder predicate list =
       (foldr combine base list) list
       where
         base = id
         combine x f
               | predicate x = f.tail
               | otherwise = id

> testDW dropWhileHigherOrder
True
```

Функция **id** — это тождественное отображение, играющее роль нейтрального элемента при композиции функций. Чтобы понять, как работает функция dropWhileHigherOrder, нужно аккуратно проследить, что является результатом свёртки. Для этого напишем вспомогательную функцию, которая вместо композиции строит список строк — названий соответствующих функций:

```
dropWhileHigherOrderStrings predicate list =
       foldr combine base list
       where
         base = ["id"]
         combine x f
               | predicate x = f++["tail"]
               | otherwise = ["id"]

> dropWhileHigherOrderStrings (<5) [1..10]
["id","tail","tail","tail","tail"]
```

Заметим, что если первый элемент исходного списка не удовлетворяет предикату, то вся композиция исчерпывается одной единственной функцией **id**:

```
> dropWhileHigherOrder (<5) [5..10]
[5,6,7,8,9,10]
> dropWhileHigherOrderStrings (<5) [5..10]
["id"]
```

Рассмотрим работу свёртки на уже знакомом примере:

```
> dropWhileHigherOrder (<3) [1,2,3,4,1,5]
[3,4,1,5]
```

Шаг	Аргументы		Результат
	x	f	
1	5	**id**	**id***
2	1	**id**	**id.tail**
3	4	**id.tail**	**id***
4	3	**id**	**id***
5	2	**id**	**id.tail**
6	1	**id.tail**	**id.tail.tail**

Теперь нужно убедиться, что полученные функции работают с бесконечными списками. Действительно:

```
> take 3 (dropWhileHigherOrder (<5) [1..])
[5,6,7]
> dropWhileHigherOrderStrings (<5) [1..]
["id","tail","tail","tail","tail"]
```

Запишем начальную часть последовательности вызовов функции combine, в которую преобразуется вызов dropWhileHigherOrder в этом примере:

```
combine
    1
    (combine
        2
        (combine
            3
            (combine
                4
                (combine
                    5
                    (combine
                        ...)))))
```

Заметим, что при обработке элементов списка, не удовлетворяющих предикату, результат функции combine *не зависит* от значения свёртки оставшейся части списка. Действительно, выбор нужного варианта её определения осуществляется без использования второго аргумента f (результат свёртки оставшейся части списка), а как только выбранным оказывается второй вариант (**otherwise = id**; очередной элемент не удовлетворяет предикату), он не нужен и для вычисления результата. В нашем примере первый элемент, не удовлетворяющий предикату — это число 5, и второй аргумент соответствующего вызова функции combine вычисляться не будет. Здесь проявляется свойство ленивости вычислений в языке Haskell.

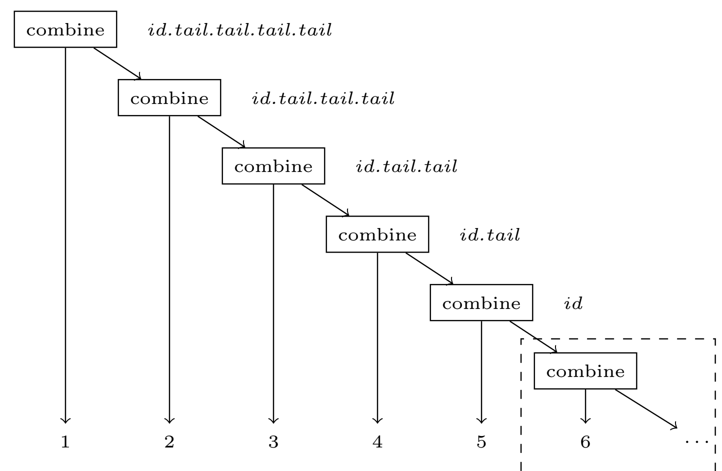


Рис. 3.1. Дерево вызовов, формируемое foldr

На рис. 3.1 показано дерево вызовов, формируемое **foldr** в этом примере. Штриховой рамкой отмечена свёртка хвоста, вычисление которой не потребуется, а курсивом написаны результаты соответствующих вызовов combine. Результатом вызова функции **foldr** является композиция **id.tail.tail.tail.tail**, которая затем применяется к исходному списку.

Интересно, что эффект ленивости можно испортить, добавив заведомо истинное условие для выбора варианта определения функции (примитивная операция **seq f True** принудительно вычислит f, а затем, отбросив результат вычисления, вернёт **True**):

```
dropWhileHigherOrderSpoiled predicate list =
       (foldr combine base list) list
       where
         base = id
         combine x f
               | seq f True && predicate x = f.tail
```

```
               | otherwise = id

> testDW dropWhileHigherOrderSpoiled
True
> dropWhileHigherOrderSpoiled (<3) [1..]

ERROR — ...
```

Результаты вычислений показывают, что в испорченной реализации свойство ленивости действительно теряется: результат свёртки оставшейся части списка теперь требуется всегда, поэтому обработка бесконечного списка невозможна.

К сожалению, у функции `dropWhileHigherOrder` есть другой недостаток: результатом свёртки может оказаться функция, состоящая из композиции огромного числа функций **tail** (их количество зависит от длины начальной части списка, подлежащей удалению), и памяти для хранения этой композиции может оказаться недостаточно. Из следующего примера видно, что `dropWhileHigherOrder` не справляется с большим, хоть и конечным, объёмом данных, хотя у **dropWhile** никаких проблем с этим нет (разумеется, вычисления продолжаются довольно долго):

```
> dropWhile (<1000000) [1..1000001]
[1000000,1000001]
> dropWhileHigherOrder (<1000000) [1..1000001]

ERROR — ...
```

3.2.4. Совмещаем композицию и применение

Новая идея — совместить построение композиции функций **tail** с одновременным применением их к списку. Вспомним, что результатом свёртки в целом, а значит и функции `combine` в частности, является функция, которая затем применяется к списку. Это означает, что функцию `combine`:

```
combine :: a → ([a]→[a]) → ([a]→[a])
combine x f
    | predicate x = f.tail
    | otherwise = id
```

можно переписать, явно указав, что она возвращает функцию над списками:

```
combine x f =
    λlist → (if predicate x then f.tail else id) list
```

или, что то же самое:

```
combine x f =
    λlist → if predicate x
              then (f.tail) list
              else id list
```

От использования безымянной λ-функции можно отказаться, перенеся её аргумент `list` непосредственно в аргументы `combine`:

```
combine :: a → ([a]→[a]) → [a] → [a]
combine x f list
    | predicate x = (f.tail) list
    | otherwise = id list
```

Очевидно, теперь можно избавиться от явной композиции и функции **id**:

```
combine x f list
    | predicate x = f (tail list)
    | otherwise = list
```

Заметим, что если эту версию трёхаргументной функции `combine` вызывать с двумя аргументами (как это фактически происходит при свёртке), то результатом будет функция над списками, т. е. именно то, что требуется.

Проведённые до сих пор изменения пока не привели к ожидаемому эффекту, так как вследствие свойства ленивости вызов **tail** будет задерживаться до тех пор, пока его результат не понадобится. Т. е. вместо явной композиции в нашей функции мы получаем неявную в трансляторе Haskell. Сделаем ещё один шаг — потребуем вычислить хвост списка, воспользовавшись сопоставлением с образцом:

```
combine x f list@(_:xs)
    | predicate x = f xs
    | otherwise = list
```

Заметим, что список `list` здесь гарантированно не пустой, так как максимальное количество вызовов `combine` равно количеству элементов в списке (все эти вызовы потребуются только в том случае, если все элементы списка удовлетворяют предикату, т. е. их все требуется удалить).

Теперь всё должно работать как следует. Приведём полученную функцию полностью:

```
dropWhileHigherOrder2 predicate list =
       (foldr combine base list) list
       where
         base = id
         combine x f list@(_:xs)
             | predicate x = f xs
             | otherwise = list

> testDW dropWhileHigherOrder2
True
> take 3 (dropWhileHigherOrder2 (<5) [1..])
[5,6,7]
> dropWhileHigherOrder2 (<1000000) [1..1000001]
[1000000,1000001]
```

Посмотрим, что происходит при выполнении следующего вызова:

```
> dropWhileHigherOrder2 (<3) [1..5]
```

Сначала выполняется первый шаг свёртки:

```
dropWhileHigherOrder2 (<3) [1..5] ⇒
(foldr combine id [1..5]) [1..5] ⇒
(combine
    1
    (foldr combine id [2..5])) [1..5]
```

Так как `combine` теперь трёхаргументная, она сразу применяется к списку:

```
combine
    1
    (foldr combine id [2..5])
    [1..5]
```

Число 1 удовлетворяет предикату, поэтому выбирается первый вариант определения `combine` (`= f xs`):

```
(foldr combine id [2..5]) [2..5]
```

Ещё два шага свёртки и два перехода от двухаргументной `combine` к трёхаргументной:

```
(combine
    2
```

```
    (foldr combine id [3..5])) [2..5] ⇒
combine
    2
    (foldr combine id [3..5]))
    [2..5] ⇒
(combine
    3
    (foldr combine id [4..5])) [3..5] ⇒
combine
    3
    (foldr combine id [4..5]))
    [3..5]
```

Как и прежде, свёртка строится до первого элемента, не удовлетворяющего предикату (<3 в нашем примере), поэтому второй аргумент последнего вызова `combine` не вычисляется, а результатом оказывается третий:

```
[3..5]
```

Получаемая последовательность вызовов показана на рис. 3.2.

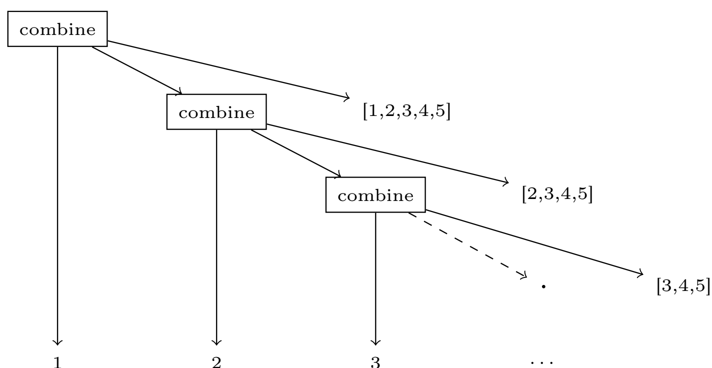


Рис. 3.2. Последовательность вызовов `combine`

Однако наш путь на этом не заканчивается. Изучая код функции `dropWhileHigherOrder2`, можно заметить, что здесь используются две копии списка — по одной из них идёт свёртка, а ко второй применяется её результат. Понятно, что очередной сворачиваемый элемент можно брать из любого из этих списков. Например, в следующем варианте он берётся из второй копии:

```
dropWhileHigherOrder2' predicate list =
       (foldr combine base list) list
       where
         base = id
         combine _ f list@(x:xs)
              | predicate x = f xs
              | otherwise = list

> testDW dropWhileHigherOrder2'
True
```

Теперь становится ясно, что содержимое первого списка (того, по которому идёт свёртка) вообще не имеет никакого значения, он может быть любым, лишь бы его длина совпадала с длиной исходного списка:

```
dropWhileHigherOrder2'' predicate list =
       (foldr combine base [1..length list]) list
       where
         base = id
         combine _ f list@(x:xs)
              | predicate x = f xs
              | otherwise = list

> testDW dropWhileHigherOrder2''
True
```

Длина списка, передаваемого свёртке, обеспечивает выполнение нужного количества операций. В принципе, эту задачу тоже можно поручить второй копии списка: для этого нужно обязать функцию `combine` проверять её пустоту. Заметим, что при этом исчезнет необходимость в присваивании аргументу `base` тождественного преобразования, так как до его использования свёртка доходить не будет, останавливаясь на последнем вызове `combine`. Окончательная версия выглядит следующим образом:

```
dropWhileHigherOrder3 predicate list =
       (foldr combine base [1..]) list
       where
         base = undefined
         combine _ _ [] = []
         combine _ f list@(x:xs)
              | predicate x = f xs
              | otherwise = list

> testDW dropWhileHigherOrder3
True
```

Здесь использование бесконечного списка `[1..]` обеспечивает выполнение того количества операций, которое необходимо в каждом конкретном случае.

3.3. Свёртка как комбинатор неподвижной точки

3.3.1. Некоторые сведения из λ-исчисления

Одной из важнейших характеристик функционального программирования является использование функций как значений первого класса (*first class values*). Это означает, что функции наряду с данными других типов — числами, логическими значениями, списками — могут быть аргументами или возвращаемыми значениями других функций, а следовательно, с ними можно выполнять различные операции, например, композицию. Такое широкое применение функций требует введения удобного их представления (нотации).

Соответствующее представление была изобретено в 30-е годы Алонзо Чёрчем. В нём в качестве вспомогательного символа используется греческая буква λ (с примерами её использования в языке Haskell мы уже встречались). λ-нотация важна не сама по себе, а как инструмент построения формальной системы λ-исчисления, которое является теоретической основой функционального программирования. В рамках этого исчисления строится понятие λ-термов и определяются способы их преобразования (*редукции*). На базе λ-термов можно вводить целые числа и арифметические операции, логические значения и операции над ними (в т. ч. условную операцию), а также типы данных, строя таким образом полноценный язык программирования. Многие функциональные языки программирования можно считать расширениями λ-исчисления, а при их трансляции иногда происходит возврат в λ-исчисление. Более подробно процесс построения языка программирования на базе λ-исчисления изложен в курсе лекций [9].

Серьёзной проблемой при этом оказывается тот факт, что функции в λ-исчислении являются *безымянными*. Дело в том, что для реализации рекурсии необходимо в теле функции использовать её имя — но что же делать, если имени нет? Оказывается, эта проблема решается благодаря существованию так называемых *комбинаторов неподвижной точки* (они ещё называются *Y-комбинаторами*). Это λ-термы специального вида, предназначенные для отыскания неподвижной точки отображения.

Напомним, что точка x называется неподвижной точкой отображения f, если $x = f(x)$. Если отображение работает на множестве функций, то и неподвижная точка тоже является функцией. Именно такие неподвижные точки ищут комбинаторы неподвижной точки. В λ-исчислении комбинаторы неподвижной точки используются для моделирования рекурсивных вызовов. В следующем пункте будет продемонстрировано их применение для реализации функции вычисления факториала.

3.3.2. Вычисление факториала

Вспомним студенческую молодость[1] и реализуем вычисление факториала с помощью комбинатора неподвижной точки. Возьмём для начала рекурсивную реализацию функции вычисления факториала, например, такую:

```
fact n
     | n > 0 = n × fact (n−1)
     | otherwise = 1
```

Теперь вынесем рекурсивный вызов из функции, введя новый аргумент — функцию, которую нужно вызывать при вычислении второго элемента произведения:

```
factNonRec f n
     | n > 0 = n × f (n−1)
     | otherwise = 1
```

Возьмём какой-нибудь комбинатор неподвижной точки:

```
fix f = f (fix f)
```

Необходимо заметить, что комбинатор такого вида не является комбинатором неподвижной точки в терминах λ-исчисления, так как в его определении явно присутствует рекурсия. Можно привести примеры более корректных в этом смысле комбинаторов, которые одновременно являются более сложными. Однако для наших целей достаточно этого упрощённого комбинатора, за подробностями о «настоящих» Y-комбинаторах можно обратиться к пособию [9].

Воспользовавшись указанным комбинатором, получаем функцию для вычисления факториала:

```
fact' = fix factNonRec
```

Протестируем полученную функцию, сравнив ее результаты с факториалами, вычисленными самым прямолинейным способом[2]:

```
testFact factImpl
    = let test_values=[5,15,27,42]
          fact n = product [1..n] in
      map factImpl test_values == map fact test_values

> testFact fact'
True
```

Вызов полученной функции `fact'` приводит к многократному вызову функции `factNonRec`, причём этот процесс останавливается при выполнении ветви **`otherwise`** `= 1`. Попытаемся посмотреть, что происходит при вызове `fact' 3`:

```
fact' 3 ⇒
(fix factNonRec) 3 ⇒
factNonRec (fix factNonRec) 3 ⇒
3 × ((fix factNonRec) 2) ⇒
3 × (factNonRec (fix factNonRec) 2) ⇒
3 × 2 × ((fix factNonRec) 1) ⇒
3 × 2 × (factNonRec (fix factNonRec) 1) ⇒
3 × 2 × 1 ⇒
6
```

Таким образом, комбинатор неподвижной точки позволил построить рекурсивную функцию вычисления факториала из нерекурсивной заготовки.

3.3.3. Использование свёртки как комбинатора неподвижной точки

Мы уже знаем, что комбинаторы неподвижной точки позволяют строить рекурсивные функции. Попробуем сделать то же самое с функцией **`dropWhile`**. Вспомним ещё раз её рекурсивную реализацию:

```
dropWhile predicate [] = []
dropWhile predicate list@(x:xs)
            | predicate x = dropWhile predicate xs
            | otherwise = list
```

Проделаем с ней ту же операцию, которой подверглась функция `fact`:

```
dropWhileNonRec f predicate [] = []
dropWhileNonRec f predicate list@(x:xs)
            | predicate x = f predicate xs
            | otherwise = list
```

Теперь оказывается, что эта функция очень похожа на `combine` в её последнем воплощении (в функции `dropWhileHigherOrder3`):

```
combine _ _ [] = []
combine _ f list@(x:xs)
     | predicate x = f xs
     | otherwise = list
```

Разница лишь в том, что `dropWhileNonRec` содержит `predicate` в качестве аргумента и явно передаёт его функции `f` — заменителю рекурсивного вызова. В `combine` это не требуется только потому, что аргумент `predicate` и так доступен из объемлющей функции. Кроме того, в `combine` имеется фиктивный первый аргумент.

Таким образом, функция свёртки в последней реализации **`dropWhile`** играет роль комбинатора неподвижной точки. Именно поэтому свёртке не нужен список с исходными данными, ей нужен просто большой список, чтобы гарантировать достаточное для отыскания неподвижной точки количество рекурсивных вызовов.

Нам остаётся, во-первых, убедиться в том, что полученная из `dropWhileNonRec` функция делает то, что требуется:

[1] Автор представляет себе студенческую молодость примерно так: изучение программирования в любом вузе начинается с λ-исчисления, и на втором или, самое позднее, на третьем занятии студенты изучают комбинаторы неподвижной точки и реализуют с их помощью факториал.

[2] Ещё несколько более или менее традиционных способов вычисления факториала приведены в [6].

© 2010 «Практика функционального программирования»

```
dropWhile' = fix dropWhileNonRec

> testDW dropWhile'
True
> take 3 (dropWhile' (<5) [1..])
[5,6,7]
```

А во-вторых, что `foldr` с тем же успехом можно применить для вычисления неподвижной точки функции `factNonRec`. Для этого достаточно добавить функции `factNonRec` фиктивный первый аргумент и передать её функции `foldr` вместо `combine`:

```
factNonRec f n
    | n > 0 = n × f (n−1)
    | otherwise = 1

fact'' = foldr (λ_ → factNonRec) undefined [1..]

> fact'' 5
120
> testFact fact''
True
```

Оформим такое использование свёртки в виде комбинатора неподвижной точки:

```
fix' f = foldr (λ_ → f) undefined [1..]

> testDW (fix' dropWhileNonRec)
True
> testFact (fix' factNonRec)
True
```

Итак, свёртка может использоваться в качестве комбинатора неподвижной точки: с её помощью можно выполнить необходимое количество вызовов нерекурсивной функции, моделируя таким образом рекурсию.

3.4. Свёртка как оператор примитивной рекурсии

3.4.1. Некоторые сведения из теории рекурсивных функций

Теория рекурсивных функций наряду с машинами Тьюринга и λ-исчислением является одним из подходов к формализации понятия вычислительного алгоритма. Основной принцип построения рекурсивных функций — рекурсивные определения, при которых значения функции для данных аргументов определяются через значения функции для «более простых» аргументов. При этом понятие «более простой» также нуждается в формализации.

Примитивно рекурсивные функции — это очень широкий и интересный класс всюду определённых функций, который можно получить с помощью рекурсивных определений. Основной задачей этого пункта будет построение класса примитивно рекурсивных функций произвольного числа переменных на множестве натуральных чисел (включая 0). В следующем пункте будет показано, как можно сформулировать понятие примитивно рекурсивной функции над списками.

Во-первых, введём набор простейших функций, из которых можно будет получить все остальные. К ним относятся:

1) Функция следования: $S(x) = x + 1$.

2) Нуль-функция: $\mathit{0}(x) = 0$.

3) Функции-проекторы: $I^n_m(x_1, \dots, x_n) = x_m$.

Функции-проекторы позволяют получить из набора аргументов аргумент с заданным номером. В дальнейшем наборы аргументов для краткости будем обозначать так:

$$(x_1, \dots, x_n) = \bar{x}.$$

Частным случаем функций-проекторов является тождественная функция: её можно записать так:

$$f(x) = x = I^1_1(x).$$

Во-вторых, необходимы *операторы*, позволяющие получать из одних функций другие. Первым из таких операторов является *оператор композиции*. С его помощью можно получить из функции f и набора функций g_i новую функцию φ, определяемую правилом:

$$\varphi(\bar{x}) = f(g_1(\bar{x}), \dots, g_k(\bar{x})).$$

Заметим, что оператор композиции позволяет выразить любые числовые константы (константные функции), для этого достаточно несколько раз применить функцию следования к нуль-функции, например:

$$5 = S(S(S(S(S(\mathit{0}(x)))))) = ((((0 + 1) + 1) + 1) + 1) + 1.$$

Здесь оператор композиции применён пять раз: первый раз он применяется к функциям S и $\mathit{0}$, а все остальные — к функции S и результату предыдущего применения.

Второй оператор называется *оператором примитивной рекурсии*. Он позволяет по заданным функциям g и h построить новую функцию ψ, определяемую следующими двумя правилами (*схема рекурсии*):

$$\begin{array}{lcl} \psi(\bar{x}, 0) & = & g(\bar{x}), \\ \psi(\bar{x}, y + 1) & = & h(\bar{x}, y, \psi(\bar{x}, y)). \end{array}$$

Важно, что рекурсия определяется только по одной переменной. Функция g задаёт базу рекурсии, а h — шаг рекурсии.

Теперь можно определить класс примитивно рекурсивных функций — это наименьшее множество, содержащее:

- функцию следования,
- нуль-функцию,
- все функции-проекторы,
- все функции, которые можно получить из вышеперечисленных с помощью *конечного* числа применений операторов композиции и примитивной рекурсии.

Как нам уже известно, к примитивно рекурсивным функциям относятся функции-константы и тождественные функции. Попробуем показать, что сложение также является примитивно рекурсивной функцией от двух аргументов. Понятно, что сложение полностью определяется следующими двумя правилами:

$$\begin{array}{lcl} x + 0 & = & x, \\ x + (y + 1) & = & (x + y) + 1. \end{array}$$

Эти правила можно записать в ином виде:

$$\begin{array}{lcl} +(x, 0) & = & I^1_1(x), \\ +(x, y + 1) & = & S(+(x, y)). \end{array}$$

Теперь видно, что функцию сложения можно представить как результат применения оператора примитивной рекурсии к функциям I^1_1 и S.

Похожим образом можно получить умножение, для этого достаточно заметить, что

$$\begin{array}{lllll} x \times 0 & = & 0 & = & \mathit{0}(x), \\ x \times (y+1) & = & (x \times y) + x & = & +(x, x \times y). \end{array}$$

Таким образом, функцию умножения можно представить как результат применения оператора примитивной рекурсии к функции $\mathit{0}$ и сложению.

Функция вычисления факториала также является примитивно рекурсивной, её можно получить так:

$$\begin{array}{lll} 0! & = & 1, \\ (n+1)! & = & (n+1) \times n! \end{array}$$

Примитивно рекурсивные функции не исчерпывают всё множество вычислимых функций. Классическим примером вычислимой функции, не являющейся примитивно рекурсивной, является функция Аккермана. Для расширения множества примитивно рекурсивных функций в рамках теории рекурсивных функций вводится третий *оператор минимизации*. Он позволяет построить по двум заданным функциям f_1 и f_2 новую функцию ω, определяемую правилом:

$$\omega(\bar{x}) = \min \{ y \,|\, f_1(\bar{x}, y) = f_2(\bar{x}, y) \} .$$

С его помощью строится множество *частично рекурсивных функций*, а после этого доказывается, что вычислительная способность этого множества полностью эквивалентна машине Тьюринга (и, соответственно, всем остальным полным вычислительным моделям). Частично рекурсивные функции могут не быть всюду определёнными, что соответствует зацикливанию машины Тьюринга на некоторых исходных данных (или отсутствию нормальной формы у некоторых λ-термов). Важно помнить, что слово «частично» в названии этих функций относится не к рекурсивности, а к их области определения. Всюду определённые частично рекурсивные функции называются *общерекурсивными*, они образуют строгое подмножество множества частично рекурсивных функций и надмножество примитивно рекурсивных. Более подробно с теорией рекурсивных функций можно познакомиться по книге В. И. Игошина [7], или в любой другой книге по математической логике и теории алгоритмов.

3.4.2. Примитивная рекурсия на списках

Теперь нас будут интересовать примитивно рекурсивные функции на списках. Место функции следования здесь занимает конструктор списка (операция присоединения элемента к списку):

$$S(x, xs) = x : xs,$$

а нуль-функция заменяется на пустой список:

$$\mathit{0}(xs) = [\,].$$

Роль функций-проекторов будет играть сопоставление с образцом, а вместо наборов аргументов будем использовать кортежи, обозначая их, как и прежде, $\bar{x}$. Оператор композиции — это стандартный механизм композиции функций, особого определения требует только оператор примитивной рекурсии:

$$\begin{array}{lll} \psi(\bar{x}, [\,]) & = & g(\bar{x}), \\ \psi(\bar{x}, y : ys) & = & h(\bar{x}, y, ys, \psi(\bar{x}, ys)). \end{array}$$

Следует обратить внимание на то, что в функции h добавился ещё один аргумент: вместо y, как у функций над множеством натуральных чисел, теперь используются y и ys (голова списка и его хвост). Такое изменение связано с тем, что в предыдущем случае получить при необходимости $y + 1$ из y можно легко, а вот извлечь из хвоста списка его голову несколько труднее. Заметим также, что ψ вызывается рекурсивно для хвоста списка.

Таким образом, оператор примитивной рекурсии позволяет из двух функций g и h указанного вида получить новую функцию над списками. Любая примитивно рекурсивная функция над списками как обычно строится из простейших с использованием конечного числа применений операторов композиции и примитивной рекурсии.

Дальнейшей целью этого пункта будет выражение оператора примитивной рекурсии через правую свёртку. В следующем пункте мы укажем соответствующие функции g и h для нескольких стандартных функций обработки списков, которые, как выяснится в результате, окажутся примитивно рекурсивными.

Итак, оператор примитивной рекурсии должен работать с тремя типами:

1) `par` — тип первого аргумента получаемой функции.
2) `val` — тип элемента списка (соответственно, тип самого списка — `[val]`).
3) `res` — тип результата обработки списка.

Он должен принимать в качестве аргументов две функции g и h следующих типов:

```
g :: par → res
h :: par → val → [val] → res → res
```

Наконец, возвращать он будет функцию ψ типа `par → [val] → res`.

Соберём все вместе и получим тип функции, реализующей оператор примитивной рекурсии:

```
primRec :: (par→res)
          → (par → val → [val] → res → res)
          → (par → [val] → res)
```

Заметим, что тип результата, вообще говоря, можно в скобки не заключать, так как результат функции зависит от того, как эту функцию вызывать: если передать ей вместо двух аргументов (функций) три (две функции и ещё один аргумент типа `par`), то получится функция, преобразующая список в результат его обработки (`[val] → res`). А если зафиксировать ещё и список, то получится константа — результат обработки типа `res`.

Идея реализации оператора примитивной рекурсии заключается в хранении на каждом шаге свёртки полной контекстной информации — результата последнего шага свёртки и обработанной к настоящему моменту части списка:

```
primRec g h x ys
   = fst (foldr combine base ys)
     where
       base = (g x, [])
       combine y (z, ys) = (h x y ys z, y:ys)
```

Мы не будем приводить доказательство корректности соответствующей функции: подробности можно уточнить в работе Хаттона [3]. Впрочем, корректность имеет место только до

© 2010 «Практика функционального программирования»

определённого предела: с бесконечными списками функции, определённые с помощью такого оператора примитивной рекурсии, работать не будут.

Сложности с реализацией оператора примитивной рекурсии связаны исключительно с ограниченностью используемых средств. Например, можно записать нерекурсивную версию этого оператора:

```
primRec' g h
    = fix funcNonRec
        where
          funcNonRec f x [] = g x
          funcNonRec f x (y:ys) = h x y ys (f x ys)
```

и подействовать на неё комбинатором неподвижной точки. Оператор примитивной рекурсии, полученный таким образом, будет лишён указанных выше недостатков. При этом можно даже воспользоваться комбинатором неподвижной точки, выраженным свёрткой. Впрочем, глубокого смысла в этих реализациях не будет, поскольку рекурсию они реализуют с помощью рекурсии.

3.4.3. Примеры реализации стандартных функций

Практически все стандартные функции обработки списков являются примитивно рекурсивными, поэтому их можно определить теми средствами, которые были предложены в предыдущем пункте.

Для задания функций g и h, используемых в схеме рекурсии, мы будем использовать либо константные функции (**const v** — функция, возвращающая значение v при любом значении аргумента), либо безымянные λ-функции. Для обозначения фиктивного аргумента (там, где он не требуется) будем использовать пустой тип **()** (тип с единственным значением, также обозначаемым как **()**).

Первым делом определим функцию, вычисляющую длину списка:

$$\begin{aligned} length([\,]) &= 0, \\ length(y:ys) &= 1 + length(ys). \end{aligned}$$

Здесь мы пытались написать правила для схемы рекурсии, но совершенно случайно почти дословно повторили обычное рекурсивное определение функции **length**. Теперь понятно, каковы должны быть функции g и h: первая из них есть константа 0, а вторая должна увеличивать значение своего последнего аргумента (результата рекурсивного вызова) на единицу, игнорируя все остальные. Дополнительный аргумент функции **length** не требуется, поэтому его можно заменить на **()**.

Окончательно получаем:

```
length' = primRec (const 0) (λ _ _ _ z → 1+z) ()
```

При желании можно ещё проверить тип этой функции и вызвать её для какого-нибудь списка:

```
> :type length'
length' :: [a] → Integer
> length' [1..100]
100
```

Совершенно аналогично пишутся функции, вычисляющие сумму и произведение элементов списка:

```
sum' = primRec (const 0) (λ _ y _ z → y+z) ()

product' = primRec (const 1) (λ _ y _ z → y*z) ()
```

Впрочем, эти функции довольно просты, попробуем реализовать что-нибудь более серьёзное, например, **map** и **filter**:

```
map' = primRec (const []) (λ f y _ z → f y : z)

filter' = primRec (const [])
            (λ predicate y _ z →
  if predicate y then z else y:z)
```

Здесь всё ещё проще: это функции с двумя аргументами, поэтому потребность в фиктивном аргументе **()** отпадает. Первые аргументы этих функций напрямую передаются функции h, используемой в схеме рекурсии, и уже в ней вызываются как функции.

Чуть больше сложностей возникает с функцией **minimum**. Во-первых, она не использует начальное значение, а во-вторых, должна сравнивать элементы списка между собой, поэтому механизму вывода типов требуется небольшая подсказка:

```
minimum' :: (Ord a) ⇒ [a] → a
minimum' = primRec (const undefined)
            (λ _ y ys z → if null ys
                            then y
                            else min y z) ()
```

Нежно любимая нами функция **dropWhile** также является примитивно рекурсивной, поэтому и для неё есть соответствующее определение:

```
dropWhile'' = primRec (const [])
               (λ predicate y ys z → if predicate y
                                       then z
                                       else y:ys)

> testDW dropWhile''
True
```

Теперь можно сравнить функцию **dropWhile** и её первую неправильную реализацию, оказавшуюся реализацией **filter**, в их исконном, теретико-алгоритмическом (рекурсивно-функциональном), виде:

```
dropWhile'' = primRec (const [])
               (λ predicate y ys z → if predicate y
                                       then z
                                       else y:ys)

filter' = primRec (const [])
               (λ predicate y  _ z → if predicate y
                                       then z
                                       else y:z)
```

Наконец, ничто не мешает определить саму функцию **foldr**:

```
foldr' = curry (
            primRec (λ (_, base) → base)
                (λ (op, _) y _ z → op y z))
```

Функция **foldr** принимает три аргумента вместо двух допустимых, поэтому первые два пришлось объединить в кортеж. Чтобы пользователь ничего не заметил, функцию, полученную в результате действия оператора примитивной рекурсии, необходимо каррировать, т. е. преобразовать из функции, принимающей пару, в функцию двух аргументов. Эту задачу выполняет стандартная функция **curry**.

Пользуясь выведенной свёрткой, можно попробовать снова реализовать **dropWhile**, взяв за основу, к примеру, реализацию с контекстом из двух списков:

```
dropWhile''' predicate list =
       snd (foldr' combine base list)
       where
         base = ([], [])
         combine x (xs, ys)
               | predicate x = (x:xs, ys)
               | otherwise = (x:xs, x:xs)

> testDW dropWhile'''
True
```

Итак, свёртка реализует оператор примитивной рекурсии, что позволяет выразить с её помощью большое количество примитивно рекурсивных функций, в том числе практически все стандартные функции обработки списков.

3.5. Заключение

Теперь можно окончательно сформулировать пределы выразительности свёрток: любая примитивно рекурсивная функция на списках может быть реализована с помощью свёртки без использования явной рекурсии. Более того, если вместе со свёрткой использовать бесконечный список, то таким образом можно выразить вообще любую частично рекурсивную функцию (это факт следует из наличия реализации комбинатора неподвижной точки в терминах свёрток и эквивалентности теории рекурсивных функций и λ-исчисления).

Следует заметить, что само использование свёртки в качестве комбинатора неподвижной точки берёт начало из теоретико-категорной работы Питера Фрейда [2], в которой он замечает, что такой комбинатор может быть выражен в виде композиции анаморфизма и катаморфизма. В работе Мейера и Хаттона [4] результаты Фрейда переводятся на язык функционального программирования и формулируются для произвольных алгебраических типов данных.

В случае списков катаморфизм оказывается свёрткой, а анаморфизм — функцией **List.unfoldr**. Ясно, что из неё легко получить бесконечный список, который мы использовали при реализации комбинатора неподвижной точки:

```
> unfoldr (λx → Just (x, x+1)) 1
[1,2,3,4,...
```

Впрочем, необходимо иметь в виду, что на практике выражение всех рекурсивных функций в терминах свёрток вряд ли полезно: ясно, что замена рекурсии на свёртку может привести к сильному усложнению кода.

Литература

[1] *Bird R. S.* Introduction to Functional Programming Using Haskell (second edition). — Prentice-Hall, 1998. http://www.comlab.ox.ac.uk/oucl/publications/books/functional/.

[2] *Freyd P.* Algebraically complete categories. — Proceedings of the 1990 Como Category Theory Conference, volume 1488 of Lecture Notes In Math, pages 95–104. Springer-Verlag. — 1990.

[3] *Hutton G.* A Tutorial on the Universality and Expressiveness of Fold. — Journal of Functional Programming, volume 9, number 4, pages 355–372, URL: http://www.cs.nott.ac.uk/~gmh/fold.pdf (дата обращения: 25 февраля 2010 г.). Cambridge University Press. — 1999.

[4] *Meijer E., Hutton G.* Bananas in space: Extending fold and unfold to exponential types. — Proceedings of the 7th SIGPLAN-SIGARCH-WG2.8 International Conference on Functional Programming and Computer Architecture, pages 324–333, URL: http://www.cs.nott.ac.uk/~gmh/bananas.pdf (дата обращения: 25 февраля 2010 г.). ACM Press. — 1995.

[5] *Pope B.* Getting a fix from the right fold. — The Monad Reader, Issue 6, URL: http://www.haskell.org/sitewiki/images/1/14/TMR-Issue6.pdf (дата обращения: 25 февраля 2010 г.). — 2007.

[6] *Ruehr F.* The Evolution of a Haskell Programmer. — URL: http://www.willamette.edu/~fruehr/haskell/evolution.html (дата обращения: 25 февраля 2010 г.).

[7] *В. И. Игошин.* Математическая логика и теория алгоритмов. — М.: Академия, 2008.

[8] *Евгений Кирпичёв.* Элементы функциональных языков // — *Практика Функционального Программирования*2009. — декабрь. — Т. №3.

[9] Русский перевод курса лекций Джона Харрисона «Introduction to Functional Programming». — Страница проекта, URL: http://code.google.com/p/funprog-ru/ (дата обращения: 25 февраля 2010 г.).

 © 2010 «Практика функционального программирования»

Мономорфизм, полиморфизм и экзистенциальные типы

Роман Душкин
darkus@fprog.ru

Аннотация

В статье описываются экзистенциальные типы данных и их применение в функциональной парадигме программирования. Даются определения необходимых понятий — мономорфизма, полиморфизма (высших рангов) и их связи с экзистенциальными типами. Примеры определений типов и функций для их обработки приводятся в основном на языке программирования Haskell.

Article describes the notion of existential types and related theoretical concepts (monomorphism, polymorphism, etc), with examples in Haskell.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/8541.html.

4.1. Введение

Настоящая статья продолжает цикл публикаций, посвящённых различным свойствам и аспектам систем типизации, используемых в функциональных языках программирования. Желательно, чтобы перед прочтением данной статьи читатель ознакомился с предыдущими материалами [6, 7].

Одним из наиболее интересных и важных понятий в парадигме функционального программирования является *параметрический полиморфизм*. В статье [7] дана классификация подвидов параметрического полиморфизма, основанная на работах [1, 5]. В целях удобства имеет смысл привести основные определения здесь, несколько расширив их.

При декларациях параметрических полиморфных типов применяются связанные *типовые переменные* (или *переменные типов*). При дальнейшем использовании таких полиморфных типов типовые переменные конкретизируются определёнными типами — происходит подстановка значения переменной в виде конкретного типа (примерно так же, как происходит подстановка конкретного значения в λ-терм при β-редукции). Другими словами, типовые переменные могут рассматриваться в качестве формальных параметров, которые получают значения аргументов в процессе конкретизации.

В зависимости от того, какие значения могут принимать типовые переменные, различают два подвида параметрического полиморфизма:

1) *Предикативный* параметрический полиморфизм проявляется тогда, когда в качестве значений типовых переменных могут быть подставлены только мономорфные типы («монотипы»), то есть такие типы, в определениях которых не используются типовые переменные.

2) *Непредикативный* параметрический полиморфизм позволяет конкретизировать типовые переменные произвольными типами (мономорфными и полиморфными), в том числе и рекурсивно: если в определении типа τ используется типовая переменная α, то при конкретизации вместо неё может быть подставлен сам тип τ. В данном случае слово «непредикативный» обозначает, что при определении сущности *возможно* использование ссылки на саму определяемую сущность, что потенциально может привести к парадоксам типа канторовского и расселовского. Иногда непредикативный полиморфизм называют полиморфизмом *первого класса*.

Существует еще одна система классификации параметрических полиморфных типов, основаная на понятии ранга. В данной классификации выделяют три подвида параметрического полиморфизма:

1) Полиморфизм *ранга* 1 (также *предварённый* полиморфизм или *let*-полиморфизм) предполагает, что в качестве типовых переменных могут быть подставлены только монотипы. Соответственно, предварённый полиморфизм может быть только предикативным. В языках семейства ML (включая язык Haskell стандарта 98 года без расширений) используется именно этот вид параметрического полиморфизма.

2) Полиморфизм *ранга* k (при $k > 1$) позволяет конкретизировать типовые переменные полиморфными типами ранга не выше (k - 1). В [2] доказано, что вывод типов возможен для k = 2, но для больших k задача вывода типов неразрешима.

3) Полиморфизм *высшего ранга* (полиморфизм *ранга* N) проявляется тогда, когда конкретизация типовых переменных может производиться произвольными типами, в том числе и полиморфными любого ранга. Непредикативный полиморфизм высшего ранга является наиболее общим видом параметрического полиморфизма.

В языках функционального программирования со статической типизацией чаще всего используется предикативный предварённый полиморфизм, так как он проще всего обрабатывается при автоматическом выводе типов. Этот вид параметрического полиморфизма был впервые реализован в языке программирования ML, после чего успешно показал себя в других языках. В частности, как уже упомянуто, стандарт Haskell-98 использует именно его.

Тем не менее, этот подвид не позволяет решать часто возникающие в программировании задачи, связанные, например, с хранением в контейнерных типах данных значений различных типов. В языках с динамической типизацией, таких как LISP, нет никаких проблем в создании *списочных структур*, элементами которых могут быть как *атомы*, так и другие списочные структуры. При этом в качестве атомов могут одновременно выступать значения разных типов — и значения истинности, и символы, и значения численных типов. Но подобные структуры данных невозможны в языке Haskell стандарта Haskell-98. Тип `[a]` при конкретизации (получении конкретного значения типовой переменной `a`) сможет содержать значения только того типа, которым конкретизирована типовая переменная. Невозможно создать список, в котором одновременно будут содержаться и числа, и символы, и иные произвольные значения.

Для преодоления этого ограничения компилятор GHC предлагает расширения языка, позволяющие использовать так называемые *экзистенциальные типы* (от англ. «*existential types*»). Эти типы и являются предметом рассмотрения данной статьи.

4.2. Пара нетривиальных задач для статически типизированных языков

Статическая типизация предполагает, что типы всех выражений, используемых в программе, известны на этапе компиляции. При этом типы могут быть либо явно указаны разработчиком, либо вычислены самим компилятором при помощи механизма вывода типов (с автоматическим выводом типов можно ознакомиться в [9], где также имеется обширный список литературы по данной тематике). Это накладывает ряд ограничений на способы определения структур данных и функций для их обработки.

Одна из главных трудностей, с которой сталкивается разработчик на языке программирования ML, Haskell или подобном, заключается в невозможности создать так называемые *неоднородные типы* (или, как их ещё называют, *гетерогенные типы*, от англ. «*heterogeneous types*»). Неоднородными типами называются такие типы, которые могут содержать в себе значения разных типов. Например, неоднородные списки могут одновременно содержать целые числа, символы, другие списки, деревья и т. д., неоднородные двоичные деревья могут содержать в своих вершинах значения совершенно произволь-

© 2010 «Практика функционального программирования»

ных типов. При этом надо отметить, что разные значения находятся в рамках одного конструктора, то есть по своей сути гетерогенные типы уже не являются «классическими» алгебраическими типами данных, как они описаны в [6].

Как упомянуто во введении, в языках с динамической типизацией создание таких типов данных не является проблемой. Язык LISP изначально оперирует именно такими структурами данных. В языке программирования JavaScript также можно без проблем определять массивы, каждый элемент которых имеет тип, отличающийся от типов других элементов.

К сожалению, функциональные статически типизированные языки программирования не позволяют просто так создавать неоднородные типы. Например, в языке Haskell сигнатура

```
f :: [a] → a
```

представляет собой сигнатуру полиморфной функции. При вызове такой функции происходит конкретизация типа `a` и все экземпляры типовой переменной в этой сигнатуре получают одно и то же значение. Другими словами, первый компонент сигнатуры `[a]` представляет собой список значений типа `a`, поэтому при конкретизации типа все элементы списка будут обязаны иметь один и тот же тип. Это требование становится понятным, если вспомнить определение списка[1]:

```
data [a] = []
         | a : [a]
```

При конкретизации типовой переменной `a` происходит подстановка конкретного типа во все места, где эта переменная используется. Так получается однородный список, элементы которого имеют один и тот же тип. Такие типы называют *мономорфными* или *монотипами*. Даже если использовать списки списков, списки деревьев и тому подобные составные контейнерные типы данных, конкретизация типовых переменных всегда будет приводить к тому, что на любом уровне вложенности будут использоваться одни и те же конкретизированные типы.

Неоднородные структуры данных используются для решения широкого спектра задач, сводимых к диспетчеризации потока разнотипных программных сущностей. Например, из потокового файла или сетевого потока последовательно принимаются записи об определённых объектах, над каждым из которых необходимо провести некоторые действия. Каким образом диспетчеризатор должен понимать, какую именно функцию вызвать для обработки очередного объекта, полученного из потока?

Хорошо, если все типы таких программных сущностей составляют достаточно ограниченное множество, тогда на языке Haskell можно определить алгебраический тип данных, каждый из конструкторов которого отвечает за одну «разновидность» обрабатываемой сущности. Но как быть, если функции-обработчики необходимо создавать динамически? И каким образом в изначально однородный список положить неоднородные данные, типы которых могут быть неизвестны на стадии компиляции?

Ответы на эти вопросы в процедурных и объектно-ориентированных языках программирования несложны. В языке C можно просто использовать список указателей на `void`, а при получении очередного объекта через такой указатель вызывать необходимую функцию для его обработки на основании некоторой метаинформации, приходящей с каждым объектом. В объектно-ориентированных языках программирования для этих целей имеется специальный метод, который так и называется — *динамическая диспетчеризация* [4].

В качестве примера можно привести задачу вывода на экран географической карты. Файл с описанием карты представляет собой поток разнообразных фигур — точек, линий, полигонов, эллипсов и т. д. Решение этой задачи на языке C++ обычно приводят в учебниках по этому языку в части, описывающей механизм наследования. Создаётся базовый класс `TFigure`, у которого имеется виртуальный метод `render`. От класса `TFigure` наследуются разнообразные классы `TPoint`, `TLine`, `TPolygon`, `TEllipse` и т. д. по количеству используемых для описания карты фигур. В функцию-диспетчеризатор передаётся список значений типа `TFigure`, для каждой из которых вызывается метод `render`, а механизм динамической диспетчеризации позаботится о том, чтобы был вызван правильный метод, связанный с конкретным типом фигуры.

Другой задачей, где традиционные строго типизированные функциональные языки терпят фиаско по сравнению с императивным подходом, является работа с изменяемым состоянием. Опасности изменяемого состояния, а также некоторые методы исключения таких опасностей из разрабатываемых программных продуктов приведены в статье [8]. Тем не менее, некоторые алгоритмы не могут быть эффективно реализованы без изменяемых состояний, а некоторые задачи вообще могут быть «извращены» при помощи использования неизменяемых программных сущностей.

Работа с изменяемыми состояниями в процедурных и объектно-ориентированных языках программирования не представляет никакой сложности, поскольку в самой императивной парадигме одним из главных методов работы с объектами является так называемое «деструктивное присваивание» — прямая замена содержимого ячейки памяти, на которую ссылается объект. В императивной парадигме программирования операция присваивания значения является столь широко используемой, что разработчик не задумывается о том, сколько раз и как часто он меняет состояния используемых в его программном продукте объектов.

Чистые функциональные языки не позволяют осуществлять деструктивное присваивание. Каждый раз, когда надо изменить какую-либо программную сущность, в памяти делается её копия с внесёнными изменениями, а о старой копии сущности, если она больше не нужна, позаботится сборщик мусора. Данный подход имеет свои несомненные достоинства, но существует и ряд недостатков. В частности, подсистема ввода-вывода, без которой не может обойтись ни один универсальный язык программирования, заведомо нарушает принципы чистоты и детерминизма функций.

Из-за необходимости реализации системы ввода-вывода в языке программирования Haskell появляется «неуклюжая» монада **IO**, которая обеспечивает ввод и вывод, то есть вводит в язык Haskell недетерминированные функции ввода и функции вывода с побочными эффектами. Получается противоречивая ситуация: язык Haskell идеологически объявлен чистым (то есть недопускающим недетерминизма и использования побочных эффектов), но в то же время подсистема ввода-вывода необходима ему для обеспечения универсально-

[1] Необходимо напомнить, что данное определение некорректно с точки зрения синтаксиса языка Haskell, оно «вшито» в язык в подобном виде. Разработчик не может создавать собственные определения типов подобного вида.

сти. Из-за этого монада **IO** в языке Haskell сделана обособленной — ввод и вывод может осуществляться только в ней. В других функциональных языках программирования подсистемы ввода-вывода также основаны на каких-нибудь допущениях.

Далее в статье будут рассмотрены приёмы использования экзистенциальных типов, дающие ответы на следующие вопросы:

1) Каким образом можно определить и в дальнейшем использовать неоднородные типы данных?

2) Что можно сделать для создания в рамках функциональной парадигмы безопасных механизмов работы с изменяемым состоянием, которые в то же время оставались бы в рамках чистоты и детерминизма?

4.3. Экзистенциальные типы и их использование в языке Haskell

Как же экзистенциальные типы позволяют ответить на поставленные в предыдущем разделе вопросы?

4.3.1. Неоднородные типы

Неоднородные контейнерные структуры данных (неоднородные типы) позволяют хранить значения различных типов, никак не связанных друг с другом. Задача создания таких структур данных достаточно легко решается в объектно-ориентированном стиле при помощи механизма наследования и динамической диспетчеризации. Но эта же задача оказывается нетривиальной для функциональной парадигмы.

В языке Haskell, например, несложно определить нужное количество алгебраических типов данных, каждый из которых является экземпляром какого-нибудь одного специального класса типов. Этот общий класс предоставит для типов общие интерфейсные методы. Однако задачи это не решит — каким образом упаковать значения этих различных типов в один контейнер? Мономорфный тип этого не позволит, поскольку при конкретизации типовой переменной будет выбран только один тип из всего множества, пусть хотя бы каждый тип из этого множества имеет экземпляр одного и того же класса.

Чтобы понять суть монотипов, необходимо более внимательно рассмотреть сигнатуры, в которых используются типовые переменные. Что означает следующая запись?

```
id :: a → a
```

В предыдущей статье цикла [7] уже показана математическая формула для этой функции, а именно:

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

Здесь используется квантор всеобщности (∀), присутствующий в нотации языка Haskell неявно (в действительности надо было бы написать так: `id :: forall a · a → a`).

В действительности транслятор языка Haskell автоматически производит универсальную квантификацию каждой типовой переменной, встречающейся в сигнатуре функции. Квантор всеобщности для каждой используемой типовой переменной вставляется транслятором языка перед всей сигнатурой, именно поэтому невозможно создание неоднородных типов — раз квантор всеобщности стоит перед всей формулой, связанная им типовая переменная должна конкретизироваться одинаковым типом во всей формуле.

Что было бы, если бы была возможность ставить квантор всеобщности в произвольном месте сигнатуры, связывая типовую переменную только в определённой её части? Это позволило бы не упоминать её в сигнатуре самого алгебраического типа.

Теперь можно обратиться к задаче из предыдущего раздела о выводе на экран географической карты. Нет необходимости определять единый алгебраический тип данных наподобие следующего:

```
data Figure = Point (Int, Int)
            | Line (Int, Int) (Int, Int)
            | Polygon [(Int, Int)]
            | Ellipse (Int, Int) Int Int
```

Это решение было бы крайне плохо масштабируемо, поскольку для добавления поддержки нового типа фигур пришлось бы дополнять этот тип, что приводило бы к необходимости значительного рефакторинга всего программного проекта. Вместо этого имеет смысл определить класс, содержащий функцию `render` (результатом выполнения которой может являться отображение фигуры на экране):

```
class Figure a where
  render :: a → IO ()
```

Для работы с этим классом необходимо подготовить набор специальных алгебраических типов данных и экземпляров класса `Figure` для них.

```
data Point = Point (Int, Int)

instance Figure Point where
  render (Point (x, y)) = ...
```

И так для каждого типа фигур, составляющих карту и выводимых на экран. Это решение уже является вполне естественным для языка Haskell и хорошо масштабируется — в любой момент можно определить дополнительные типы фигур и реализовать для них экземпляры класса `Figure`.

Но как же быть с организацией неоднородной структуры данных? Здесь и приходят на помощь экзистенциальные типы. Для создания неоднородного списка, который будет содержать все считанные из файла геометрические фигуры, необходимо определить экзистенциальный тип-обёртку:

```
data GeoObject = forall a · Figure a ⇒ GeoObject a
```

Квантор всеобщности, обозначаемый в языке Haskell ключевым словом `forall`, скрывает типовую переменную `a` из области видимости всего типа `GeoObject`, но при этом указывает, что тип для конкретизации может быть произвольный, главное, чтобы он являлся экземпляром класса `Figure`. Это ограничение на наличие экземпляра важно, что будет показано впоследствии.

Ключевое слово `forall` не входит в стандарт Haskell-98, поэтому для его использования необходимо подключить дополнительные языковые расширения, например, так:

```
{-# LANGUAGE ExistentialQuantification #-}
```

Теперь при помощи типа `GeoObject` можно определить неоднородный список объектов, составляющих географическую карту:

```
geolist :: [GeoObject]
geolist =
  [GeoObject (Point (0, 0)),
```

```
    GeoObject (Line (1, 1) (2, 2)),
    GeoObject (Polygon [(1, 1), (1, −1), (−1, −1), (−1, 1)]),
    GeoObject (Ellipse (0, 0) 1 1)]
```

Таким образом можно собирать неоднородный список из описаний структур данных, считываемых динамически, например, из файла. Для обработки такого списка можно без проблем построить функцию-диспетчеризатор:

```
dispatch :: [GeoObject] → IO ()
dispatch [] = return ()
dispatch ((GeoObject g):gs) = do render g
                                 dispatch gs
```

Или ещё проще:

```
dispatch :: [GeoObject] → IO ()
dispatch = sequence_ · map render
```

Как теперь видно, благодаря использованию экзистенциального типа из функции `dispatch` исчезает типовая переменная. Но сама по себе эта функция уже не может считаться мономорфной — типовая переменная вместе с квантором всеобщности скрыта в определении экзистенциального типа `GeoObject`. Таким образом функция `dispatch` оказывается полиморфной функцией ранга 2, поскольку её параметром является полиморфный объект ранга 1 — список элементов, каждый из которых сам по себе является полиморфным объектом ранга 0, то есть имеет монотип.

Функция `dispatch` не знает, значения каких типов могут содержаться в качестве объектов неоднородного списка типа `[GeoObject]`. Это общее свойство полиморфизма — использующие программные сущности не могут знать типов используемых объектов (поскольку эти типы вообще могут быть не конкретизированы в случае полиморфизма высших рангов). Поэтому всё, что может делать функция с получаемыми на вход объектами неизвестных типов, это применять к ним какие-либо заведомо применимые функции. Именно поэтому важно либо иметь общий интерфейс для таких типов (ограничение на наличие экземпляра одного и того же класса), либо на вход диспетчеризатору фактическим параметром передавать функцию для обработки значений соответствующего типа.

4.3.2. Работа с изменяемыми состояниями в функциональном стиле

Один из «отцов-основателей» языка Haskell С. Л. Пейтон-Джонс в своё время озаботился трудностями манипулирования изменяемым состоянием в этом языке и разработал монаду `ST` (от англ. «State Transformer» — преобразователь состояний), совмещающую в себе свойства монад **`IO`** и `State`. Детально библиотека, реализующая данную монаду, описана в работе [3].

В контексте данной статьи монада `ST` интересна тем, что реализует работу с изменяемым состоянием, но в совершенно функциональном стиле, сохраняя все важные свойства и принципы функциональной парадигмы: независимость результата от порядка вычислений, ссылочную прозрачность, ленивость, возможность извлечения объектов из монады `ST` (то есть, возможность получить «чистое» значение в результате вычисления с состоянием в монаде `ST`). Как видно, эти свойства вполне отвечают требованиям, поставленным в предыдущем разделе, и позволяют заявить, что в языке Haskell имеется чисто функциональный инструмент для эффективного решения императивных задач.

В цели настоящей статьи не входит подробное описание монады `ST`. Интересен другой её аспект — реализация монады `ST` основана на параметрическом полиморфизме ранга 2.

Тип «*действия*» (от англ. «action») в монаде `ST` таков:

```
ST s a
```

Здесь тип `a` возвращается «действием» типа `s`. Самое интересное — сигнатура функции, позволяющей извлечь значение из монады `ST`:

```
runST :: (forall s · ST s a) → a
```

Как видно, в этой сигнатуре используется параметрический полиморфизм ранга 2. Квантор всеобщности для типовой переменной `s` находится не на самом верхнем уровне, как для типовой переменной `a` (для неё квантор всеобщности в сигнатуре не показан, поскольку он подразумевается), а на один уровень ниже. Это скрывает типовую переменную `s`, а используемые для работы конкретные типы становятся неизвестны вне монады `ST`, что и обеспечивает безопасность типизации.

Можно рассмотреть простой чисто учебный пример[2]. Следующая функция вычисляет произведение элементов заданного списка.

```
import Data.STRef
import Control.Monad
import Control.Monad.ST

productST :: Num a ⇒ [a] → a
productST xs =
  runST $ do n ← newSTRef 1
             mapM_ (λx → modifySTRef n (* x)) xs
             readSTRef n
```

Вызов функции `productST` приводит к выполнению последовательности из трёх монадических действий, записанных после ключевого слова **`do`**. Результат выполнения этой последовательности находится в монаде `ST`, поэтому для получения «чистого» значения из этой монады используется функция `runST`. Сама функция `productST` ничего не знает о том, как именно и с какими возможными побочными эффектами производятся вычисления внутри монады `ST`, всё это сокрыто от неё посредством использования экзистенциального типа и параметрического полиморфизма ранга 2.

Последовательность монадических действий в функции `productST` написана во вполне императивном духе. Первое действие — запись в именованное состояние («переменную») `n` начального значения `1`. Далее производятся циклические вычисления при помощи функции **`mapM_`**, для каждого элемента `x` из списка `xs` выполняется действие `modifySTRef n (*x)`. Это действие модифицирует значение «переменной» `n` при помощи заданной функции (в данном случае — `(* x)`, то есть операции умножения на очередное значение из списка). Императивный аналог этого фрагмента кода — `n = n×x`.

Третье монадическое действие считывает значение, хранящееся в «переменной» `n`, и возвращает его. Далее это значение, обёрнутое монадой `ST`, попадает на вход функции `runST`, которая, как уже сказано, возвращает его в чистом виде.

Чтобы понять способ выполнения циклических монадических действий, имеет смысл привести определения используемых для этого функций (благо они совершенно краткие):

[2] Пример этот, конечно, искусственный — реализовать подобную функцию очень легко и без использования монады `ST`. Тем не менее, он иллюстрирует использование императивного стиля в функциональном языке.

```
mapM_ :: Monad m ⇒ (a → m b) → [a] → m ()
mapM_ f xs = sequence_ (map f xs)

sequence_ :: Monad m ⇒ [m a] → m ()
sequence_ xs = foldr (≫) (return ()) xs
```

Здесь функции **map** и **foldr** являются стандартными. Знак подчёркивания (_) в наименованиях монадических функций указывает на то, что функции игнорируют результаты вычислений, но используют только побочные эффекты выполняемых монадических действий. Функция **sequence_** выполняет последовательность монадических действий, хранящихся в заданном списке. Функция **mapM_** строит такой список при помощи заданной функции `f` и исходного списка, после чего передаёт построенный список монадических действий функции **sequence_**.

Теперь читатель вполне готов реализовать другие функции обработки списков в подобном квази-императивном стиле.

Таким образом использование в определении монады `ST` экзистенциального типа позволяет решить важную задачу — сокрытие реального типа действий, используемых в монаде. Обработка этих действий может производиться только средствами, предоставленными монадой (функциями, написанными её разработчиками). Это значит, что налицо определение *абстрактного типа данных*. И действительно, экзистенциальные типы в данном отношении используются в качестве инструмента повышения уровня абстрактности — реальный тип используемых выражений неизвестен, а потому их обработка может вестись только посредством имеющегося в наличии «интерфейса». Если обратиться к более раннему примеру с динамической диспетчеризацией, то в нём это свойство экзистенциальных типов проявляется ещё нагляднее, поскольку определённый класс `Figure` предоставляет интерфейс для обработки неоднородного списка.

4.4. Заключение

В данной статье лишь кратко рассказано про отдельные аспекты применения параметрического полиморфизма высших рангов и экзистенциальных типов в языке программирования Haskell. Предложенные примеры демонстрируют решения задач, которые не могут быть эффективно решены традиционными методами функционального программирования (использование неоднородных структур данных и работа с изменяемым состоянием в функциональном стиле). Это открывает читателю новые направления изучения систем типов, используемых в информатике как в теории, так и на практике.

Дополнительно надо отметить, что одной из важнейших теоретических основ для изучения и использования параметрического полиморфизма является так называемая «*Система F*». Этому формализму будет посвящена одна из грядущих статей.

Автор благодарит своих коллег Д. А. Антонюка и Е. Р. Кирпичёва за помощь, оказанную ими при подготовке данной статьи.

Литература

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

[2] *Kfoury A. J. and Wells J. B.* Principality and decidable type inference for finite-rank intersection types // *ACM Symposium on Principles of Programming Languages (POPL)*. — 1999. — Pp. 161–174.

[3] *Launchbury J. and Peyton-Jones S. L.* Lazy functional state threads // *Programming Languages Design and Implementation*. — 1994.

[4] *Lippman S. B.* Inside the C++ Object Model. — Addison-Wesley, 1996.

[5] *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: 25 февраля 2010 г.)). http://www.cis.upenn.edu/~bcpierce/tapl.

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

[7] *Душкин Р. В.* Полиморфизм в языке Haskell // *«Практика функционального программирования»*. — 2009. — Т. 3, № 3. — С. 67–81.

[8] *Кирпичёв Е. Р.* Изменяемое состояние: опасности и борьба с ними // *«Практика функционального программирования»*. — 2009. — Т. 1, № 1. — С. 29–49.

[9] *Кирпичёв Е. Р.* Элементы функциональных языков // *«Практика функционального программирования»*. — 2009. — Т. 3, № 3. — С. 83–196.

 © 2010 «Практика функционального программирования»

Сечения композиции как инструмент бесточечного стиля

Денис Москвин
dmoskvin@fprog.ru

Аннотация

В статье обсуждается роль оператора композиции функций и его левого и правого сечения при программировании на языке Haskell в так называемом «бесточечном» стиле. Показано, что разнообразные композиции функции нескольких аргументов с функцией одного аргумента единообразно выражаются через последовательности таких сечений.

We discuss the function composition operator and its left and right sections in point-free style Haskell programming. We show that the variety of compositions of unary and n-ary functions are expressible as a sequence of such sections.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/8243.html.

5.1. Введение

Существуют два совершенно эквивалентных по возможностям исчисления, образующих теоретическую основу функционального программирования. Это лямбда-исчисление и комбин*а*торная логика. В первом случае функции не имеют имён — они описываются как преобразования над их аргументами. Во втором случае, наоборот, функции имеют имена, но не имеют аргументов — функции описываются как применения базовых комбинаторов.

Программирование на прикладных функциональных языках представляет собой некоторый компромисс между этими крайностями. Бесточечный стиль программирования тяготеет к комбин*а*торной логике. Для него зачастую характерна удивительная краткость записи. Многие, однако, жалуются на низкую читабельность программ записанных в этом стиле. Хотя некоторую часть таких претензий можно признать обоснованными, тем не менее во многих ситуациях бесточечная запись может ясно и лаконично выразить мысль программиста [2, 3].

Бесточечный стиль программирования уже обсуждался на страницах журнала [5]. Напомним, что первое систематическое описание такого подхода к программированию дано в знаменитой лекции Джона Бэкуса «Can programming be liberated from the Von Neumann style?» [1]. На практике, помимо Haskell, бесточечный стиль используют в языках семейства ML и в наследниках APL — языках J и K.

В данной статье мы продемонстрируем важную роль оператора композиции при написании бесточечных программ на языке Haskell. (Отметим, что в комбин*а*торной логике этот оператор также играет существенную роль, являясь базовым комбинатором для многих систем этой логики.) Мы рассмотрим применение левого и правого сечений оператора композиции для работы с функциями нескольких аргументов в бесточечном стиле. Будет показано, что разнообразные композиции вызовов функции нескольких аргументов с функцией одного аргумента единообразно выражаются через последовательности таких сечений.

Здесь стоит отметить, что в статье мы используем термин «функция n аргументов» для фактического обозначения функции n или большего числа аргументов. Для Haskell такой подход вполне допустим, поскольку «лишние» аргументы можно «спрятать» с помощью частичного применения функции. Например, утверждение, что функция **zipWith ::** `(a → b → c) → [a] → [b] → [c]` требует функции двух аргументов в качестве своего первого аргумента, не может рассматриваться как запрет на вызов **zipWith** `(λx y z → x - y + z)`. Хотя лямбда-функция здесь явно задана как функция трёх аргументов (её тип **Int** → **Int** → **Int** → **Int**), но в данном вызове она используется как функция двух аргументов с типом **Int** → **Int** → (**Int** → **Int**).

Также следует отметить, что мы будем использовать символ равенства = как для определения функций, так и для выражения эквивалентности двух конструкций, допустимых в синтаксисе Haskell. Обычно тип такого использования будет оговариваться, если только он не ясен из контекста.

В тексте статьи имеются два отступления — про функторы и про контрафункторы. Они не являются необходимыми для понимания основной идеи — при первом прочтении их можно безболезненно пропустить. Однако они позволяют наметить связь между рассматриваемыми здесь сечениями оператора композиции и некоторыми конструкциями теории категорий.

5.2. Бесточечный стиль: сечения и композиция

Напомним, что при программировании в бесточечном стиле мы опускаем аргументы (точки применения) функции в левой и в правой частях ее определения. То есть вместо

```
sum :: (Num a) ⇒ [a] → a
sum xs = foldr (+) 0 xs
```

в бесточечном стиле мы пишем

```
sum :: (Num a) ⇒ [a] → a
sum = foldr (+) 0
```

Такое отбрасывание аргументов допустимо, только если самый правый аргумент в левой части является самым правым и в правой части (и при этом больше нигде в правой не используется). В нашем примере это аргумент `xs`.

К сожалению, подобная ситуация, удобная для реализации бесточечной версии определения функции, встречается довольно редко. Чаще самый правый из аргументов функции используется не в правой позиции, а внутри некоторого выражения. Здесь нам приходит на помощь оператор композиции функций

```
( · ) :: (b → c) → (a → b) → a → c
(f · g) x = f (g x)
```

Прочитав это определение в обратную сторону, легко увидеть, что у нас есть способ «вытащить» аргумент `x` из скобок. Например, для

```
foo x = bar (baz (qux x))
```

преобразования, использующие это определение,

```
bar (baz (qux x))
= bar ((baz · qux) x)
= (bar · (baz · qux)) x
```

позволяют перейти к эквивалентному бесточечному определению

```
foo = bar · baz · qux
```

Проиллюстрированный этим примером метод перехода от аппликаций к композиции является одним из основных при построении бесточечных определений функций. Именно поэтому для бесточечного стиля характерно очень широкое использование оператора композиции.

Работа с подобными композиционными цепочками выглядит несложной, когда мы имеем дело с функциями одного аргумента. Эта же техника легко обобщается на выражения, в которых функции нескольких аргументов приведены к «одноаргументному» виду частичными применениями. Например,

```
f = product · take 3 · map succ
```

перемножает (**product**) первые три элемента списка (**take** 3), предварительно увеличенных на единицу (**map succ**):

```
> f [1,2,3,42]
24
```

Бесточечное программирование возможно не только для функций, но и для операторов. Специфика бесточечного стиля для операторов связана с понятием сечений, обсуждавшемся, например, в [5]. Например, для функции

```
toSeconds min sec = min × 60 + sec
```

«частично» бесточечная[1] версия

```
toSeconds min = (min × 60 +)
```

базируется на эквивалентности

```
min × 60 + sec
= (min × 60 +) sec
```

Скобки здесь обязательны, поскольку именно они задают левое сечение оператора сложения.

Чтобы продолжить освобождение от аргументов, можно выполнить следующие эквивалентные преобразования

```
(min × 60 +)
= (+) (min × 60)          -- (1)
= (+) (( × 60) min)       -- (2)
= ((+) · ( × 60)) min     -- (3)
```

позволяющие перейти к полностью бесточечному определению

```
toSeconds = (+) · ( × 60)
```

Преобразование (1) базируется на возможности записать любой оператор в функциональной форме, заключив его в скобки. Преобразование (2) основано на переходе к правому сечению оператора умножения. Наконец, преобразование (3) демонстрирует использование определения оператора композиции функций `( · )` для «протаскивания» переменной **min** через правую скобку.

Последнее, полностью бесточечное определение функции `toSeconds` не выглядит интуитивно понятным. Неприятность заключается в том, что левый аргумент композиции (оператор сложения) представляет собой функцию двух аргументов. Если мы хотим научиться читать и использовать бесточечные выражения с использованием функций нескольких аргументов, то нам необходимо достаточно хорошо разобраться с тем, как их «обрабатывает» оператор композиции. Ниже мы увидим, что сечения этого оператора окажутся весьма полезными для достижения такой цели.

5.3. Левое сечение композиции

Рассмотрим несколько простых примеров левого сечения композиции, то есть выражения `(f · )`. Запустим сессию GHCi, и будем сравнивать тип функции и тип левого сечения композиции этой функцией[2]:

```
> :type sin
sin :: (Floating a) ⇒ a → a
> :type (sin · )
(sin · ) :: (Floating a) ⇒ (l → a) → l → a

> :type and
and :: [Bool] → Bool
> :type (and · )
(and · ) :: (l → [Bool]) → l → Bool

> :type length
length :: [a] → Int
> :type (length · )
(length · ) :: (l → [a]) → l → Int
```

Легко увидеть общий паттерн: для произвольной функции

```
f :: a → b
```

тип левого сечения композиции имеет вид

```
(f · ) :: (l → a) → l → b
```

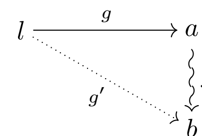
или, эквивалентно,

```
(f · ) :: (l → a) → (l → b)
```

Можно сформулировать это так: сечение `(f · )` преобразует переданную в качестве аргумента функцию `g :: l → a` в функцию `g' :: l → b`; это превращение происходит с помощью функции `f :: a → b`:

```
g' = (f · ) g
```

Описанное превращение удобно изображать с помощью диаграмм. При этом исходную функцию (`g`) будем изображать на диаграммах обычной стрелкой, результирующую (`g'`) — стрелкой из точек, а вспомогательную `f` — волнистой стрелкой:



Можно сформулировать наше наблюдение с меньшей степенью формализма: тип сечения `(f · ) :: (l → a) → (l → b)` получается приписыванием стрелочки `(l → )` слева (отсюда и использование типовой переменной `l` — left), к аргументу и возвращаемому значению типа функции `f :: a → b`.

5.3.1. Отступление: функторы, механика за сценой

Стоит отметить, что тип функциональной стрелки `l → r` в Haskell имеет два типовых параметра (`l` и `r`); эту стрелку допустимо записывать на уровне типов как `(→ ) l r`. Аналогично, скажем, двухпараметрический тип пары `(a,b)` можно записать как `(,) a b`. Допускается частичное применение конструктора типа на уровне типов — та стрелка `(l → )`, которую мы неформально «приписывали» выше, имеет в Haskell законное формальное определение: `(→ ) l`. Это уже однопараметрический тип: второй параметр типа связан типом `l`.

Интересно, что этот тип, как и многие библиотечные однопараметрические типы (тип списка, **Maybe**, **IO**) является функтором: в `Control.Monad.Instances` определено

```
instance Functor ((→) l) where
  fmap f = (f · )
```

то есть рассматриваемое нами сечение композиции есть не что иное, как `fmap` для нашей стрелки `(l → )`! Это нетрудно понять, сравнив тип `fmap` для неё

```
λf → (f · ) :: (a → b) → ((→) l) a → ((→) l) b
```

с типом функции `fmap`, скажем, для списка,

[1]Мы будем использовать термин «бесточечный» и для случая, когда у функции опущены не все, а только часть аргументов.

[2]Для того, чтобы подчеркнуть интересующие нас свойства, имена типовых переменных здесь несколько изменены по сравнению со стандартным выводом GHCi. В определенной позиции используется имя типа `l` от слова *left* (левый), причины этого станут ясны ниже.

```
fmap :: (a → b) → [] a → [] b
```

или с общим определением

```
class Functor f where
 fmap :: (a → b) → f a → f b
```

Заметим, что в теории категорий функтор `(l → )` называют ковариантным hom-функтором [6].

5.3.2. Дважды левое сечение

Рассмотрим теперь, какие полезные конструкции можно извлечь из такой связи между функциональной стрелкой со связанным левым параметром и левым сечением композиции. Напомним: тип левого сечения `(f·) :: (l → a) → (l → b)` получается «приписыванием» стрелочки `(l → )` слева к аргументу и возвращаемому значению типа функции `f :: a → b`. Что будет, если осуществить эту операцию два раза подряд? Получающаяся конструкция имеет тип

```
((f · ) · ) :: (l2 → (l1 → a))
            → (l2 → (l1 → b))
```

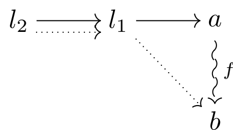
Если абстрагироваться по `f` и убрать некоторые лишние скобки, получим

```
λf → ((f · ) · ) :: (a → b) → (l2 → l1 → a)
                               → (l2 → l1 → b)
```

или, в более удобном для практических целей виде,

```
λf g → (f · ) · g :: (a → b) → (l2 → l1 → a)
                               → (l2 → l1 → b)
```

Для `f :: a → b` и `g :: l2 → l1 → a` получаем типизацию `(f·)·g :: l2 → l1 → b`. Соответствующая диаграмма имеет вид



Напомним, что функцию `g` мы изображаем на диаграммах обычными стрелками, результирующую конструкцию — стрелками из точек. В «точечном» стиле вызов конструкции `(f·)·g` на аргументах `x :: l2`, `y :: l1` эквивалентен вызову

```
f (g x y)
```

Итак, *конструкция* `(f·)·g` *порождает функцию двух аргументов, на которых вызывается* `g`, *а результат этого вызова обрабатывается затем* `f`.

Примеры.

Бесточечное определение `takeWhile`. Имеем библиотечную функцию

```
fst :: (a, b) → a
```

которая возвращает первый элемент пары, и библиотечную функцию

```
span :: (a → Bool) → [a] → ([a], [a])
```

которая для заданных предиката и списка возвращает пару списков, первый — длиннейший префикс исходного списка, все элементы которого удовлетворяют предикату, второй — оставшаяся часть исходного. Тогда библиотечная функция

```
takeWhile :: (a → Bool) → [a] → [a]
```

которая должна вернуть длиннейший префикс списка, все элементы которого удовлетворяют предикату, может быть определена в бесточечном стиле так:

```
takeWhile = (fst · ) · span
```

Бесточечное определение `mapM`. Имеем библиотечные функции

```
sequence :: Monad m ⇒ [m a] → m [a]
map :: (a → b) → [a] → [b]
```

Тогда библиотечная функция

```
mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
```

может быть определена в бесточечном стиле так

```
mapM = (sequence · ) · map
```

Поиск без использования `Maybe`. Имеем библиотечные функции

```
fromMaybe :: a → Maybe a → a
find :: (a → Bool) → [a] → Maybe a
```

Тогда функция с типом `(Num a) ⇒ (a → Bool) → [a] → a`, возвращающая 0 при неудачном поиске в списке чисел и найденное число при удачном, может быть определена в бесточечном стиле так

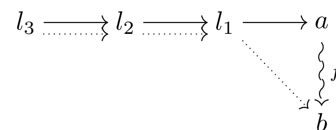
```
(fromMaybe 0 · ) · find
```

5.3.3. Многократное левое сечение

Если применить операцию левого сечения композиции не два раза подряд, а три или четыре, мы получим возможность «прицеплять» функцию `f`, как обработчик результата функции трёх или четырёх аргументов. Для трёх применений имеем

```
λf g → ((f · ) · ) · g
   :: (a → b) → (l3 → l2 → l1 → a)
              → (l3 → l2 → l1 → b)
```

Соответствующая диаграмма имеет вид



В «точечном» стиле вызов конструкции `((f·)·)·g` на аргументах `x :: l3`, `y :: l2`, `z :: l1` эквивалентен вызову

```
f (g x y z)
```

Приведем пример использования конструкции `((f·)·)·g`. Выражение

```
((product · ) · ) · zipWith
     :: (Num c) ⇒ (a → b → c) → [a] → [b] → c
```

работает как

```
zipWith :: (a → b → c) → [a] → [b] → [c]
```

только над результирующим списком `[c]` выполняется свертка умножением

```
product :: (Num c) ⇒ [c] → c
```

Проверим в GHCi:

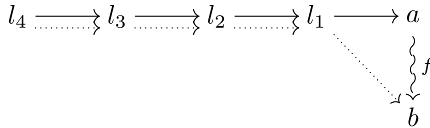
```
> (((product · ) · ) · zipWith) (+) [1,2,3] [3,2,1]
64
```

Действительно `(1 + 3)×(2 + 2)×(3 + 1) = 64`.

Для четырёх последовательных применений левого сечения композиции получаем

```
λf g → (((f · ) · ) · ) · g
   :: (a → b) → (l4 → l3 → l2 → l1 → a)
              → (l4 → l3 → l2 → l1 → b)
```

Диаграмма



В «точечном» стиле вызов конструкции `(((f · ) · ) · ) · g` на аргументах `x :: l4`, `y :: l3`, `z :: l2`, `v :: l1` эквивалентен вызову

```
f (g x y z v)
```

Ясно, что эта техника легко обобщается на случай произвольного числа аргументов у функции `g`.

Итак, левое сечение композиции при последовательном применении (начиная с некоторой функции) порождает конструкцию, позволяющую осуществить вызов этой функции над результатом функции того количества аргументов, сколько сечений композиции использовалось.

5.4. Правое сечение композиции

Чтобы породить более богатое семейство бесточечных композиционных конструкций, полезно воспользоваться правым сечением композиции `( · f)`. Начнём, как обычно, с примеров:

```
> :type cos
cos :: (Floating a) ⇒ a → a
> :type ( · cos)
( · cos) :: (Floating a) ⇒ (a → r) → a → r

> :type and
and :: [Bool] → Bool
> :type ( · and)
( · and) :: (Bool → r) → [Bool] → r

> :type length
length :: [a] → Int
> :type ( · length)
( · length) :: (Int → r) → [a] → r
```

Видно, что для произвольной функции

```
f :: a → b
```

тип правого сечения композиции имеет вид

```
( · f) :: (b → r) → a → r
```

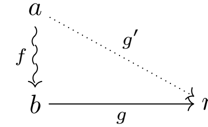
или, эквивалентно,

```
( · f) :: (b → r) → (a → r)
```

Можно сформулировать это так: сечение `( · f)` преобразует переданную в качестве аргумента функцию `g :: b → r` в функцию `g' :: a → r`; это превращение происходит с помощью функции `f :: a → b`:

```
g' = ( · f) g
```

В виде диаграммы:



Менее формально: тип правого сечения `( · f) :: (b → r) → (a → r)` получается «приписыванием» стрелочки `(→ r)` *справа* к аргументу и возвращаемому значению типа функции `f :: a → b` и *перестановкой аргументов* (в отличие от левого сечения, где последовательность аргументов сохранялась).

5.4.1. Отступление: контрафункторы

Функциональная стрелка со связанным правым аргументом `(→ r)` не может быть напрямую формализована в Haskell. В библиотеке `category-extras` Эдварда Кметта [4] имеется определение

```
newtype ContraF r l = ContraF {runContraF :: l → r}
```

упаковывающее стрелку `(l → r)` в двухпараметрический тип `ContraF`. Частичное применение этого типа `ContraF r` является полным эквивалентом стрелки со связанным правым аргументом `(→ r)`.

Из-за перестановки аргументов тип правого сечения композиции не может являться функтором. В теории категорий, однако, для этого случая вводится понятие *контравариантного* функтора. В библиотеке `category-extras` контравариантный функтор реализован как класс типов

```
class ContraFunctor f where
 contramap :: (a → b) → f b → f a
```

При этом для частичного применения типа `ContraF r` имеется объявление экземпляра[3]

```
instance ContraFunctor (ContraF r) where
 contramap f = ( · f)
```

Это объявление создаёт связь между функциональной стрелкой со связанным правым параметром и правым сечением композиции, аналогичную связи между их левыми собратьями.

5.5. Конструкции, порождаемые обоими сечениями

Теперь мы можем осуществлять и левое и правое сечение, при этом стрелки к типу будут подписываться слева и справа соответственно. Во втором случае следует также не забыть переставлять аргументы.

Например, если сначала применить правое сечение, а потом левое, получим

```
λf → (( · f) · )
     :: (a → b) → (l2 → (b → r1))
                → (l2 → (a → r1))
```

или

```
λf → (( · f) · )
     :: (a → b) → (l2 → b → r1)
                → (l2 → a → r1)
```

[3] В теории категорий функтор `(→ r)` называют контравариантным hom-функтором [6].

Если сначала применить левое сечение, а потом правое, получим

```
λf → ( · (f · ))
     :: (a → b) → ((l1 → b) → r2)
                → ((l1 → a) → r2)
```

Два правых сечения дадут

```
λf → ( · ( · f))
     :: (a → b) → ((a → r1) → r2)
                → ((b → r1) → r2)
```

Здесь `a` и `b` оказались на месте, потому что имела место двукратная перестановка: сначала `a` поменялось с `b`, затем `(b → r1)` с `(a → r1)`.

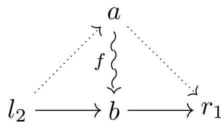
Глядя на последние два примера, можно заметить, что каждое дополнительное правое сечение приводит к увеличению на единицу порядка второго функционального аргумента. С другой стороны, левые сечения не изменяют этого порядка. Поскольку на практике функции высоких порядков используются крайне редко, мы не будем рассматривать конструкции, в которых больше двух правых сечений. Остальные конструкции изучим подробнее в следующих подразделах.

5.5.1. Правое сечение, потом левое

Правое сечение композиции, а затем левое над его результатом `(( · f) · )`, порождают для `f :: a → b` и `g :: l2 → b → r1` конструкцию `( · f) · g :: l2 → a → r1`:

```
λf g → ( · f) · g
     :: (a → b) → (l2 → b → r1)
                → (l2 → a → r1)
```

Соответствующая диаграмма имеет вид



Напомним, что функцию `g` мы изображаем на диаграммах обычными стрелками, результирующую конструкцию — стрелками из точек. В «точечном» стиле вызов конструкции `( · f) · g` на аргументах `x :: l2`, `y :: a` эквивалентен вызову

```
g x (f y)
```

Итак, *конструкция* `( · f ) · g` *порождает функцию двух аргументов, первый из которых передается напрямую в* `g`, *а последний перед такой передачей предварительно обрабатывается* `f`.

Примеры.

Бесточечное определение `replicate`. Имеем библиотечные функции

```
repeat :: a → [a]
take :: Int → [a] → [a]
```

Тогда библиотечная функция

```
replicate :: Int → a → [a]
```

может быть определена в бесточечном стиле так

```
replicate = ( · repeat) · take
```

Отображение над хвостом списка. Имеем библиотечные функции

```
map :: (a → b) → [a] → [b]
tail :: [a] → [a]
```

Тогда функция с типом `(a → b) → [a] → [b]`, осуществляющая отображение переданной функции на хвост переданного списка, может быть записана в бесточечном виде так

```
( · tail) · map
```

Увеличение элементов списка перед «зиповкой». Имеем библиотечные функции

```
map :: (a → b) → [a] → [b]
succ :: (Enum a) ⇒ a → a
zipWith :: (a → b → c) → [a] → [b] → [c]
```

Тогда функция с типом `(Enum a) ⇒ (a → b → c) → [a] → [`
осуществляющая однократное увеличение элементов первого списка, может быть записана в бесточечном виде так

```
( · map succ) · zipWith
```

Фильтрация элементов списка перед свёрткой. Имеем библиотечные функции

```
filter :: (a → Bool) → [a] → [a]
(>) :: (Ord a) ⇒ a → a → Bool
foldr1 :: (a → a → a) → [a] → a
```

Тогда функция с типом `(Num a, Ord a) ⇒ (a → a → a) → [`
осуществляющая фильтрацию элементов списка по условию `(> 3)` перед его свёрткой, может быть записана в бесточечном виде так:

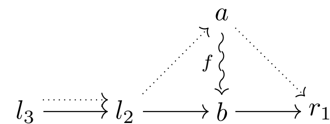
```
( · filter (> 3)) · foldr1
```

5.5.2. Сначала правое сечение, а потом левые

Если применить операцию правого сечения композиции, а затем выполнить над его результатом несколько левых, мы получим возможность «вставлять» функцию `f`, как предварительный обработчик последнего аргумента функции нескольких переменных. Для трёх сечений получаем

```
λf g → (( · f) · ) · g
   :: (a → b) → (l3 → l2 → b → r1)
              → (l3 → l2 → a → r1)
```

Соответствующая диаграмма имеет вид



В «точечном» стиле вызов конструкции `(( · f) · ) · g` на аргументах `x :: l3`, `y :: l2`, `z :: a` эквивалентен вызову

```
g x y (f z)
```

Приведем пример использования конструкции `(( · f) · ) · g`. Выражение

```
(( · map succ) · ) · foldr
```

это тот же **`foldr`**, только над однократно увеличенными элементами списка. Действие **`map succ`** посредством обсуждаемой конструкции перенаправляется на последний (третий) аргумент функции

 © 2010 «Практика функционального программирования»

```
foldr :: (a → b → b) → b → [a] → b
```

то есть на список. Проверим в GHCi:

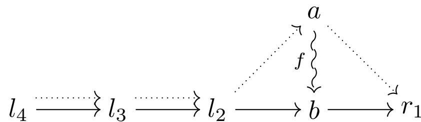
```
> (((·map succ)·)·foldr) (×) 1 [1,2,3]
24
```

Действительно `2×3×4 = 24`.

Для четырёх сечений (правое, левое, левое, левое) получаем

```
λf g → (((·f)·)·)·g
   :: (a → b) → (l4 → l3 → l2 → b → r1)
              → (l4 → l3 → l2 → a → r1)
```

Диаграмма



«Точечный» эквивалент конструкции `(((·f)·)·)·g` на аргументах `x :: l4`, `y :: l3`, `z :: l2`, `v :: a` таков

```
g x y z (f v)
```

Эта техника легко обобщается на случай произвольного числа аргументов у функции `g`.

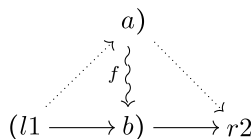
Итак, правое сечение композиции некоторой функцией, за которым следует несколько последовательных левых сечений порождает конструкцию, позволяющую «вставлять» указанную функцию как предварительный обработчик последнего аргумента функции того количества аргументов, сколько сечений использовалось.

5.5.3. Сначала левое сечение, а потом правое

Левое сечение композиции, а затем правое над его результатом `(·(f·))`, порождают для `f :: a → b` и `g :: (l1 → b) → r2` конструкцию `g·(f·) :: (l1 → a) → r2`:

```
λf g → g·(f·)
     :: (a → b) → ((l1 → b) → r2)
                → ((l1 → a) → r2)
```

Соответствующая диаграмма имеет вид



Напомним, что функцию `g` мы изображаем на диаграммах обычными стрелками, результирующую конструкцию — стрелками из точек. В «точечном» стиле вызов конструкции `g·(f·)` на аргументе `h :: l1 → a` эквивалентен вызову

```
g (f·h)
```

Итак, *конструкция* `g·(f·)` *порождает функцию, представляющую собой функцию функционального аргумента, и осуществляющую вызов* `g` *на композиции функции* `f` *и этого функционального аргумента.*

Пример.

Имеем библиотечные функции

```
map :: (a → b) → [a] → [b]
digitToInt :: Char → Int
```

Последняя определена только для символов `'0'..'9'`, `'a'..'f'`, `'A'..'F'`. Тогда конструкция

```
map·(digitToInt·) :: (a → Char) → [a] → [Int]
```

работает как отображение **`map`**, только над каждым элементом списка *после*[4] вызова функции типа `a → Char`, передаваемой в это выражение в качестве функционального аргумента, выполняется ещё вызов **`digitToInt`**.

Проверка в сессии GHCi даёт

```
> (map·(digitToInt·)) succ "ABCDE"
[11,12,13,14,15]
> (map·(digitToInt·)) succ "ABCDEF"
[11,12,13,14,15,*** Exception:
        Char.digitToInt: not a digit 'G'
```

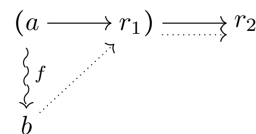
Вызов функции **`succ`**, передаваемой как аргумент конструкции, происходит *до* вызова функции **`digitToInt`**, встроенной в конструкцию (фактически в **`map`** передаётся композиция **`digitToInt`**·**`succ`**). Поэтому во втором тестировании и выбрасывается исключение: `succ 'F' = 'G'`, а на значении `'G'` функция **`digitToInt`** не определена.

5.5.4. Дважды правое сечение

Правое сечение композиции, повторённое два раза `(·(·f))`, порождает для `f :: a → b` и `g :: (a → r1) → r2` конструкцию `g·(·f) :: (b → r1) → r2`:

```
λf g → g·(·f)
     :: (a → b) → ((a → r1) → r2)
                → ((b → r1) → r2)
```

Соответствующая диаграмма имеет вид



Напомним, что функцию `g` мы изображаем на диаграммах обычными стрелками, результирующую конструкцию — стрелками из точек. В «точечном» стиле вызов конструкции `g·(·f)` на аргументе `h :: b → r1` эквивалентен вызову

```
g (h·f)
```

Итак, *конструкция* `g·(·f)` *порождает функцию, представляющую собой функцию функционального аргумента, и осуществляющую вызов* `g` *на композиции этого функционального аргумента и функции* `f`.

Пример:

Рассмотрим конструкцию, очень похожую на конструкцию из предыдущего примера:

```
map·(·digitToInt) :: (Int → c) → [Char] → [c]
```

Она работает как отображение **`map`**, только над каждым элементом списка помимо вызова функции типа `Int → c`, передаваемой в это выражение в качестве функционального аргумента, выполняется ещё *предварительный* вызов **`digitToInt`**.

Проверка в сессии GHCi даёт

[4] Термины «после» и «до» здесь используются для маркирования последовательности аппликаций; мы говорим что в вызове `f (g x)` функция `f` применяется *после* `g`, невзирая на возможные тонкости операционной семантики.

```
> (map · ( · digitToInt)) succ "ABCDE"
[11,12,13,14,15]
> (map · ( · digitToInt)) succ "ABCDEF"
[11,12,13,14,15,16]
```

Вызов функции **succ**, передаваемой как аргумент конструкции, происходит *после* вызова функции **digitToInt**, встроенной в конструкцию (фактически в **map** передаётся композиция **succ** · **digitToInt**). Поэтому в отличие от прошлого примера во втором тестировании исключения на символе `'F'` не происходит: `digitToInt 'F' = 15` и `succ 15 = 16`.

5.6. Дополнительные техники

Если взглянуть на разобранные выше цепочки сечений, может показаться, что многие полезные конструкции остались вне нашего рассмотрения. Из диаграмм видно, что функция-обработчик `f`, модифицирующая функцию нескольких аргументов `g`, может подключаться либо как постобработчик результата `g`, либо как «предварительный» обработчик последнего из аргументов `g`. А как действовать в случае, если требуется организовать предобработчик другого (не последнего) аргумента `g`?

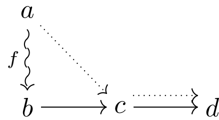
Оказывается, здесь нам помогает тот факт, что все функции в Haskell могут рассматриваться как функции одного аргумента. Предположим, нас интересует бесточечное представление для конструкции

```
λf g x y → g (f x) y
```

Её тип

```
(a → b) → (b → c → d)
        → (a → c → d)
```

и диаграмма



Легко увидеть, что в этой конструкции функция `g` может трактоваться как функция одного аргумента (типа `b`): `g :: b → (c → d)`, то есть тип конструкции сводится к типу простой композиции

```
(a → b) → (b → t)
        → (a → t)
```

где введено обозначение `t = (c → d)`. То есть конструкция может быть в бесточечном стиле записана как

```
λf g  → g · f
```

несмотря на то, что функция `g` является функцией двух аргументов. Этот подход работает и в других аналогичных случаях; важно лишь осознавать, что любой аргумент функции может рассматриваться как последний.

Здесь можно вспомнить пример, приведенный в начале статьи — функцию

```
toSeconds min sec = min × 60 + sec
```

Функция одного аргумента `(×60)` применяется к первому аргументу функции двух аргументов `(+)`. Отсюда сразу получается бесточечное определение

```
toSeconds  = (+) · (×60)
```

которое раньше выводилось как результат долгих преобразований.

Коротко коснёмся композиции сечений композиции. Независимо от того, используются правые сечения или левые, такая конструкция описывает композицию трёх функций

```
λf g h → f · g · h
```

позволяя избавиться в ней от одного из функциональных аргументов. Так композиция левых сечений избавляет от последнего аргумента

```
λf g h → f · g · h =
λf g   → (f · ) · (g · )
```

Композиция правых сечений избавляет от первого аргумента

```
λf g h → f · g · h =
λg h   → ( · h) · ( · g)
```

Композиция левого и правого (или правого и левого) сечения позволяет избавиться от среднего аргумента

```
λf g h → f · g · h =
λf h   → (f · ) · ( · h) =
λf h   → ( · h) · (f · )
```

Более сложные конструкции из операторов композиций также могут быть предметом рассмотрения. Более подробно с этой темой можно ознакомиться в [3, 7]. Например, в [7] демонстрируется, как по конструкции

```
λf g h x y z →  f( g (h x y z))
```

эффективно получить ее полный бесточечный эквивалент

```
( · (( · ) · ( · ) · ( · ))) · ( · ) · ( · ) · ( · ) · ( · )
```

5.7. Заключение

Мы выяснили, что сечения оператора композиции являются полезными инструментами для построения широкого спектра бесточечных выражений. Сформулируем основные практические выводы.

Если мы начинаем цепочку левых сечений композиции с левого сечения композиции некоторой функцией `f`, мы получаем конструкцию `(...(((f · ) · ) · )... · )` со следующими свойствами.

- Эта конструкция применима к функции `g` того количества аргументов, сколько сечений использовалось.
- Функция `f` работает в этой конструкции как постобработчик результата функции нескольких переменных `g`.

Для случаев, когда `g` является функцией одного, двух, трех и четырех аргументов, имеем, соответственно, эквивалентности (в «точечной» форме):

```
(f · )              g x       = f (g x)
((f · ) · )         g x y     = f (g x y)
(((f · ) · ) · )    g x y z   = f (g x y z)
((((f · ) · ) · ) · ) g x y z v = f (g x y z v)
```

Если мы начинаем цепочку левых сечений композиции с правого сечения композиции некоторой функцией `f`, мы получаем конструкцию `(...((( · f) · ) · )... · )` со следующими свойствами.

- Эта конструкция применима к функции `g` того количества аргументов, сколько сечений использовалось.

- Функция `f` работает в этой конструкции как предварительный обработчик последнего аргумента функции нескольких переменных `g`.

Если же нам требуется организовать предварительный обработчик любого другого (не последнего) аргумента функции нескольких переменных, мы делаем этот аргумент последним искусственно, рассматривая всё, что идёт после этого аргумента, как возвращаемое (функциональное) значение. Для случая, когда `g` является функцией двух аргументов, «точечные» эквивалентности для всевозможных вариантов «подключения» предобработчика `f` таковы:

```
((·f)·) g x y = g x (f y)
(·f)    g x y = g (f x) y
```

Для случая, когда `g` является функцией трех аргументов:

```
(((·f)·)·) g x y z = g x y (f z)
((·f)·)    g x y z = g x (f y) z
(·f)       g x y z = g (f x) y z
```

Для случая, когда `g` является функцией четырех аргументов:

```
((((·f)·)·)·) g x y z v = g x y z (f v)
(((·f)·)·)    g x y z v = g x y (f z) v
((·f)·)       g x y z v = g x (f y) z v
(·f)          g x y z v = g (f x) y z v
```

В заключение хотелось бы ещё раз подчеркнуть, что бесточечный стиль — всего лишь одна из техник программирования на Haskell. Разумный программист понимает, что никакой техникой не стоит злоупотреблять в ущерб другим. Компактность, возникающая благодаря использованию этого стиля, имеет свою цену: глядя на бесточечную конструкцию иногда трудно восстановить количество и тип необходимых аргументов. Поэтому разобранные в статье конструкции можно порекомендовать использовать в ситуациях, когда их компонентами являются широко известные библиотечные функции из семейств `fmap`, `fold`, `scan`, **`zipWith`** и им подобных.

Литература

[1] *Backus J.* Can programming be liberated from the von neumann style? // *Communications of the ACM*. — 1978. — Vol. 21, no. 8. — Pp. 613–641.

[2] *Gibbons J.* A pointless derivation of radixsort // *Journal of Functional Programming*. — 1999. — Vol. 9, no. 3. — Pp. 339–346. http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/radix.ps.gz.

[3] *Haskell Wiki*. Pointfree. — URL: http://www.haskell.org/haskellwiki/Pointfree (дата обращения: 25 февраля 2010 г.).

[4] *Kmett E.* Various modules and constructs inspired by category theory. — URL: http://comonad.com/haskell/category-extras/dist/doc/html/category-extras/index.html (дата обращения: 25 февраля 2010 г.).

[5] *Кирпичёв Е.* Элементы функциональных языков // — *Практика функционального программирования*2009. — декабрь. — Т. №3.

[6] *Маклейн С.* Категории для работающего математика. — ФизМатЛит, 2004.

[7] *Москвин Д.* Функциональные типы и композиция функций в Хаскелле // *RSDN Magazine*. — 2007. — Т. №3. http://rsdn.ru/article/haskell/typesH.xml.