Практика функционального программирования

Выпуск 7 • Апрель 2011

ISSN 2075-8456

MarginCon'11

«Какой язык изучать завтра?»

Конференция, посвящённая немейнстримным языкам программирования

25 июня 2011

г. Омск, Россия

http://margincon.ru/

Информационный спонсор —
журнал «Практика функционального программирования»

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: Бойко Банчев
Лев Валкин
Алексей Вознюк
Вадим Залива
Илья Ключников
Олег Смирнов
Максим Сохацкий
Александр Темерев

Выпускающий редактор: Евгений Кирпичёв

Редактор: Лев Валкин

Корректор: Алекс Отт

Иллюстрации: **Обложка**
© Ира Горд

Шрифты: **Текст**
Minion Pro © Adobe Systems Inc.
Обложка
Days © Александр Калачёв, Алексей Маслов
Cuprum © Jovanny Lemonad

Ревизия: 666 (2010-10-10)

Сайт журнала: http://fprog.ru/

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

Пожертвования

Продолжаем традицию публикации записок, приложенных к пожертвованиям.

Статьи интересные. Надеюсь, следующий номер выйдет раньше, чем через полгода :)	Yandex.Money
от Василия	PayPal
Дякую файно!	PayPal
C++ наше всё!	PayPal
$$↑$$	PayPal
Интересный журнал, спасибо!	PayPal
Так держать, ребята!	PayPal
Спасибо за интересный журнал. Ждем следующих выпусков.	PayPal
way to go	PayPal
Молодцы! Так держать! С нетерпением жду следующего номера. Владимир Лебедев.	PayPal
(счастья (вам))	SMS
Впервые отправляю платное смс, но для хорошего дела	SMS
Молодцы, ребята. Продолжайте в том же духе. Годно!	SMS
Merry christmass, Santa	SMS
большое спасибо за ваш журнал.	SMS
Bolshushee spasibo za interesnie stat'i! Za f.p. budushee! Anton.	SMS
Ну давайте же уже очередной номер выпускайте. Ждем.	SMS
Давайте уже расширяйте аудиторию.	SMS
thank you	SMS
Будьте счастливы!	SMS
Когда там уже седьмой номер-то выйдет, а?	SMS
fmap fmap fmap	SMS
Итого	**22367.90**

Спасибо за поддержку!

fprog.ru/donate

Оглавление

От редактора

Седьмой номер был небыстр и нелегок в производстве; долгими студёными зимними вечерами, не покладая рук, трудились авторы и редакторы над его созданием. Но, пожалуй, труды себя оправдали — по разнообразию и качеству содержимого этот номер беспрецедентен; низкий поклон авторам. Мы уверены, что читатели откроют для себя из статей много нового, и надеемся, что этот номер окажется большим шагом к выполнению нашей основной задачи: *повысить осведомлённость о не-мейнстримных языках и методиках программирования в русскоязычном сообществе и изменить отношение к ним.*

Статьи

Позволим себе пару комментариев относительно статей номера.

- Бойко Банчев расскажет о сущности, истории и современном состоянии языка РЕФАЛ — детища Валентина Фёдоровича Турчина; языке незаслуженно подзабытом, но на десятилетия опередившем своё время и пустившем корни во множество современных ЯП (как, например, Mathematica).
- Александр Темерев даст урок использования Circumflex — «обезжиренного» и гибкого веб-фреймворка, написанного на Scala российской командой во главе с Борисом Окунским.
- Вадим Залива расскажет об опыте прототипирования алгоритма обнаружения движения на Mathematica. Хотя Mathematica и не является в полном смысле этого слова функциональным языком программирования, но программирование в этой среде задействует множество не очень распространенных пока в мейнстриме, но очень важных и заслуживающих внимание средств: интерактивный интерпретатор, гомоиконность (единство кода и данных), сопоставление с образцами, *wholemeal programming* (оперирование целыми структурами данных, а не их частями), акцент на «чистое» программирование, dataflow programming (*Manipulate*) и т. п.
- Алексей Вознюк расскажет об использовании продолжений для упрятывания асинхронности кода за обыкновенным «последовательным» синтаксисом. Эта тема особенно актуальна в контексте взрывного роста Веба и сетевых сервисов, но интересна и потому, что открывает целый пласт неожиданных структур управления. Рекомендуем перед прочтением ознакомиться с недавно опубликованным в «Библиотечке ПФП[1]» переводом статьи «Паттерны использования `call-with-current-continuation`».
- Илья Ключников, занимающийся суперкомпиляцией в Институте прикладной математики им. М. В. Келдыша, расскажет о суперкомпиляции (кстати, изобретенной также В. Ф. Турчиным в контексте языка РЕФАЛ), построив поразительно компактный и понятный суперкомпилятор простого языка, и покажет, что она применима не только для удивительно мощной оптимизации многих классов программ, но имеет и другие неожиданные применения. Обязательно прочитайте и приложение[2]!

Печатный номер

Как и все предыдущие номера, данный номер доступен для заказа[3] в издательстве Самиздал. Стоимость номера — 200 рублей; осуществляется доставка в Россию и по всему миру (в Москве можно забрать прямо из издательства). В блогах можно найти немало постов от гордых владельцев комплектов ПФП с фотографиями, разве не завидно? ;) Напоминаем, что все доходы от продаж уходят на услуги типографии — так что мы будем по-прежнему рады, если вы угостите нас кофе[4] за наши труды.

Реклама

Мы по-прежнему предлагаем рекламодателям очень дешевую площадку для публикации вакансий в номере, и напоминаем, что едва ли в Рунете есть издание для программистов с такой же «концентрированно сильной» аудиторией более 10 тысяч человек. Пишите на ad@fprog.ru!

Сообщество

Ну а для тех, кому журнала мало, напоминаем: бесценный источник функциональных новостей — само русскоязычное сообщество функциональных программистов. Следите за блогами в твиттере[5] и Russian Lambda Planet[6]!

Приятного чтения!

С самыми чистыми пожеланиями,
Евгений Кирпичёв, `jkff@fprog.ru`

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

[2] http://fprog.ru/2011/issue7/ilya-klyuchnikov-supercompilation/supercompilation-addendum.pdf

[3] http://wikers.ru/catalog/samizdal_journals/

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

[5] http://twitter.com/fprogblogs

[6] http://fprog.ru/planet/

Язык Рефал — взгляд со стороны

Бойко Банчев
boyko@fprog.ru

Аннотация

Знакомство с языком Рефал полезно программисту хотя бы потому, что этот функциональный язык не похож ни на один из других — среди них он занимает особое место и по возрасту, и по происхождению, и по назначению, и по стилю. Достойно сожаления то что, несмотря на свои качества, язык не очень популярен.

Статья знакомит читателя с Рефалом. Язык так прост, что его описание почти целиком вмещается в статью — за исключением стандартных функций, которых тоже немного. Простота сама по себе — положительное качество, но читатель убедится, что оно не единственно.

Помимо описания самого Рефала, представлен взгляд автора на место, достоинства и слабые стороны языка.

Knowing the Refal language is useful to a programmer, if for nothing else than for the language's uniqueness — with respect to its age, its origin, its intended purpose, and style. It is regrettable that, in spite of its qualities, the language is not very popular.

This article gives an introduction to Refal.

Обсуждение статьи ведется по адресу:
http://fprog.ru/2011/issue7/boyko-banchev-refal/discuss/.

1.1. Вместо введения

Рефал — язык программирования. Сам я, знакомясь с новым языком, прежде всего хочу увидеть пример программы на нем, так что давайте начнем с примера.

```
$ENTRY Go {= <Read>}
Read {, <Card>:
       {   0 = ;
         e.1 = <Prout <Pal e.1>>
               <Read>}}
Pal {          = 'si';
         s.1 = 'si';
  s.1 e.2 s.1 = <Pal e.2>;
         e.1 = 'no'}
```

Это — программа на варианте языка, именуемом Рефал-5. На нем записаны и все остальные примеры, и вообще, главным образом он рассматривается далее.

Программа читает одну за другой строки текста из стандартного устройства ввода, для каждой строки определяя, является ли она палиндромом. `Go`, как можно догадаться — то же самое, что `main` в C — обязательное имя главной функции. Она объявлена при помощи ключевого слова `$ENTRY`, чтобы была видна «извне». `Card` — функция чтения строки со стандартного ввода. Функция же `Read` обращается к ней, пока та выдает строку текста `e.1` (0 есть признак конца «файла» ввода), которую она передает `Pal`. После того, как последняя выдаст `'si'` или `'no'`, `Read` обращается к самой себе для получения новой строки: хвостовая рекурсия. `Go` запрещено быть рекурсивной, поэтому цикл чтения-обработки она вынуждена возложить на кого-нибудь.

`Pal` сначала проверяет, не является ли цепочка, ее аргумент, пустой или состоящей из единственной литеры `s.1` — тогда ясно, что она — палиндром, и выдается соответствующий результат. Если это не так, цепочка сопоставляется с образцом `s.1 e.2 s.1`, успешно сопоставляющимся с любой цепочкой, у которой первая и последняя литера одинаковы. Успех сопоставления влечет вычисление `Pal` над серединой `e.2` цепочки, так что в конце концов `Pal` выдаст то, что она выдает при работе над укороченной цепочкой — опять хвостовая рекурсия. Любой другой состав входной цепочки означает, что она — не палиндром; в этом случае цепочка успешно сопоставится с образцом `e.1` и будет выдан ответ `'no'`.[1]

1.2. Функциональный язык — а какой именно?

Два обстоятельства делают Рефал интересной темой обсуждения в этом журнале. Во-первых, это — язык функционального стиля. Во-вторых, насколько мне известно, это единственный язык, который может быть сочтен «русским языком программирования» — если у языка программирования может быть национальность. Дело прежде всего в том, что Рефал создан в России (СССР) и полностью самобытен в своих идеях, но также и в том, что принцип вычислений в нем подобен алгорифмам Маркова.

Чтобы увидеть место Рефала в функциональном программировании, представляется нужным задаться вопросом «Что такое функциональный стиль программирования?». На этот вопрос ответить нелегко, и это — к счастью. К счастью, потому что трудность ответа свидетельствует об относительной развитости данного направления, и в частности — об имеющемся в нем разнообразии и взглядов на суть программирования, и проистекающих из них практических решений в виде конкретных языков, приёмов программирования и т. д.

Действительно, какие разновидности функционального стиля можно выделить? Согласно λ-исчислению, программирование есть *абстрагирование и применение*, т. е. образование некоторых функций и их применение. При этом смысл функций, их аргументов, равно как и смысл применения, несущественны. Существенно то, что программы — выражения, построенные исключительно из этих двух видов операций. Разумеется, конкретные языки программирования дополняют эту общую модель разного рода особенностями, будь то единообразие представления всей программы (Lisp, REBOL), типовые системы с автоматизированным выводом (ML, Haskell) или полная абстрактность понятия типа (Russell). Все же суть одна: порождение функций и применение.

Другая модель вычислений сосредотачивается на применении — поэтому назовем ее *аппликативной*. В ней применение функций имеет место преимущественно или даже только над другими функциями, и при том оно — единственный способ получить любую новую функцию; образование абстрагированием отсутствует. Это — модель комбинаторного исчисления, FP/FL, в значительной степени APL/J/K, или же безаргументного стиля, скажем в Haskell. При этом «настоящие данные и результат» программы только подразумеваются. Их можно косвенно определить как «то, над чем программа работает, когда ее запустят» и «то, что она выдает», но непосредственно на них не ссылаются.

Некоторым информатикам аппликативная модель кажется ненужно сложной, поэтому вместо общей операции применения они предпочитают ограничиться одной только композицией функций. Как в аппликативном стиле, все порождаемые программой объекты — функции, но порождение происходит применением только композиции. Саму операцию композиции тогда записывать не приходится: она выражается просто сочленением текстов композируемых функций, а потому данный стиль называют *конкатенативным*. Наиболее заметными представителями можно назвать Joy, Factor и (подмножество) PostScript.

На рисунке показано определение функции нахождения среднего арифметического значения списка чисел в разных функциональных стилях и языках.

Итак, есть по крайней мере три очень разных варианта функционального стиля. При этом они получаются один из другого как бы сужением допускаемых возможностей. (Сужение, кстати, не проявляется ограничением, но не будем обсуждать, почему это так.) Где среди них Рефал? Самое интересное, что нигде: он представитель — и этим он особенно примечателен — другого направления мышления, имеющего мало общего с λ-исчислением или с рассмотрением функций как полноправных значений и результатов в программе.

Рефал функционален, потому что действие программы в нем состоит в вычислении функции, вызванных из нее функций и т. д. и, как правило, функции не имеют побочных эффектов. Вместе с тем, в отличие от большинства функциональных языков, основное внимание уделяется не функциям, а данным,

[1] «Палиндромная функция» `Pal` имеет в Рефале статус факториальной функции в большинстве других функциональных языков: обязательный первый пример рекурсии. Противиться этому освященному мифическими человековеками правилу я не стану, но предпочитаю дать не отдельную функцию, а полную программу.

абстрагировано

Lisp:

```
(lambda (ns) (/ (reduce #'+ ns) (length ns)))
```

Oz:

```
fun {$ A} {FoldL A Number.'+' 0}/{Length A} end
```

Haskell:

```
\ns -> sum ns / genericLength ns
```

аппликативно

FP:

```
÷∘[/+,length]
```

Nial:

```
/[sum,tally]
```

Haskell:

```
liftM2 (/) sum genericLength
```

конкатенативно

Joy:

```
[sum] [size] cleave /
```

Cat:

```
count [vec_sum] dip /
```

PostScript:

```
dup 0 exch {add} forall exch length div
```

Рис. 1.1. Функции нахождения среднего арифметического значения

точнее — связи между структурой обрабатываемых данных и структурой вычислений. Еще точнее, функции в Рефале определяются при помощи механизма сопоставления входных данных с образцом, так что строение данного входного значения непосредственно определяет, какие вычисления над ним производить: это видно уже во вводном примере.

Следовательно, Рефал можно определить как *сопоставительный функциональный* язык. При этом, однако, следует остерегаться прямой аналогии с тем, что знакомо по языкам типа Haskell и ML. Сопоставление в этих языках связано с определением типов и наличием в этих типах конструкторов, т. е. с необходимостью построения некоторого уровня (или нескольких уровней) абстракции над вводимыми данными. В Рефале же функции работают прямо над входным текстом, преобразуя его, наводя структуру и далее таким же способом обрабатывая получившееся. Не строятся иерархии типов и конструкторов. В той мере, в какой структура нужна, она возводится непосредственно из данных и над ними. При этом значения элементарных данных понимаются либо конкретно как текст или числа, либо, более абстрактно, как символы — толкование условно и является вопросом соглашения или осмысления программы, но не касается ее организации.

В связи со сказанным нужно иметь ввиду и то, что Рефал намного старше упомянутых языков, и в частности сопоставления, основанного на конструктивных (алгебраических) типах.

Хотя подход Рефала к программированию имеет свои недостатки, и о них будет сказано далее, по сравнению с ориентированными на функции подходы он более прагматичен — и в этом смысле более привлекателен — своей непосредственной направленностью на данные.

1.3. Начала программирования на Рефале

1.3.1. Структура и выполнение программ

Программа на Рефале состоит из одной или более функций. Каждая функция задается именем и последовательностью *предложений* в фигурных скобках. Любое предложение имеет две части, между которыми стоит знак равенства `=`:

образец `=` *выражение*

Предложения отделяются одно от другого точкой с запятой.

У любой функции на Рефале ровно один аргумент. Его значение задается любым допустимым в Рефале выражением.

Вычисление функции состоит в сравнении ее аргумента с образцами, рассматриваемыми один за другим по порядку предложений. Неуспех сопоставления приводит к попытке сопоставления со следующим образцом. Как только найдется соответствие, вычисляется выражение за знаком `=`, и значение выдается как результат функции. Неуспех последнего, а значит всех сопоставлений, есть ошибка применения функции.

В самом простом виде выражение является константой, но чаще оно состоит в вызове одной или нескольких функций. Аргумент данной функции сам есть выражение, так что он может быть вызовом другой или той же самой функции и так далее. Таким образом, вызов функции имеет последствием одно или больше сопоставлений, за которыми следуют, в общем случае, новые вызовы функций.

Среди функций программы одна отмечена особым именем как главная. Выполнение программы начинается с нее, а дальше продолжается, как только что было описано.

Подчеркнем, что функции никогда не вызываются сами по себе, а только в результате активирования вычисления некоторого выражения вследствие успешного сопоставления. Другой особенностью является отсутствие глобальных переменных, так же как и вложенности функций. Все это приводит к тому, что контекст вычисления любого выражения строго локальный: в выражении могут встречаться только переменные из соответствующего образца, которые и получают значения именно благодаря успешному сопоставлению.

1.3.2. Значения

Весьма существенным в программировании на Рефале является понимание значений данных и их структуры.

Объект данных может быть атомарным или составным. Атом — это *литера*, *слово* или *число*. Пример слов: `"как тихо"` и `"b12"`; поскольку второе из них «простое», его можно записать и без кавычек: `b12`. (Простыми считаются слова, являющиеся допустимыми в качестве имен программных объектов: они начинаются буквой и состоят из букв, цифр и литер `-` и `_`.) Пример литерных значений: `'x'`, `'5'` и `'+'`.

Ввиду их особой роли в синтаксисе языка, для некоторых литерных значений используется запись со знаком `\`: `\'`, `\"`, `\n`, `\(`, `\<` и т. д. Через `\x` можно записывать литеры в шестнадцатеричном ASCII кодировании, например `\x20` — знак пробела.

Составные данные задаются *последовательностями*. Последовательность может быть пустой или состоять из атомов и других последовательностей. Так как любой атом есть последовательность с одним членом, можно считать что всякое значение — последовательность.

Последовательность можно записать перечислением ее членов, как в

```
ok 13 '*' "it's" "simple"
```

Пробелы между отдельными элементами последовательности надо понимать как применения операции сцепления.

Сцеплением атомы один с другим не сливаются, а только выстраиваются в последовательность. В этом смысле уместно рассматривать его как конструктор значений-последовательностей, именно — *конструктор сцепления*.

Для вложения одной последовательности в другую применяется *конструктор вложения*, записываемый круглыми скобками около включаемого значения. Это и единственная роль круглых скобок в Рефале, поэтому их называют *структурными скобками*. Так, например,

```
the five (boxing (wizards jump)) quickly
```

— последовательность из четырех элементов, один из которых — последовательность `boxing (wizards jump)` у которой второй элемент — последовательность `wizards jump`.

Таким образом, все нетривиальные последовательности — те, которые непусты и не являются атомами (а это и есть все составные значения) — образованы конструкторами сцепления и вложения. Всякая непустая последовательность есть не что иное, как дерево, у которого в концевых узлах стоят атомы, а в остальных — обособленные (под-)последовательности.

Единичные кавычки используются не только для обозначения отдельных литер, но и вообще для последовательностей литер, так что

```
'tomorrow'
```

— то же самое что

```
't' 'o' 'm' 'o' 'r' 'r' 'o' 'w'
```

или

```
'to' 'morrow'
```

Заметим, что в последнем случае та же последовательность получена сцеплением двух более коротких: так как аргументы сцепления есть последовательности, а не атомы, то они под действием сцепления сливаются.

Можно считать, что единичные кавычки предназначены для последовательностей (а не одиночных экземпляров) литер и что нет способа записать отдельную литеру, кроме как частью последовательности.[2] Последовательности литер играют ту же роль, что тип цепочек в других языках: их можно сцеплять, в них можно искать, из них можно выделять составляющие их литеры или части большей длины.

Слова, с другой стороны, цепочками не являются. Как значения они атомарны, без элементов, но стандартными функциями из слова можно получить цепочку, а из цепочки образовать слово.

Пустую последовательность удобно записать через `''` («пустая последовательность литер»), но во многих случаях ее даже не записывают, а подразумевают. Во всяком случае, `''` — единственный способ задать пустую последовательность в явном виде. `()` пустой последовательностью не является — это последовательность с одним элементом, а сам он — пустая последовательность. Также и пустое слово `""` — это не пустая последовательность, а именно слово, или же последовательность с одним элементом, являющимся пустым словом, т. е. словом длины 0.

С практической точки зрения о последовательностях важно отметить, что они представлены в памяти так, что доступ к их элементам симметричен относительно двух возможных направлений. Такое представление может опираться на двунаправленные списки или на другую структуру, обеспечивающую указанную симметричность, вплоть до последовательного («векторного») расположения.

В Рефале можно работать с неограниченно большими целыми числами, однако они атомами не являются. «Большое» целое число представляется последовательностью целых атомов, которые в этом случае называют «макроцифрами». Отрицательное число, независимо от его величины, тоже представляется не непосредственно, а последовательностью, начинающейся с атома `'-'`. Стандартные арифметические функции языка работают и с одноатомными, и с многоатомным числами, однако нужно иметь ввиду, что последние сами по себе не являются структурно обособленными значениями. Так, для отделения одного числа от следующего за ним, в общем, нужно использовать структурные скобки, а `'-' 357` может быть как записью отрицательного, так и просто последовательностью двух атомов.

1.3.3. Выражения

Выражения — это та конструкция языка, из которой путем вычислений получаются значения. В Рефале все значения — последовательности из атомов и подпоследовательностей, что отражено в записи константных выражений. Выражения общего вида отличаются от константных только включением переменных и вызовов функций.

Обращение к функции записывается в угловых скобках, где на первом месте стоит ее имя, а затем записан аргумент.

Значение переменной или вызова функции участвует в формировании значения соответствующего выражения точно так же, как если бы оно стояло непосредственно в этом выражении на месте переменной или вызова. Вложение при таком включении не подразумевается, его нужно всегда выражать явным образом структурными скобками.

Рассмотрим выражение

```
foo 'bar' e.baz <f qux etc> 'all' that stuff
```

В нем `e.baz` — переменая, а `f qux etc` — вызов функции `f` с аргументом `qux etc`. Допустим, значение `e.baz` — `'corge'`, а вызов `f` возвращает `'gra' ult 'garply'`. Тогда значением выражения является

```
foo 'barcorgegra' ult 'garplyall' that stuff
```

В любом случае, включаемые замещением переменной или вызова функции последовательности теряют свою обособленность, входя в результат не сами собой, а своими элементами. Если указанное выражение изменить на

```
foo 'bar' (e.baz) <f qux etc> 'all' that stuff
```

то его значением будет

```
foo 'bar' ('corge') 'gra' ult 'garplyall' that stuff
```

1.3.4. Функции, образцы, переменные

Угловые скобки в вызове функции имеют двоякую роль. Можно понимать `<` как «операцию вызова», без которой имя

[2] Исключением являются литеры, записанные через \: см. раздел *Пример: синтаксический анализ выражений языка Рефал.*

функции было бы просто словом, представляющим себя самого. Знак же `>` обозначает конец выражения, являющегося аргументом вызова — ведь аргумент, хотя и один, может быть любым выражением.

Когда части аргумента вызова функции следует рассматривать как отдельные значения, можно применить вложение, т. е. обособление структурными скобками. Тогда условно можно считать, что у функции несколько аргументов. Наиболее часто для этих целей заключают в скобках каждый «аргумент», возможно, за исключением последнего. Соответственным образом в определении функции надо составлять и образцы распознавания аргумента. Так, функция

```
inv3 {(e.1) (e.2) e.3 = (e.3) (e.2) e.1}
```

меняет местами первый и последний из своих трех аргументов. При обращении к ней

```
<inv3 ("they" "both") ("wanted" "to go") "there">
```

несмотря на то, что формально аргументом является лишь одна последовательность, но она расчленяется образцом нужным образом, и получаем

```
("there") ("wanted" "to go") "they" "both"
```

В предложениях Рефала скобки слева знака `=` являются частями образца, а справа — формируемого значения. В общем, образцы описывают структуру и состав успешно сопоставляющихся с ними выражений-аргументов при помощи констант, переменных и структурных скобок. Константы и скобки соответствуют тем же самым элементам сопоставляемого образцу выражения, а переменные получают значения соответствующих частей того же выражения. Так, если в образце встречается `(e.x '.')`, то переменная `e.x` сопоставляется с некоторой частью аргумента только в том случае, когда эта часть обособлена в подпоследовательность, оканчивающуюся точкой; сама точка в значение `e.x` не входит.

Ход и успешность сопоставления определяются не только структурой образца, но и типами участвующих в нем переменных. Тип переменной задается буквой с точкой в начале ее имени. Задание типа обязательно и указывает на вид структуры значений, которые могут сопоставляться с переменной.

Тип `e.`, как в примере выше, отвечает любому значению — пустой или непустой последовательности. Тип `s.` (см. пример в начале статьи) соответствует атому («символу»). Рассмотрим в качестве примера еще функцию

```
Flatten {   = ;
    s.h e.t = s.h <Flatten e.t>;
  (e.h) e.t = <Flatten e.h> <Flatten e.t>}
```

которая образует последовательность из всех своих атомов, сохраняя относительный порядок между ними (плоская проекция содержания последовательности). Согласно второму и третьему предложениям, если в начале аргумента стоит атом, он непосредственно переносится в результат, а если подпоследовательность, то ее часть результата вычисляется рекурсивным вызовом. В обоих случаях остаток аргумента обрабатывается рекурсивно. Тип `s.` переменной дает возможность вычленить именно атом, а не бо́льшую часть аргумента. Как и выше, скобки в образце используются для извлечения содержимого подпоследовательности, пропуская сами скобки.

Первое же предложение функции `Flatten` говорит, что для пустого аргумента пустым является и результат. «Пустое» значение в образце или в вычисляемой части предложения можно было бы задать и явным образом через `''`. Для вызова функции с пустым аргументом также можно записать `''` или не писать ничего (но скобки вызова все-таки нужны). К примеру, `<Flatten>` является правильным вызовом.

Взглянув еще раз на функцию `Pal` в самом первом примере, обратим внимание, что повторное употребление имени переменной в образце соответствует не только структурно подобным, а именно одинаковым значениям данных частей аргумента. Механизм сопоставления таков, что аргумент соответствует образцу только в этом случае. Иллюстрацией тому является и следующая функция:

```
In-seq {s.x e.1 s.x e.2 = T (e.1) e.2;
       (e.x) e.1 e.x e.2 = T (e.1) e.2;
                    e.x = F}
```

Будем считать, что у нее два аргумента, и она проверяет, входит ли первый во второй. Точнее, когда первый аргумент является атомом, то функция проверяет, встречается ли он на верхнем уровне второго аргумента (в этом предложении мы отклоняемся от описанного выше соглашения о разделении аргументов скобками). Если же первый аргумент — подпоследовательность, то все ее елементы должны встречаться один за другим в том же порядке во втором аргументе, опять же на верхнем уровне. Условившись передавать успех поиска словом `T`, а неуспех — `F`, функция возвращает один из этих атомов. В случае успеха она также передает в точку вызова контекст, т. е. префикс и суффикс найденного, заключая префикс в структурные скобки. Функцию можно использовать для поиска вхождения одной цепочки в другую, но она применима и к более разнообразным аргументам.

Приведем примеры вызовов `In-seq` и соответствующие результаты:

```
<In-seq go a le go rist>             →  T (a le) rist
<In-seq a a le go rist>              →  T () le go rist
<In-seq (le go) a le go rist>        →  T (a) rist
<In-seq (le go) a (le go) rist>      →  F
<In-seq ((le go)) a (le go) rist>    →  T (a) rist
<In-seq ((le go)) a ((le go) rist)>  →  F
```

Уславливание относительно `T` и `F` как здесь, является типичным. В Рефале булевых значений нет, поэтому приходится их имитировать. Впрочем, в конкретном случае можно было бы обойтись без флага истинности и возвращать только префикс и суффикс или пустое значение, поскольку в случае успеха результат будет непустым, и значит, отличить успех от неуспеха всегда возможно. Действительно, даже если и префикс, и суффикс пусты — первый аргумент в точности равен второму — получим `()`, а не `''`.

Если к сказанному еще условиться заключать в скобки не первый, а второй аргумент функции `In-seq`, то ее определение можно упростить до следующего:

```
In-seq {e.x (e.1 e.x e.2) = (e.1) e.2;
                     e.z = }
```

Сделаем следующее наблюдение. Соглашения о различении аргументов нужны, вводятся легко и не слишком перегружают определения функций, однако по одному виду этих определений, да и по вызовам тех же функций в принципе невозможно разобраться, сколько у функции аргументов и каковы они в структурном или содержательном отношении. Тем более, что эти соглашения бывают разными. В этом смысле программы обладают низкой степенью самоочевидности. Как ми-

нимум, мы с необходимостью приходим к выводу, что в Рефале требование к документированию программ нужно ставить особенно жестко.

Наряду со структурными типами `e.` и `s.` имеется и третий: `t.`. Он соответствует «терму» — понятию, которым обозначают либо атом, либо выражение в скобках.

В совокупности три типа отражают три степени общности значений: атомы являются также и термами, а термы — выражениями общего вида. Это учитывается, в частности, при упорядочении предложений функции, а также при выборе переменных в рамках образца, так как переменная более общего вида имеет тенденцию сопоставляться более успешно и с более крупными отрывками значений-аргументов. Крайним случаем является уже встречавшееся `e. = …`, где переменная сопоставляется с любым выражением. Это — идиоматическое предложение для обеспечения перехвата некоторого значения целиком или успешности выполнения функции.

Наличие термов как самостоятельного структурного типа, формально говоря, не является существенным для распознавания, так как образцы для любых структур можно составлять и без него. Однако оно дает удобную абстракцию для скрытия возможной неатомарности на сколь угодно высоком уровне. Так, любую последовательность можно понимать как однородную, состоящую из термов. Вообще, любой узел в иерархии любого составного значения можно рассматривать как терм, тем самым считая его атомарной сущностью, независимо от его действительного состава.

В следующем примере снова определяется функция поиска, но на этот раз первый аргумент есть терм, в каждом предложении обозначенный через `t.x`, который ищется во втором аргументе на любом уровне вложения. При этом считается, что нужно найти значение, равное `t.x` в целом, а не как в `In-seq` в виде подпоследовательности. Возвращается только атом `T` либо `F`.

```
In-term {     t.x = F;
  t.x e.1 t.x e.2 = T;
      t.x s.1 e.2 = <In-term t.x e.2>;
  t.x (e.1) e.2, <In-term t.x e.1>: {
                    T = T;
                    F = <In-term t.x e.2>}}
```

Логика действия функции, отраженная в предложениях в соответствующем порядке, такова. В пустой последовательности `t.x` не встречается. Если `t.x` встречается на верхнем уровне значения второго аргумента, поиск успешен. Если же нет, но первый член последовательности есть атом, то отбрасываем его и ищем далее рекурсивным вызовом. Если, наконец, последовательность начинается подпоследовательностью `e.1`, производим поиск в `e.1`. Если он успешен, то на этом и заканчиваем, а если нет — ищем еще в остатке аргумента `e.2`.

Здесь, впрочем как и в функции `Read` программы для распознавания палиндромов, применяется так называемая *«с»-конструкция* (от предлога «с»). В ней сначала вычисляется выражение над переменными, получившими значения благодаря успешному сопоставлению образца. Затем значение этого выражения сопоставляется по заданному блоку предложений, а результат вызванного сопоставлением вычисления нового выражения выдается как результат исходной функции. Блок предложений играет роль вложенной безымянной функции, которая как бы применяется для уточнения и ветвления основного сопоставления.

Есть еще *конструкция «где»*. По виду она подобна «с»-конструкции и тоже имеет уточняющий сопоставления смысл, но вместо блока и связанного с ним ветвления вторичное сопоставление идет только в одном направлении. Довольно типичный пример использования «где» — проверка принадлежности успешно сопоставленного значения данному множеству. Допустив, что аргумент некоторой функции — цепочка литер, предложение

```
e.1 s.c s.c e.2, 'aeiou': e.3 s.c e.4 = s.c
```

сначала присвоит `s.c` значение дважды подряд встречающейся литеры. Затем новым сопоставлением оно проверит, является ли уже найденное `s.c` одной из гласных букв латиницы и только если это имеет место, выдается результат. Если же вторичное сопоставление неуспешно, предложение попытается удовлетворить основное сопоставление повторно (удлиняя сопоставляемую с `e.1` часть), после чего, в случае успеха, опять произойдет вторичное сопоставление, и т. д.

Если по данному предложению сопоставить цепочку `'attendee'`, произойдет следующее. Сначала, согласно образцу `e.1 s.c s.c e.2`, переменные `e.1`, `s.c` и `e.2` получат значения соответственно `'a'`, `'t'` и `'endee'`. Затем идет попытка сопоставить цепочку `'aeiou'` с образцом `e.3 s.c e.4`, но так как в `'aeiou'` нет `'t'` (значение `s.c`), сопоставление неуспешно. Это приводит к повторному поиску сопоставления с образцом `e.1 s.c s.c e.2`, в результате которого переменным `e.1`, `s.c` и `e.2` будут присвоены `'attend'`, `'e'` и `''`. Теперь уже сопоставление `'aeiou'` с `e.3 s.c e.4` успешно, поскольку `'e'` — значение `s.c` — входит в эту цепочку. Буква `'e'` и выдается в конечном счете предложением. При сопоставлении же цепочки `'Mississippi'` переменной `s.c` значение присваивается три раза, и каждый раз присвоенное отбрасывается конструкцией «где».

Ввиду схожести «с»- и «где»-конструкций и отсутствия ветвления во второй, ее можно рассматривать как частный случай первой. Однако в чем-то она и мощнее: повторение попытки сопоставления, как описано выше, возможно только с применением конструкции «где», но не «с».

Несколько конструкций «где» можно применять последовательно, а конструкции «с» — естественным образом вставлять одну в другую. Два вида конструкций можно и сочетать. Во всех случаях имеет место последовательное уточнение сопоставления.

Заметим что, применяя несколько «где» одну за другой, получаем конвейер поисков сопоставлений с возвратным поведением (backtracking). Точки поиска-возврата вложены одна в другую: откат из любой из них (неуспех сопоставления) служит сигналом возобновления поиска в предыдущей.

С формальной точки зрения конструкции «с» и «где» являются лишь синтаксическими удобствами, без которых можно обойтись, но их практическая полезность высока: они помогают выражаться более четко и непосредственно, уменьшая, в частности, вынужденное введение функций вспомогательного характера.

1.4. Пример: синтаксический анализ выражений языка Рефал

Рассмотрим несколько больший пример программирования на Рефале. Приведенный ниже текст является модулем, ре-

ализующим синтаксический анализ константных выражений языка Рефал. По заданной цепочке текста функция `expr`, следуя правилам грамматики языка, находит записанное в цепочке значение-последовательность.

Из всех функций модуля только `expr` является доступной вне него. Чтобы ссылаться на нее из другого места программы, нужно добавить определение

```
$EXTRN expr;
```

Для выполнения основной работы `expr` вызывает `parse`, предваряя свой аргумент пустой последовательностью `()`, содержание которой растет по ходу распознавания в цепочке значений членов результирующей последовательности. Если цепочка успешно переводится в последовательность на Рефале, `parse` выдаст эту же последовательность в скобках, которые `expr` снимет. Если же цепочку удается перевести не целиком, `expr` получит либо слово `F`, либо результат в скобках с непустым остатком цепочки за скобками — в любом случае цепочка оказалась неправильным выражением и `expr` выдаст `F`.

Вспомогательные функции `name` и `number` распознают соответственно слова простого вида (без `"`) и числа (макроцифры). Функция `quoted` переводит подцепочку вида `'…'` в цепочку литер на Рефале, а `"…"` — в слово общего вида. Ответственность функции `special` — распознание литер, заданных через \; сама она вызывает `xpair` для шестнадцатерично заданных и `srep` — для остальных особых литер.

Для представления различных классов литер в обрабатываемой цепочке используются функции с `letter` по `xdigit`. Они вызываются только с пустым аргументом и не вычисляют ничего, но тем не менее возвращают значения. Такие функции принято применять в роли «глобальных констант» (которыми в собственном смысле язык не располагает).

Функция `size` — оболочка `Lenw`, отбрасывающая лишнюю часть результата последней. (`Lenw` — стандартная функция для нахождения длины последовательности, но, кроме необычного имени, она отличается еще более необычным результатом: часть его — та же последовательность.)

Использованы также стандартные функции `Lower` — для преобразования содержимого цепочки в строчные буквы — и `Implode_Ext` — для получения символьного атома из цепочки.

Любопытной особенностью Рефала-5 является возможность использовать \-запись литер не только между кавычками `'…'` и `"…"`, но также и самостоятельно — тогда подразумевается наличие единичных кавычек. Например, `\" \n` и `\"\n` обе равнозначны `'\"' '\n'` или же `'\"\n'`, а `\'A\'`, так же как и `\' A \'`, обозначает последовательность `' A '` из двух литер `'` со словом `A` между ними. Это учитывается в анализе выражений, при вызове `special` и из `quoted`, и непосредственно из `parse`, а также, для упрощения и сокращения текста, используется в определениях функций везде, где уместно.

```
letter {= 'abcdefghijklmnopqrstuvwxyz'}
digit {= '0123456789'}
namedlm {= '-_'}
spechar {= '\'\"\\()<>tnrx'}
xdigit {= <digit> 'abcdef'}

size {e.x, <Lenw e.x>: s.n e.z = s.n}
```

```
$ENTRY
expr {e.x, <parse () e.x>: {(e.res) = e.res;  e.z = F}}

parse {
  (e.x) = (e.x);
  (e.x) \' e.r1, <quoted \' () e.r1>:
    {F = F;  (e.q) e.r2 = <parse (e.x e.q) e.r2>}
  (e.x) \" e.r1, <quoted \" () e.r1>:
    {F = F;  s.q e.r2 = <parse (e.x s.q) e.r2>}
  (e.x) \\ s.c1 e.r1, <special s.c1 e.r1>:
    {F = F;  s.c2 e.r2 = <parse (e.x s.c2) e.r2>}
  (e.x) s.c e.r1, s.c: {
    s.a, <letter>: e.1 s.a e.2, <name (s.a) e.r1>: s.n e.r2
      = <parse (e.x s.n) e.r2>;
    s.d, <digit>: e.1 s.d e.2, <number (s.d) e.r1>: s.n
        e.r2
      = <parse (e.x s.n) e.r2>;
    '(', <parse () e.r1>:
    {(e.y) ')' e.r2 = <parse (e.x (e.y)) e.r2>; e.z = F}
    ' ' = <parse (e.x) e.r1>;
    s.z = (e.x) s.c e.r1}}

name {
  (e.x) s.c e.r, <letter> <digit> <namedlm>: e.1 s.c e.2
    = <name (e.x s.c) e.r>;
  (e.x) e.r = <Implode_Ext e.x> e.r}

number {
  (e.x) s.c e.r, <digit>: e.1 s.c e.2
    = <number (e.x s.c) e.r>;
  (e.x) e.r = <Numb e.x> e.r}

quoted {
  s.d (e.x) s.d e.r, s.d:
    {\' = (e.x) e.r;  \" = <Implode_Ext e.x> e.r}
  s.d (e.x) \\ s.c1 e.r1, <special s.c1 e.r1>:
    {F = F;  s.c2 e.r2 = <quoted s.d (e.x s.c2) e.r2>}
  s.d (e.x) s.c e.r = <quoted s.d (e.x s.c) e.r>;
  e.x = F}

special {
  s.c e.r1, <spechar>: e.1 s.c e.2, s.c: {
    'x', e.r1: s.a s.b e.r2, <xpair <Lower s.a s.b>>:
      {F = F;  s.x = s.x e.r2}
    e.z = <srep s.c> e.r1}}

xpair {
  s.1 s.2, <xdigit>: e.1 s.1 e.2, <xdigit>: e.3 s.2 e.4
    = <Chr <+ <* 16 <size e.1>> <size e.3>>>;
  e.z = F}

srep {\' = \';  \" = \";  \( = \(;  \) = \);  \< = \<
      \> = \>;  't' = \t;  'n' = \n;  'r' = \r;  \\ = \\}
```

Из допустимых данной реализацией анализатора выражений исключены имена переменных и вызовы функций. Собственно грамматический анализ этих частей не вызвал бы затруднений, и даже по большей части уже сделан. Имя переменной за знаком `.` — либо простое слово, либо число; имя функции — любое слово, а аргумент — любое выражение. Анализ всего этого и так уже реализован. Однако непосредственно представить результат перевода неконстантных выражений невозможно: пришлось бы либо как-то условиться относительно толкования результата, либо перейти с программирования на уровень метапрограммирования, который в таком случае должен быть заранее обеспечен языком. Впрочем, о

второй из этих возможностей упоминается в следующем разделе.

1.5. Дополнительные возможности

Программирующий на Рефале имеет в своем распоряжении некоторые средства императивного программирования, хотя они ограничены и считается хорошим стилем воздерживаться от их использования насколько возможно.

Прежде всего, императивным является, конечно, ввод-вывод. В отношении ввода-вывода через файлы и стандартные устройства в Рефале нет ничего, на что стоило бы обращать внимание знакомому с такими же средствами в популярных императивных языках.

Более интересен и несколько необычен механизм «закапывания/выкапывания».

Выделенной для этой цели функцией можно создать особый объект данных и назначить ему имя. По назначенному имени в других функциях объекту можно присвоить значение или извлечь его. Можно также присвоить значение, сохранив («закопать») уже имеющееся, и делать это сколько нужно раз. И наоборот, сохраненные значения можно извлекать в обратном запоминанию порядке. Таким образом, объект данного типа есть на самом деле стек. Более того, в нем можно сохранять любые значения, а так как функциями управления объектами можно пользоваться везде, то и все эти объекты доступны из любой точки программы. Другими словами, они являются глобальными именованными стеками универсального назначения.

Считается, что стеки глобальных значений могут ускорить доступ к данным, устраняя необходимость явной передачи последних между функциями, однако по этому пути можно очень легко прийти к программе, «нефункциональной» не только с точки зрения стиля выражения, но в каком угодно смысле.

В целях поддержки создания больших программ их можно разбить на модули, для каждого модуля определяя доступные из других модулей имена его функций. Пример модуля уже был показан в предыдущем разделе.

В каждом модуле может быть определена и сделана доступной функция `Go` — это будет означать, что модуль может быть главным при некотором выполнении программы. Какому из таких модулей действительно быть главным — уточняется лишь в момент запуска программы: исполнение начинается с функции `Go` этого модуля.

Диалектом Рефал-5 активно поддерживается возможность метапрограммирования. Через встроенную функцию отрывок программы можно перевести в похожую, но все же отличающуюся форму: это названо «погружением в метакод». И наоборот, полученный указанным или другим способом метакод при помощи другой функции «подымают из метакода» и тут же исполняют. Так как метакодовым представлением можно орудовать как данными, то можно подвергнуть преобразованию любую часть программы, включая сам преобразователь, а также можно и порождать-исполнять совсем новые куски. Таким способом можно организовать разного рода кусочное, контролированное исполнение, в отладочных и других целях.

Данный процесс — отнюдь не интерпретирование текста. Погружение можно делать лениво, помечая, а не преобразуя соответствующие части, с тем чтобы подъем тоже ничего не преобразовал. Кроме того, закодированная программа частично выполняется, насколько это возможно в зависимости от наличия в ней переменных с неопределенными значениями, так что при подъеме выполняется только некоторое остаточное множество действий.

Поддержка метапрограммирования отсутствует в более поздних диалектах, то ли ради простоты реализации, то ли для упрощения определения самого языка.

В реализации Рефала предусмотрена возможность трассировки. О ней идет речь в следующем разделе.

1.6. Трассировщик

Интересной и практически ценной особенностью трассировщика является то, что точки прерывания задаются указанием функции и образца аргумента, при соответствии с которым происходит прерывание. При этом образец не обязан совпадать с образцом, используемым в определении функции.[3]

Введение новых, «трассировочных» образцов и переменных превращает трассировщик в инструмент исследования программы, при помощи которого без каких бы то ни было изменений в ней самой можно проводить наблюдения над ее поведением с разных точек зрения. Рассмотрим для примера построение двоичного дерева поиска из целых чисел. Дерево либо пусто, либо имеет вид

(*левое поддерево*) *число* (*правое поддерево*)

Функция `insert` вталкивает число в дерево:

```
insert {s.x = () s.x ();
  s.x (e.1) s.y (e.2), <Compare s.x s.y>: {
    '—' = (<insert s.x e.1>) s.y (e.2);
    s.c = (e.1) s.y (<insert s.x e.2>)}}
```

Функция же `build` строит дерево из последовательности чисел в структурных скобках, накапливая результат во «втором аргументе»:

```
build {  () e.r = e.r;
  (s.x e.z) e.r = <build (e.z) <insert s.x e.r>>}
```

Допустим, в программе она вызывается так:

```
<build (3 1 6 7 2 5 4)>
```

Запустив программу через трассировщик, установим такую точку прерывания:

```
set <build (5 4) e.r>
```

Это момент, в котором в изначально пустом дереве уже вставлено все за исключением последних двух чисел, `5` и `4`. Частичный результат-дерево будет доступен через переменную `e.r`. Командой `g` исполняем программу до указанной точки останова и написав `p e.r` печатаем значение `e.r`:

```
(() 1 (() 2 ())) 3 (()6 (() 7 ()))
```

Это же, но в контексте вызова функции, в котором оно имеет место, можно получить и командой `p v` («показать текущее выражение»).

Почти то же самое, на чуть нижнем уровне вычисления, можно получить, выбрав точку останова другим образом:

```
set <insert 2 e.r>
```

Теперь дерево (`p e.r`) состоит только из первых четырех чисел (`3`, `1`, `6` и `7`), а числу `2`, являясь аргументом `insert`, лишь

[3] Е. Кирпичёв указывает на то, что трассировщик системы программирования языка Erlang работает аналогичным образом.

предстоит быть добавленным. `p v` показывает, что вычисляемое выражение состоит из вызова `build` и вложенного в нем вызова `insert`.

Установив точку прерывания

```
set <insert 4 t.1 t.rt t.3>
```

можно проследить путь включения числа `4`: скомандовав `g` и затем `p t.rt` получим `3`; повторив то же самое еще два раза, получим еще `6` и `5` — включение происходит в (под)деревья с корневыми узлами `3`, `6` и `5`, в этом порядке.

Еще возможность — проследить все включения узлов `s.n` в поддеревья, состоящие к моменту включения из единственного узла `s.rt` (листевые узлы):

```
set <insert s.n () s.rt ()>
```

— в данной точке программа остановится четыре раза, со значениями `s.n` и `s.rt` соответственно `1` и `3`, `7` и `6`, `2` и `1` и `4` и `5`.

Так же прост перехват включений узлов `s.n` в поддеревья с корневыми узлами `s.rt`, у которых пусто только левое поддерево:

```
set <insert s.n () s.rt (t.1 t.2 t.3)>
```

(Для данного примера единственная пара таких узлов — `5` и `6`.

Функция `build` здесь определена в хвосто-рекурсивной форме, а ее результат накапливается явным образом в ее аргумент. Ее можно было бы определить и несколько проще:

```
build { = ;  s.x e.z = <insert s.x <build e.z>>}
```

и вызывать

```
<build 3 1 6 7 2 5 4>
```

однако в такой форме ее не так удобно трассировать. Хвостовая рекурсия не только более эффективна, но и лучше уживается с наблюдением поведения программы.

1.7. Диалекты

Рефал был создан во второй половине 1960-х годов В. Турчиным, сначала в качестве метаязыка для описания семантики других языков, а вскоре затем — и как универсальный язык программирования для обработки текстовой и символьной информации.

Практическое применение Рефала со временем привело к совершенствованию элементов первоначального — не очень удобного — синтаксиса, а также к формированию диалектов языка. Для удобства обозначения исходного языка, который и оказался общим подмножеством диалектов, ввели понятие Базисный Рефал.

В настоящее время основными диалектами языка считаются Рефал-2, Рефал-5, Рефал-6 и Рефал+. Рефал-2 [15] — самый ранний, появившийся почти непосредственно после рождения языка. На протяжение десятка или больше лет он оставался единственным диалектом и фактически отождествлялся с самым языком. С современной точки зрения его синтаксис громоздок и устарел [14]. Хотя реализации Рефала-2 имеются и сегодня, он не развивался уже много лет и, если он где-то и используется, то это незаметно.

Рефал-5 [16, 2, 21] создан в середине 1980-х тем же В. Турчиным.[4] Он тоже на сегодня не развивается,[5] но остается в употреблении. Его следует считать классическим диалектом Рефала. (Напомним, что в этой статье рассматривается почти исключительно Рефал-5.)

Рефал-6 [17, 10] был задуман как реализация Рефала-5, но в конце концов вырос в самостоятельный диалект. Через него в язык вошли обработка неуспехов сопоставлений и функций и другие удобства, а также, на библиотечном уровне, ассоциативные таблицы и другие дополнительные структуры данных. На сегодня он тоже не развивается. Любопытная особенность: единственно в этом диалекте реализованы вещественные числа и арифметика над ними. Вместе с тем, однако, в нем нет средств для ввода таких чисел или извлечения их значений по текстовой цепочке с соответствующим содержанием.

Рефал+ (также Рефал Плюс) [18, 8] является самым продвинутым диалектом, и притом единственным, который развивался (правда, не сам язык, но его реализация) в последние несколько лет. Дополнительные по сравнению с остальными диалектами свойства Рефала+ сохраняют стиль программирования на Рефале и общий вид программ, но повышают их выразительность и лаконичность. Некоторые из этих дополнений — таблицы, перехват неуспехов — подобны имеющимся в Рефале-6.[6] Примеры других свойств: неограниченно большие целые числа как атомарные значения, а не последовательности макроцифр; возможность формировать признак успеха сопоставления на основе успехов и неуспехов частичных сопоставлений; типизирование и обязательное прототипирование функций (в терминах структурных типов — атомов, термов и последовательностей — значений Рефала).

По-видимому, часть нововведений Рефала+ привела к значительному увеличению количества синтаксических правил и ухудшению понимаемости определения языка и программ на нем. В грамматике Рефала+ имеется примерно 100 синтаксических категорий (в языке C их немного больше 60). К тому же, среди имен этих категорий встречаются «забор», «перестройка», «огражденная тропа», «образцовое распутье», «жесткое выражение» и «жесткий край». Последнее вызывает симпатию к неординарности мышления авторов диалекта, но вряд ли способствует овладению языка.

1.8. Достоинства и недочеты

Во время своего создания язык Рефал был в нескольких важных отношениях весьма передовым и даже опережающим свою современность. *Лежащие в его основе идеи, тем более их сочетание, предвосхитили тенденции в развитии программирования, проявившиеся лишь десятки лет спустя.* Перечислим основные достоинства Рефала.

- Декларативный, а не командный и не основанный на понятии состояния, стиль программирования. Рефал — один из первых и очень немногих таких языков.

- Сопоставление с образцом как способ определения функций и структурирования вычислений разветвлением. И в этом отношении Рефал сильно опередил другие языки. Можно дополнить, что применение сопоставления в программировании исследовалось в то же самое время еще очень ограниченно и отнюдь не в декларативном контексте.

[4] К тому времени автор языка работал уже в США, где некоторое время преподавал Рефал в Городском университете Нью-Йорка (The City University of New York).

[5] Хотя планы на то есть — см. [6], раздел «Задачи».

[6] На ранних стадиях своего развития два диалекта взаимно обогащались.

- У функции лишь один аргумент и результат. Разнообразие и общность применимости достигаются не количеством аргументов, а приданием подходящего строения единственному аргументу функции. Так же и насчет результата. Этот принцип хорошо известен по современным функциональным языкам, где ему следуют практически везде, но сорок с лишним лет назад господствовали другие представления.
- Неограниченная последовательность с симметричным доступом к элементам — основная структура данных в языке. Последовательности являются гораздо более ценной структурой по сравнению с (однонаправленными) списками.[7]
- Реализация посредством компилирования исходной программы в программу на языке виртуальной машины, которая затем интерпретируется.
- Автоматическое выделение и освобождение памяти.

Последние два свойства тоже не так часто встречались во время становления Рефала.

Удивительно, но данная совокупность свойств как будто относится к языку, изобретенному сегодня, и к тому же продвинутому!

У Рефала, однако, есть и немало недостатков. Часть их относится к языку вообще, другая — к Рефалу как к представителю функционального стиля. Рассмотрим наиболее существенные.

У Рефала отсутствует возможность построения и именования структур данных и тем более — задания определенных программистом типов. «Знание типов» со стороны языка исчерпывается различением атомарного значения от составного (последовательности) и видов атомарных значений (числа, литеры и символы). Это приемлемо для небольших программ, но сильно затрудняет создание крупных.

Проблемы возникают даже на уровне основных видов значений. За исключением Рефала+, большое целое число, и даже небольшое отрицательное — не атомарное значение, а последовательность. Из последнего вытекает необходимость рассматривать два вида чисел — атомарные и составные, а также вкладывать составное число в другую последовательность, скобками отгораживая его от окружающего контекста — без этого произошло бы сцепление. И, конечно же, из-за этих усложнений возрастает ожидаемость ошибок в текстах программ.

Аскетизм в отношении типов проявляется и в отсутствии в языке булевых значений и булевой арифметики. Вместе с тем нет и какой бы то ни было формы условного ветвления, если не считать выбора предложения в рамках данной функции на основе успешности сопоставления с одним из нескольких образцов. К примеру, сравнение по величине двух чисел состоит в применении функции `compare`, выдающей `'+'`, `'-'` или `'0'`, с последующим сопоставлением результата с этими тремя образцами. Однако успех/неуспех сопоставления — всего лишь неполноценная и часто неуклюжая имитация булевой арифметики и выбора действия.

Для выражения повторяющихся действий в функциональном языке естественно рассчитывать на рекурсию. Во многих языках, однако, для часто возникающих схем повторения предусматриваются специализированные под них конструкции высокого уровня, или по крайней мере стандартные функции, которые, вбирая в себя рекурсию, скрывают ее от программиста. Такими являются, скажем, определители списков (list comprehensions), функции вида *map*, *zip*, *fold* и пр. В Рефале подобного вида конструкций нет и поэтому рекурсию приходится выражать всегда в явной форме. Это приводит к большому количеству вспомогательных функций, из-за чего программа имеет тенденцию становиться чересчур раздробленной, громоздкой, а ее смысл — расплывчатым.

Заметим, что проблема раздробленности и расплывания смысла усугубляется тем, что функции не могут быть вложенными,[8] а значит, отношение подчиненности или различение смысловых уровней нельзя полноценно передать структурой текста программы.

С другой стороны, почти недоступна и возможность создания функций высшего порядка. Это потому, что функции в Рефале не есть значения: нельзя создавать безымянные функции, тем более замыкания. Самое близкое к функции-значению, что является возможным — взять имя или адрес данной функции: в этом отношении Рефал не превосходит С.

Функции стандартной библиотеки[9] тоже не имеют ничего общего с функциональным стилем программирования. На самом деле, библиотеку трудно отнести даже вообще к Рефалу. Трудно объяснить почти полное отсутствие в ней функций для работы с текстовыми цепочками, равно как и с последовательностями — ведь именно это и есть данные в языке. Отсутствуют даже арифметические функции нахождения абсолютного значения, меньшего или большего из двух чисел и обращения знака.

У механизма сопоставления имеется то неудобство, что в рамках образца нельзя выразить ни альтернирование, т. е. ветвление сопоставления, ни повторение, за исключением одинаковых частей, цитируемых одной и той же переменной. Другими словами, обобщение сопоставлений, аналогичное легко выражаемому примерно аппаратом регулярных выражений, в Рефале невозможно. Вследствие того некоторые задачи, которые очень легко решить регулярными выражениями, требуют неестественно больших и запутанных программ на Рефале.

Представляется весьма полезным (но в Рефале не так) иметь возможность обращаться с образцами как с данными. Использование переменных образцов повысило бы гибкость сопоставлений. С другой стороны, образцы можно было бы использовать и как определители типов аргументов и результатов функций.

Недостатком Рефала, препятствующим его применению в современном программировании, является и «замкнутость» языка — прежде всего, отсутствие программного интерфейса к другим языкам и средств обмена данными через Интернет.

Наконец, имеющиеся описания действующих реализаций языка несколько неполны и в какой-то степени устарели.

Все приведенные выше критические замечания относятся в полной мере к Рефалу-5. Рефал-6 и Рефал+ восполняют только небольшую часть указанных пробелов и только частичным образом.

[7] Кстати, спискам и сегодня в Haskell, ML и т. д. уделяется, похоже, незаслуженно большое внимание по сравнению с последовательностями с прямым доступом. По-моему, это — атавизм.

[8] За исключением вложения блоков «с»-констукций, если рассматривать их как функции.

[9] Точности ради, библиотечными они являются в Рефале-6 и Рефале+, а в Рефале-5 они считаются встроенными.

1.9. Перспективы применения и развития

Несмотря на описанные недостатки, Рефал можно успешно применять в нескольких областях. Может быть самая главная из них — преподавание и изучение начал программирования, а также применений программирования для решения задач в других дисциплинах. Основанием тому служит простота языка и легкость моделирования решений многих не очень сложных задач, где естественно работать с текстовой или символьной информацией. Примеры такого применения даются в [13, 12, 11].

Рефал можно также с успехом использовать для решения многих практических задач анализа и преобразования текста, в том числе для преобразования текстов на формальном (алгоритмическом, описательном и др.) языке и для анализа текстов на естественном языке. Например, преобразование может состоять в снабжении тегами HTML/XML или TEX некоторого текста на основе распознания в нем определенной структуры. Или такой текст, уже снабженный тегами, скажем, HTML, переразметить в LATEX, или переформатировать. В этом отношении интерес может представлять [3], где защищается перспективность применения Рефала для работы с семейством языков XML.

Также благоприятными для Рефала областями являются распознавание и алгоритмы над абстрактными структурами — такими как, например, в области искусственного интеллекта — где нужно интенсивно применять поиск по структурному или содержательному образцу.

Нельзя не отметить, что в СССР Рефал уже имел активное использование в разработке программного обеспечения: систем компьютерной алгебры, компиляторов, программ для специализированных компьютеров, в том числе встраиваемых в изделия космической промышленности [9]. Был также создан экспериментальный Рефал-процессор. Считается, что Рефал сыграл решающую роль для успеха изготовления адаптивной технологии построения серии компиляторов с языка Фортран для специализированных компьютеров. Похоже, что немало из перечисленных работ велись в закрытых или ограниченного доступа научных и научноприкладных центрах, и поэтому публикаций, свидетельствующих о них, почти нет. Некоторые все же можно найти в библиографическом хранилище сайта [20].

Вместе с самым языком В. Турчиным предложен и общий метод глобального оптимизирующего преобразования программ, названный им «суперкомпиляцией» (от англ. *supervising compilation*), в основе которого лежат методы, похожие на абстрактную интерпретацию. Установлено, что благодаря такой глобальной оптимизации из некоторых программ можно получить чувствительно более быстродействующие варианты. Идея суперкомпиляции описана первоначально в [4] и позднее, более подробно, в [5].

Суперкомпилятор под именем Scp4 распространяется с вебсайта Рефала-5 [16] и на сегодня является, по всей видимости, самым достопримечательным применением языка Рефал.

Хотя долгое время суперкомпиляция развивалась в среде только Рефала, в последнее время идут исследования по применению метода к другим функциональным языкам, прежде всего Haskell.

Упомянем две разработки, чья цель — сочетание Рефала с широкоиспользуемыми программами [1]. Refal-SciTE — интегрированная среда программирования на Рефале-5, основанная на очень компактном и гибко настраиваемом текстовом редакторе SciTE. Refal-PHP — система программирования, объединяющая использование Рефала и PHP; там, где как правило используется PHP, доступным становится и Рефал.

Разработчики Рефала+ нашли перспективным реализовать Рефал (в принципе, все главные диалекты) переводом в C++, Java и возможно других языках [7]. Для этой цели они определили промежуточный язык, целевой для трансляторов всех диалектов, который и переводится в упомянутые языки. Окончательно исполняемая программа получается вызовом транслятора конкретного языка.

Понятно, что применение Рефала ограничено из-за приведенных в предыдущей части недостатков языка, поэтому очень жаль, что ни один из основных диалектов на настоящий момент не развивается. Все же попытки осовременивания языка делаются. Так, Рефал-7 [19] вводит вложенные и безымянные функции, а также функции высшего порядка, чем стремится поставить язык, можно сказать, в истинно функциональном контексте: ведь действительно надо считать неестественным в функциональном языке неприятие функций в роли равноправных данных.

К примеру только заметим что, приняв на вооружение безымянные функции, открывается возможность выражать, скажем, композицию таких функций в виде функции, и вообще программировать непосредственно функциями. Хотя дух Рефала несколько иной — ориентированный прежде всего на данные — но программирование на уровне функций является слишком ценной формой абстракции, чтобы пренебречь ею.

Другой отпрыск под именем D-Refal [22] задался целью повысить практическую применимость языка в работе с текстом, совершенствуя механизм образцов. Он вводит конструкции группирования, альтернирования, повторения и отрицания образцов, наподобие образующим операциям регулярных выражений. Программисту предоставляются и определяемые программистом типы: это, по существу, именованные образцы, которые цитируются в других образцах, но их сила больше, чем только в сокращении записи, поскольку они могут быть и рекурсивными, а значит — выражать неограниченную повторность некоторой части сопоставления. Кроме того, D-Refal прилагает усилия в направлении повышения быстродействия сопоставления путем устранения избыточности в нем с помощью ссылок на значения и отсечений. О практической направленности свидетельствуют также введение вещественных чисел и принятие Юникода в роли алфавита обрабатываемого текста.

К сожалению, упомянутые два диалекта не имеют ни популярности основных, ни даже статуса «официально признанных»: на сайте «сообщества Рефала» [20] они не упомянуты и, по-видимому, не обсуждались. Впрочем, и тоже к сожалению, в данном сообществе Рефал уже годами никак и не обсуждается.

Независимо от указанных и, возможно, других улучшений Рефала в будущем, или даже отсутствия улучшений, кажется привлекательным реализовать Рефал как программную библиотеку для языка C. Это позволило бы применять Рефал в качестве расширяющего, сценарного языка почти везде, и заодно — через программирование на C — быть расширяемым самому. Чрезвычайно успешный пример такой симбиотической

реализации — язык Lua. Структура Рефала тоже благоприятствует данному подходу.

1.10. Аналогии

Подходя к концу, вкратце сравним Рефал с другими языками, решающими подобные задачи.

Prolog — хорошо известный пример тоже декларативного языка и тоже основанного на сопоставлении. Однако (оставим в стороне другие различия) в программе на Prolog-е предметом сопоставления является цель, результат, и действие программы состоит в поиске аргументов, при которых цель удовлетворяется, или же в проверке того, удовлетворяется ли она при заданных аргументах. В Рефале же сопоставление касается аргументов, а цель не формулируется явным образом. Похоже что в ряде случаев «прямой» подход Рефала приводит к более наглядным программам, чем «обратный» Prolog-а, но также возможно, что в других случаях как раз наоборот. Что точно известно, так это то, что разница существенным образом сказывается на стиле программирования. Рефал, однако, однозначно выигрывает своими симметричными последовательностями против однонаправленных списков Prolog-а.

Нельзя не упомянуть и Snobol. Этот язык был еще в 1970-е и остается мощным средством программирования в той же области, что и Рефал — текст и символьные преобразования. В Snobol-е тоже основа вычислений — сопоставления, но тем сходство с Рефалом кончается. В этом языке образцы — полноправные данные иерархической структуры, включающие, помимо прочего, вызовы функций и присваивания переменным, которые могут быть безусловными или зависеть от успеха сопоставления. Успешные сопоставления сопровождаются заменой распознанной части текстовой цепочки новым текстом. В этом смысле, хотя Snobol совсем не функциональный язык, он даже в большей степени «марковский» чем Рефал. В отношении сопоставительной семантики Snobol, несмотря на возраст, очень продвинут. С другой стороны, быстродействие программ на Рефале, как правило, чувствительно выше.

TXL и OmniMark — два современных языка, в которых программа представляет собой совокупность правил распознавания и преобразования текста. В отличие от Рефала управление последовательностью действий в них осуществляется на событийном принципе: выбирается правило с успешным сопоставлением (распознавание — это активирующее работу данного правила событие), в нем происходит некоторое преобразование текста, затем опять распознающим событием выбирается правило и т. д., пока возможно. Можно сказать, что оба языка работают на несколько более высоком семантическом уровне. TXL можно считать чисто функциональным, хотя в нем, как в Рефале, функции не есть данные. Практически ценной особенностью OmniMark-а является возможность режимного переключения с общего распознавания на текст, размеченный в SGML, а значит и XML или HTML.

1.11. Благодарности и уточнения

Мне очень приятно выразить благодарность редакторам журнала ПФП за приглашение написать эту статью, за ценные советы по улучшению ее содержания, а также за языковую помощь по превращению моего текста в не слишком ужасное для читателя испытание.

Все оставшиеся несовершенства выражения — по вине ограниченности моего владения русским языком.

Также хочу поблагодарить рецензентов статьи за их полезные замечания, которые помогли сгладить неточности и неясности изложения и дополнить его.

Наконец, хочу подчеркнуть, что все мнения здесь — лично мои и ни в какой мере не претендуют на разделяемость в том или ином сообществе программистов.

Литература

[1] Refal-SciTE и Refal-PHP. — URL: http://refal.net/~belous (дата обращения: 11 апреля 2011 г.).

[2] *Turchin V. F.* Refal-5 programming guide & reference manual. — URL: http://refal.botik.ru/book/html (дата обращения: 11 апреля 2011 г.).

[3] *Turchin V. F.* Refal: the language for processing XML documents. — URL: http://www.math.bas.bg/bantchev/place/refal/refal-for-xml.pdf (дата обращения: 11 апреля 2011 г.).

[4] *Turchin V. F.* A supercompiler system based on the language Refal // *ACM SIGPLAN Notices*. — 1979. — Vol. 14, no. 2. — Pp. 46–54. — (URL: http://www.math.bas.bg/bantchev/place/refal/sc-based-on-refal.djvu (дата обращения: 11 апреля 2011 г.)).

[5] *Turchin V. F.* The concept of a supercompiler // *ACM Transactions on Programming Languages and Systems*. — 1986. — Vol. 8, no. 3. — Pp. 292–325. — (URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.6414 (дата обращения: 11 апреля 2011 г.)).

[6] «Институт Рефала». — URL: http://refal.botik.ru (дата обращения: 11 апреля 2011 г.). — Онлайн библиотека и другие ресурсы по Рефалу, нормальным алгорифмам и пр.

[7] Викисайт посвящённый развитию Рефала (в основном Рефал+). — URL: http://wiki.botik.ru/Refaldevel (дата обращения: 11 апреля 2011 г.).

[8] *Гурин Р. Ф.* Язык программирования Рефал Плюс. — URL: http://wiki.botik.ru/Refaldevel/RefalPlusBook (дата обращения: 11 апреля 2011 г.).

[9] *Ефимов Г. Б. Зуева Е. Ю.* Из истории развития и применения компьютерной алгебры в Институте прикладной математики им. М. В. Келдыша. — URL: http://www.ict.edu.ru/ft/004313/prep2003_27.pdf (дата обращения: 11 апреля 2011 г.).

[10] *Климов А. В.* Программирование на языке Рефал. — URL: http://refal.net/~arklimov/refal6/manual.htm (дата обращения: 11 апреля 2011 г.). — (описание Рефала-6).

[11] *Корлюков А. В.* Лекционные записки по Рефалу. — URL: http://www.refal.net/~korlukov/refbook (дата обращения: 11 апреля 2011 г.).

[12] *Немытых А. П.* Введение в Рефал с примерами задач для школы. — URL: ftp://www.botik.ru/pub/local/scp/refal5/nemytykh_informatica_N9_2008_pp25-32.pdf (дата обращения: 11 апреля 2011 г.).

[13] *Немытых А. П.* Лекционные записки по Рефалу. — URL: http://www.botik.ru/pub/local/scp/ugp/seminars.html (дата обращения: 11 апреля 2011 г.).

Литература

[14] Примеры программ на Рефале-2. — URL: http://www.cnshb.ru/vniitei/sw/refal (дата обращения: 11 апреля 2011 г.).

[15] Рефал-2 – главный сайт. — URL: http://refal.net/~belous/refal2-r.htm (дата обращения: 11 апреля 2011 г.).

[16] Рефал-5 – главный сайт. — URL: http://botik.ru/pub/local/scp/refal5/refal5.html (дата обращения: 11 апреля 2011 г.).

[17] Рефал-6 – главный сайт. — URL: http://refal.net/~arklimov/refal6 (дата обращения: 11 апреля 2011 г.).

[18] Рефал+ – главный сайт. — URL: http://rfp.botik.ru (дата обращения: 11 апреля 2011 г.).

[19] *Скоробогатов С. Ю.* Язык Refal с функциями высшего порядка. — URL: http://iu9.bmstu.ru/science/refal.php (дата обращения: 11 апреля 2011 г.).

[20] Сообщество «Рефал/Суперкомпиляция». — URL: http://refal.net (дата обращения: 11 апреля 2011 г.). — Ссылки на реализации и публикации и другие ресурсы.

[21] *Турчин В. Ф.* РЕФАЛ-5. Руководство по программированию и справочник (перевод несколько устаревшего в отношении определения языка варианта книги [2]). — URL: http://refal.org/rf5_frm.htm (дата обращения: 11 апреля 2011 г.).

[22] Язык программирования D-Refal. — URL: http://ulm.uni.udm.ru/~soft/d-refal (дата обращения: 11 апреля 2011 г.).

Circumflex — веб-фреймворк на Scala comme il faut

Александр Темерев
atemerev@fprog.ru

Аннотация

Веб-фреймворк Circumflex — одна из сравнительно недавних разработок на Scala, и, на взгляд автора, гораздо менее известная, чем она того заслуживает. В статье описываются отличия Circumflex от других веб-фреймворков, применение языковых средств Scala для создания необходимых фреймворку DSL, подход к структурированию Scala-проектов, применяемый разработчиками фреймворка, и другие его интересные особенности.

Circumflex is a relatively new player among Scala web frameworks; it appears to be much less popular than it deserves, from author's point of view. This article outlines the differences between Circumflex and other web frameworks, its employment of Scala language features to build useful DSLs, its clever approach to structuring of Scala code and other interesting features.

Обсуждение статьи ведется по адресу:
http://fprog.ru/2011/issue7/alexander-temerev-circumflex/discuss/.

2.1. Немного истории

Со времён триумфального явления Ruby on Rails типичный веб-фреймворк включает в себя следующие компоненты:

Роутер — компонент, делегирующий создание ответов на HTTP-запросы с различными URL разным обработчикам. Часто включает в себя некий DSL, с помощью которого можно задавать шаблоны URL и передаваемых по HTTP данных, извлекать из них нужные параметры и вызывать нужную функцию-обработчик. Иными словами, это пример специализированного pattern matching, хорошо знакомого функциональным программистам.

Один из наиболее популярных подходов к роутингу в веб-фреймворках впервые был реализован в Sinatra[1], где средствами Ruby был создан мини-DSL вида:

```
get '/hello/:name' do |n|
  "Hello #{n}!"
end
```

После Sinatra подобные реализации стали появляться и в других фреймворках, различаясь лишь адаптацией DSL к синтаксису используемого языка программирования. В Scala такой подход реализован как минимум в двух фреймворках: Scalatra, ранее известном как Step, и герое данной статьи — Circumflex.

Вот как аналогичный роутер будет выглядеть здесь:

```
class SomeRouter extends RequestRouter {
    get("/hello/:name") = "Hello, " + param("name") + "!"
}
```

Этого уже достаточно для написания REST-сервисов (поскольку вместо `get()` можно написать и `post()`, и `delete()`, кроме того, имеются и средства работы с HTTP-заголовками и куками), однако большинство веб-разработчиков хотят отдавать на HTTP-запросы некий HTML, и очень не хотят собирать его руками. Из этого очевидно следует, что фреймворку нужен…

Шаблонизатор (*template engine*) — механизм, который из шаблона и данных генерирует некоторый размеченный текст (обычно HTML/XHTML). Поскольку разработка шаблонизатора, по-видимому, требует программистов с определённым складом ума, их обычно делают отдельно, а авторы веб-фреймворков используют готовые решения (хотя, конечно, есть и исключения). По большому счёту, шаблонизатор — это функция вида `process(template: String, data: Any): String`; тоже тривиальное явление для функциональщиков.

Единообразия в используемых различными веб-фреймворками шаблонизаторах нет. Некоторые уходят разумом в чистый XSLT, и скорбь о них сравнима со скорбью о тех, кто использует Lisp. Другие придумывают собственные решения разной степени повёрнутости (Freemarker, Velocity); ещё более некоторые разрешают применение в шаблонах непосредственно кода на основном языке программирования (многочисленные *SP, т. е. JSP, ASP, SSP, и т. д.), кое-кто стремится к максимальной «сухости» выражения шаблонов, не повторяя ничего более одного раза (Haml, Jade); прочие дают выбор между разными вариантами (Scalate). Обычно интегрировать поддержку нового шаблонизатора в веб-фреймворк — дело одного-двух дней, а то и быстрее. Circumflex поддерживает Freemarker, а также (экспериментально) Scalate и входящие в его состав синтаксисы (Mustache, Scaml, Jade, SSP). Рендеринг темплейта, как и положено, осуществляется одной строчкой кода, вроде такой:

```
ftl("/path/to/template.ftl") // данные для рендеринга
    берутся из контекста
```

Впрочем, в недалёком будущем, когда весь веб станет асинхронным, а AJAX и Comet будут применяться не эпизодически, а в качестве основы основ, шаблонизаторы на стороне сервера окажутся ненужными, а вместо них будут использоваться решения, работающие прямо в браузере, на JavaScript. Из активно применяемых на данный момент автор знает два: jQuery templates[2] и Pure.js[3], причём больше всего ему нравится именно последний, сочетающий, на его взгляд, преимущества XSLT и отсутствие его недостатков. При этом серверу достаточно отдавать браузеру лишь одну стартовую HTML-страницу, а всё остальное, в идеальном случае, можно реализовать с помощью REST-сервисов.

Подсистема хранения данных (*persistence framework*). До Ruby on Rails этот компонент не считался неотъемлемой частью веб-фреймворка, и каждый придумывал свои способы борьбы с SQL. После Rails многие фреймворки обладают интегрированным ORM-движком; в самом Rails использовался паттерн Active Record, который приводил (и приводит) к невозможности применения одних и тех же объектов как для моделирования, так и для persistence, в случае сколько-нибудь сложной объектной модели — не все приложения являются data-centric. С другой стороны, заметная их часть всё-таки являются таковыми, а использование в ORM объектов предметной области прямо без модификаций, как правило, всё равно является утопией (Java-программисты наверняка вспомнят ту или иную разновидность интерфейса `Identifiable`, например, или мучения с Hibernate и его отображением реляционных связей в коллекции) — хотя попытки такие есть; интересующиеся могут посмотреть на Squeryl. Так или иначе, Active Record используется и в Circumflex ORM. Зато в сочетании с синтаксическими возможностями Scala становится возможным реализовать вот такой DSL (пример скопирован из документации Circumflex):

```
// Подготовить отношения, которые будут участвовать в
    запросах
val ci = City as "ci"
val co = Country as "co"

// Выбрать все города Швейцарии, вернуть Seq[City]:
SELECT (ci.*) FROM (ci JOIN co) WHERE (co.code LIKE "ch")
    ORDER_BY (ci.name ASC) list

// Выбрать страны и количество городов в них,
    вернуть Seq[(Option[Country], Option[Long])]:
SELECT (co.* → COUNT(ci.id)) FROM (co JOIN ci) GROUP_BY
    (co.*) list
```

Стоило ли огород городить, если на выходе всё равно получается нечто, что очень похоже на SQL? Выражаясь языком Lurkmore, «вы так говорите, как будто SQL — это что-то плохое». Получившийся DSL является, тем не менее, корректным Scala-кодом, и может быть использован внутри Scala-программ со всеми возможностями языка, правильной типизацией, явно заданной (опять же с помощью Scala-кода) схе-

[1] http://www.sinatrarb.com

[2] http://api.jquery.com/category/plugins/templates/

[3] http://beebole.com/pure/

мой данных и некоторыми другими преимуществвами, вроде таких:

```
class City extends Record[Long, City] {
  val country = "country_code".TEXT
        .REFERENCES(Country)
        .ON_DELETE(CASCADE)
}

class Country extends Record[String, Country] {
  def cities = inverseMany(City.country)
}
```

Двумя строчками кода мы создали двустороннюю one-to-many ассоциацию, отображаемую в корректный DDL со столь любимыми адептами реляционной модели cascading deletes и внешними ключами. Все остальные ожидаемые означенными адептами вещи тоже работают. Подробную информацию можно найти в документации Circumflex-ORM[4], которая, как и вся прочая документация этого фреймворка, великолепна.

2.2. Show me your code!

Что ж, напишем один из стандартных тестовых проектов для веб-фреймворка: *TODO List* (второй стандартный проект — блог, но его делать немного дольше). Джентльменские соглашения и знаменитая видеопрезентация Ruby on Rails требуют, чтобы создание подобного приложения занимало не более 15 минут. Что ж, постараемся уложиться.

Приборы и материалы:

- Scala 2.8.1
- SBT 0.7.4

Установив SBT согласно прилагаемой инструкции[5], инициализируем проект, запустив `sbt` в его каталоге и ответив на задаваемые вопросы, например, так: `project name: todo`, `organization: fprog.ru`, `version: 1.0`, `Scala version: 2.8.1`, `sbt version: 0.7.4`. После чего нужно создать Scala-класс с конфигурацией проекта: `project/build/TodoProject.scala`, в котором прописать зависимости проекта — помимо самого Circumflex, нам понадобятся СУБД (возьмём H2 Database), система логгирования (Logback) и веб-сервер, точнее, сервлетный контейнер (Jetty). Таким образом, класс будет выглядеть примерно так:

```
import sbt._

class TodoProject(info: ProjectInfo) extends
    DefaultWebProject(info) {

  val cxVersion = "2.0.1"
  val jetty7 = "org.eclipse.jetty" % "jetty-webapp" %
      "7.0.2.v20100331" % "test"

  override def libraryDependencies = Set(
    "ru.circumflex" % "circumflex-web" % cxVersion %
        "compile->default",
    "ru.circumflex" % "circumflex-orm" % cxVersion %
        "compile->default",
    "ru.circumflex" % "circumflex-ftl" % cxVersion %
        "compile->default",
    "com.h2database" % "h2" % "1.3.153",
    "ch.qos.logback" % "logback-core" % "0.9.27",
    "ch.qos.logback" % "logback-classic" % "0.9.27"
  ) ++ super.libraryDependencies
}
```

Теперь запустим в каталоге проекта `sbt update`, и через некоторое время, зависящее от скорости вашего интернет-соединения, но в любом случае довольно продолжительное, SBT выкачает все зависимости и положит их в каталог `lib_managed`. Отлично, теперь можно начинать писать код.

На этом месте наши читатели разделятся на тех, кто работает с текстовыми редакторами и тех, кто работает с IDE. Первым ничего делать не надо, а вторым нужно найти способ открыть проект в IDE. Поскольку единственной сколько-нибудь годной IDE для программирования на Scala на данный момент является лишь IntelliJ IDEA с установленным Scala Plugin (шаманские прелести его установки мы здесь расписывать не будем), автор рекомендует использовать IDEA processor для SBT[6]. Так или иначе, попробуем для начала сделать «Hello, world».

Немного энтерпрайза: написанное на Circumflex является веб-приложением в смысле J2EE, и может существовать в J2EE-среде. Слова «enterprise edition» в аббревиатуре J2EE должны как бы говорить нам, что без XML и деплоймент-дескрипторов мы не обойдёмся. Так и вышло: нам нужно создать файл `src/main/webapp/WEB-INF/web.xml`, в котором указать, что все запросы к приложению будут проходить через класс `CircumflexFilter`. Делается это так:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
     xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <filter>
    <filter-name>Circumflex Filter</filter-name>
    <filter-class>ru.circumflex.web.CircumflexFilter
        </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>Circumflex Filter</filter-name>
    <url-pattern>*</url-pattern>
  </filter-mapping>
</web-app>
```

Принеся жертву кровавым богам энтерпрайза, можно заняться, собственно, написанием хэллоуворлда. В каталоге `src/main/scala` создаём файл `router.scala` со следующим содержимым:

```
package ru.fprog.todo

import ru.circumflex._, core._, web._, freemarker._

class MainRouter extends RequestRouter {
  get("/") = "Hello, world!"
}
```

Теперь надо указать, что именно этот роутер будет обслуживать наши запросы, для чего создаём конфигурационный файл `src/main/resources/cx.properties`, содержащий одну строку: `cx.router=ru.fprog.todo.MainRouter`. Запускаем в каталоге проекта `sbt` и пишем там `jetty-run`. Заходим на `http://localhost:8080/` и видим там жизнерадостный хэллоуворлд.

[4] http://circumflex.ru/docs/orm/assembly.html
[5] http://code.google.com/p/simple-build-tool/wiki/Setup
[6] https://github.com/mpeltonen/sbt-idea

Тут некоторые читатели, не имеющие до сих пор опыта работы с Java, должны были в ужасе закрыть эту статью и заняться решением задач из Project Euler на хаскеле. Что делать, наследие Java-среды тяжко, и объёмы предварительной конфигурации для новых проектов остаются велики, несмотря на все усилия последних лет — и именно за подобную ерунду многие Java-программисты получают деньги. Scala-сообществу же пока лень переписывать то, что уже работает на JVM многие годы. Но есть и плюсы: совместимость со стандартными J2EE-контейнерами, лёгкость деплоймента и широчайшие возможности по автоматизации всего этого энтерпрайза. Выбирать вам, хотя некоторые просто уходят на Python. Ну и попутного ветра им, а мы продолжим написание нашего туду-листа, потому что за хэллоуворлд много денег не получить.

Займёмся моделью данных. Для работы с persistence, как вы уже, наверное, догадались, необходимо сконфигурировать СУБД. Она у нас встроенная, поэтому конфигурация ограничится добавлением следующих строк в `cx.properties`:

```
orm.connection.driver=org.h2.Driver
orm.connection.url=jdbc:h2:~/todo
orm.connection.username=sa
orm.connection.password=
```

(URL в данном случае оканчивается директорией и префиксом имени файлов, в которых будет располагаться база данных; если захламлять домашний каталог не хочется, то можно выбрать любой другой.)

Создадим файл `src/main/scala/model.scala` и напишем там, собственно, класс и companion object к нему для самих туду-листов:

```
package ru.fprog.todo

import ru.circumflex.orm._
import java.util.Date

// соотвествует таблице todo_entry в базе данных
class TodoEntry extends Record[Int, TodoEntry] with
    IdentityGenerator[Int, TodoEntry] {

  // поля в таблице объявляются почти как в SQL;
  // id генерируется автоинкрементом
  val id = "id".INTEGER.NOT_NULL.AUTO_INCREMENT
  // текст записи
  val text = "text".VARCHAR(255).NOT_NULL
  // время создания записи
  val start = "start".TIMESTAMP.NOT_NULL
  // время пометки записи как «выполненной»
  val end = "end".TIMESTAMP

  // конструктор инициализирует поля с помощью
  // специального оператора :=
  def this(text: String) {
    this()
    this.text := text
    this.start := new Date
  }

  // указываем первичный ключ
  def PRIMARY_KEY = id

  // и companion object, где прописаны индексы
  // и прочие объекты БД, относящиеся к таблице
  def relation = TodoEntry

  // если запись помечена как «выполнена»,
  // у неё определено время выполнения
  def isDone: Boolean = this.end.get.isDefined
  // метод, который «выполняет» задачу, присваивая
  // полю end текущее время
  def markDone() { this.end := new Date }
  // экспортируем в XML, точнее, в XHTML. По-хорошему,
  // нужно сериализовать в JSON и работать с шаблонами
  // на стороне клиента, но пока «шаблон» будет прямо
      здесь:
  val sdf = new SimpleDateFormat("MMM dd, yyyy")
  def toXml =
      <li id={"l" + this.id()}>
        {sdf.format(this.start())}: {this.text()}
        <button class="done" id={"d" +
            this.id()}>Done</button>
      </li>
}

// объект-компаньон с дополнительными сведениями
object TodoEntry extends TodoEntry with Table[Int,
    TodoEntry] {
  // определяем два индекса на полях start и end —
      пригодятся
  val idxStart = "idx_start".INDEX("start")
  val idxEnd = "idx_end".INDEX("end")
}
```

Если всё компилируется (`sbt compile` в каталоге проекта), то можно создать базу данных с этой схемой (состоящей из одной таблицы). Для этого напишем новую команду `sbt` — это несложно, нужно всего лишь создать новый метод в классе конфигурации проекта. Сначала — кусок кода, создающий схему (дописываем в `model.scala`):

```
object CreateSchema {
  def main(args: Array[String]) {
    val ddl = new DDLUnit(TodoEntry)
    ddl.DROP_CREATE
  }
}
```

Теперь — собственно метод (точнее, lazy val), регистрирующий команду `sbt` (в `TodoProject.scala`)

```
lazy val schema =
    runTask(Some("ru.fprog.todo.CreateSchema"),
    runClasspath)
  .dependsOn(compile, copyResources) describedAs "Create
      database schema"
```

Теперь запускаем в корне проекта `sbt schema` и убеждаемся, что всё работает хорошо и база данных создалась там, где нужно.

Время переходить к веб-части проекта. Откроем `router.scala`, где томится наш хэллоуворлд, удаляем все, что там есть, и вместо этого пишем такой обработчик:

```
post("/api/task") = {
  val text = param("text")
  val task = new TodoEntry(text)
  task.save()
  // результат — кусок XML с датой создания и текстом
      todo-записи
  task.toXml
}
```

(не будем возиться с JSON-сериализацией, а сразу же будем возвращать XML, даже XHTML, чтобы потом вставить

его джаваскриптом в DOM — в классическом духе AJAX. Уф. С таким количеством buzzwords успех приложения обеспечен). Обработка ошибок здесь отсутствует, желающие могут обернуть обработчик в `try...catch` самостоятельно.

Компилируем, запускаем консоль `sbt`, там — `jetty-run`, и проверяем получившееся с помощью `curl`. Например, так:

```
curl -d 'text=Test' http://localhost:8080/api/task
```

На выходе должен быть, очевидно, кусок XHTML, например, такой:

```
<li>Mar 28, 2011: Test <button class="done"
    id="d{this.id()}">Done</button></li>
```

Но записалось ли задание в базу данных? Проверим, создав метод, который выводит все несделанные задания в туду-листе. Напишем для начала соответствующий селектор в `model.scala` (в companion object):

```
object TodoEntry {
  ...
  def todoEntries = {
    val te = TodoEntry AS "te"
    SELECT(te.*) FROM (te) WHERE (te.end IS_NULL) list
  }
}
```

Потом добавим обработчик в `router.scala`:

```
// Вполне функционально!
get("/api/tasks") =
    TodoEntry.todoEntries.map(_.toXml).mkString("\n")
```

Компилируем, перезапускаем Jetty, проверяем:

```
curl http://localhost:8080/api/tasks
```

Результат должен совпадать с предыдущим методом. Несложно также добавить ещё одно задание и убедиться, что и оно присутствует в выдаче. Другое дело, что веб-приложения обычно смотрят браузером, а не с помощью curl. Следовательно, нужны шаблоны. Сделаем шаблон на FreeMarker[7], для чего создадим файл `src/main/resources/template.ftl`, и напишем там следующее:

```
[#ftl]
<!DOCTYPE html>
<html>
<head>
    <title>Todo</title>
</head>
<body>
<h1>Todo</h1>
<input type="text"/><button id="add">Add</button>
<ul>
[#list todo as entry]
  <li id="l${entry.id}">
    ${entry.start?date}: ${entry.text}<button class="done"
        id="d${entry.id}">Done</button>
  </li>
[/#list]
</ul>
</body>
</html>
```

Убираем из `router.scala` GET-обработчик `/api/tasks`, он нам больше не нужен, и добавляем вместо него такой:

```
get("/") = {
  //параметр контекста, который потом передаётся шаблону
  'todo := TodoEntry.todoEntries
  ftl("template.ftl")
}
```

Компилируем, запускаем Jetty, заходим на http://localhost:8080[8] и видим отображенный шаблон. Нда. Призов за дизайн, конечно, он не возьмёт, но взять CSS в руки вы можете и самостоятельно. Другое дело, что кнопки тоже не работают, а ведь был обещан AJAX. Надеваем резиновые перчатки, скачиваем свежий jQuery[9] в `/src/main/webapp/public/js/jquery.js`, создаём рядом с ним файл `todo.js`, открываем его и принимаемся за дело. Начнём с кнопки *Add*, тем более что обработчик для добавления новых заданий уже реализован:

```
function onAddButtonPress() {
    var text = $('input').val();
    jQuery.ajax({
        type: 'POST',
        url: '/api/task',
        data: { 'text' : text },
        success: function(response) {
            var dom = $(response);
            $('ul').append(dom);
        },
        error: function(response) {
            "Error happened: " + response;
        },
        dataType: 'html'
    });
}
```

Чуть сложнее с кнопкой *Done*, потому что обработчика, отмечающего задания как выполненные, у нас пока нет. Для этого нужно, во-первых, написать в объекте `TodoEntry` (`model.scala`) селектор для получения задания по идентификатору:

```
def byId(id: Integer) = {
  val te = TodoEntry AS "te"
  SELECT(te.*) FROM (te) WHERE (te.id EQ id) unique
}
```

Во-вторых, нужно, собственно, реализовать обработчик:

```
post("/api/done") = {
  val id = param("id").toInt
  val task = TodoEntry.byId(id)
  task.end := new Date
  task.save()
  "OK"
}
```

В-третьих, немного джаваскрипта:

```
function onDoneButtonPress(target) {
    var id = event.target.id.substr(1);
    jQuery.ajax({
        type: 'POST',
        url: '/api/done',
        data: { 'text' : text },
        success: function(response) {
            $('li#l' + id).remove();
        },
```

[7] http://freemarker.sourceforge.net

[8] http://localhost:8080

[9] http://jquery.com

```
        error: function(response) {
            "Error happened: " + response;
        }
    });
}
```

Назначаем кнопкам обработчики событий:

```
$(document).ready(function() {
    $('button#add').bind('click', onAddButtonPress);
    $('button.done').live('click', onDoneButtonPress);
}
```

Дописываем в `template.ftl` ссылки на jQuery и наш код на JavaScript:

```
<script language="javascript" src="/js/jquery.js"></script>
<script language="javascript" src="/js/todo.js"></script>
```

Компилируем, запускаем — работает! По крайней мере, должно. Исходный код этого приложения также выложен на github.[10]

Говоря о Circumflex, нельзя не сделать лирическое отступление и отметить его документацию. Прежде всего, это один из представителей занесённого в Красную Книгу вида проектов, которые используют literate programming — точь-в-точь как завещал нам дедушка Кнут. Для этого они сделали пригодным для практического применения и портировали на Scala проект Docco,[11] который позволит забыть про Javadoc и его аналоги как страшный (очень страшный) сон. Качество получаемой документации продемонстрировано на http://circumflex.ru/api[12] (да, это API-документация). После того, как вы подберёте с пола упавшую челюсть, можно скачать проект Circumflex-Docco, Maven-плагин к нему, и наслаждаться тем же самым в собственном коде. Или не наслаждаться, потому что даже самые удобные утилиты по экстракции и просмотру документации не освобождают вас от её, документации, написания.

Поскольку у Scala есть довольно продвинутая система пространств имён, которая поначалу вызывает лёгкое раздражение, но впоследствии оно сменяется недоумением вида «почему раньше так не делали?» — Circumflex использует её на полную катушку и, в частности, отказывается от доставшегося нам от Java анахронизма в виде повторения структурой каталогов проекта его структуры пакетов (packages). То, что в 1996 году казалось гениальным техническим решением, сейчас кажется карго-культом — в конце концов, почему бы не поручить рутинную работу определения, в каком файле находятся какие пакеты, компилятору? (да, скорость компиляции Scala-кода вызывает подозрение о наличии у Мартина Одерски финских или эстонских корней). Так или иначе, в Circumflex код группируется вертикально и горизонтально в соответствии с функциональностью, в одном файле могут находиться разные пакеты, и хотя это звучит ересью, выглядит всё очень хорошо. Особенно в сочетании с документацией. Начиная новый Scala-проект, необязательно с использованием Circumflex, стоит задуматься — не начать ли применять ту же систему?

Авторы Circumflex придерживаются в разработке принципа «ничего лишнего, максимум гибкости». По мнению автора статьи, это гораздо лучше, чем принципы «включить в себя всё, что только можно — потому что это энтерпрайзи» (привет, Lift[13] и Vaadin[14]), или «convention over configuration — мы всё решим за вас, и вы сможете написать блог за 5 минут; вот над чем-то кроме блога придётся попотеть» (привет, Ruby on Rails и Django). Хотя, конечно, решать вам — а для этого ничего не надо, кроме свежей версии Scala, SBT и немного здорового хакерского духа. Увидимся на IPO.

[10] http://github.com/atemerev/todo
[11] http://jashkenas.github.com/docco/
[12] http://circumflex.ru/api
[13] http://liftweb.net/
[14] http://vaadin.com/home

Разработка алгоритма обнаружения движения в среде программирования *Mathematica*

Вадим Залива
lord@crocodile.org

Аннотация

В данной статье мы рассмотрим пример использования среды *Mathematica* для быстрого прототипирования простого алгоритма обнаружения движения. Кроме общих приёмов работы с *Mathematica*, мы познакомим читателя с некоторыми понятиями из области машинного зрения, цифровой обработки изображении и сигналов. При разработке мы применим некоторые из подходов функционального программирования, поддерживаемых *Mathematica*.

In this article, we will use Mathematica to develop a prototype of a simple motion detection algorithm. We will introduce the basics of using Mathematica environment and use this exercise to illustrate some basic concepts from the fields of machine vision, digital image processing and signal processing. We will be using some of the functional programming capabilities provided by Mathematica.

Обсуждение статьи ведется по адресу:
http://fprog.ru/2011/issue7/vadim-zaliva-motion-detection/discuss/.

3.1. Задача

Недавно передо мной встала задача разработать программу, которая позволяет использовать мобильный телефон (iPhone) в качестве системы сигнализации. Телефон при помощи видеокамеры наблюдает за помещением и, если обнаружено движение, подает звуковой сигнал, фотографирует движущийся объект и передает изображение в Интернет. Ключевая часть этой программы — это алгоритм определения движения. Конечно, можно было бы его сразу написать его на Objective-C и производить отладку на телефоне, но такой процесс был бы довольно-таки трудоемким. Вместо этого мы решили использовать среду программирования *Mathematica*[1] для проверки и отладки алгоритма, который затем будет переписан на Objective-C.

Пакет *Mathematica* — продукт фирмы *Wolfram Research* — является мощным вычислительным инструментом, используемым в академической и инженерной среде. Работа обычно ведется в *записных книжках* — интерактвных документах содержащих формулы, графики, данные и текст. Они обычно сохраняются в файлах с расширением *.nb*. Код выполняется по мере его ввода и результаты выводятся попеременно с кодом. Это чрезвычайно удобный режим для интерактивной работы. Сложные функции можно оформить в виде отдельных *модулей* (файлов с расширением *.m*), на которые можно ссылаться из записных книжек.

Пакет *Mathematica* умеет работать с видеокамерой, но поскольку нашей целью было быстрое прототипирование алгоритма, то мы не воспользовались этой возможностью, а работали с заранее записанным видеофайлом. Также, поскольку этот алгоритм планировалось в дальнейшем реализовывать на другом языке, мы старались применять простейшие базовые функции, а не использовать более мощные встроенные функции пакета. Так, например, мы не воспользовались стандартными функциями работы с изображениями для масштабирования кадров и перевода их в серое цветовое пространство (grayscale).

Первая рабочая версия данного алгоритма, включая запись тестового видеоролика (созданную при помощи того же iPhone), была разработана примерно за 2 часа. Это демонстрирует удобство пакета *Mathematica* для быстрого решения подобных задач. В дальнейшем алгоритм был реализован и использован в программе iSentry для iPhone[2].

Для начала мы записали тестовый видеоролик, на котором производилась отладка алгоритма. Это шестисекундный ролик, на котором на столе под лампой стоит детская игрушка (динозавр). Через некоторое время в кадре появляется iPhone, брошенный на стол. Еще через некоторое время в кадре появляется рука, которая двигает игрушку. Ролик можно скачать отдельно[3].

3.2. Алгоритм

Работа с видео и картинками

Запустим пакет *Mathematica* и создадим пустую *записную книжку*. Допустим, что ролик сохранён в файле с именем *movement.mov* и находится в той же директории, что и наша *записная книжка*. Теперь мы можем набрать нашу первую строку кода на *Mathematica*:

```
> raw = Import[ToFilename[NotebookDirectory[],
    "movement.mov"], "Data"];
```

Синтаксис пакета *Mathematica* использует для указания аргументов функций квадратные, а не круглые скобки. Функция *NotebookDirectory* возвращает полный путь каталога, в котором расположена текущая *записная книжка*. Параметр *ToFileName* формирует абсолютный путь из каталога и имени файла. Обычно функция *Import* сама пытается «угадать» формат файла, но можно его указать и явно. Для каждого поддерживаемого типа файлов эта функция умеет загружать разные элементы. В данном случае нас интересуют данные в виде массивов чисел, что мы и указываем при помощи второго параметра «*Data*». Результат присваивается переменной с именем *raw*. Точка с запятой после выражения подавляет вывод результата выполнения непосредственно под строкой с выражением. Результатом выполнения функции будет список кадров.

Чтобы узнать количество кадров в прочитанном файле, можно воспользоваться функцией *Length*. Результатом работы функции будет количество кадров (элементов списка), сохранённое в переменной *nframes*.

```
> nframes = Length[raw]
196
```

В случае вложенных структур данных *Length* считает количество элементов на самом верхнем уровне. В данном случае получилось 196 кадров. Если же мы захотим получить все измерения (предполагая что, имеет место регулярная многомерная структура), то можно воспользоваться функцией *Dimensions*:

```
> Dimensions[raw]
{196, 240, 320, 3}
```

Теперь рассмотрим подробнее, как представлены кадры. Базовая структура данных — это список. В данном случае у нас есть список из 196 элементов (кадров). Каждый кадр — это матрица размером 240 строк по 320 элементов каждая. В *Mathematica*, в отличие от языков типа R, нет специальной структуры данных для матриц — они представлены как вложенные списки. Каждый элемент представляет собой список из трёх значений, соответствующих RGB-компонентам цветовой модели. Каждая компонента может принимать значения от 0 до 255. Чтобы узнать значение цвета первого пикселя в первой строке первого кадра, достаточно написать следующее выражение (поскольку квадратные скобки используются в нотации вызова функций, для доступа к элементам массивов используются двойные квадратные скобки):

```
> raw[[1,1,1]]
{42, 33, 20}
```

В *Mathematica* есть встроенный тип данных *Image*, который мы будем использовать для визуализации данных. Все, что нам нужно знать об этом типе — это то, что можно его создать из матрицы значений пикселей (первый кадр) с помощью следующей команды:

```
> Image[raw[[1]], "Byte"]
```

[1] Wolfram Research, Inc., Mathematica, Version 8.0, Champaign, IL (2010).
[2] http://itunes.apple.com/us/app/isentry/id396777365
[3] http://www.crocodile.org/lord/MotionDetection/movement.mov

Перевод в серую шкалу

Для определения движения информация о цвете нам не нужна, и поэтому нам будет гораздо проще работать не с цветной картинкой, представленной 3-мя компонентами, а с изображением в серой шкале, где цвет представлен одним значением. Для перевода RGB-цвета в серую шкалу мы воспользуемся известной формулой:

$$I = 0.3 \times R + 0.59 \times G + 0.11 \times B$$

Эта формула основана на чувствительности человеческого глаза к различным частям спектра [1]. Поскольку в нашем случае картинки будет обрабатывать программа, а не человеческий глаз, то точные значения не так важны, и поэтому можно было бы использовать 33% для каждого компонента.

Преобразование всех кадров в серую шкалу выполняется следующей командой:

```
> orig = Map[(# . {0.3,0.59,0.11})&, raw, {3}];
```

Мы используем функцию *Map*, которая применяет указанное выражение к каждому элементу структуры данных. В функциональных языках обычно эта функция определена для списков и применяет выражение к элементам списка первого уровня. В *Mathematica* она также позволяет работать с вложенными списками произвольной глубины. Функция *Map* имеет три параметра. Первый параметр — это *функция*. Второй параметр — это список произвольной вложенности. Третий, опциональный параметр указывает, к каким уровням вложенности списка применять указанную функцию. В нашем случае его значение *{3}* обозначает, что применять функцию нужно к элементам 3-го уровня вложенности. Напомним, что в нашей структуре данных первый уровень — это кадры, второй — строки кадра, а третий — пиксели в строке. То есть, функция будет применена к каждому пикселю, который будет ей передан как список из трёх элементов. Поскольку у нас нет отдельной функции для перевода RGB в серую шкалу, то первым параметром *Map* мы передаем то, что в *Mathematica* называется *pure function*, а в других языках известно как *анонимная функция*, или *лямбда-выражение*. В нашем примере она определена при помощи оператора '&'. Выражение перед '&' является телом функции. На формальные параметры можно ссылаться, используя имена *#1, #2, #3* и так далее. Для удобства *#1* можно сокращать до *#*. Вычисления, которые мы делаем, довольно просты: список компонентов цвета для каждого пикселя мы поэлементно умножаем на коэффициенты {0.3,0.59,0.11} и суммируем то, что получилось (используя скалярное произведение векторов). Результатом работы всего выражения будет список кадров, в котором каждый элемент представлен уже не тройкой RGB, а одним значением оттенка серого цвета.

Image умеет работать и с такими изображениями, и мы можем вывести первый кадр, чтобы взглянуть на результат:

```
> Image[orig[[1]], "Byte"]
```

Масштабирование

Исходные кадры имели размер 320x240 пикселей. Для нашей задачи даже такое довольно-таки низкое разрешение излишне. Кроме того, поскольку наше приложение будет работать на мобильном телефоне, где процессорные ресурсы ограничены и использование CPU связано с расходом батарейки, то имеет смысл уменьшить кадры, по крайней мере, в два раза по каждому измерению, что уменьшит количество пикселей в четыре раза. В принципе, нам не обязательно масштабировать пропорционально, но для простоты мы будем использовать пропорциональное масштабирование, так что фактор масштабирования у нас будет одинаков для обоих измерений:

```
> scaleFactor = 2;
```

Процесс масштабирования картинки (с уменьшением размера) на множитель *scaleFactor* называется *downsamping*. Обычно этот процесс делается в 2 этапа:

1) Применение фильтра нижних частот (low-pass filter) для удовлетворения теоремы Котельникова [2] (в англоязычной литературе известной как *Nyquist–Shannon sampling theorem*);
2) Уменьшение количества пикселей выбором элементов с шагом *scaleFactor*.

Рассмотрим эти шаги подробнее. Не вдаваясь в теорию информации, достаточно сказать, что на шаге 1 мы применим так называемый усредняющий фильтр. Интуитивное определение этого фильтра довольно простое: каждая точка будет заменена на среднее значение ее окрестности с заданным радиусом. В более общем случае, определение значения точки через ее окрестность — довольно частая операция в обработке изображений и обычно выражается через *свертку*. В функциональном анализе свертка двух функций f и g определяется как:

$$[f * g](x) = \int_{-\infty}^{\infty} f(u)g(x-u)du$$

Для двухмерных растровых изображений мы можем записать дискретную форму этой формулы:

$$F(x,y) = \sum_i \sum_j f(i,j)g(x-i, y-j)$$

В этой формуле *f(x,y)* — это значения пикселей изображения до применения свертки. *F(x,y)* это функция, которая дает значения после применения свертки. Собственно свертка задается функцией *g(x,y)*. Для большинства практических применений функция *g* опередена на небольшом диапазоне значений (обычно в несколько пикселей). Ее удобно представлять с помощью матрицы, которую иногда называют *ядром свертки*. Применение такой операции свертки к изображениям очень хорошо распараллеливается и часто выполняется при помощи GPU.

Теперь представьте, что мы возьмем матрицу размером $(2 * \text{scaleFactor} + 1) \times (2 * \text{scaleFactor} + 1)$. В нашем случае в ней будет $(2*\text{scaleFactor} + 1)^2$ элементов. Установим каждый элемент этой матрицы равным $\frac{1}{(2*\text{scaleFactor}+1)^2}$.

```
> meanKernel = BoxMatrix[scaleFactor]/((2*scaleFactor +
    1)^2);
> MatrixForm[meanKernel]
```

$$\begin{pmatrix} \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \end{pmatrix}$$

Такую матрицу можно получить, выполнив следующую команду:

```
> meanKernel =
    BoxMatrix[scaleFactor]/((2*scaleFactor+1)^2);
> MatrixForm[meanKernel]
```

Это и есть наше ядро свертки для усредняющего фильтра с радиусом *scaleFactor*. Если его применить к каждому пикселю изображения, то мы получим среднее значение из окрестности 25-ти соседних пикселей, включая текущий.

Функция *ListConvolve* позволяет применить ядро к списку значений. Количество измерений списка и ядра могут быть произвольными, но должны совпадать. Дополнительный параметр уточняет, как обрабатывать значения около краев, когда применяемый фильтр выходит за границы данных. Детальное описание опций *ListConvolve* можно посмотреть в документации по *Mathematica*. Используя уже знакомую нам функцию *Map*, можно применить *mean* фильтр ко всем кадрам:

```
> origc = Map[ListConvolve[meanKernel, #, {-1, -1}, 0]&,
    orig];
```

Как и ожидалось, результирующая сглаженная картинка выглядит «размытой». Не зря *mean* фильтр еще называют сглаживающим фильтром.

```
> Image[origc[[1]], "Byte"]
```

Теперь можно выполнить второй шаг и выбрать пиксели с шагом *scaleFactor*. Выбор элементов из списка можно сделать при помощи функции *Take*. В нашем случае мы указываем ей два критерия выбора — по одному на каждое измерение. Критерий задан в виде списка вида *{начальное_значение, конечное_значение, шаг}*.

```
> origdim = Dimensions[orig]
{196, 240, 320}
> bwm = Map[Take[#, {1, origdim[[2]], scaleFactor}, {1,
    origdim[[3]], scaleFactor}]&, origc];
```

В результате получаем массив кадров уменьшенного размера:

```
> Dimensions[bwm]
{196, 120, 160}
```

Давайте посмотрим, что у нас получилось. На этот раз мы не просто визуализируем изображение одного из кадров, а применим функцию под названием *Manipulate*. Это один из очень удобных и мощных инструментов в *Mathematica*, позволяющий визуализировать данные, динамически изменяя значения параметров. Опять же, мы не будем вдаваться во все возможности и способы использования этого механизма, а покажем его простейший вариант, который очень уместен в нашем случае:

```
> Manipulate[Image[bwm[[f]], "Byte"], {f, 1, nframes, 1}]
```

Первым параметром передается выражение. В нашем случае оно преобразует кадр с номером *f* из списка *bwm* в формат *Image*. Заметьте, что в отличие от *Map*, это не функция, а просто фрагмент кода, и, соответственно, мы не используем оператор '*&*'. Второй параметр — это список, состоящий из имени переменной, начального значения, конечного значения и шага. Функция *Manipulate* позволяет пользователю изменять значения переменной, и при каждом изменении пересчитывает выражение с новыми значениями, показывая результат. Можно изменять более одного значения, но в нашем случае нам достаточно номера кадра. Двигая ручку *f*, пользователь может покадрово прокручивать получившееся видео. Можно также перейти к произвольному кадру, введя его номер, или включить механизм автоматической прокрутки (анимацию).

Мы закончили предварительную обработку данных, и теперь готовы перейти собственно к алгоритму определения движения.

Оценка количества движения

Как мы будем определять движение? Простейший подход — смотреть, насколько изменилось изображение между кадрами. Просто сравнивать кадр с предыдущим будет недостаточно, так как программа будет чувствительна к мелким вариациям освещения и цифровому «шуму» камеры. Более надежный подход — сравнивать текущий кадр с усредненным значением нескольких предыдущих. Это будет довольно легко реализовать и при работе в режиме реального времени, накапливая несколько последних кадров в специальном буфере и используя их для сравнения. Такое сглаживание кадра по нескольким предыдущим называется частным случаем *каузального фильтра (causal filter)*. Название подчеркивает, что его значение зависит только от текущего и предыдущих кадров, но не от будущих.

Для начала определим функцию, которая берет несколько кадров и строит на основании их «средний» кадр. Определение довольно-таки очевидное:

```
> avgFrame[d_] := Total[d]/Length[d];
```

Список кадров указан формальным параметром *d_*. В *Mathematica* существует довольно мощный механизм выбора подходящей функции на основании типов и количества параметров. Формальные параметры задаются в виде *шаблонов (patterns)*. Все, что нужно знать на данном этапе — это то, что в определении функции имена формальных параметров должны содержать символ подчеркивания («_»), а в теле функции на них нужно ссылаться без этого символа. Нашей функции *avgFrame* передается на вход список кадров в виде матриц. Функция *Total* их суммирует поэлементно. Получившуюся матрицу мы делим на количество кадров, тем самым получив на выходе матрицу средних значений пикселей заданных кадров.

Теперь можно определить функцию, реализующую сглаживающий фильтр. На вход передаем список кадров и размер окна сглаживания (количество кадров). На этот раз используем операцию *Map* не по самим кадрам, а по их индексам. Из исходного массива мы выбираем подмножество элеменов, используя двойные квадратные скобки (оператор *Part*). Диапазон выбираемых значений задается в формате *от ;; до* (оператор *Span*).

```
>avgFilter[l_, w_] := Map[avgFrame[
   l[[Max[1,#-w] ;; #]]]&, Range[Length[l]]];
```

Теперь мы можем применить наш фильтр. Значение окна подбирается экспериментально в зависимости от желаемой «чувствительности» алгоритма и частоты получения кадров. Допустим, мы будем использовать значение 10. При частоте 30 кадров в секунду это соответствует одной третьей доле секунды.

```
> bwms = avgFilter[bwm, 10];
```

Таким образом мы получили кадры, которые «сглажены» по времени. Если их визуализировать, то в местах, где нет движения, изображение мало отличается от оригинального, а в местах, где наблюдается движение, движущиеся части немного размыты. Сам по себе этот промежуточный результат малоинтересен. Более интересно взглянуть на разницу между несглаженным по времени кадром (*bwm*) и сглаженным (*bwms*). На статических кадрах разницы не будет, и результатом будет кадр с нулевыми значениями пикселей (черный). На кадрах, где присутствует движение, мы увидим изменившуюся часть. Посмотрим на пару показательных кадров:

В кадре номер 50 телефон движется, «въезжая» в кадр:

```
> Image[bwm[[50]]-bwms[[50]], "Byte"] // ImageAdjust
```

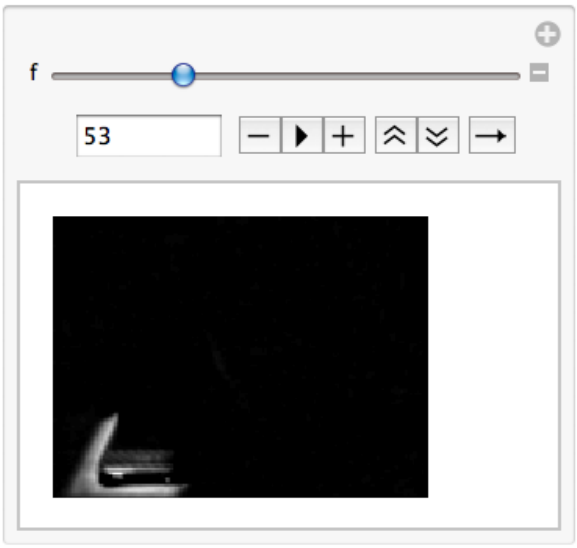
В кадре 139 мы видим движущуюся руку:

```
> Image[bwm[[139]]-bwms[[139]], "Byte"] // ImageAdjust
```

Кроме знакомой функции *Image*, тут использована еще функция *ImageAdjust*. Эта функция масштабирует значения пикселей, чтобы они были в диапазоне от 0 до 1, что позволяет лучше рассмотреть темные кадры. Кроме того, тут использован оператор «//». Это постфикс-нотация применения функций. Запись *F[x] // G* эквивалентна *G[F[x]]*.

И, как обычно, можно воспользоваться функцией *Manipulate*, чтобы посмотреть на произвольные кадры, интерактивно прокручивая список.

```
> Manipulate[ImageAdjust[Image[bwm[[f]]—bwms[[f]], "Byte"],
     {f, 1, nframes, 1}]]
```



Попробуем представить объем изменений в виде одного числа. Количество движения в указаном кадре корреллирует с тем, насколько текущий кадр (из списка *bwm*) отличается от усредненного значения нескольких предыдущих (из списка *bwms*). Оценить разницу этих двух кадров удобнее всего с помощью среднеквадратичного отклонения разницы соответствующих пикселей, поделенного на среднее значение пикселя в текущем кадре:

$$\mathrm{CV(RMSD}(x,y)) = \frac{\mathrm{RMSD}(x,y)}{\bar{x}} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i - y_i\right)^2}}{\bar{x}}$$

На *Mathematica* это можно выразить вот так:

```
> errs = Map[RootMeanSquare[Flatten[bwm[[#]]—bwms[[#]]]] /
              Mean[Flatten[bwm[[#]]]],
            Range[nframes]];
```

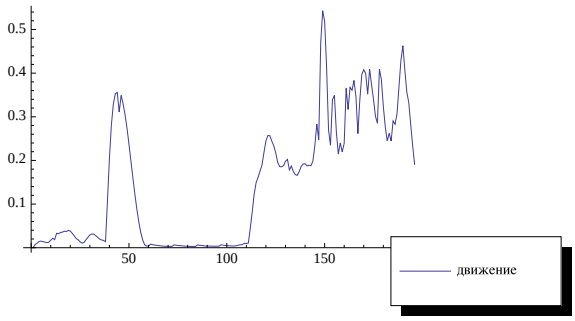
Заметим, что мы уменьшаем размерность данных при вызове функции *Mean*, используя функцию *Flatten*. В противном случае мы получим не одно среднее значение по всем пикселям кадра, а вектор средних значений столбцов.

В переменной *errs* у нас должен был получится список значений, отражающих количество движения в каждом кадре. Попробуем нарисовать график этих значений. Команда *Needs* позволяет подгружать внешние модули. В данном случае мы используем модуль *PlotLegends*, позволяющий показывать на графике легенду.

```
> Needs["PlotLegends'"]
```

Собственно рисование графика выполняется функцией *ListLinePlot*.

```
> ListLinePlot[errs, PlotLegend —> {"движение"},
     LegendPosition —> {0.8, —0.8}]
```



Можно заметить два всплеска активности: первый в диапазоне кадров 40—60, а второй — начиная с кадра 110.

Сглаживание сигнала при помощи *линейной регрессии*

Используя получившийся график, уже можно попробовать обнаружить движение, но для начала имеет смысл его немного сгладить, чтобы уменьшить влияние случайного шума. Сглаживать будем опять же при помощи *каузального фильтра*. На этот раз применим для сглаживания линейную регрессию. Идея довольно проста — в качестве сглаженного значения точки будем использовать ее регрессию по нескольким предыдущим точкам. Для этого постараемся найти прямую, которая ближе всего описывает предыдущие точки. Искомое значение — точка на этой прямой с абсциссой, соответствующей текущему (последнему) значению.

Рассмотрим пример, взяв произвольный диапазон значений из списка *errs*. Допустим, что диапазон задан начальным и конечным значениями *f* и *t*.

```
> f = 3; t = 13;
```

Сформируем список точек, заданных парой координат. Первая координата — порядковый номер. Вторая — собственно значение. Список первых координат легко получить при помощи функции *Range*, список вторых — используя оператор *Span (;;)* из списка *errs*. Их можно поместить в список из двух элементов, используя фигурные скобки. Если рассматривать получившуюся структуру как матрицу, то ее размерность будет 2 строки и 11 столбцов. Транспонировав ее при помощи функции *Transpose*, мы получим нужный нам список пар:

```
> fdata = Transpose[Range[f, t], errs[[f ;; t]]]
{{3,0.0104027},{4,0.0139062},{5,0.0151486},
 {6,0.0138507},{7,0.012893},{8,0.0117535},
 {9,0.0125816},{10,0.0172805},{11,0.0215209},
 {12,0.0187361},{13,0.0330796}}
```

Применим линейную регрессию. Для начала мы это сделаем при помощи функции *Fit*. Первым параметром передадим пары значений. Второй параметр задает тип многочлена, коэффициенты, которого мы собираемся вычислить. Третий перечисляет переменные многочлена. Результатом является выражение, использующее эти переменные.

```
> fit = Fit[data, {1, x}, x]
0.00495072 + 0.00143972 x
```

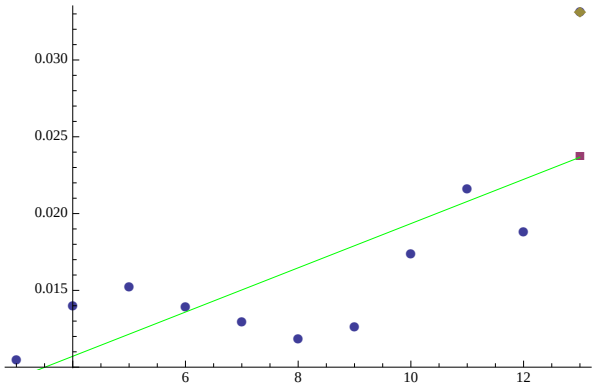
Теперь можно применить это выражение, используя операцию подстановки (/.). Эта операция применяет указанные правила подстановки к заданному выражению. Например, можно посчитать последнюю точку:

```
> lastt = {t, fit /. x —> t}
{13, 0.0236671}
```

Литература

Итак, мы можем отобразить наш пример:

```
> Show[{ListPlot[{fdata, {lastt}}, {Last[fdata]},
    PlotMarkers -> Automatic],
      Plot[fit, {x, f, t}, PlotStyle -> Green]}]
```



Синие точки — исходные данные. Зеленая линия — прямая, которой мы их аппроксимировали. Коричневая точка — несглаженное значение. Красная — сглаженное. Мы не будем рассматривать подробно функции *Plot*, *ListPlot* и *Show*, лишь обратим внимание, что *Show* используется для отображения нескольких графиков в общей системе координат.

Оформим этот механизм в виде функции. Сначала определим функцию *lmsmoothPoint*, которая берет список значений равномерно распределенных точек и на основании их вычисляет сглаженное значение последней. Отметим, что здесь используется несглаженное значение, как часть набора данных для регрессии. Мы допускаем, что в переданном нам наборе данных могут быть неопределенные значения. Мы их просто опускаем, используя функцию *Select* совместно с предикатом *NumberQ*, который возвращает *FALSE* для нечисловых значений, таких как, например, иногда используемый в *Mathematica* символ *Indeterminate*. Если после отбрасывания неопределенных значений у нас останется недостаточно данных для линейной регрессии (меньше двух точек), то функция вернет неопределенное значение:

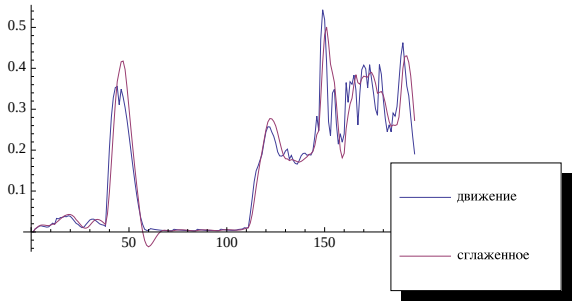
```
> lmsmoothPoint[d_] := Module[{data,rdata},
    data = Transpose[{Range[Length[d]],d}];
    rdata = Select[data,NumberQ[N[#[[2]]]]&];
    If[Length[rdata]<2,
        Last[d],
        Fit[rdata,{1,x},x] /. x-> Length[d]
    ]
]
```

Следующим шагом можно определить функцию, которая принимает набор данных и размер окна сглаживания и применяет сглаживание при помощи линейной регрессии:

```
> lmsmooth[l_,w_] :=
    Map[lmsmoothPoint[l[[Max[1,#-w];;#]]] &,
        Range[Length[l]]]
```

А теперь применим ее к нашим данным и нарисуем сглаженное и несглаженное значения:

```
> serrs = lmsmooth[errs, 10];
> ListLinePlot[{errs, serrs}, PlotLegend -> {"движение",
    "сглаженное"},
    LegendPosition -> {0.8, -0.8}]
```



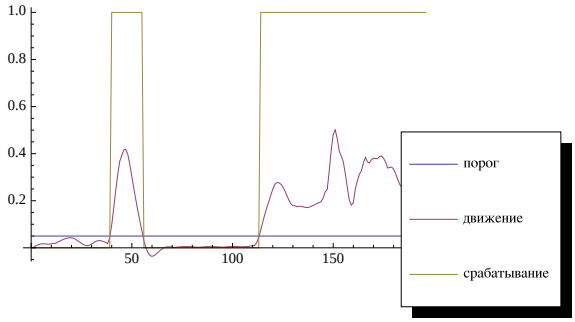
Определение движения

Последний шаг — собственно определение наличия движения в виде логического значения *да/нет*. Мы будем использовать пороговое значение, задаваемое пользователем. Диапазон этого значения будет *[0..1]*. Один из способов получить такие бинарные значения в зависимости от того, что больше — элемент списка или порог — это отнять пороговое значение от элементов списка и применить к ним функцию *UnitStep*. Эта функция возвращает 0 для значений, меньших 0, и 1 для остальных.

```
> motionThreshold = 0.05;
> motionFlag = UnitStep[serrs - motionThreshold];
```

Теперь можно визуализировать окончательный результат работы нашего алгоритма:

```
> ListLinePlot[{Table[motionThreshold, {nframes}], serrs,
    motionFlag},
     PlotLegend -> {"порог", "движение", "срабатывание"},
     LegendPosition -> {0.8, -0.8}]
```



Заключение

Мы не ставили себе задачи обучить вас языку *Mathematica*. Мы умышленно упростили многие понятия и не вдавались в детали внутреннего представления функций и данных в *Mathematica*. Нашей целью было показать стиль работы с этим пакетом в исследовательской работе на примере разработки этого несложного алгоритма. Надеемся, что заинтересованный читатель самостоятельно сможет изучить множество других возможностей этого пакета, используя материалы, перечисленные в приведенном ниже списке литературы.

Алгоритм, который мы разработали, также был упрощен в учебных целях. В тоже время, даже в таком виде он вполне работоспособен.

Литература

[1] *(uni)quotation*. (uni)quotation website. — URL: http://uniquation.ru/ru/ (дата обращения: 11 апреля 2011 г.). — 2011.

[2] *Wikipedia*. Nyquist–shannon sampling theorem — wikipedia, the free encyclopedia. — URL: http://en.wikipedia.org/wiki/Nyquist–Shannon_sampling_theorem (дата обращения: 11 апреля 2011 г.). — 2011.

Как написать LDAP-сервер на Erlang

Максим Сохацкий, Олег Смирнов
maxim@synrc.com, oleg.smirnov@gmail.com

Аннотация

Статья выведена из распространения по просьбе авторов.

Как написать LDAP-сервер на Си

Лев Валкин
vlm@fprog.ru

Аннотация

Ответ на статью Максима Сохацкого (опубликованную несколькими страницами ранее), в котором делается попытка показать, что написание прототипа LDAP-сервера на языке Си ничуть не сложнее, чем на Эрланге. Проводится сравнение полученных решений.

This is a response to the preceding Maxim Sokhatsky's article. We're aiming to show that creating a LDAP server prototype in C is not any more complex than writing it in Erlang. We compare the resulting C and Erlang solutions.

Обсуждение статьи ведется по адресу:
http://fprog.ru/2011/issue7/lev-walkin-ldap/discuss/.

5.1. Экскурс в историю

LDAP — это очень интересная технология, корни которой восходят к началу восьмидесятых годов, к стандартам ITU X.500, к протокольному стеку OSI. В современном мире от стека протоколов OSI осталась практически только «модель ISO/OSI» (1984), повсеместно изучаемая в университетах, но в восьмидесятых годах предсказать грядущее торжество TCP/IP было под силу немногим. Тогда стек OSI виделся как унифицирующая перспектива, устраняющая бардак взаимодействия между десятками проприетарных (AppleTalk, NetWare) или полувоенных (ARPANET) технологий. Предполагалось, что все вендоры полностью перейдут на OSI-стек, заменив им свои технологические решения.

Такого, разумеется, не произошло, и стек OSI почил в бозе, оставив нам многочисленные осколки былого величия. Одним из таких осколков остаётся ASN.1 (Abstract Syntax Notation One) — технология, позволяющая абстрактно описывать структуры данных для их последующей передачи по сетям. Набор стандартов ASN.1 состоит из одного документа, описывающего, собственно, язык описания структур данных, и нескольких сопутствующих стандартов, описывающих *способы кодирования* данных. Например, чтобы описать личное дело члена партии КПСС, можно воспользоваться следующим описанием:

```
Person ::= SEQUENCE {
    name              UTF8String,
    gender            ENUMERATED { male, female },
    dateOfBirth       GeneralizedTime,
    cpsuMemberSince   GeneralizedTime,
    beenAbroad        BOOLEAN DEFAULT false
}
```

Стандартных (ГОСТ, между прочим!) способов кодирования этой информации система стандартов ASN.1 предлагает более десятка, если считать все вариации:

BER (Basic Encoding Rules и его производные: CER, DER) — байт-ориентированный, двоичный способ кодирования с подходом TLV (tag-length-value), при котором каждый элемент предваряется специфичной меткой, за которой идёт длина, а затем, собственно, значение.

XER (XML Encoding Rules и его производные: BASIC-XER, CANONICAL-XER, EXTENDED-XER) — фактически, кодирование с помощью разметки XML. Самый неэффективный вид кодирования с точки зрения количества байт, необходимых для представления значений.

PER (Packet Encoding Rules и его производные: Unaligned PER, Aligned PER) — этот вид кодирования базируется на представлении значений с помощью отдельных битов, и является самым компактным из всех.

ASN.1 имеет и современные аналоги, которые, в общем, предлагают похожий подход к описанию сложных структур данных для их последующей передачи по сети или хранения. Одни из самых модных — Facebook Thrift и Google protocol Buffers. ASN.1 — наиболее распространенный из них (так, в любом web-браузере и в любом мобильном телефоне используются протоколы, описанные в терминах ASN.1), самый компактный с точки зрения эффективности представления данных (если брать PER), но также и самый сложный для реализации.

Люди, создавшие и развивающие стандарт ASN.1, подразумевают, что сложность стандарта должна быть скрыта в *компиляторе* ASN.1. Компилятор должен принимать на вход описание в виде ASN.1-нотации. На выходе компилятор должен генерировать структуры данных или иерархию классов на конечном языке, а также сопутствующий код для превращения этих структур в плоский набор байтов (сериализатор и десериализатор).

Сложность ASN.1 такова, что до сих пор нет ни одного свободно распространяемого компилятора, полностью поддерживающего все стандарты. Это часто приводит к тому, что создателям систем, использующих протоколы, описанные в терминах ASN.1, приходится вместо использования компилятора делать собственные сериализаторы и десериализаторы своих структур или иерархий классов, например, в BER, убивая огромное количество времени и делая поистине невероятное количество ошибок.[1]

Языки Эрланг и Си находятся в выигрышном положении, потому что для них существуют достаточно развитые компиляторы ASN.1. Компилятор ASN.1 для Эрланга называется asn1rt[2] и входит в базовую поставку языка. Компилятор `asn1rt` поддерживает BER и PER. Наиболее развитый свободный компилятор для Си так же бесхитростно называется `asn1c`,[3] и доступен на GitHub: http://github.com/vlm/asn1c.

5.2. Компилируем LDAP-спецификацию

Неудивительно, что имея корни в X.500, LDAP унаследовал от него и способ кодирования данных. Протокол LDAP описывается при помощи ASN.1. Формальное описание протокола находится в RFC 4511.

В отличие от Максима, мы не будем «пробегаться по содержимому» RFC 4511 и склеивать цитаты протокола ASN.1 вручную. При инсталляции `asn1c` устанавливает в систему несколько утилит, одна из которых помогает извлекать ASN.1-документы из RFC. Выяснилось, что это довольно формализуемая задача, что позволяет автоматизировать процесс. Данная утилита называется `crfc2asn1.pl`. Используя `crfc2asn1.pl` в связке с curl,[4] нам нет необходимости даже заглядывать внутрь RFC документа, чтобы извлечь из него ASN.1-спецификацию:

```
$ curl -s http://tools.ietf.org/rfc/rfc4511.txt | crfc2asn1.pl
Found LDAP-V3 at line 2983
=> Saving as LDAP-V3.asn1
$
```

После сохранения в файл ASN.1-модуль нужно скомпилировать компилятором:

```
$ asn1c -fcompound-names LDAP-V3.asn1
Compiled LDAPMessage.c
Compiled LDAPMessage.h
Compiled MessageID.c
Compiled MessageID.h
...
Copied /usr/local/share/asn1c/converter-sample.c
Generated Makefile.am.sample
$
```

[1] http://www.google.com/search?&q=asn.1+vulnerability

[2] http://www.erlang.org/doc/man/asn1rt.html

[3] Здесь я должен сделать обязательный «дисклеймер»: я являюсь автором `asn1c` и разрабатываю его с 2003 года. Подробнее см. http://lionet.info/asn1, http://lionet.livejournal.com/tag/asn1c — *Авт.*

[4] http://en.wikipedia.org/wiki/CURL

Компилятор `asn1c` вытащил из ASN.1-спецификации множество типов данных и сгенерировал большое количество исходных файлов. Кроме того, он породил файл `Makefile.am.sample`, который можно напрямую использовать, чтобы собрать рабочую программу для перекодирования LDAP-пакетов между кодировками BER, PER, XER (XML). Такая программа может понадобиться при отладке протокола LDAP, но сейчас нам интересно написать LDAP-сервер самостоятельно. Поэтому мы выкинем из проекта вежливо предложенный нам исходник `converter-sample.c` и изменим файл `Makefile.am.sample` так, чтобы командой `make` собирался наш файл `ldap-server.c`, который мы вскоре напишем сами.

```
$ rm converter-sample.c
$ mv Makefile.am.sample Makefile
$ sed -i '' -e s/converter-sample/ldap-server/ Makefile
$ sed -i '' -e s/progname/ldap-server/ Makefile
$ make
cc -I. -o LDAPMessage.o -c LDAPMessage.c
...
cc -I. -o per_opentype.o -c per_opentype.c
make: *** No rule to make target 'ldap-server.o'.  Stop.
$
```

Как видно, сборка проекта прошла практически успешно. Не хватает самой малости — функции `main()`, которая бы использовала всю эту машинерию, сгенерированную компилятором `asn1c`. Откроем новый файл `ldap-server.c` и напишем в нём примитивный код, который принимает из стандартного входа LDAP-пакет и печатает его функцией `asn_fprint()`:

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include "LDAPMessage.h"

LDAPMessage_t *receive_ldap_message(int fd) {
    char buffer[8192];
    ssize_t buffer_len = 0;
    LDAPMessage_t *msg = 0;
    asn_dec_rval_t rv;

    buffer_len = read(fd, buffer, sizeof(buffer));
    assert(buffer_len > 0);

    rv = ber_decode(0, &asn_DEF_LDAPMessage, (void**)&msg,
         buffer, buffer_len);
    assert(rv.code == RC_OK);
    assert(rv.consumed == buffer_len);

    return msg;
}

int main() {
    LDAPMessage_t *msg;

    msg = receive_ldap_message(fileno(stdin));
    asn_fprint(stderr, &asn_DEF_LDAPMessage, msg);

    return 0;
}
```

Замечания. Мы использовали `read()`, а не `fread()`, чтобы потом научить этот код работать с сокетами. Мы подразумевали, что пакеты от клиентов будут небольшого размера, поэтому для начала нам хватит буфера размером 8 КБ. Мы игнорируем ошибки `read()` (такие как `EINTR`), и рассчитываем, что клиент не будет посылать нашему игрушечному серверу сразу несколько запросов в одном пакете. Функция `ber_decode()` умеет обрабатывать входной пакет BER по частям, поэтому у неё довольно сложный интерфейс.[5]

Теперь запустим `make` ещё раз и получим выполняемый файл `ldap-server`. Как же протестировать его работоспособность? Сконфигурируем LDAP-клиент, встроенный в стандартную для Mac OS X программу Address Book. Чтобы не возиться с суперпользовательскими правами, порт для LDAP-сервера укажем 3389, а не стандартный 389. На иллюстрациях 5.1, 5.2 показано, как это сделать.

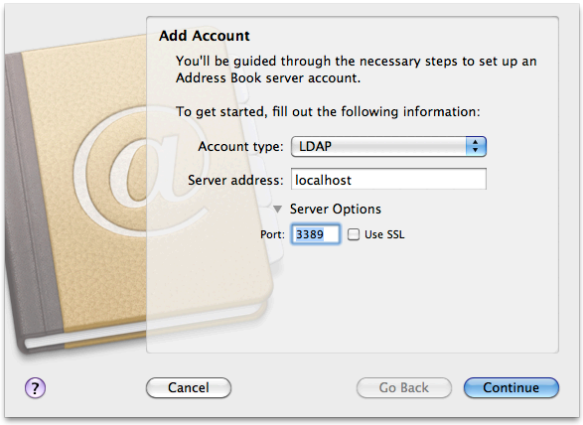


Рис. 5.1. Добавление LDAP-сервера в Address Book

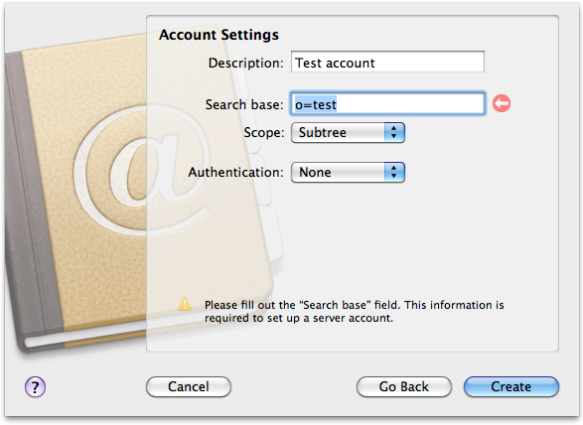


Рис. 5.2. Настройка LDAP-сервера в Address Book

После добавления адреса нашего LDAP-сервера в Address Book, теперь надо каким-то образом заставить наш `ldap-server` отвечать на TCP-порту 3389. Легче всего это сделать командой nc.[6] Подключим вход `ldap-server` к выходу `nc` и попробуем поискать что-нибудь в Address Book (рис. 5.3).

```
$ nc -l 3389 | ./ldap-server
LDAPMessage ::= {
    messageID: 1
    protocolOp: BindRequest ::= {
        version: 3
```

[5] Подробнее см. в http://lionet.info/asn1c/asn1c-usage.pdf.
[6] http://en.wikipedia.org/wiki/Netcat

```
        name: 74 65 73 74
        authentication:
    }
}
$
```

Как видно на рис. 5.3, Address Book ничего не возвращает. Зато в терминале мы получили отладочный вывод команды `asn_fprint()`, из которого следует, что мы получили сообщение `BindRequest`.

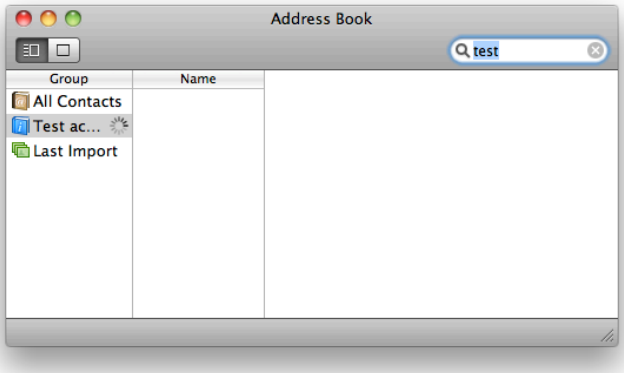


Рис. 5.3. Тестовый поиск не возвращает результатов

Небольшое отступление. В приведённой выше отладочной печати `ldap-server`, поле `BindRequest.name` распечаталось в виде набора чисел, а не строки «*test*», введенной в строке поиска Address Book. Связано это с тем, что спецификация LDAP консервативно подходит к спецификации типов для строковых данных, не задействуя в должной мере возможности стандарта ASN.1. Воспользуемся информацией из пункта 4.1.2 RFC 4511 и заменим в файле `LDAP-V3.asn1` строку

```
LDAPString ::= OCTET STRING -- UTF-8 encoded
```

на строку

```
LDAPString ::= [UNIVERSAL 4] IMPLICIT UTF8String -- UTF-8 encoded
```

чтобы `asn1c` смог сгенерировать более понятную печать строковых значений. Перекомпилировав спецификацию, мы получим более приятный глазу результат:

```
$ nc -l 3389 | ./ldap-server
LDAPMessage ::= {
    messageID: 1
    protocolOp: BindRequest ::= {
        version: 3
        name: test
        authentication:
    }
}
```

Итак, своего рода инвертированный «hello world!» для LDAP написан. Теперь нужно улучшить программу, научив ее отвечать на сообщения каким-нибудь нетривиальным способом.

5.3. Нетривиальный LDAP-сервер

Воспользуемся предложением Максима Сохацкого и построим сервис, возвращающий один и тот же фиксированный набор данных вне зависимости от условий поиска. Для этого нам необходимо завести слушающий TCP-сокет и научиться отвечать на сообщения `BindRequest`, `SearchRequest`, и `UnbindRequest`.

Во-первых, научим LDAP-сервер сериализовывать подготовленные структуры `LDAPMessage_t` в файловый дескриптор.

```
int output(const void *buffer, size_t size, void *key) {
    return write(*(int *)key, buffer, size);
}

void send_ldap_message(int fd, LDAPMessage_t *msg) {
    der_encode(&asn_DEF_LDAPMessage, msg, output, &fd);
}
```

Во-вторых, добавим функцию, возвращающую готовое клиентское соединение.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

int accept_single_connection(int port) {
    struct sockaddr_in sin;
    int err;
    int lsock, sock;
    int opt_true = ~0;

    memset(&sin, 0, sizeof sin);
    sin.sin_family = AF_INET;
    sin.sin_port = htons(3389);
    sin.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    lsock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    setsockopt(lsock, SOL_SOCKET, SO_REUSEPORT, &opt_true, sizeof(opt_true));
    err = bind(lsock, (struct sockaddr *)&sin, sizeof sin);
    assert(err == 0);
    err = listen(lsock, 1);
    assert(err == 0);

    sock = accept(lsock, 0, 0);
    assert(sock > 0);
    close(lsock);

    return sock;
}
```

В-третьих, заменим примитивную функцию `main()` на что-то более функциональное, в общечеловеческом смысле.

```
1  int main() {
2      LDAPMessage_t *req_bind, *req_search, *req_unbind;
3      LDAPMessage_t *rsp_bind, *rsp_search1, *rsp_search2, *rsp_search_done;
4
5      int sock = accept_single_connection(3389);
6
7      req_bind = receive_ldap_message(sock);
8      assert(req_bind);
9      assert(req_bind->protocolOp.present == LDAPMessage__protocolOp_PR_bindRequest);
10     assert(req_bind->protocolOp.choice.bindRequest.version == 3);
11     asn_fprint(stderr, &asn_DEF_LDAPMessage, req_bind);
12
13     rsp_bind = bind_response_ok(req_bind->messageID, &req_bind->protocolOp.choice.bindRequest.name);
14     asn_fprint(stderr, &asn_DEF_LDAPMessage, rsp_bind);
15     send_ldap_message(sock, rsp_bind);
16
17     req_search = receive_ldap_message(sock);
18     assert(req_search->protocolOp.present == LDAPMessage__protocolOp_PR_searchRequest);
19     xer_fprint(stderr, &asn_DEF_LDAPMessage, req_search);
20
21     rsp_search1 = search_result_entry(req_search->messageID, "Lev Walkin", "vlm@fprog.ru");
22     asn_fprint(stderr, &asn_DEF_LDAPMessage, rsp_search1);
23     send_ldap_message(sock, rsp_search1);
24
25     rsp_search2 = search_result_entry(req_search->messageID, "Olga Bobrova", "oley@fprog.ru");
26     asn_fprint(stderr, &asn_DEF_LDAPMessage, rsp_search2);
27     send_ldap_message(sock, rsp_search2);
28
29     rsp_search_done = search_result_done(req_search->messageID,
               &req_search->protocolOp.choice.searchRequest.baseObject);
30     asn_fprint(stderr, &asn_DEF_LDAPMessage, rsp_search_done);
31     send_ldap_message(sock, rsp_search_done);
32
33     req_unbind = receive_ldap_message(sock);
34     xer_fprint(stderr, &asn_DEF_LDAPMessage, req_unbind);
35
36     return 0;
37 }
```

В-четвёртых, определим функции, обозначенные выше жирным шрифтом. Три функции `bind_response_ok`, `search_result_entry` и `search_result_done` занимаются составлением ответа на соответствующие запросы LDAP-сервера.

```
LDAPMessage_t *bind_response_ok(int messageId, LDAPDN_t *dn) {
    LDAPMessage_t *msg = calloc(1, sizeof *msg);
    BindResponse_t *resp;

    msg->messageID = messageId;
    msg->protocolOp.present = LDAPMessage__protocolOp_PR_bindResponse;
    resp = &msg->protocolOp.choice.bindResponse;
    asn_long2INTEGER(&resp->resultCode, BindResponse__resultCode_success);
    OCTET_STRING_fromBuf(&resp->matchedDN, dn->buf, dn->size);
```

```
    OCTET_STRING_fromString(&resp->diagnosticMessage, "OK");

    return msg;
}

PartialAttribute_t *make_partial_attribute(char *key, char *value) {
    PartialAttribute_t *pa = calloc(1, sizeof *pa);

    OCTET_STRING_fromString(&pa->type, key);
    ASN_SEQUENCE_ADD(&pa->vals,
        OCTET_STRING_new_fromBuf(&asn_DEF_AttributeValue, value, -1));

    return pa;
}

LDAPMessage_t *search_result_entry(int messageId, char *name, char *email, char *jpegFilename) {
    LDAPMessage_t *msg = calloc(1, sizeof *msg);
    SearchResultEntry_t *entry;

    msg->messageID = messageId;
    msg->protocolOp.present = LDAPMessage__protocolOp_PR_searchResEntry;
    entry = &msg->protocolOp.choice.searchResEntry;
    OCTET_STRING_fromString(&entry->objectName, name);

    ASN_SEQUENCE_ADD(&entry->attributes, make_partial_attribute("cn", name));
    ASN_SEQUENCE_ADD(&entry->attributes, make_partial_attribute("mail", email));

    return msg;
}

LDAPMessage_t *search_result_done(int messageId, LDAPDN_t *dn) {
    LDAPMessage_t *msg = calloc(1, sizeof *msg);
    SearchResultDone_t *done;

    msg->messageID = messageId;
    msg->protocolOp.present = LDAPMessage__protocolOp_PR_searchResDone;
    done = &msg->protocolOp.choice.searchResDone;
    asn_long2INTEGER(&done->resultCode, LDAPResult__resultCode_success);
    OCTET_STRING_fromBuf(&done->matchedDN, dn->buf, dn->size);
    OCTET_STRING_fromString(&done->diagnosticMessage, "OK");
    return msg;
}
```

Собрав всё это вместе, мы получаем примерно 180 строк кода, подходящего для Address Book в качестве LDAP-сервера.

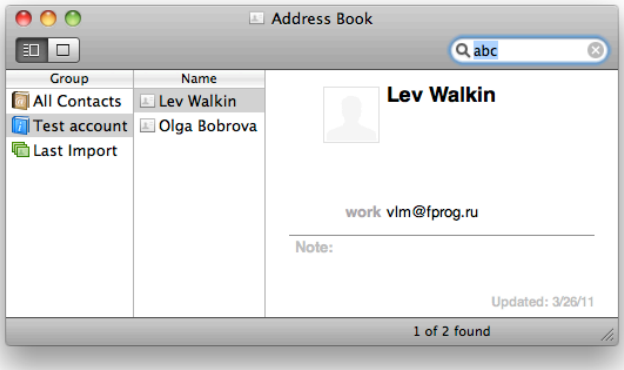


Рис. 5.4. Поиск возвращает два результата

После поиска консоль содержит отладочную печать всех посланных и принятых LDAP-сообщений. Часть отладочных сообщений напечаталась в формате XER (XML Encoding Rules, одной из стандартных кодировок ASN.1) путём использования функции `xer_fprint()`.

```
$ ./ldap-server
LDAPMessage ::= {
   messageID: 1
   protocolOp: BindRequest ::= {
      version: 3
      name: test
      authentication:
   }
}
LDAPMessage ::= {
   messageID: 1
   protocolOp: BindResponse ::= {
      resultCode: 0 (success)
      matchedDN: test
      diagnosticMessage: OK
   }
}
<LDAPMessage>
   <messageID>2</messageID>
   <protocolOp>
      <searchRequest>
         <baseObject>o=test</baseObject>
         <scope><wholeSubtree/></scope>
         <derefAliases><neverDerefAliases/></derefAliases>
         <sizeLimit>0</sizeLimit>
         <timeLimit>30</timeLimit>
         <typesOnly><false/></typesOnly>
         <filter>
            <or>
               <substrings>
                  <type>givenname</type>
                  <substrings><initial>abc</initial></substrings>
               </substrings>
               <substrings>
                  <type>sn</type>
                  <substrings><initial>abc</initial></substrings>
               </substrings>
               <substrings>
                  <type>mail</type>
                  <substrings><initial>abc</initial></substrings>
               </substrings>
               <substrings>
                  <type>cn</type>
                  <substrings><initial>abc</initial></substrings>
               </substrings>
            </or>
         </filter>
         <attributes>
            <selector>givenName</selector>
            <selector>sn</selector>
            <selector>cn</selector>
            <selector>mail</selector>
            <selector>jpegPhoto</selector>
         </attributes>
      </searchRequest>
   </protocolOp>
</LDAPMessage>
LDAPMessage ::= {
   messageID: 2
   protocolOp: SearchResultEntry ::= {
      objectName: Lev Walkin
      attributes: PartialAttributeList ::= {
         PartialAttribute ::= {
            type: cn
            vals: vals ::= { Lev Walkin }
         }
         PartialAttribute ::= {
            type: mail
            vals: vals ::= { vlm@fprog.ru }
         }
      }
   }
}
LDAPMessage ::= {
   messageID: 2
   protocolOp: SearchResultEntry ::= {
      objectName: Olga Bobrova
      attributes: PartialAttributeList ::= {
         PartialAttribute ::= {
            type: cn
            vals: vals ::= { Olga Bobrova }
         }
         PartialAttribute ::= {
            type: mail
            vals: vals ::= { oley@fprog.ru }
         }
      }
   }
}
LDAPMessage ::= {
   messageID: 2
   protocolOp: SearchResultDone ::= {
      resultCode: 0 (success)
      matchedDN: o=test
      diagnosticMessage: OK
   }
}
<LDAPMessage>
   <messageID>3</messageID>
   <protocolOp>
      <unbindRequest></unbindRequest>
   </protocolOp>
</LDAPMessage>
$
```

Если добавить ещё строк пятнадцать, то мы сможем даже заставить Address Book отображать фотографию, ассоциированную с карточкой. Полный код для LDAP-сервера доступен по адресу: http://github.com/vlm/ldap-server-example.

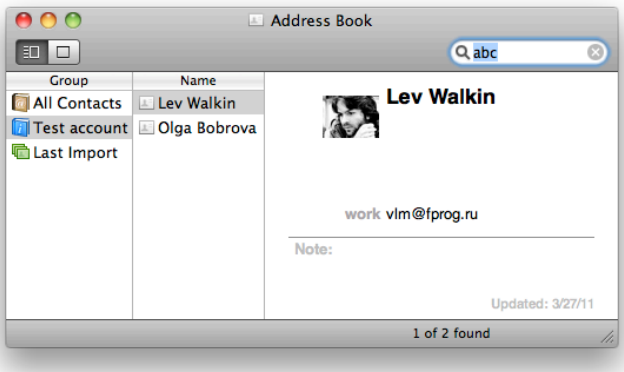


Рис. 5.5. Поиск возвращает результат с фотографией

5.4. Анализ решения

Кода на Си получилось немного больше, чем кода на Эрланге в статье Максима Сохацкого. Можно было бы поспорить с выводом Максима о «выразительности языка Эрланг». Сложно говорить о разнице в выразительности, когда разница в объёме кода исчисляется десятком процентов.

Вообще, делать выводы о выразительности того или иного языка, сравнивая API какой-либо доступной библиотеки — занятие неблагодарное. Тем более, если библиотека очень богатая. Поэтому мы не будем сравнивать мощность ASN.1-компиляторов, а постараемся обойтись сравнением языков и используемых в них подходов.

Разрабатывая LDAP-сервер на Си, мы заранее знали, что почти всю неблагодарную механику скроет использование ASN.1-компилятора. Поэтому задача «написать игрушечный LDAP-сервер» представлялась очень простой. Однако видимая простота оказалась коварной.

Распределение памяти

Области существования объектов в LDAP-сервере содержат перекрытия. Посмотрим на листинг `main()` на с. 39. Объект `req_bind` (строка 7) используется при формировании ответа `resp_bind` в строке 13. Объект `req_search` (строка 17) тоже используется, но далеко не в первом ответе на SearchRequest, а гораздо ниже, в строке 29.

Если в качестве реакции на ошибки клиента мы не полагались бы на `assert()`, а попытались корректно завершить работу с клиентом, то появилась бы необходимость аккуратно покидать вложенные области видимости, удаляя за собой объекты в определённом порядке. Это не большая проблема для программистов на C, привыкших аккуратно работать с памятью, но это опасная ловушка для времени: программист, вместо разработки логики приложения, занимается микроменеджментом памяти.

Ещё раз подчеркнём важную разницу между необходимостью освобождать память и необходимостью следить за пересекающимися областями существования.

Код на Эрланге не только чуть проще из-за того, что в нём нет кода по освобождению памяти, но ещё и чуть-чуть *корректнее*, так как автоматическая сборка мусора устраняет целый класс проблем.

Эти же аргументы можно привести в защиту любых языков с автоматической сборкой мусора, но надо отметить, что свободные развитые ASN.1-компиляторы существуют только для Эрланга и Си. Это делает Эрланг фактически единственной бесплатной системой с автоматической сборкой мусора для разработки сервисов, базирующихся на ASN.1.

Работа с сетью

Базовые примитивы работы с сетью в Си недостаточно защищены от побочных эффектов среды выполнения. Например, функция `read()` может вернуться с кодом ошибки `EINTR`, в зависимости от метода обработки сигналов, использующегося в приложении. Если код на Си упакован в библиотеку, то создатель библиотеки не может рассчитывать, что в основной программе обработчики сигналов будут установлены должным образом, и наверняка должен быть готов к тому, что `EINTR` придёт в любой момент.

Иными словами, успешность работы с функцией, работающей с сокетом, зависит от того, внутри какой программы — *рядом с каким ещё кодом* — выполняется код нашей гипотетической библиотеки.

Облагородить дизайн программы на C можно, вынеся работу с сокетами на уровень центрального ядра программы, а библиотеки подключать к этому ядру с помощью механизма диспетчеризации событий. Такой интерфейс между библиотекой и программой нетривиален. Ядру необходимо реагировать на желание библиотеки открыть и закрыть сокет, регистрировать на интерес к возможности чтения или записи в сокет, «договариваться» с библиотекой о том, кто из них отвечает за буферизацию данных. Такое «разделение труда» между ядром и библиотекой требует развитого и стабильного фреймворка. К сожалению, общепринятого фреймворка в этой области нет[7], что вынуждает создателей библиотек, рассчитывающих на переносимость, предоставлять слишком низкоуровневый интерфейс к своей функциональности, а создателей приложений вынуждает сопрягаться с доступными библиотеками на чрезвычайно низком уровне абстракции.

В Эрланге работа с сетью абстрагирована от особенностей операционной системы. Модуль на Эрланге, желающий работать с сетью, использует сокеты напрямую, без необходимости сопрягаться с «центральным кодом» приложения. Легковесные процессы в Эрланге работают независимо друг от друга и не имеют общей памяти, а сокеты связаны с породившими их процессами. То есть, при аварийном завершении процесса сокет будет автоматически закрыт.

Сокеты в Эрланге — довольно высокоуровневая абстракция. Например, можно заставить сокет присылать данные процессу только тогда, когда данных наберётся на полное сообщение, независимо от TCP-сегментации. Если данные в TCP-потоке представляют собой набор фрагментов, предварённых одно-, двух- или четырёхбайтовой длиной фрагмента, то нужным образом настроенный эрланговский сокет будет копить данные из TCP-потока, и передаст пакет в процесс только тогда, когда полностью соберёт фрагмент указанной длины.

Кроме того, эрланговские сокеты могут «резать» поток согласно соглашениям о длине, принятым в ASN.1[8], CORBA, FastCGI, HTTP, да и просто построчно. Этот механизм и был использован в статье Максима Сохацкого. В макросе `TCP_OPTIONS` задаётся пожелание, чтобы сокет присылал полностью оформленные фрагменты, пригодные для использования в `asn1ct:decode/3` без промежуточной буферизации.

```
-define(TCP_OPTIONS, [binary, {packet,asn1}, ...]).
```

Расширение программы

Игрушечный LDAP-сервер на Си, разработанный в рамках этой статьи, может посоревноваться с эрланговской версией в лаконичности и простоте. Однако расширение функциональности игрушеченого сервера на Си потребует непропорционально большего количества усилий. Вот с чем придётся столкнуться тем, кто попытается это сделать:

Полноценная работа с TCP-потоком LDAP-сообщений.
В прототипе на Си мы сделали несколько допущений. Например, в функции `receive_ldap_message()` мы предполагаем, что клиентское сообщение не превысит

[7] Можно сослаться на libev/libevent, ACE и Boost.Asio.

[8] Если протокол использует *definite length* вариант кодирования в BER.

8 КБ. Также мы считаем, что клиент посылает новые сообщения только после того, как дождется предыдущего ответа, а его способ работы с TCP-стеком не вызывает сегментирования BER-фрагментов на несколько пакетов. Эти допущения приемлемы для прототипа, но в случае расширения прототипа до полноценного сервера нам необходимо будет самостоятельно обеспечивать буферизацию и склейку приходящих данных перед отправкой их в `ber_decode()`. В Эрланге эти задачи уже решены на уровне API работы с сетью.

Работа с множеством клиентов. В версии на Си необходимо будет перейти на какой-то фреймворк асинхронной работы с сокетами, полностью заменив линейную последовательность действий в программе на конечный автомат. Альтернативные пути, такие как порождение отдельного потока (`pthread_create`) или процесса (`fork`), сопряжены с не меньшим количеством трудностей, вызванных обеспечением корректности доступа к общим данным. В Эрланге достаточно на каждого клиента порождать новый процесс стандартной функцией `spawn`, как сделано в коде Максима.

Использование внешних ресурсов. Если LDAP-серверу в процессе работы понадобится обращаться к внешним ресурсам (к ImageMagick, к базе данных MySQL, и т. д.), то это не только усложнит конечный автомат LDAP-сервера, но и потребует нетривиального сопряжения с использующимися для доступа к внешнему ресурсу библиотеками. Например, каким образом сопрягаться с `libmysql`? Эрланг предлагает подход, при котором внешняя синхронная функциональность запускается в отдельных процессах операционной системы, а коммуникация с ними происходит через стандартный ввод/вывод этих процессов. Кроме убирания синхронности, этот подход ещё и защищает от ошибок внутри библиотек: если внешний процесс умирает, Эрланг перезапускает его автоматически.

Диагностика внутреннего состояния сервера. В версии на Си необходимо будет отвечать на SNMP-запросы, имитировать HTTP-сервер или придумывать какой-то свой протокол выдачи внутреннего состояния для диагностических или административных нужд. Если же сервер разработан на Эрланге, к нему всегда можно подключиться эрланговской консолью и произвести ручную интроспекцию любого выполняющегося кода без остановки LDAP-сервера. Кроме того, в поставку Эрланг входят встроенные HTTP- и SNMP-серверы, поэтому собрать из них более традиционный интерфейс диагностики потребует лишь несколько десятков строк кода.

5.5. Заключение

Надеюсь, нам удалось показать, что тестировать платформы путём использования встроенной функциональности — вещь довольно бесполезная. Кода на Си получилось ненамного больше, чем кода на Эрланге, что неплохо, особенно учитывая период обучения, необходимый новичку для того, чтобы начать использовать Эрланг.

Однако как только мы начинаем говорить о развитии системы, низкоуровневость инструмента даёт о себе знать: наш прототип LDAP-сервера на Си исключительно тяжело превратить в полноценный, стабильный продукт. Наличие встроенных библиотек для организации серверов и клиентов HTTP, SNMP, ASN.1, а также поддержка безопасного способа сопряжения со сторонними программами делает Erlang идеальным средством для прототипирования приложений. А развитые языковые средства, нацеленные на создание систем в условиях постоянных программных ошибок, позволяют легко превращать прототипы в надёжные и функциональные (во всех смыслах этого слова) приложения.

Продолжения в практике

Алексей Вознюк
swizard@fprog.ru

Аннотация

В данной статье определяется проблемная область, задачи которой успешно решаются при помощи продолжений, анализируются традиционные методы решения таких задач и рассматриваются несколько практических примеров с эффективным использованием продолжений. *This article defines a subject area whose problems can be conveniently solved with the use of continuations; we analyze the traditional methods of solving these problems and consider a few practical examples with efficient use of continuations.*

Обсуждение статьи ведется по адресу:
http://fprog.ru/2011/issue7/alexey-voznyuk-continuations-in-practice/discuss/.

6.1. Введение

Полагаю, что многие читатели журнала знакомы с MIT-овским курсом «Structure and Interpretation of Computer Programs» [1] («Структура и интерпретация компьютерных программ» [4] в русском переводе). В курсе (в классическом его варианте) слушателя знакомят с языком Scheme, с помощью которого в дальнейшем и подают материал.

Язык Scheme был выбран авторами не случайно: он отлично подходит для демонстрации всех рассматриваемых в курсе концепций программирования. Он прост, но в то же время элегантен и очень выразителен. Стандарт языка (на текущий момент R6RS[1]) весьма лаконичен: основной упор делается не на практичность, а на гибкость, благодаря которой в Scheme легко добавлять новые возможности. Словом, язык со всех сторон хорош в плане легкости изучения и освоения.

Однако в стандарте Scheme задекларирована одна весьма любопытная возможность языка, которая в SICP не рассматривается. Речь идет о *продолжениях* (continuations).

И, если все остальные аспекты Scheme довольно просты для понимания, то с продолжениями у многих начинающих схемеров возникают проблемы. Разумеется, в Интернете можно найти массу материалов по этой теме, однако (в большинстве своем) это либо заметки вида: «О, я, кажется, разобрался с продолжениями! Смотрите…», либо статьи о том, как это работает (continuation passing style, аналогии с `setjmp/longjmp` и т. п.).

В итоге не складывается четкого понимания практического смысла продолжений. Кто-то считает **call/cc** аналогом **return-from** из Common Lisp, кто-то — аналогом **goto**, а кто-то вообще видит в продолжениях механизм для реализации исключений. А если практический смысл неочевиден, то большинству схемеров проще продолжениями не пользоваться вовсе: и без них вполне успешно создаются элегантные и нормально работающие программы.

В данной статье я предлагаю воспользоваться одной хитростью начинающих Haskell-программистов. Не секрет, что у большинства из них далеко не сразу получается осознать монады. Однако это не мешает ими продуктивно пользоваться, например, конструкцией **do** и монадой List — достаточно ознакомления с паттернами их использования и чтения чужого кода. Через некоторое время активной практики внезапно (или постепенно) приходит и осознание того, как эти механизмы работают и что еще с ними можно делать.

Далее я постараюсь аналогично применить данный подход к теме статьи.

6.2. Control Flow

Рассматривая ассемблерный листинг исходного кода некоторой программы, можно достаточно уверенно представлять себе, каким именно образом процессор будет исполнять код. В большинстве случаев инструкции выполняются строго последовательно, ровно в том порядке, как они приведены в листинге, за некоторыми единичными исключениями. К таким исключениям, например, относятся безусловные и условные переходы или вызов подпрограмм. Выполняя подобную инструкцию, процессор перемещается в другое место листинга, начиная затем, выполнять инструкции уже оттуда.

[1] http://www.r6rs.org/

Такое «движение» по коду будем называть «ходом исполнения программы» (*Control Flow*), а мгновенное перемещение в другую точку «передачей управления».

Подобный поток выполнения явно или неявно существует в каждом языке программирования, в том числе и в функциональных. У многих программистов с опытом даже развивается такой навык, как «отладка взглядом»: отслеживание хода выполнения программы при помощи беглого просмотра глазами.

Однако далеко не все элементарные «шаги» в программе исполняются через обычное процессорное вычисление. Выполнение некоторых команд требует взаимодействия с внешним миром, в котором жизнь течет с разной скоростью. Достаточно часто процессор вынужден бессмысленно простаивать, ожидая результата извне.

Примеров таких команд очень много — это практически любое обращение в машине за пределы процессора, а именно *Input/Output, IO*. Жесткий диск, сеть, клавиатура, даже оперативная память — любой такой запрос вгоняет процессор в вынужденный простой.

Конечно же, существует множество различных техник и приемов по организации кода для работы с вводом и выводом.

6.2.1. Последовательное выполнение

Это самый очевидный вариант, тем не менее, очень часто даже самый практичный. Несмотря на то, что программа блокируется в ожидании данных из «большого мира», зачастую ей и не надо больше ничего выполнять, пока запрашиваемые данные не будут получены.

Однако недостаток данного подхода также очевиден: существуют вынужденные периоды бездействия, во время которых можно было бы выполнять другую полезную работу.

6.2.2. Параллельное выполнение

Первая идея, которая приходит в голову: запускать взаимонезависимые части программы в параллель. Например, в разных процессах или нитях. Это несложно, пока в программе существуют независимые друг от друга участки алгоритма, которые не разделяют никаких общих ресурсов.

Под физической природой «параллельного выполнения» можно подразумевать разные процессы:

1) **«Честная многопроцессорность»**: Программы честно выполняются на разных процессорах, каждый из которых организует свой поток выполнения.

2) **«Вытесняющая многозадачность»**: Физически процессоров меньше, чем потоков выполнения, а планирование заданий осуществляется операционной системой. Существует специализированный поток: планировщик задач, который, во время вынужденного бездействия процессора по причине *IO* (либо по истечении некоего кванта времени) приостанавливает текущую задачу (запоминая необходимый для ее возобновления контекст) и перекидывает поток исполнения на другой участок кода. При этом собственно в суть выполняемых программ диспетчер задач не вникает.

Оба варианта, помимо собственно необходимости привлечения «сторонних сил», разумеется, имеют свои накладные расходы.

В первом случае все равно не удается полностью загрузить процессоры полезной работой, а во втором возможны лишние

переключения контекста по причине истечения кванта времени, которые, в свою очередь, тоже требуют ресурсов процессора.

Однако на практике программист часто может самостоятельно разобраться, когда есть угроза простоя процессора, и можно пока заняться другой работой.

6.2.3. Мультиплексирование

В информационных технологиях мультиплексирование подразумевает объединение нескольких потоков данных (виртуальных каналов) в один. В нашем же случае, мы таким образом «сливаем» несколько логических *Control Flow* в одном физическом потоке.

Для этого, правда, придется произвести крупный рефакторинг своих программ. Переключение между задачами происходит вручную (по инициативе приложения), поэтому:

1) контекст выполнения переключаемой задачи нужно уметь сохранять;

2) за «подвисшими» задачами (в режиме ожидания) надо уметь следить (не готов ли уже результат выполнения).

Если с контекстом все более или менее понятно (его можно сохранить в специальной структуре, в замыкании, в базе данных с доступом по ключу *сессии* и т. п.), то для второго пункта популярны следующие сценарии решения:

- **Polling** (периодический опрос): состояние приостановленных задач регулярно проверяется, и выполнение продолжается, как только появляется такая возможность.

- **Event loop** (событийный цикл): самый распространенный вариант мультиплексирования. С помощью специального событийного механизма (например, используя популярные системные вызовы `select(2)`, `poll(2)`, `kqueue(2)`, `WaitForMultipleObjects` и т. д.) программа регистрирует типы событий, в которых она заинтересована. Затем, вызов специальной функции оповещает о сработавших событиях или блокируется в их ожидании.

Мультиплексирование практически всегда предпочтительней многопоточности в задачах, где суммарное количество простоев (ожиданий внешних событий) заметно больше рабочей нагрузки на процессор. Здесь дело в том, что распараллеливание по ядрам или процессорам теряет смысл (большую часть времени программа все равно не делает ничего), а с диспетчера задач операционной системы снимается большое количество работы, так как физически нить только одна.

Опять же, с точки зрения архитектуры иногда оказывается правильней, когда само приложение решает, в какой именно момент следует переключать задачи. Операционной системе можно только «намекать» на важность той или иной нити (при помощи системы приоритетов), а при мультиплексировании распределять процессорное время между задачами можно по любым, сколь угодно сложным алгоритмам.

Другое дело, что всю «черную работу» приходится выполнять руками. Это, а также необходимость специальным образом переписывать алгоритмы зачастую является причиной, по которой многие программисты предпочитают многопоточность.

6.2.4. Демультиплексирование

Существует множество различных способов припрятать мультиплексор подальше и «спрямить» ход выполнения программы, *Control flow*, чтобы алгоритм не так сильно разносился в различные места листинга программы. Давайте разберем некоторые удачные варианты.

Паттерн программирования «Reactor»

Событийный цикл (event loop) дополняется **диспетчером**, который сам обрабатывает все происходящие события и диспетчеризирует пользовательские обработчики, которые к ним привязаны. Тем самым от пользовательского кода абстрагируется собственно вся реализация событийного механизма, и от разработчика требуется лишь запрограммировать реакцию на интересующие его события.

Реализации этого паттерна существуют для всех языков программирования, из широко известных можно отметить такие библиотеки, как:

- **Boost::Asio** и **ACE** для C++.
- **POE** или **AnyEvent** для Perl.
- **EventMachine** для Ruby.
- **Twisted** для Python.
- **java.nio**, **JBoss Netty** и **Apache MINA** для Java.

Однако, несмотря на существенное упрощение организации самого событийного цикла, контекст все равно остается рваным (в обработчиках его приходится носить с собой) и ход выполнения программы все равно выглядит вывернутым наизнанку. К счастью, в языках, которые поддерживают полноценные замыкания, существует возможность эти недостатки некоторым образом скрыть.

Node.js

Несмотря на то, что этот фреймворк также является реализацией паттерна «Reactor» для JavaScript, разработчикам удалось добиться очень элегантного решения проблемы рваного контекста:

```
var net = require('net');
net.createServer(function (socket) {
  socket.write("Echo server\r\n");
  socket.on("data", function (data) {
    socket.write(data);
  });
}).listen(8124, "127.0.0.1");
```

Контекст выполнения совершенно прозрачно протаскивается в замыканиях, благодаря полноценной поддержки оных в языке. Тем самым появляется возможность читать код последовательно, а не прыгая по раскиданным по проекту обработчикам.

Тем не менее, данный подход также не является панацеей от всех неудобств, привнесенных в программу мультиплексором, хотя и заметно облегчает жизнь. В той же реализации итеративных алгоритмов, содержащих несколько взаимосвязанных блокирующих вызовов (например, нетривиальная маршрутизация данных), даже облагораживающие программу замыкания не помогут избежать «спагетти» в коде – все потому, что стандартные управляющие структуры (try/catch, циклы, даже структура управления «;») больше не применимы и их приходится эмулировать.

Green Threads

Достаточно элегантный вариант решения проблемы могут предложить нам языки, чьи рантаймы поддерживают так называемые «зеленые нити» на уровне своей платформы. Грубо говоря, для подобной поддержи рантайм должен:

1) Уметь сохранять, поддерживать и восстанавливать контексты выполнения программ при создании, переключении и завершении пользовательских нитей (между прочим, в этом месте мы уже недалеко от продолжений).

2) Иметь где-то в фоне невидимый глобальный событийный цикл, умеющий обрабатывать все поддерживаемые языком конструкции и функции, которые могут требовать блокировки.

В данном контексте становится понятно, что зеленая нить (в отличие от «реальной» нити операционной системы с поддержкой вытесняющей многозадачности) умеет отдавать управление другой нити только в нескольких, единичных случаях:

1) В процессе выполнения интерпретатор наткнулся на блокирующий вызов (sleep, I/O или искусственная блокировка, например, на мьютексе).

2) Программа собственноручно попросила об этом (аналогом вызова `yield`).

3) Плюс рантаймы некоторых языков (например, *Erlang*) умеют по своей инициативе выполнять неявный `yield`, эмулируя тем самым некоторое подобие вытесняющей многозадачности. Триггером к такому переключению может служить достижение лимита количества вызовов функций.

Очевидным плюсом зеленых нитей является удачное комбинирование многих положительных сторон мультиплексирования (которым, по сути, оно и является) с некоторыми положительными сторонами многопоточного программирования. Все алгоритмы остаются строго последовательными, не инвертируя *Control flow* и не запутывая отладку; приложение остается физически однопоточным, тем самым, избавляя разработчика от необходимости синхронизировать доступ к общим ресурсам; тредов можно генерировать любое количество, пока хватит памяти, не опасаясь перегрузить системный планировщик задач и так далее.

При всех своих плюсах, зеленые нити имеют свои серьезные минусы:

- Временем отклика (`latency`) следует управлять вручную с помощью `yield` или аналогов, потому что нить с кодом, который не содержит блокирующих вызовов, никогда не отдаст управление другой нити. Даже для рантаймов с эмуляцией вытесняющей многозадачности это тоже справедливо: программа может осуществлять вызов «чужого» (*foreign*) кода, который может сколь угодно долго не отдавать управление обратно.

- Зачастую весьма неочевидно, по каким правилам передается управление, когда после очередного выполнения мультиплексора в очереди на возобновление оказывается несколько нитей.

- Набор событий (и, следовательно, набор блокирующих вызовов) определяется возможностями глобального цикла рантайма языка и, в большинстве случаев, ограничен только некоторой коллекцией *IO*. И нет возможностей для расширения этого набора.

Последнему пункту следует посвятить отдельный раздел, поскольку понимание данной проблемы однозначно подведет нас к интуитивному пониманию смысла продолжений. Давайте рассмотрим элементарную, но практическую задачу в следующем примере.

6.3. Пример: сетевая файловая система.

ТЗ: требуется реализовать сервер элементарной сетевой файловой системы. Все, что он умеет — это асинхронно читать данные из заданных файлов (имена их заранее известны и они всегда существуют). Для минимизации затрат по трафику используется проверка по контрольной сумме.

1) Сервер принимает запросы по TCP на заданный порт.

2) Протокол состоит из двух этапов, при этом все пакеты в течение этих двух шагов должны иметь одинаковый `request_id`.

3) Этап первый: проверка контрольной суммы блока данных.

   - Запрос представляет собой пакет формата:

     `|packet_size|request_id|0|filename|offset|length|`

     где цифра «0» означает команду: получить контрольную сумму блока данных из файла `filename`, начиная со смещения `offset` и размером `length`.
   - В ответ ожидается пакет следующего формата:

     `|packet_size|request_id|crc|`

4) Второй этап: принятие клиентом решения, передавать блок данных или нет.

   - Положительное решение — пакет формата:

     `|packet_size|request_id|1|`

     где цифра «1» означает команду: передать блок.
   - Ответ на это представляет собой пакет формата:

     `|packet_size|request_id|data|`
   - Отрицательное решение — пакет формата:

     `|packet_size|request_id|2|`
   - Ответа на данное решение не требуется.

5) Протокол полностью асинхронный: один клиент может совершить любое количество запросов, не дожидаясь ответа на предыдущие.

6) Для упрощения условия считается, что оперативная память ничем не ограничена, и `request_id` всегда уникальный.

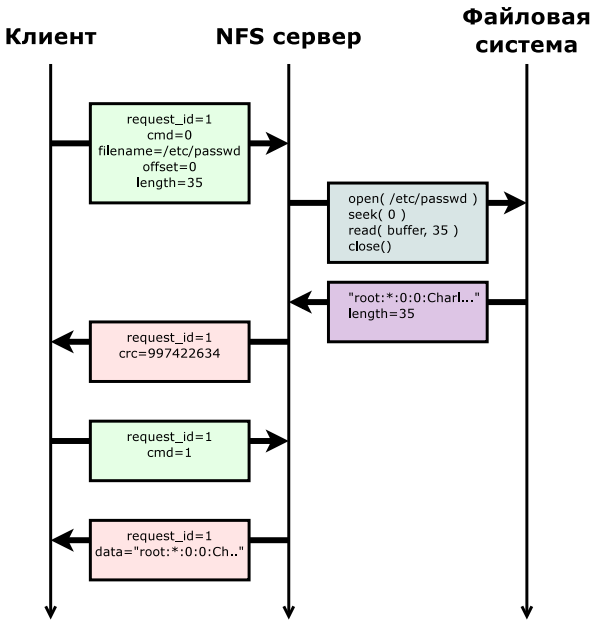


Рис. 6.1. Пример сессии протокола

С примером одной такой сессии можно ознакомиться на диаграмме 6.1.

Можно прикинуть, какой должна быть архитектура такого сервера, построенного вокруг `select(2)` или аналогичного механизма. Единственная проблема, что даже для подобного игрушечного проекта кода получается очень много, и программировать его следует с особой аккуратностью.

Следует признать, что реализация данной задачи с использованием зеленых нитей жизнь облегчит ненамного. Учитывая асинхронную природу запросов от одного клиента, контекст выполнения все равно оказывается рваным, и алгоритм приходится расписывать в нескольких различных местах. Один из возможных вариантов реализации выглядит следующим образом:

1) На каждое клиентское соединение порождается одна нить, которая в бесконечном цикле считывает запросы (используется аналог блокирующего `recv(2)`) и одна нить, которая в бесконечном цикле передает ответы (блокирующим `send(2)`) из некой очереди ответов.

2) Из каждого полностью считанного запроса добывается `request_id` и проверяется его «актуальность» по глобальной таблице.

   - Если это первый запрос протокола (в таблице нет записей, соответствующих этому `request_id`, то порождается отдельная нить-обработчик, которая аналогом блокирующего `read(2)` считывает `length` октетов из файла `filename`, начиная со смещения `offset`, в буфер. В глобальной таблице этот буфер сохраняется под ключом `request_id`. В очередь для нити-передатчика кладется сформированный ответ, содержащий контрольную сумму данных в буфере.
   - Если это второй запрос в протоколе и команда была равна «1», то дли нити-передатчика создается пакет с данными буфера. Для команды «2» ничего не создается. В любом случае, в глобальной таблице запись под ключом `request_id` уничтожается.

Данная реализация, разумеется, намного проще для восприятия, нежели «лапша» из условий вокруг `select(2)` или каша из коллбеков событий вокруг реактора. Однако даже на такой простой задаче обнаруживаются, как минимум, два неприятных места, требующих от программиста специальных мер по реструктуризации программы, реализующей протокол.

Первое такое место — это выделение отдельной нити со специальной очередью под рассылку ответов. Действительно, из-за асинхронной природы протокола нить-обработчик не имеет возможности только что считанный буфер сразу же отослать обратно клиенту. На больших данных `send(2)` заблокируется до момента окончания передачи на физическом уровне и управление будет отдано другой зеленой нити, которая, в свою очередь, может начать отсылать другие данные этому же клиенту. Так как сокет один и тот же, пакеты окажутся битыми.

Одно из очевидных решений данной проблемы — использование инструментов для межпоточной синхронизации. Например, запись в сокет можно защитить мьютексом. Даже в том случае, когда рантайм реализует зеленые нити в один поток, такой подход сможет гарантировать атомарность операции отсылки пакетов. Второе очевидное решение – специальная выделенная нить-передатчик — продемонстрировано в вышеприведенной реализации.

Второе место, требующее реструктуризации алгоритма, опять же, связано с асинхронной природой протокола. Несмотря на то, что алгоритм обработки запросов в пределах одного `request_id` строго последовательный (запрос → CRC → запрос → данные или конец), программа должна уметь принимать на обработку запросы с различными `request_id`, не дожидаясь окончания обработки предыдущих. Хотя шагов в последовательности *request*/*response* всего два, в реализацию все равно пришлось вводить дополнительную сущность: глобальную таблицу идентификаторов запросов.

И здесь достаточно легко предсказать, чем обернется дальнейшее развитие и усложнение протокола рассматриваемой сетевой файловой системы: в записи глобальной таблицы под ключом `request_id` сначала появляется текущее состояние некоего конечного автомата, затем дополнительные данные, в которых хранится контекст обработки текущего шага. На этапе очередного рефакторинга конечный автомат преобразуется в пару реактор + набор событий: и вуаля, мы снова вернулись к тому, от чего пытались спастись зелеными нитями. Зеленые нити удобно абстрагируют один уровень стека протоколов, но не позволяют произвести этот же фокус со следующим, который пользователь определяет сам.

Как вариант, конечно, можно запускать неумирающую в пределах одного `request_id` нить, которая будет обслуживать одну сессию протокола. Через систему межпроцессных коммуникаций (например, *pthread conditions* можно передавать ей все приходящие пакеты в пределах ее сессии и забирать уходящие очереди. Но здесь надо быть крайне аккуратным, чтобы не запутаться во всех синхронизирующих примитивах для такого количества нитей.

6.4. Продолжения

6.4.1. Общий вид

В этом месте статьи у нас уже должно сформироваться интуитивное понимание возможностей некого волшебного инструмента, к необходимости наличия которого подводят

предыдущие разделы. Давайте попытаемся их явно сформулировать: нужно научиться записывать в «явном», «плоском», «последовательном» виде алгоритмы, содержащие в себе асинхронные процессы.

Такие процессы логично реализуются через конечный автомат (или надстройку над ним), используя переключение состояний в местах *разрыва контекста*, когда, прежде чем переходить к другому действию, следует дождаться какой-то реакции «извне». Примеров в реальной практике огромное количество: ожидание пользовательского ввода в UI, реализация протоколов для взаимодействия, работа с оборудованием и так далее. Основным недостатком конечного автомата здесь является сильное преобразование исходного алгоритма при реализации, причем в сторону его запутанности и сложности для восприятия.

Нам же нужен инструмент, который вводит дополнительный уровень абстракции над автоматом. Пряча его где-то внутри, он позволяет алгоритму «склеиваться» обратно. Неплохим примером являются платформы с поддержкой зеленых нитей: спрятав какой-нибудь `libevent` внутри рантайма, они предоставляют средства для псевдо-многозадачности, позволяя писать алгоритмы, общающиеся с операционной системой, в нормальной последовательности, без разрывов контекста. Тем не менее, поддержка зеленых нитей сама по себе не даёт возможностей для расширения подобных абстракций дальше — на пользовательском уровне.[2]

В принципе, должна уже вырисовываться общая картина того, как именно должен работать такой механизм. В точках разрыва контекста выполнения (там, где вводятся состояния конечного автомата) нужно уметь сохранять полный их *снимок* в некий отдельный полноправный объект. Можно провести аналогию с *hibernate* в ноутбуках, когда, перед обесточиванием, все содержимое оперативной памяти сбрасывается на жесткий диск, чтобы после включения можно было продолжить работу с операционной системой с той же самой точки, когда было произведено выключение, без необходимости загрузки с нуля. Или с *save game* / *load game* в компьютерной игре. Грубо говоря, в программе в местах разрыва контекста нужно уметь создать такой *снимок*, уметь его куда-нибудь сохранить, и, когда появится необходимая информация, вернуться к выполнению алгоритма с сохраненного места — «продолжить».

Такой механизм известен под термином «продолжения» (*continuations*). Очевидно, что это не *goto*, не исключения, а совершенно другой инструмент для иных целей.

6.4.2. Продолжения в Scheme

В Scheme создание такого *снимка* (получение текущего объекта продолжения) известно как `call-with-current-continuation` или, сокращенно, `call/cc`. Выглядит это так:

```
(call/cc (lambda (k) ...))
```

В переменной *k* в замыкание передается тот самый объект продолжения, представляющий собой *снимок* программы на текущем месте. Обратите внимание, что на этом этапе никакой специальной магии не происходит и *Control Flow* программы никак не изменяется. Поведение аналогично коду **`(apply (lambda (k)...)(list (somehow-get-the-current-continuation)))`**.

Полученный объект `k` можно сохранить и далее вызывать из произвольного места как функцию (синтаксически):

```
(k arg1 arg2 ...)
```

Но, в отличие от вызова функции, будет иметь место совершенно другой эффект. Произойдут три важных события:

1) Вне зависимости от того, где именно было вызвано продолжение, управление перейдет к тому месту в программе, где физически находится `call/cc`. При этом восстановится весь стек вызовов и все локальные переменные — как будто был произведен откат во времени (разумеется, за исключением переменных, чьи значения могли быть изменены с помощью деструктивного присваивания типа `set!`).

2) После такого перехода (вызова продолжения) управление обратно передано **не будет** — по аналогии с `goto`, а не вызовом функции.[3] Это безусловный переход в программе на заранее сохраненное состояние, с потерей текущего.

3) Переданные продолжению аргументы будут возвращены как множественные результаты `values` от формы `call/cc`. При этом функция, переданная `call/cc` повторно вызвана не будет! Рассмотрим пример:

   ```
   (call/cc (lambda (k) (+ 4 (k 0) 7 8)))
   ```

   Что здесь произойдет, по шагам:

   (a) Функция `call/cc` захватывает текущее продолжение и вызывает переданное замыкание, передавая ему это продолжение.
   (b) В замыкании начинают вычисляться аргументы для последующего вызова функции сложения.
   (c) На вычислении аргумента `(k 0)` произойдет немедленная потеря всего контекста и безусловный переход к месту, где было захвачено продолжение: форме `call/cc`.
   (d) Аргумент `0`, с которым была произведена «разморозка» продолжения, будет возвращен формой `call/cc`.
   (e) Результатом выполнения всей формы будет `0`.

   Подобная возможность позволяет нам передавать разного рода данные обратно из точки, где мы вызываем продолжение, в точку, где мы его захватывали. Одно такое возвращаемое значение можно использовать напрямую, как будто это результат вызова `call/cc`, а множественные возвращаемые значения следует перехватывать с помощью `call-with-values` или специальным синтаксисом `receive` из *SRFI-8* [3]. Например:

   ```
   (call-with-values (lambda () ...) proc)
   ```

   Здесь сначала будет выполнен *thunk* (первый аргумент), затем будут собраны все значения, которые были им возвращены (ноль, одно или несколько) и выполнится функция `proc`, которой все эти значения будут переданы в качестве аргументов.

[2] Платформа может предоставлять другие возможности, за счет которых такое расширение будет все же возможно — к примеру, в Erlang за счет удобных средств по организации программы в виде множества процессов, обменивающихся сообщениями, можно легко выстроить такой «стек абстракций» в виде конвейера взаимодействующих процессов.

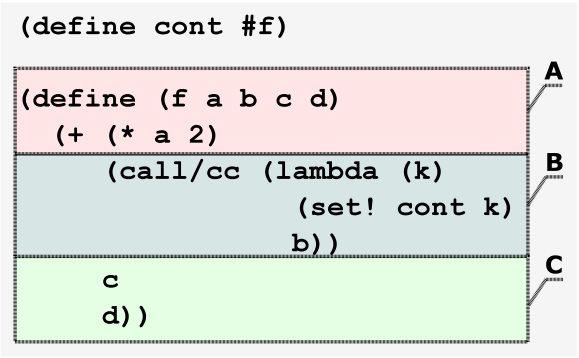


Рис. 6.2. call-with-current-continuation

Давайте рассмотрим на примере. На рис. 6.2 приведен размеченый тремя областями код на Scheme. Стандартный вызов функции `(f 1 2 3 4)` даст результат вычисления параметров: 11. Помимо этого, в области **B** происходит захват и сохранение текущего продолжения в глобальной переменной `cont`.

В момент выполнения `call/cc` был произведен *снимок* выполнения программы на текущий момент. В переменной *k* (и, соответственно, `cont`) теперь находится вся область **A**: стек вызовов содержит вызов `f`, вызывается функция `+` с четырьмя аргументами, и для нее уже вычислен первый аргумент: `(* a 2)`.

Теперь, что произойдет, если продолжение, сохраненное в `cont` будет вызвано с произвольным аргументом, допустим, `(cont 100)`?

1) Из продолжения восстановится все сохраненное состояние (область **A**).

2) Вся конструкция `call/cc` вернет `100`, не вызывая на этот раз свой аргумент-продолжение.

3) Продолжится выполнение программы с этого места: довычисляются оставшиеся аргументы функции сложения, считается результат и происходит возврат из функции `f`.

Результатом `(cont 100)` будет `2 + 100 + 3 + 4 == 109`.

Но давайте рассмотрим более практический пример, на котором процесс должен стать более понятным.

6.4.3. Конвертация коллбека в итератор

Существует два популярных паттерна в программировании для проектирования абстрактного интерфейса для обхода коллекций: итератор (курсор) и коллбек, иначе называемые «активный» и «пассивный» итераторы. В первом случае мы создаем специальный объект, у которого запрашиваем очередной элемент коллекции, во втором – передаем функции итерации алгоритм для обработки очередного элемента (в виде замыкания или функционального объекта). Соответственно, итераторы более широко применяются в объектно-ориентированных языках (Java, C++/STL, C#, Python, etc), а коллбеки — в языках, удобно поддерживающих замыкания (ruby, все лиспы, и т. д.).

[3] В отлиие от т. н. delimited continuations (http://okmij.org/ftp/continuations/undelimited.html), которые возвращают значение и могут быть использоваться как полноправные функции.

Оба подхода имеют свои плюсы и свои минусы, но в статье мы не будем на них останавливаться. Для одних задач лучше подходят итераторы (паузы в итерации), для других — коллбеки (RAII), но идеального инструмента на любой случай жизни нет.[4]

Поэтому иногда возникает задача над уже существующим API построить другое: превратить итератор в коллбек или наоборот.

Интрига здесь заключается в том, что если итератор в коллбек превращается достаточно тривиально, то обратную операцию произвести невозможно без продолжений или специализированных конструкций языка (вроде синтаксиса `yield` в Python, который преобразует специально оформленную функцию в объект-генератор).

Давайте рассмотрим следующую проблему: от некоторого модуля или библиотеки мы получили следующий интерфейс для преобразования дерева: `map-tree`:

```
(define (map-tree proc tree)
  (if (list? tree)
      (map (lambda (node) (map-tree proc node)) tree)
      (proc tree)))
```

В статье реализация данной функции предельно упрощена для наглядности, но в реальной жизни это может быть что-нибудь сложное, вроде процедуры для обхода огромных *xml*-файлов, но имеющее похожий интерфейс.

Допустим, перед нами встала задача, не решаемая (классическим подходом) данной функцией: имея два дерева нужно найти длину их общего префикса, вне зависимости от их конкретной структуры. Грубо говоря, изобразить аналог:

```
(list-prefix-length (flatten tree-a) (flatten tree-b))
```

но через использование функции `map-tree`. Для языка с ленивым порядком вычислений (например, Haskell), скорее всего, подойдет и такое решение (если `map-tree` нестрогая), но в других языках, пожалуй, единственный вариант реализации — это полный проход по дереву со сборкой списка, что-то вроде:

```
(define (tree->list tree)
  (let ((lst '()))
    (map-tree (lambda (node)
                (set! lst (cons node lst)))
              tree)
    (reverse! lst)))
```

Процедура повторяется для обоих деревьев и затем выполняется поиск общего префикса. Вариант рабочий, но крайне ресурсозатратный в том случае, если деревья очень большие, а общий префикс может быть либо крайне малым, либо отсутствовать вообще. Интуитивно понятно, что корректней будет каким-нибудь волшебным образом запустить в параллель две `map-tree` на обоих деревьях и сравнивать попарно получаемые элементы. Такой волшебный способ существует и, собственно, называется «продолжения».

Давайте сначала представим, как бы мы решали такую задачу, если бы у нас **уже** был итератор, выдающий по одному элементу от дерева. Наверное, логично было бы абстрагировать

[4] Вообще-то есть еще iteratee: http://john-millikin.com/articles/understanding-iteratees/. В последнее время хаскельный мир отказывается от ленивого IO и переходит на iteratee. Это в каком-то смысле гибрид активных и пассивных итераторов, обладающий преимуществами обоих. Правда, в языке без алгебраических типов и без удобного синтаксиса замыканий использовать их крайне некомфортно.

проход по обоим деревьям специальной версией комбинатора `fold`:

```
(define (trees-prefix-length tree-a tree-b)
  (call/cc
   (lambda (return)
     (fold-trees (lambda (a b len)
                   (if (equal? a b)
                       (1+ len)
                       (return len)))
                 0
                 (make-iterator tree-a)
                 (make-iterator tree-b)))))
```

В этом коде используется один `call/cc`, но исключительно для нужд досрочного возвращения результата, аналогично `return-from` в Common Lisp (см. паттерн *Escape from a Loop* из статьи *Call with Current Continuation Patterns* [2]).

Соответственно, комбинатор выглядел бы следующим образом:

```
(define (fold-trees proc seed iterator-a iterator-b)
  (call-with-values (lambda () (iterator-a))
    (lambda (leaf-a next-iterator-a)
      (call-with-values (lambda () (iterator-b))
        (lambda (leaf-b next-iterator-b)
          (if (and next-iterator-a next-iterator-b)
              (fold-trees proc
                          (proc leaf-a leaf-b seed)
                          next-iterator-a
                          next-iterator-b)
              seed))))))
```

А теперь давайте подумаем, как использовать подобный код с `map-tree`.

Основная идея состоит в том, чтобы научиться «выпрыгивать» из середины коллбека в `map-tree` наружу, вынося текущий элемент, а затем, как ни в чем не бывало, передавать управление обратно. Соответственно, нам нужно здесь два продолжения: на одно мы возвращаем управление из коллбека, а другое захватываем прямо там и передаем вовне. Выглядеть это будет так:

```
(define (tree-gen tree flow)
  (map-tree
   (lambda (leaf)
     (set! flow (call/cc (lambda (k) (flow leaf k))))
     leaf)
   tree)
  (flow #f #f))
```

Замечание: вообще, главная хитрость в такого рода задачах — не возвращаться из подобных функций обычным образом. Давайте разберем по шагам, что здесь происходит:

1) Функция `tree-gen` запускает `map-tree`, передавая в качестве обработчика собственное замыкание.

2) Как только `map-tree` выдает замыканию очередной элемент `leaf`, происходит связывание текущего продолжения с аргументом `k` в `(lambda (k)(flow leaf k))`.

3) Управление немедленно возвращается наружу в продолжение, находящееся в переменной `flow`, при этом ему передается обрабатываемый элемент и только что сохраненное продолжение.

4) Как только внешние силы вернут управление обратно (по переданному наружу продолжению `k` — сам вызов `(flow leaf k)` завершиться не может, поскольку `flow` — продолжение), ожидается, что будет указана (возвращена) новая точка перехода наружу. Эта точка перезаписывает переменную `flow` — в следующий раз управление будет переданно именно туда.

5) Таким образом, в переменной `flow` последовательно оказываются продолжения вида «выдать текущий элемент и переключиться на остаток дерева».

6) Как только работа `map-tree` будет завершена, функция `tree-gen` не возвращается обычным образом, а снова выкидывает управление наружу, передавая вместо точки возврата ложь (`#f`), сигнализируя тем самым, что итерация окончена и возвращаться далее некуда.

То есть, по сути, управление постоянно передается туда-сюда между *генератором* (`map-tree`) и *потребителем* (кодом, последовательно вызывающим в цикле возвращенные из `map-tree` продолжения, пока не получит ложь `#f`). *Потребитель* сохраняет свое продолжение и передает его *генератору* (через вызов его продолжения); тот, в свою очередь, сохраняет свое и возвращает *потребителю*, — получается такое жонглирование.

Соответственно, алгоритм работы с `tree-gen` очень похож на использование обычного итератора и выглядит так:

1) Инициализация итератора: `(call/cc (lambda (k)(tree-gen tree k)))` — сразу получаем пару в `values`: первый элемент и следующее продолжение `map-flow` (захваченное внутри `map-tree`).

2) Получение очередного элемента: `(call/cc map-flow)` — точно так же получаем в `values` элемент и следующее продолжение из `tree-gen`.

3) Итерация заканчивается, когда очередной вызов `(call/cc map-flow)` возвращает `#f` вместо очередного продолжения.

Теперь, превратив `map-tree` в итератор, нет проблем написать функцию, аналогичную стандартной `fold` из *srfi-1*, только для параллельной поэлементной обработки двух деревьев. Соответственно, пользовательское замыкание будет получать очередной элемент из первого дерева, очередной элемент из второго и аккумулятор `seed`:

```
(define (fold-trees proc seed cont-a cont-b)
  (call-with-values (lambda () (call/cc cont-a))
    (lambda (leaf-a next-cont-a)
      (call-with-values (lambda () (call/cc cont-b))
        (lambda (leaf-b next-cont-b)
          (if (and next-cont-a next-cont-b)
              (fold-trees proc
                          (proc leaf-a leaf-b seed)
                          next-cont-a
                          next-cont-b)
              seed))))))
```

А теперь давайте проведем небольшой фокус: переименуем в коде `fold-trees` все подстроки **cont** на **iterator**, уберем вызовы `call/cc`, и перед нами окажется классический функциональный код, написанный с применением итераторов, который мы приводили в начале главы!

Теперь, построив комбинатор `fold` для двух деревьев, перепишем `trees-prefix-length` для нашего варианта на продолжениях.

```
(define (trees-prefix-length tree-a tree-b)
  (call/cc
   (lambda (return)
     (fold-trees (lambda (a b len)
                   (if (equal? a b)
                       (1+ len)
                       (return len)))
                 0
                 (lambda (k) (tree-gen tree-a k))
                 (lambda (k) (tree-gen tree-b k))))))
```

Давайте рассмотрим на диаграмме как конкретно работает программа на примере `(trees-prefix-length '(1 (3))'((1)7))`: диаграмма 6.3.

Черными стрелками показан ход выполнения программы при вычислении выражения. Зеленым цветом отмечены функции в стеке вызовов, которые используются непосредственно для вычисления, а голубым, фиолетовым, желтым и красным — скопированное состояние программы в процессе захвата продолжения. Серая стрелка указывает на блок со скопированным состоянием в процессе текущего `call/cc`. Все переменные, содержащие объект продолжения, на схеме взяты в треугольные скобки, чтобы их было проще отличать от обычных переменных и функций.

6.4.4. Итоги

Итак, теперь мы имеем инструмент, способный в произвольном месте «замораживать» и откладывать выполнение программы «до лучших времен». Вообще, для общего ознакомления с паттернами применения продолжений, я рекомендую прочитать статью Даррелла Фергюсона и Деуго Дуайта *Паттерны использования «call with current continuation»* [2], перевод которой доступен в библиотеке ПФП.

Нам же продолжения позволят, в том числе, откладывать выполнение блокирующего `read(2)` до момента, когда `select(2)` сигнализирует о наличии в дескрипторе данных для чтения!

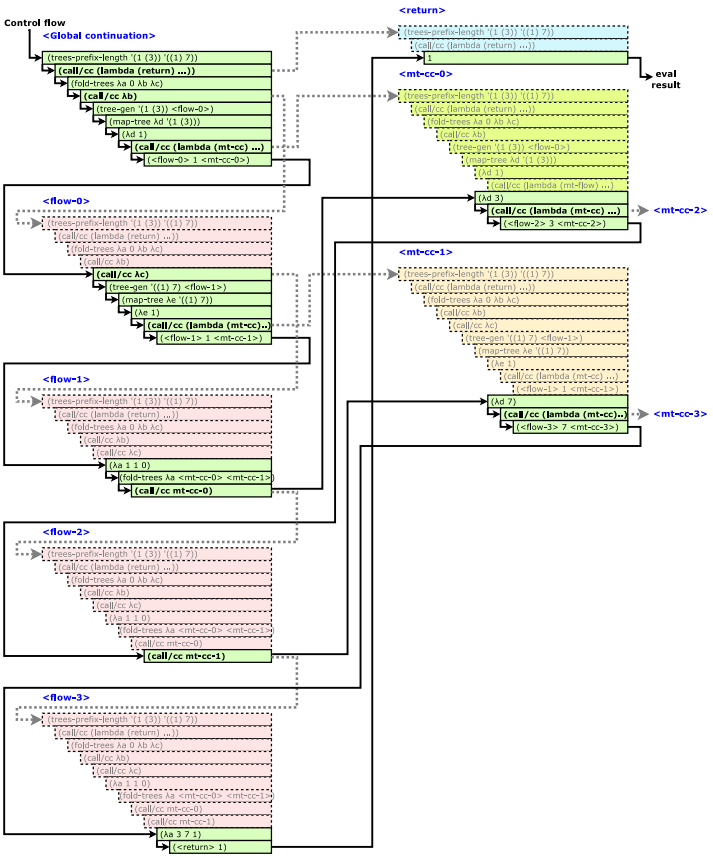


Рис. 6.3. tree-prefix-length, ход выполнения `(trees-prefix-length '(1 (3)) '((1) 7))`

6.5. Практика #1: nfs.scm

6.5.1. Введение

Самое время продемонстрировать, как с помощью продолжений реализуется проект «Сетевая файловая система», описанный в разделе 6.3.

В сервере, несмотря на всю его примитивность, большое количество асинхронных действий, которые серьезно мешают реализовать проект традиционными способами (мультиплексирование, reactor, многопоточность):

- `accept(2)`: следует отложить до момента, когда `select(2)` просигнализирует о наличии данных на серверном сокете для чтения.
- `recv(2)/read(2)`: следует отложить до момента, когда `select(2)` просигнализирует о наличии данных на клиентском сокете и, соответственно, на дескрипторе открытого файла для чтения.
- `send(2)`: следует отложить до момента, когда `select(2)` сигнализирует о возможности записи в клиентский сокет.

- Обработчик запросов имеет смысл запускать только после того, как весь пакет запроса будет считан из клиентского сокета.
- Формировать ответный пакет можно лишь после того, как из файла будет считан буфер нужного размера.
- Следующий в сессии запрос можно начинать обрабатывать только после того, как клиенту будет отослан ответ на предыдущий запрос этой сессии.

Все эти действия нуждаются во взаимной синхронизации. Поэтому однопоточная реализация с использованием мультиплексирования превращается в полный кошмар, а многопоточный вариант на блокирующемся вводе/выводе (включая использование зеленых нитей) все равно, либо нуждается в программировании автомата для обработки асинхронного протокола, либо в нетривиальной конструкции из межпоточных синхронизаций.

Очень простая, но полностью работоспособная реализация сервера на языке Scheme (я использовал реализацию Guile[5] как наиболее доступную) с активным использованием продолжений занимает около сотни строк кода. В коде применяется популярный трюк с перебрасыванием хода выполнения программы между *диспетчером* и несколькими *сопрограммами*. Сопрограмма, натыкаясь на потенциально блокирующийся вызов, при помощи `call/cc` приостанавливает свое выполнение, сохраняя свое продолжение в специальной таблице и возвращая управление диспетчеру. А как только диспетчер обнаруживает, что сопрограмма получила возможность исполняться далее без блокировок (например, опросом системного мультиплексора `select(2)`), он передает ей управление обратно. Какого конкретно события ожидает продолжение, диспетчер понимает по тому, в какой именно таблице оно записано: продолжения в `queue-read` ожидают, когда из указанного дескриптора можно будет читать без блокировки, а продолжения из `queue-write` — когда в дескриптор можно будет писать (то есть, в диспетчере «зашито» знание о таблицах `queue-read` и `queue-write`).

6.5.2. Архитектура

Давайте вкратце обрисуем то, что у нас должно получиться, и разберем смысл структур данных.

Сам алгоритм работы предельно упрощен:

1) Ожидание подключения клиента.
2) Прием данных (в три этапа: считывание длины пакета, считывание данных и парсинг).
3) Обработка шага протокола.
4) Формирование ответа и отправка его обратно клиенту.

Идея реализации состоит в том, чтобы каждый такой шаг выполнять с собственным отдельным *Control Flow*: независимо от остальных. Для этого мы обрамляем каждый пункт в отдельную *сопрограмму*, тем самым обозначая псевдопараллельность их выполнения.

Теперь, что касается основных структур данных:

- `queue-read` и `queue-write`: хэш-таблицы, ключом в которых служит файловый дескриптор (*порт* в терминах Scheme), а значением — набор продолжений, ожидающих события на этом дескрипторе. Соответственно, в диспетчере захардкожено, что продолжения из первой таблицы ожидают возможности неблокирующего чтения из портов, а из второй — неблокирующей записи в порты, и при обнаружении такой возможности при помощи `select(2)` диспетчер «будит» соответствующие продолжения.

- `queue-reqs`: практически аналогичная по смыслу хэш-таблица, кроме того, что ключами здесь служат уникальные идентификаторы сессий протокола: `request-id`.

- `current-flow`: «глобальный» *Control Flow*, в котором работает диспетчер. Перед тем, как отдать управление очередной сопрограмме, диспетчер должен сохранить в `current-flow` свое продолжение, на которое сопрограмма обязана будет вернуть управление, когда возжелает приостановить собственное выполнение или закончить работу.

  Для того, чтобы было возможно использовать вложенные сопрограммы (то есть, вызывать одну «приостанавливаемую» процедуру из другой), `current-flow` можно сохранять в воображаемом «стеке» с помощью `call-with-saved-flow`: она сначала «припрятывает» текущее значение `current-flow`, а затем, на выходе, восстанавливает его.

- **Пакет** структурно будет ровно таким, как было описано в соответствующем разделе 6.3. Для упрощения процедуры парсинга, поля из фрейма пакета, в котором находятся данные, вычитываются стандартной лисповой функцией `read`. Это накладывает некоторые незначительные ограничения: например, строки должны быть обрамлены кавычками, чтобы предотвратить их считывание в качестве символов.

6.5.3. Базовый механизм

Весь механизм диспетчера/сопрограмм реализуется тремя базовыми функциями: `call-with-flow`, `call-with-async` и `resume/cc`.

- `(call-with-flow thunk)`: запускает сопрограмму:
  1) Функция обрамляет вызов переданного ей замыкания `thunk` в `call/cc`, в котором она инициализирует значение `current-flow` только что захваченным продолжением.
  2) Если выполнять `call-with-flow` с функцией, не использущей `call-with-async` и не вызывающей продолжение из глобальной переменной `current-flow`, то ее поведение будет аналогично тому, как если бы ее (функцию) вызвали напрямую.
  3) Таким образом, во время выполнения `(thunk)` в глобальной переменной `current-flow` будет находиться продолжение, соответствующее остатку программы после `(call-with-flow thunk)`. Затем, встретив блокирующий вызов, мы сможем при помощи `call-with-async` отложить в сторонку выполнение `thunk` до более подходящего момента, а пока продолжить выполнять этот остаток программы — получается подобие многозадачности.
  4) По завершению `thunk` с возвращенными ей результатами происходит вызов продолжения `current-flow` — то есть, выполнение программы, окружавшей `(call-with-flow thunk)`, продолжает-

[5] http://www.gnu.org/s/guile/

ся так, как будто этот вызов вернул указанные результаты.[6]

- `(call-with-async thunk queue key)` добавляет текущее замыкание в таблицу `queue` под ключом `key` и возвращает управление из сопрограммы, определенной `call-with-flow` продолжением, находящимся в `current-flow`. Управление может быть возвращено обратно в сопрограмму, когда произойдет событие, определенное семантикой таблицы `queue`. Например, если это была `queue-read`, диспетчер вернет управление в тот момент, когда из дескриптора, заданном `key`, можно будет осуществить неблокирующее чтение.

- `(resume/cc queue key . vals)` выбирает одно продолжение из таблицы `queue` под ключом `key` и воскрешает его с аргументами `vals`. Какое конкретно продолжение из набора под ключом `key` воскрешать, и кому из них будут переданы аргументы `vals` — в нашем проекте не важно (конкретно в коде выбирается первое продолжение из набора), но теоретически можно использовать здесь любую стратегию, например, *Priority Queue*.

  Перед перекидыванием *Control Flow* по выбранному продолжению в глобальной переменной `current-flow` сохраняется текущее, чтобы следующая передача управления из сопрограммы, по факту, вернулась обратно. А в параметрах `vals` можно передавать сопрограмме любые данные, которые она получит как множественные результаты от `call/cc`.

```
(define current-flow #f)

(define (call-with-saved-flow thunk)
  (let ((old-flow current-flow))
    (call-with-values thunk
      (lambda vals
        ;; collect all parameters in the list 'vals'
        (set! current-flow old-flow)
        ;; invoke 'values' with parameters in 'vals'
        (apply values vals)))))

(define (call-with-flow thunk)
  (call-with-saved-flow
   (lambda ()
     (call/cc
      (lambda (flow)
        (set! current-flow flow)
        (call-with-values thunk
          (lambda vals (apply current-flow vals))))))))

(define (call-with-async thunk queue key)
  (call-with-values
      (lambda ()
        ;; Push current continuation ("then call (thunk)")
        ;; onto queue[key] and suspend execution.
        ;; When we're resumed, (thunk) will be called
        ;; (e.g. a potentially blocking read(2), but it
        ;; will actually be non-blocking because it's
        ;; resumed only when the corresponding port
        ;; is reported "ready for read" by select(2))
        (call/cc
          (lambda (k)
            (hash-set! queue key
                       (cons k (hash-ref queue key '())))
            (current-flow #f))))
    ;; pass all the results returned by 'call/cc' to
        'thunk'
    thunk))

(define (resume/cc queue key . vals)
  (let* ((ks (hash-ref queue key))
         (k (car ks))
         (rest-ks (cdr ks)))
    (if (null? rest-ks)
        (hash-remove! queue key)
        (hash-set! queue key rest-ks))
    (call-with-saved-flow
     (lambda ()
       (call/cc (lambda (next)
                  (set! current-flow next)
                  ;; pass 'vals' to restored continuation
                  (apply k vals))))))
```

Итак, раскрываем основную идею:

1) Отдельный алгоритм программы оформляется в сопрограмму с помощью `call-with-flow`.

2) Как только алгоритм натыкается на потенциально блокирующий вызов (допустим, к примеру, `accept(2)`) — при помощи `call-with-async`:

   - выбирается момент, когда желательно управление вернуть обратно, и, соответствующим образом выбирается специальная таблица (для `accept(2)` нам интересен факт доступности сокета для чтения, следовательно, таблица будет `queue-read`);
   - в выбранную таблицу добавляется текущее продолжение;
   - происходит выход из *сопрограммы*, которую мы ограничили блоком `with-flow`.

3) Диспетчер работает в отдельной сопрограмме, блокируясь на `select(2)`. Как только мультиплексор сигнализирует о том, что произошли какие-то из запрошенных другими сопрограммами событий, из соответствующих таблиц по очереди выбираются и «воскрешаются» продолжения этих сопрограмм.

6.5.4. Серверная часть

Давайте рассмотрим функцию инициализации сервера (точка входа в программу):

```
(define (run-nfs-server listen-port)
  (let ((server-socket (socket AF_INET SOCK_STREAM 0)))
    (bind server-socket AF_INET INADDR_ANY listen-port)
    (listen server-socket -1)
    (call-with-flow
      (lambda () (nfs-server-acceptor server-socket)))))
```

После инициализации серверного сокета запускается сопрограмма обработки входящих соединений:

```
(define (nfs-server-acceptor server-socket)
  (let* ((client (accept/cc server-socket))
```

[6] Несмотря на то, что в нашей реализации механизма `call-with-flow` умеет возвращать текущему продолжению результаты `thunk`, в коде nfs-сервера эта возможность реально не используется.

```
        (client-socket (car client))
        (client-data (cdr client)))
    (run-requests-reader client-socket client-socket)
    (nfs-server-acceptor server-socket)))
```

С виду она ничем не отличается от алгоритма, использующего многопоточность и блокирующиеся системные вызовы: тот же самый бесконечный цикл, в котором `accept(2)` принимает входящие соединения и запускает обработчики для них. Здесь вся магия скрыта в функции `accept/cc`:

```
(define queue-read (make-hash-table))

(define (accept/cc port)
  (call-with-async
    (lambda () (accept port)) queue-read port))
```

Здесь выполняются следующие действия: при помощи вызова `call-with-async` в таблице `queue-read` под ключом `port` регистрируется текущее продолжение («выполнить `(accept port)`») и происходит немедленный выход из блока `with-flow` в текущее продолжение диспетчера. По логике алгоритма сопрограмма должна быть продолжена в тот момент, когда диспетчер при помощи `select(2)` убедится, что порт (сетевой дескриптор в данном случае) стал доступен для чтения. Соответственно, запустив сервер в **REPL** мы должны:

1) Немедленно получить управление обратно:

```
guile> (run-nfs-server 1115)
#f
```

2) Убедиться, что замороженное продолжение хранится в таблице `queue-read`:

```
guile> (hash-map→list cons queue-read)
((#<input-output: socket 18> #<continuation 2018 @
     8643e50>))
```

Теперь, чтобы сервер заработал, нам нужена собственно диспетчерская сопрограмма, выполняющая мультиплексирование и передачу управления рабочим сопрограммам.

```
(define (multiplex-loop)
  (multiplex)
  (multiplex-loop))

(define (multiplex)
  (apply
   (lambda (ready-read ready-write . rest)
     (resume/ccs queue-read (vector→list ready-read))
     (resume/ccs queue-write (vector→list ready-write)))
   (select (queue-ports queue-read)
           (queue-ports queue-write)
           #())))

(define (queue-ports queue)
  (list→vector (hash-map→list (lambda (k v) k) queue)))

(define (resume/ccs queue keys . vals)
  (for-each (lambda (key)
              (apply resume/cc queue key vals))
            keys))
```

Теперь, запустив диспетчер с помощью функции `multiplex-loop`, можно ожидать от сервера выполнения полезной работы.

- `multiplex` собирает все ключи из таблиц `queue-read` и `queue-write` (ключами в этих таблицах являются файловые дескрипторы открытых сокетов или файлов) и применяет к ним системный вызов `select(2)`.
- Этот мультиплексор отдает команду операционной системе заблокировать выполнение программы до тех пор, пока на каком-нибудь дескрипторе (или нескольких) из `queue-read` не появятся данные для чтения (и, соответственно, `read(2)` будет неблокирующим) или на каком-нибудь дескрипторе (или, опять же, нескольких) из `queue-write` не появится возможность выполнить неблокирующий `write(2)`.
- По выходу из `select(2)` в функции `multiplex` оказывается список дескрипторов, для которых возможны неблокирующие `IO` операции. Для каждого такого дескриптора из таблицы `queue-read` или `queue-write` достается соответствующее ему продолжение: буквально, сопрограмма, выполнение которой было временно заморожено до тех пор, пока на интересующем ее дескрипторе вероятен блокирующий ввод-вывод.
- Собранные программы в функции `resume/ccs` по очереди «размораживаются», одна за другой, в цикле, с использованием трюка, который мы применяли в разделе «конвертация коллбека в итератор»:
  - сохраняется текущее продолжение,
  - устанавливается в качестве «точки возврата»,
  - размораживается сопрограмма,
  - снова прерванная или завершившаяся сопрограмма передает управление на «точку возврата».

6.5.5. Клиентская часть

Давайте теперь рассмотрим поближе обработчик клиентских соединений `run-requests-reader`.

```
(define (run-requests-reader in-port out-port)
  (call-with-flow
   (lambda ()
     (define (reader)
       (call-with-values
         (lambda () (read-packet/cc in-port))
         (lambda (request-id . data)
          (if request-id
              (begin
                (touch-worker request-id out-port)
                (apply resume/cc queue-reqs request-id
                     data)
                (reader))))))
     (reader)
     (close-port in-port)
     (close-port out-port))))
```

Обработчик запускается в отдельной сопрограмме, а это означает, что асинхронные вызовы внутри нее, реализованные через `call-with-async`, будут возвращать управление из обработчика клиентских соединений на уровень выше — в серверный цикл приема соединений.

Реализация обработчика содержит в себе цикл, считывающий из соединения запросы — `read-packet/cc`, и передающий их далее в сопрограммы третьего уровня, реализующие уже собственно логику протокола. Цикл будет завершен, когда `read-packet/cc` вернет `#f`.

Как видно из названия функции `read-packet/cc`, она, как и рассмотренная в предыдущем разделе `accept/cc`, реализует выходную точку в сопрограмме `run-requests-reader`. При этом собственно асинхронный вызов в стеке будет только один: это системный `read-string!/partial`, а остальное — совершенно обычные алгоритмы по сборке пакета из бинарного потока данных.

```
(define (read-packet/cc port)
  (call/cc
   (lambda (escape)
     (define (read-frame/escape len)
       (let ((buffer (read-frame/cc port len)))
         (if buffer buffer (escape #f))))
     (define (parser port)
       (let ((value (read port)))
         (if (eof-object? value)
             '()
             (cons value (parser port)))))
     (let* ((packet-length
              (bytes->value (read-frame/escape 4)))
            (packet
              (read-frame/escape packet-length)))
       (apply values
              (call-with-input-string packet parser))))))

(define (read-frame/cc port len)
  (let ((buffer (make-string len)))
    (define (reader offset)
      (if (>= offset len)
          buffer
          (let ((bytes (read!/cc buffer port offset len)))
            (if bytes (reader (+ offset bytes)) #f))))
    (reader 0)))

(define (read!/cc buffer port start end)
  (call-with-async
    (lambda ()
      (read-string!/partial buffer port start end))
    queue-read
    port))
```

Начиная с конца:

- `read!/cc` выполняет классический `read(2)`: читает какое-то количество октетов в заданный буфер, но сначала обязательно убеждается, что чтение из дескриптора не будет блокирующим. Для этого она вызовом `call-with-async` регистрирует дескриптор и свое продолжение в таблице `queue-read`, а затем выбрасывает управление наружу ближайшего `call-with-flow`.

- `read-frame/cc` делает то же самое, но функция гарантированно считывает *len* октетов из файлового дескриптора. Для этого она в цикле выполняет столько вызовов `read!/cc`, сколько потребуется (ведь один такой вызов может считать меньше октетов, чем было запрошено).

- `read-packet/cc` считывает пакет за два вызова `read-frame/cc` (первый фрейм фиксированной длины 4 содержит в себе размер пакета, а второй фрейм будет содержать собственно пакет) и далее анализирует его. В программе считается, что данные в пакетах всегда целостны и корректны, поэтому никакой специальной обработки ошибок не происходит. `Call/cc`, реализуемый здесь, служит исключительно в целях досрочного возврата результата из цикла (см. аналогичный пример из функции `trees-prefix-length` в разделе 6.4.3).

Сами алгоритмы элементарны, но страшно представить, во что бы они превратились, если бы их надо было реализовывать конечным автоматом вокруг неблокирующегося `recv(2)`.

Итак, благодаря продолжениям нам уже удалось в стандартном линейном виде записать три блока нашего проекта: прием клиентских соединений сервером, считывание полностью сформированных и распарсенных пакетов из них и собственно сопрограмму диспетчера. В то же время, на системном уровне программа является чистейшим КА с кучей состояний и хаотичными переходами в разные стороны от `select(2)` и обратно.

Однако подобного уровня абстракции над системными вызовами можно добиться и обычной реализацией с использованием многопоточности (реальной или «зеленой», синтаксически это не имеет значение) и блокирующимся вводом/выводом. Поэтому сейчас мы рассмотрим применение нашего механизма уже на пользовательском уровне, для «схлопывания» автомата по обработке протокола сервера.

6.5.6. Реализация протокола

Протокол в пределах одной сессии `request_id` обрабатывает специальная сопрограмма. На каждый уникальный `request_id` цикл в `run-requests-reader` создает отдельную сопрограмму, которая для сохранения продолжений в своих асинхронных вызовах использует таблицу `queue-reqs`. Эту же таблицу обработчик клиенстких соединений использует для размораживания продолжений-worker'ов, как только он получает в руки полностью считанный и разобранный пакет-запрос.

То есть, используется тот же самый уже рассмотренный нами в разделах выше механизм `call-with-flow` + `call-with-async` + `resume/cc` (который выполняется из `run-requests-reader`, принимая разобранный пакет):

```
(define (run-worker request-id out-port)
  (define (worker-async thunk)
    (call-with-async thunk queue-reqs request-id))
  (call-with-flow
   (lambda ()
     (worker-async
      (lambda (cmd filename offset len)
        (let* ((f (open-file filename "r"))
               (buf (begin (seek f offset SEEK_SET)
                           (read-frame/cc f len))))
          (close-port f)
          (write-packet/cc out-port
                           (thing->lisp request-id)
                           (thing->lisp (string-hash buf)))
          (worker-async
           (lambda (cmd)
             (if (= cmd 1)
                 (write-packet/cc out-port
                                  (thing->lisp request-id)
                                  buf)))))))))))
```

Продолжения позволили нам в этих двух десятках строк кода полностью уместить всю логику по обработке протокола сервера сетевой файловой системы, включая также и работу с файлами! Код практически полностью идентичен наивному однопоточному синхронному варианту, но, тем не менее, он

полностью соответствует техзаданию. Обработка полностью асинхронна как на сетевом уровне и уровне файловой системы (сервер будет корректно работать даже на GPRS-соединении с файлами на дискете, обслуживая любое количество клиентов), так и на уровне сессий `request_id`: в пределах одного клиентского соединения одновременно их также может поддерживаться сколь угодно много. Ведь, по сути, это старая добрая однопоточная программа с мультиплексированием внутри.

Несложно прикинуть, что при дальнейшем расширении протокола не будет особых проблем при внесении изменений в программу. К тому же, нет необходимости менять диспетчер. Вся логика компактна и находится перед глазами, а ход выполнения прослеживается точно таким же образом, как и в обычном алгоритме без всякой асинхронности.

6.5.7. Схема работы

Давайте рассмотрим на схеме, как именно перекидывается управление в программе между сопрограммами и диспетчером: диаграмма 6.4. Несмотря на то, что физически программа полностью однопоточная, построеная вокруг мультиплексора `select(2)`, диаграмма напоминает ход работы многопоточной реализации или кода, подобного Erlang.

6.5.8. Итог

Полностью весь код сервера можно найти в файле *nfs.scm*, который прилагается к статье.

Разумеется, проект слишком примитивен, чтобы приносить какую-то реальную пользу, но он отлично демонстрирует практическую проблемную область для успешного применения продолжений. На самом деле, ни один другой подход не является в данном случае более удачным. Но если же вдруг кто-либо сможет продемонстрировать другой подход к проблеме, который:

- или окажется короче в реализации,
- или будет содержать меньше лишних технических сущностей, отвлекающих внимание (разнообразные флаги, переменные состояний, примитивы синхронизации и т. п.),
- или окажется существенно производительней,

то я буду рад увидеть эти варианты в комментариях к статье.

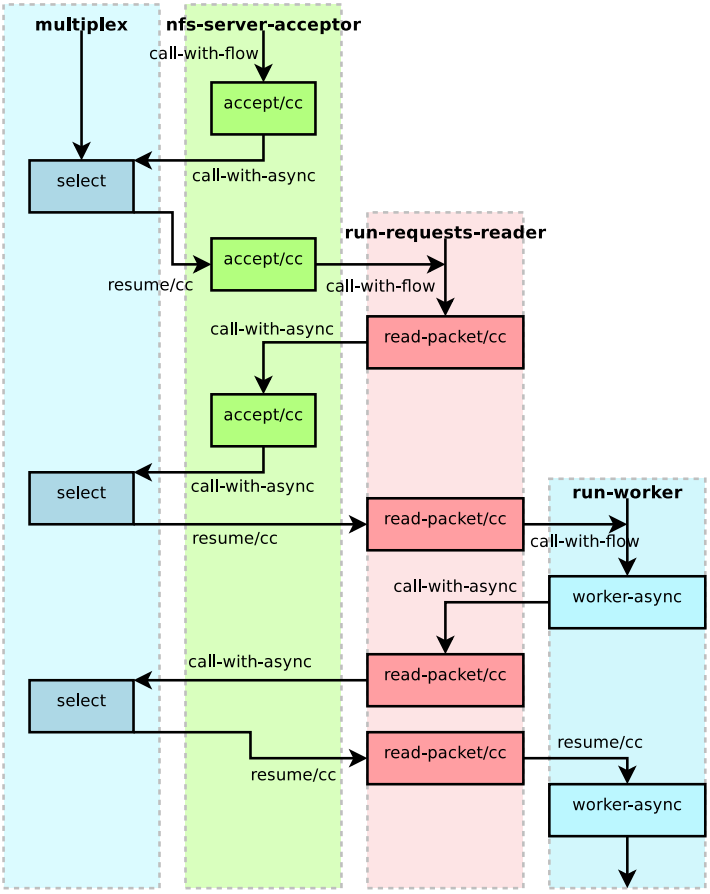


Рис. 6.4. Ход выполнения программы

6.6. Практика #2: zeromq/cc

6.6.1. Описание

В качестве развития темы программирования протоколов с помощью продолжений, я хочу представить свою ветку проекта cl-zmq[7] под именем *cl-zmq-tools*, которая, вероятно, либо войдет в состав основной библиотеки, либо будет развиваться дальше в рамках отдельного модуля.

Сам *cl-zmq* представляет собой низкоуровневый биндинг к библиотеке ZeroMQ[8] для Common Lisp, а *cl-zmq-tools* реализует задачу по повышению уровня абстракции ее интерфейса до привычного в языке. Код можно посмотреть в ветке *cl-zmq-tools* основного репозитория[9].

Библиотека *ZeroMQ* представляет собой высокопроизводительную платформу обмена сообщениями, абстрагируя транспортный уровень за универсальным программным интерфейсом. ZeroMQ-сокет можно прозрачно использовать для обмена сообщениями между разными нитями одного процесса,

[7] http://repo.or.cz/w/cl-zmq.git
[8] http://www.zeromq.org/
[9] http://repo.or.cz/w/cl-zmq.git/shortlog/refs/heads/cl-zmq-tools

разными процессами одной машины и различными машинами в сети.

Однако, несмотря на то, что уровень абстракции, предоставляемый библиотекой, выше, чем обычные системные средства коммуникации, все основные используемые техники для программирования асинхронных процессов (например, маршрутизатор сообщений на сокетах **XREQ** и **XREP**;[10] когда нужно уметь асинхронно читать из обоих) остаются такими же, как те, что мы рассматривали в первых разделах. Это:

- **Многопоточность**: (не рекомендуемый вариант) когда на прием сообщений из каждого сокета работают отдельные нити, между которыми, в свою очередь, каким-либо образом налажена коммуникация.
- **Мультиплексирование**: (рекомендуемый вариант) с использованием функции библиотеки `zmq_poll(3)`, который аналогичен по использованию системному `poll(2)`.

Интерфейс *cl-zmq-tools*, помимо разнообразного синтаксического сахара к биндингам, предоставляет два механизма для упрощения мультиплексирования:

1) **Реактор**, реализуемый поверх `zmq_poll(3)`, предоставляющий интерфейс для подписки на события `can-read`, `can-write` и `set-timeout`/`clear-timeout`.
2) **Сопрограммы** *zeromq-cc*, реализуемые с помощью модуля для CL *cl-cont* поверх реактора, описанного пунктом выше.

Конечно же, в данном разделе мы будем рассматривать второй интерфейс.

6.6.2. zeromq-cc

Логически, механизм практически эквивалентен тому, который мы спроектировали для сервера сетевой файловой системы, только с поправкой на Common Lisp + *cl-cont* вместо Scheme с нативной поддержкой продолжений.

Внутри блока `with-zmq-layout` с реактором вводится блок `with-zmq/cc`, который использует сгенерированный этим макросом реактор вокруг `zmq_poll` для построения инфраструктуры под поддержку сопрограмм.

Собственно сопрограмма вводится блоком `with-flow` в рамках `with-zmq/cc`, а диспетчер генерируется DSL-ем автоматически и работает прозрачно для пользователя. Помимо этого, интерфейс предоставляет `'/cc`-варианты функций для получения и отправки сообщений из пакета *cl-zmq-tools*.

Давайте рассмотрим, как это выглядит на примере уже упоминавшегося выше маршрутизатора **XREQ** ↔ **XREP**:

```
(defpackage :xreq-xrep-router
  (:use :cl :zmq-tools :zmq-cc))

(defun run-router (ctx)
  (with-zmq-layout
    ((xreq-sock zmq:xreq :bind "inproc://xreq")
     (xrep-sock zmq:xrep :bind "inproc://xrep"))
    (:context ctx :reactor t)

    (with-zmq/cc ()
      (with-flow
        (loop
          (send-parts/cc xrep-sock
                         (recv-parts/raw/cc xreq-sock))))
      (with-flow
        (loop
          (send-parts/cc xreq-sock
                         (recv-parts/raw/cc
                             xrep-sock)))))))
```

Выглядит предельно просто, особенно, по сравнению с вариантом, предлагаемом в официальной документации к *ZeroMQ*: rrbroker.lisp;[11] хотя функционально все то же самое. Физически это такой же цикл с конечным автоматом вокруг `zmq_poll(3)`, но логически самостоятельные блоки со своим отдельным ходом выполнения удобно оформляются в виде сопрограмм `with-flow`.

6.6.3. Классический ввод-вывод в zeromq-cc

Помимо *ZeroMQ*-сокетов, библиотечный мультиплексор `zmq_poll(3)` поддерживает и стандартные файловые дескрипторы (аналогично системному `poll(2)`), позволяя, тем самым, смешивать в одном потоке традиционный ввод-вывод с обработкой сообщений.

Это позволяет, например, создавать различного рода прокси и адаптеры между системами, из которых одни работают с использованием *ZeroMQ* в качестве коммуникационного транспорта, а другие работают с протоколами поверх стандартного TCP/IP.

Фреймворк *zeromq-cc* предоставляет набор функций, параллельный интерфейсу POSIX для сетевой коммуникации:

- **socket-accept/cc**: аналог `accept(2)`, приостанавливает сопрограмму до тех пор, пока сокет не станет доступным для чтения.
- **socket-connect/cc**: аналог `connect(2)` приостанавливает сопрограмму до тех пор, пока сокет не станет доступным для записи.
- **socket-receive/cc**: аналог `recv(2)` приостанавливает сопрограмму до тех пор, пока сокет не станет доступным для чтения.
- **socket-send/cc**: аналог `send(2)` приостанавливает сопрограмму до тех пор, пока сокет не станет доступным для записи.

В качестве иллюстрации возможностей инструмента давайте рассмотрим маленький пример из поставки *zeromq-cc*: небольшой HTTP-сервер с балансировкой нагрузки. Проект состоит из двух частей: HTTP-фронтенд и рабочие процессы, общающиеся друг с другом посредством *ZeroMQ*-сообщений. Балансировка нагрузки осуществляется с помощью схемы zmq-сокетов **XREQ** ↔ **REP**. Общий алгоритм работы следующий:

1) Фронтенд принимает HTTP-запрос от клиента.
2) Оборачивает его в *ZeroMQ*-сообщение и отсылает очередному worker-у.
3) Рабочий процесс обрабатывает запрос и формирует HTTP-ответ.

[10] См. пример из официальной документации: http://zguide.zeromq.org/page:all#A-Request-Reply-Broker.

[11] https://github.com/imatix/zguide/blob/master/examples/Common%20Lisp/rrbroker.lisp

4) Ответ оборачивается в *ZeroMQ*-сообщение и возвращается фронтенду.

5) Фронтенд пересылает ответ клиенту.

Интерфейс *zeromq-cc* позволяет в простом и понятном виде записать эту нетривиальную логику в однопоточном варианте вокруг `zmq_poll(3)`-мультиплексора.

```
(defun run-frontend (ctx &key
                     (workers-bind-addr "inproc://workers")
                     (http-port 45080))
  (with-listen-tcp (srv-sock http-port)
    (with-zmq-layout
        ((workers-sock zmq:xreq :bind workers-bind-addr))
        (:context ctx :reactor t)
      (let ((routing-table (make-hash-table :test 'equal)))
        (with-zmq/cc ()
          (with-flow
            (let ((cnt-sock (socket-accept/cc srv-sock))
                  (headers (make-receive-buffer)))
              (loop for received = (socket-receive/cc
                                    cnt-sock headers)
                    until (zerop received)
                    do
                    (when (headers-read-p headers)
                      (let* ((http-reply
                              (wait-routed-reply/cc
                                 routing-table
                                 workers-sock
                                 headers))
                             (http-reply-length
                              (length http-reply)))
                        (loop for sent = (socket-send/cc
                                          cnt-sock
                                          http-reply
                                          (or sent 0))
                              while (< sent
                                       http-reply-length))
                        (return))))
              (socket-close cnt-sock)))
          (with-flow
            (loop
               for (delimiter ident-raw reply) =
                     (recv-parts/raw/cc workers-sock)
               for ident =
                   (map 'string #'code-char ident-raw)
               do (funcall (gethash ident routing-table)
                           reply)
               do (remhash ident routing-table)))))))))
```

При использовании XREQ/XREP[12]-сокетов нам необходимо в начале пакета передавать пустой фрейм: *delimiter*, чтобы отметить конец стека адресов получателей. Так как в системе маршрутизации у нас только один узел, то *delimiter* мы просто ставим в голову очереди.

```
(defun/cc wait-routed-reply/cc (routing-table sock request)
  (let ((ident (string (gensym))))
    (send-parts/cc sock (list #() ident request))
    (call/cc (lambda (k)
               (setf (gethash ident routing-table) k)))))

(defun run-worker (ctx &key
                   (workers-bind-addr "inproc://workers"))
  (with-zmq-layout
      ((workers-sock zmq:rep :connect workers-bind-addr))
      (:context ctx)
    (loop
       for (ident headers) =
             (zmq-recv-parts/raw workers-sock t)
       for reply =
             (make-http-reply headers)
       do (zmq-send-parts workers-sock
                          (list ident reply)))))
```

(Полностью код можно найти в дистрибутиве библиотеки.[13])

Для решения проблемы соответствия приходящего в **XREQ**-сокет ответа ожидающему его клиенту применена уже известная нам по проекту **nfs.scm** техника с сохранением продолжения клиента в специальной таблице и размораживанием его оттуда как только появятся необходимые данные. Ключом в этой таблице служит автоматически генерируемый идентификатор, которым помечаются *ZeroMQ*-сообщения.

Все остальное выглядит довольно просто: две сопрограммы в блоке `with-zmq/cc` на фронтенде (одна для обработки клиентского http-соединения, вторая — для коммуникации с бэкендами) и цикл с блокирующим IO в процессах-обработчиках.

Конечно, проект не является полноценным http-сервером, но он наглядно демонстрирует возможности *zeromq-cc* в «облагораживании» смешанного ввода/вывода вокруг мультиплексора `zmq_poll(3)` в одном потоке.

6.6.4. Event-driven IO vs call/cc

Коль скоро *zeromq-tools* предоставляет два различных механизма для программирования асинхронных процессов: **reactor** и **zeromq-cc**, то было бы крайне интересно сравнить оба подхода в решении одной и той же задачи.

В примерах к дистрибутиву библиотеки[14] приведены две различных реализации простого эхо-сервера. Сервер обслуживает одновременно два сокета — один традиционный *TCP* и второй — *ZeroMQ* типа **REP**: все данные, поступающие к серверу без изменений должны быть отправлены обратно клиенту. При этом количество одновременно обслуживающихся клиентов не должно быть искусственно ограничено.

Даже несмотря на то, что в Common Lisp есть полноценные замыкания (которые, как мы уже выяснили в предыдущих разделах, крайне удобны для событийных механизмов), это не спасает ситуацию: реализация сервера, с использованием *zeromq-cc* вдвое (31 строка с комментариями) короче реализации на реакторе (64 строки с комментариями). При этом на выходе получается очень похожий код (напомню, что библиотека *zeromq-cc* реализована поверх реактора) практически идентичного быстродействия. Читатели ради интереса могут попробовать реализовать этот же демонстрационный пример на чистом *cl-zmq* вокруг `zmq_poll(3)` или даже многопоточный вариант: скорее всего, объем кода кардинально возрастет, а быстродействие не изменится (а то и вовсе деградирует).

Даже на таком синтетическом и предельно упрощенном примере отчетливо видно преимущество подхода с продолжениями над всеми остальными в программировании асинхронных процессов. В реальных (даже небольших) проектах разница в объеме кода и его стройности выходит еще ощутимей.

Однако мультиплексирование ввода/вывода и «спрямление» конечных автоматов в пользовательских протоколах —

[12] http://api.zeromq.org/master:zmq-socket

[13] http://repo.or.cz/w/cl-zmq.git/tree/refs/heads/cl-zmq-tools:/examples

[14] http://repo.or.cz/w/cl-zmq.git/tree/refs/heads/cl-zmq-tools:/examples

это далеко не единственная область применения продолжений. Пожалуй, самая известная общепризнанная практика — использование продолжений в web-фреймворках.

6.7. Продолжения в web

Разработка интерактивных веб-сайтов с динамическим содержанием — это многолетний опыт боли, страданий и унижения. В основе web лежит протокол HTTP, который с момента своего изобретения (1990-ые гг.) не претерпел никаких существенных изменений. Он синхронный, текстовый и без состояния (каждый последующий запрос независим от предыдущих). Поэтому для интерактивных сайтов клиентское состояние поддерживается различными кустарными методами, которых, в свою очередь, немного — в связи с многочисленными ограничениями HTTP. Существует всего две разновидности этих методов: хранение состояния на стороне сервера и на стороне клиента. Наиболее популярные из них следующие:

6.7.1. Client side

HTTP GET

Клиентское состояние протаскивается в строке URL запроса. Способ самый старый и надежный, у него нет проблем с кнопками *Back*/*Forward* в браузере, но обладающий огромным количеством недостатков:

- адресная строка в браузере обезображивается;
- состояние не может быть уникальным для каждого клиента (работа с учетными записями);
- много данных в URL не поместится;
- все передаваемые значения параметров находятся перед глазами;
- и т. п.

Кроме того, чтобы в сложных многостраничных интерфейсах проносить состояние в URL, все навигационные ссылки на каждой странице такого интерфейса должны выглядеть по-разному для каждой комбинации проносимых параметров. Допустим, очередной фрейм интерфейса содержит такого рода навигацию:

```
Choose your sex:<br>
<a href="?step=2&sex=m">male</a><br>
<a href="?step=2&sex=f">female</a><br>
<a href="?step=2&sex=u">other, sorry</a>
```

Несмотря на то, что все ссылки производят переход на одну и ту же страницу, скрипту передается различный контекст в URL, отчего их содержимое различается. По большому счету, это не так страшно, однако какой-нибудь поисковый паук может быть крепко сбит с толку (особенно если передаваемые параметры не влияют на отображение следующего фрейма интерфейса).

HTTP POST

Данные протаскиваются в POST области HTTP-запроса. Классическим образом в браузере POST-запрос можно сформировать только из HTML-формы, однако с помощью *javascript* формы на странице можно замаскировать под обычные гиперссылки. В отличие от GET-запроса, в POST можно передавать большой объем информации, но реакция на *Back*/*Forward* навигацию уже на порядок хуже: только современные браузеры могут корректно обрабатывать POST *Back*/*Forward* длиной в один шаг, и далеко не все могут полноценно работать со стеком истории переходов.

Тем не менее, с помощью форм и сгенерированного для них набора `<hidden>` полей намного удобней организовывать многостраничные анкеты, требующие от пользователя ввода большого количества информации. При этом строка URL не загромождается никакими лишними данными. Поэтому, например, интерфейсы многих интернет-банкингов построены исключительно на POST-переходах, даже для простейших диалогов вроде «Вы уверены? Да/Нет». Такие банкинги несложно узнать: при нажатии на кнопку *Back* браузер все время переспрашивает, действительно ли вы хотите еще раз осуществить POST-запрос. Иногда, кстати, подобное действие вызывает исключение в серверном программном обеспечении, не ожидающем повторного получения данных из предыдущей формы.

Следует отметить, что существуют фреймворки, которые в данном направлении зашли еще дальше допотопных интернет-банкингов. Это, например, ASP, который не гнушается сериализовать собственное состояние в скрытое поле POST-формы. При этом это состояние (особенно после Base64-кодирования) может достигать нескольких мегабайт, что порядком озадачивает обладателей доступа в интернет с узким исходящим каналом или помегабайтной тарификацией.

Browser cookies

Практически все современные браузеры позволяют серверному приложению сохранить небольшое количество текстовых данных на локальной машине и получить их затем обратно при следующем обращении. По сути, это такой же набор параметров вида *поле=значение*, как и в POST-запросе, но автоматически передающийся браузером каждый раз, когда осуществляется доступ к этой же странице.

По большому счету, никаких особых минусов в этой технологии нет, за исключением ограничений на размер и количество кук. Тем не менее, официального ограничения в четыре килобайта на куку зачастую недостаточно для сложных серверных приложений, особенно учитывая текстовую природу сохраняемых данных. Например, картографические сервисы (треки, маршруты и т. п.), разнообразные редакторы (особенно мультимедиа-редакторы), многопользовательские интерактивные сервисы и т. д.

6.7.2. Server side

В отличие от хранения данных контекста на стороне клиента, на сервере можно поддерживать данные любого размера и любой природы, вплоть до ресуров, типа открытых файловых дескрипторов. Соответствие сохраненных на сервере контекстов каждому клиенту обычно делается по специальному ключу: сгенерированной при первом обращении сессии.

В зависимости от выбранной архитектуры для генерации динамического содержимого веб-страницы, контексты сохраняются различными способами. Для классического подхода «один запрос — одна интерпретация скрипта» (CGI, стандартный PHP) данные приходится куда-нибудь сериализовать во внешнее хранилище, например, в базу данных или memcached. В случае с FastCGI или сервером приложений контекст можно хранить прямо в памяти, обращаясь к нему по ключу сессии.

Разумеется, поддержка клиентского контекста на стороне сервера намного активней потребляет ресурсы машины, нежели когда контекстные данные носит сам клиент, но за счет своего удобства и гибкости, этот подход весьма широко распространен. Как минимум, авторизация клиентов практически во всех веб-фреймворках сделана через пользовательские сессии.

6.7.3. Абстрагирование автомата через продолжение

Информация, изложенная во вводной части, подводит нас к хорошо знакомому нам паттерну применения продолжений: схлопывание конечного автомата в линейный алгоритм.

И действительно, вся классическая схема программирования под веб, благодаря примитивности протокола HTTP, это бесконечная имплементация частных состояний автомата и каша из обработок тонны флагов:

- Залогинен ли клиент?
- По каким столбцам выставлена сортировка таблицы?
- Какой раздел / пункт меню выбран в данный момент?
- Какой язык интерфейса выставлен?
- Каков статус фоновой задачи (например, готовы ли результаты для тяжелого поискового запроса)?
- И т. д.

Серверный код либо превращается в одно огромное дерево `if`-ов, либо используется какой-нибудь веб-фреймворк или библиотека, несколько упрощающая жизнь. Однако полностью абстрагировать подобный автомат, как мы уже выяснили в предыдущих разделах, под силу только продолжениям.

В самом деле, только с их помощью становится возможным писать серверную часть в линейном стиле, вроде:

```
(defun/cc sell-car ()
  (when (ask-yes/no "Do you want to buy a car?")
    (loop
       with all-cars = (load-all-cars)
       for cars = (reduce (lambda (cars proc)
                            (funcall proc cars))
                          (list #'choose-car-manufacturer
                                #'choose-car-model
                                #'choose-car-year
                                #'choose-car-color)
                          :initial-value all-cars)
       do
         (loop for car in cars
           when
             (ask-yes/no "Do you want to buy this car: ~a?"
                         car)
           do (return-from sell-car car)))))
```

Это нечто вроде экспертной системы по подбору автомобиля, но реализованной в совершенно стандартном Common Lisp-синтаксисе, подобно какому-нибудь консольному диалогу. За кадром остается вся HTTP-машинерия: запросы и ответы, обработка заголовков, разбор параметров форм и так далее.

Абстрагировав логику приложения от автомата, далее можно без проблем применять другие паттерны проектирования, например, MVC.

6.7.4. Использование на практике

Несмотря на то, что оформить автомат из серверного приложения, написанного в классическом стиле, нетрудно и собственноручно (используя технику, рассмотренную в предыдущих разделах), в мире существует некоторое количество уже готовых относительно популярных веб-фреймворков, основанных на продолжениях. В основном, такие фреймворки реализованы для языков, нативно поддерживающих продолжения (Scheme, Smalltalk), либо для языков, имеющих модули для их прозрачного добавления (Common Lisp).

- **Seaside**[15] (Smalltalk)
- **UncommonWeb**[16] (Common Lisp)
- **Weblocks**[17] (Common Lisp)
- **Racket web server**[18] (Racket scheme)

Далее в статье будет рассмотрен демонстрационный проект с использованием фреймворка **Weblocks** для Common Lisp.

6.8. Практика #3: weblocks chat

6.8.1. О Weblocks

Технически *Weblocks* — это библиотека-надстройка над популярным Common Lisp веб-сервером *Hunchentoot*.

Философия веб-фреймворка **Weblocks** строится вокруг трех основных механизмов: продолжения (control flow), виджеты (widgets) и модель «отображение + хранилище» (view, store). Несмотря на то, что все эти механизмы тесно связаны (реализованы друг через друга) и могут продуктивно использоваться все вместе, в некоторых моментах паттерны использования получаются несколько ортогональными и дублирующимися. Однако фреймворк симметрично развивается во всех направлениях, ничем не ограничивая пользователя, который имеет возможность применять только один или два поддерживаемых механизма из трех в своем проекте, не испытывая при этом никаких неудобств.

Давайте подробней рассмотрим эти механизмы.

Control flow: сложная логика в несколько строк

При самом первом обращении клиента *Weblocks* генерирует для него отдельную сессию. На одну такую клиентскую сессию запускается один поток управления, отображающий некоторую иерархию *виджетов* (виджеты рассматриваются в следующем разделе). Каждый из таких виджетов поддерживает собственные потоки управления, продолжения из которых (захваченные через `call/cc`) могут сохраняться и «размораживаться» при последующем обращении пользователя к серверу.

Здесь применяется техника, похожая на уже хорошо известную нам по рассмотренным выше проектам — «диспетчер + несколько сопрограмм». Диспетчером в данном примере оказывается *Weblock*'овский контроллер *Hunchentoot*, который берет на себя всю черную работу по управлению сессиями, пользовательскими продолжениями и механизмом их устаревания (garbage collection). Сопрограммы стартуют при первом обращении клиента (продолжение корневого виджета *root*), а их точка входа (entrypoint) задается при декларации веб-приложения `defwebapp` ключом **:init-user-session**.

Схематическую иллюстрацию можно посмотреть на диаграмме 6.5.

Как мы уже убедились в разделах выше, сопрограмма позволяет организовывать асинхронный по своей натуре код в

[15] http://seaside.st/
[16] http://common-lisp.net/project/ucw/
[17] http://weblocks.viridian-project.de/
[18] http://racket-lang.org/

совершенно естественном стиле. Благодаря этому в *Weblocks* возможны следующие фокусы с элементарными вещами, которые, тем не менее, в традиционном веб-программировании привносят массу дискомфорта, связанного с необходимостью написания большого количества кода в разных местах проекта:

- **Подтверждения и уведомления**. Никаких флагов и редиректов, совершенно стандартный код:

```
(when (eq (do-confirmation "Are you sure?" :type
     :yes/no) :yes)
  (remove-item ...))
```

- **Организация логики UI по аналогии с классическим десктопным** — используя те же самые паттерны, включая подобие делегатов и т. п.:

```
(render-link
  (lambda/cc (&rest args)
    (do-information "Now something bad will happen!"))
    "Please don't click this link")
```

- **Вся бизнес-логика проекта в явном виде**:

```
(loop
  (let ((user (yield (do-login))))
    (cond ((admin-p user) (show-admin-panel))
          (user (show-user-data))
          (t (do-information "Restricted area,
               sorry")))))
```

- **Многостраничные формы**: визарды, экспертные системы, пользовательские анкеты и так далее:

```
(when (and (yield (show-eula-0))
           (yield (show-eula-1))
           (yield (show-eula-2)))
  (show-requested-data))
```

И многое другое.

Виджеты: строительные блоки

Основная единица в *Weblocks* для отображения (рендеринга) информации в браузер — это *виджет* (widget). Пользовательский интерфейс предполагается составлять (в том числе, иерархически) из виджетов как из строительных блоков (само название фреймворка *Weblocks*: **web** + **blocks** намекает на это).

Фреймворк при старте клиентской сопрограммы создает один корневой виджет `root`, на котором далее предполагается размещать пользовательские виджеты. Виджетом в *Weblocks* может считаться:

- Обычная Common Lisp строка:

```
(render-widget "I am a widget too")
```

  После рендеринга в HTML виджет-строка помещается в контейнер `<div>`, который получает CSS-классы *widget* и *string*.

- Функция или замыкание, способная служить аргументом к вызову `call/cc`. Например:

```
(render-widget
  (lambda (k)
    (render-widget "Are you sure?")
    (render-link (lambda (&rest args) (answer k t))
         "yes")
    (render-link (lambda (&rest args) (answer k nil))
         "no")))
```

  При рендеринге такого объекта создается `<div>`-контейнер CSS-класса *widget*, внутри которого окажутся все элементы, отрендеренные после выполнения замыкания.

- Объект-потомок класса `weblocks:widget`, охарактеризованный метаклассом `weblocks:widget-class`. Такие объекты можно создавать макросом `defwidget` и задавать им свои правила рендеринга, задавая соответствующую специализацию методу `render-widget-body`:

```
(defwidget show-hide ()
  ((show-p :initform nil
           :accessor show-p)))

(defmethod render-widget-body
    ((widget show-hide) &key &allow-other-keys)
  (if (show-p widget)
      (progn
        (render-widget "Hidden string!")
        (render-link (lambda (&rest args)
                       (setf (show-p widget) nil))
                     "hide again"))
      (render-link (lambda (&rest args)
                     (setf (show-p widget) t))
                   "show hidden string")))
```

  Рендеринг такого класса обязательно дает `<div>`-контейнер, получающий CSS-класс с таким же именем, как и CLOS-класс в программе, а также рекурсивно все родительские классы. Например, рендеринг класса `show-hide` даст следующий контейнер:

```
<div class="show-hide widget">...</div>
```

Как уже видно из примеров, при рендеринге виджетов не приветствуется явное задание HTML-кода, вместо этого дизайнеру или верстальщику предлагается активно и плотно использовать CSS для размещения контента на странице. Благодаря тому, что вся страница состоит из виджетов, а все виджеты рендерятся в *div*-ы с корректной автоматической простановкой классов, верстать вручную сырую HTML-разметку нет необходимости. Тем не менее, эта возможность присуствует в фреймворке, для этого достаточно в методах рендеринга воспользоваться возможностями модуля *CL-WHO*:

```
(render-widget
  (lambda (k)
    (with-html
      (:h1 (str "Page title"))
      (:hr)
      (:p (str "Some text")))))
```

Или даже отсылая «сырой» HTML в стандартный поток вывода, который будет перенаправлен фреймворком *Weblocks* в браузер.

Объединяя виджеты и продолжения, мы получаем возможность невероятно быстро и легко создавать веб-интерфейсы, применяя обычные практики реализации десктопного GUI: показывать/прятать виджеты, отображать их в цикле, строить сложные иерархические меню и так далее.

Фреймворк по возможности активно использует *AJAX* — для частичного обновления интерфейса, но, тем не менее, не требует обязательной поддержки JavaScript'а на клиенте. Наличие работающего JavaScript'а проверяется при инициализации пользовательской сессии. При его наличии, обновление DOM-дерева происходит инкрементально, а при отсутствии, выполняется полная перерисовка страницы после каждого запроса.

Ни строчки html: модель view / store

Для реализации сложных enterprise-проектов в фреймворке существует еще один слой абстракции над виджетами и продолжениями. По аналогии с широко-известным паттерном **MVC** построена своя собственная конструкция, определяющая специальные инструменты, API и DSL для быстрой сборки веб-приложений со сложной логикой.

- **Store** — это интерфейс для абстракции источника данных. В библиотеке уже присутствуют реализации для следующих бэкендов:
  - Elephant (CLOS ORM для Common Lisp со своими собственными бэкендами)
  - CLSQL (модуль для доступа к базам данных)
  - Postmodern (биндинг к postgresql)
  - CL-PREVALENCE (персистентное in-memory хранилище)
  - Memory (обычное хранилище в памяти, которое сбрасывается при старте)
- **View** — это DSL для отображения данных в браузере. Задает правила, по которым для записи *Store* будут сгенерированы формы для ввода, редактирования и виджеты для просмотра (отдельно, или в составе коллекции). В составе библиотеки находится некоторое количество инструментов для упрощения этой задачи, в том числе механизм *scaffold*, генерирующий view или formview автоматически по описанию класса CLOS, пользуясь метаобъектным протоколом в Common Lisp.
- **Presentation** — DSL для описания алгоритма, по которому информация запрашивается у пользователя и, затем, превращается в данные проекта. В *Weblocks*, в том числе, предлагаются следующие типы:
  - password: данные вводятся вслепую, на выходе — текст.
  - file-upload: пользователь указывает файл, на выходе — его содержимое на сервере.
  - date: дата и время.
  - url: ссылка.
  - и многое другое...

Дизайн системы направлен на то, чтобы программист сосредотачивался на описании объектной области (модели) задачи: традиционное ООП с CLOS и логикой на Common Lisp. А персистентность и веб-интерфейс должны как бы доставаться бесплатно — генерироваться, по возможности, автоматически, с минимальным количеством ручного кода.

В реальной жизни все, разумеется, несколько сложнее, чем в теории. В любом случае, модель «View + Store» в *Weblocks* подробно в данном описании рассматриваться не будет, так как она достаточно сложна сама по себе, и нормальное ее описание претендует на отдельную статью.

Кроме того, для использования данной модели программист вообще может и не подозревать о продолжениях и прочих технических деталях реализации.

6.8.2. Проект: weblocks-chat

Описание

Давайте попробуем с помощью *Weblocks* реализовать некий небольшой, но полностью работоспособный проект, который явно продемонстрирует выгоду от использования веб-фреймворков, основанных на продолжениях и от *Weblocks* в частности.

В качестве такого проекта будет создан интерактивный веб-чат со следующим ТЗ:

1) Клиент перед входом в чат должен ввести свой никнейм. При этом в пределах чата все никнеймы обязаны быть уникальными.
2) Залогиненый клиент должен периодически либо что-то говорить, либо иным образом уведомлять сервер о том, что он «жив». При отсутствии такого пинга в течении некоторого времени (одна минута) пользователь должен быть разлогинен и выброшен из чата.
3) Пользователь должен иметь возможность произносить фразы, которые будут видны всем (broadcast) или только указанным собеседникам (multicast).
4) В архитектуру проекта должна быть заложена основа для возможного горизонтального масштабирования системы в будущем.
5) Интерфейс пользователя должен быть максимально интерактивным, с активным использованием AJAX. Для получения новых сообщений от сервера должна быть выбрана технология, подразумевающая минимальное время реакции, например, *HTTP Long polling* (*Comet*).

Конечно же, вышеприведенные требования были некоторым образом «подогнаны» под демонстрационный пример, однако они весьма приближены к реальным требованиям, которые встречаются в реальных проектах.

Архитектура

Сразу два пункта ТЗ мы будем одновременно закрывать с помощью уже знакомой нам библиотеки *ZeroMQ*. Заодно, кстати, еще раз будут продемонстрированы «в бою» продолжения из *zeromq-cc*.

Для реализации broadcast и multicast каналов сообщений будут использованы *ZeroMQ*-сокеты **publish** и **subscribe**. Для получения реплик собеседников, клиент с никнеймом *user* подписывается на два канала:

- **msg.all.**, для получения «широковещательных реплик», предназначенных всем пользователям одновременно.

- **msg.user.**, для получения «приватных» реплик, предназначенных только для глаз пользователя *user*.

Соответственно, обратный процесс будет выглядеть параллельно:

- Если *user* собирается сказать реплику всем пользователям одновременно, он публикует фразу в канал **msg.all.**.
- Если *user* разговаривает с одним или несколькими собеседниками, сообщения уходят в соответствующие их никнеймам списки рассылки.

Общая архитектура будет разнесена на развязаные по функционалу модули (в угоду возможному будущему масштабированию) и будет выглядеть следующим образом:

- **Ядро системы**: автономный модуль, отвечающий за глобальное состояние чата. Он хранит список текущих пользователей, принимает от них «пинги» и разлогинивает их по истечении некоторого периода бездействия.
- **Веб-фронтенд**: собственно *Weblocks*-приложение, отрисовывающее пользовательское GUI.

Модули проекта общаются между собой исключительно с помощью *ZeroMQ*-сообщений. Поэтому веб-фронтендов может быть несколько, и при этом они будут взаимонезависимы.

Core

Давайте на первом этапе займемся той частью проекта, которая пока никак не связана с интерфейсом: ядром системы. Формализуем алгоритм, по которому будет работать чат:

- Модуль обслуживает три *ZeroMQ*-сокета в режиме сервера (bind):
  1) **Pub**: используется для публикации сообщений в различные каналы подписок.
  2) **Sub**: используется для ретрансляции пользовательских сообщений обратно в **Pub** (паттерн, аналогичный `zmq_forwarder`).
  3) **Rep**: служебный сокет, второе звено в паттерне Request/Reply, используется для получения пользовательских запросов.

  Этот макет можно, не откладывая, переносить в код на Common Lisp:

```
(defun start-core ()
  (with-zmq-layout
    ((sock-pub zmq:pub :bind *zmq-pub-addr*)
     (sock-sub zmq:sub :bind *zmq-sub-addr*
                       :option (zmq:subscribe "msg."))
     (sock-usr zmq:rep :bind *zmq-usr-addr*))
    (:context *zmq-context* :reactor t)
    (with-zmq/cc () ...)
```

  Сразу же можно организовать ретрансляцию сообщений из `sock-sub` в `sock-pub`, как это уже было рассмотрено в разделе про *zeromq-cc*:

```
;; Route user messages
(with-flow
  (iter (send-parts/cc sock-pub (recv-parts/raw/cc
       sock-sub))))
```

- На служебном **Rep**-сокете модуль ожидает пакет-запрос вида `(user cmd)`, где *user* — это имя пользователя, а *cmd* — команда серверу, одна из:
  - **login**: осуществить попытку залогиниться под никнеймом *user*. Если попытка успешная (должна соблюдаться уникальность никнеймов в пределах чата), то реакция должна быть следующая:
    1) Вернуть ответ «*accept*» и текущий список залогиненых пользователей.
    2) В список рассылки **sys.login.** опубликовать никнейм нового, только что залогиненого пользователя.

    Если попытка неуспешная, то следует вернуть ответ «*reject*».
  - **ping**: напомнить серверу о своем существовании. Если в течение продолжительного времени *core*-модуль не получает от клиента *ping*-а, пользователь будет разлогинен.
  - **exit**: служебная команда, заставляющая модуль выполнить выход из бесконечного цикла обработки сообщений и завершить работу.

Команды **ping** и **exit** реализованы тривиально и не представляют особого интереса (их код можно посмотреть в файле *core.lisp*), а вот в реализации обработки **login** применена любопытная идея, которая возможна только при использовании продолжений и сопрограмм.

Сразу после успешного логина клиента создается и запускается сопрограмма, не выполняющая ничего, кроме как `sleep/cc` (адаптированый под продолжения вариант системного `sleep(3)`) в цикле. Время сна на каждой итерации равно таймауту, после которого следует разлогинить пользователя, если от него не было получено ни одного *ping*-а. Соответственно, в конце итерации данный факт проверяется и, если клиент действительно «протух», то он будет выброшен из чата.

Данное решение напоминает одновременно и многопоточное программирование, и подход, практикующийся в Erlang'е с порождением процесса на любую, сколь угодно малую задачу. Но только без многоуровневых конструкций для синхронизации или IPC.

```
(case-string cmd
  ("login"
   (if (gethash user users)
       (send-parts/cc sock-usr (reject-user-pkt))
       (progn
         (send-parts/cc sock-usr (accept-user-pkt user))
         (send-parts/cc sock-pub (list "sys.login."
              user))
         ;; Look after expired logins
         (without-call/cc
          (let ((user user))
           (with-flow
             (iter
               (for elapsed = (- (get-universal-time)
                                 (gethash user users)))
               (until (>= elapsed
                    *inactivity-timeout-sec*))
               (sleep/cc (* (- *inactivity-timeout-sec*
```

```
                        elapsed) 1000000)))
    (send-parts/cc sock-pub
                   (list "sys.expired." user))
    (remhash user users))))))))
```

В данном примере для организации вложенных сопрограмм используется небольшой трюк с *CPS*-трансформацией кода, которую производит библиотека `cl-cont`. Если в примере с nfs-сервером нам пришлось реализовывать стек продолжений, который позволял из вложенной сопрограммы возвращаться в корректное место (на уровень выше), то здесь я просто в нужном месте (старт другой сопрограммы в рамках активной) просто «выключаю» преобразование кода в *Continuation Passing Style* декларацией блока `without-call-cc`. Тем самым, внутренняя сопрограмма стартуется как бы в «глобальном контексте».

Конструкция `(let ((user user))...)` используется в данном примере для копирования переменной, потому что замыкание не производит полного клонирования, а значение будет перезаписано на другой итерации бесконечного цикла по обработке пакетов (издержки императивного программирования).

О разлогиненом пользователе модуль сообщает в список рассылки **sys.expired.**, чтобы клиентские интерфейсы смогли отреагировать на данное событие (убрать пользователя из контакт-листа).

Вся реализация алгоритма (местами нетривиального) уместилась в 50 строк кода благодаря использованию библиотеки *zeromq-cc*, с преимуществами которой мы познакомились в предыдущем разделе. Дальнейшее усовершенствование модуля может включать в себя, например, использование каких-либо персистентных хранилищ (баз данных) вместо хэш-таблицы для манипуляции пользователями, введение механизма аутентификации, горизонтальное масштабирование при помощи системы маршрутизаторов *ZeroMQ*-сообщений и т. п. — все это достаточно тривиально реализуется в рамках выбраной архитектуры (сопрограммы вокруг мультиплексора).

Давайте теперь перейдем к веб-интерфейсу.

Web

Weblocks работает в режиме сервера приложений, обслуживая один или несколько независимых вебсайтов. Декларация нового проекта осуществляется при помощи специального DSL `defwebapp`. В нашем проекте будет использоваться только один сайт, поэтому он будет размещен в корне домена:

```
(defwebapp weblocks-chat
    :prefix "/"
    :description "Comet webchat for weblocks"
    :init-user-session 'init-user-session
    :autostart nil
    :ignore-default-dependencies nil
    :public-files-path (compute-public-files-path
                        :weblocks-chat "static")
    :dependencies '((:stylesheet "weblocks-chat"))
    :debug t)
```

Более подробно:

- **:prefix** задает префиксную часть URL-адреса, по которому weblocks будет обслуживать данный проект.
- **:description** задает `<title>` страницы.
- В **:init-user-session** должна быть указана функция, которая будет служить точкой входа для сопрограммы, обслуживающей клиента.
- **:public-files-path** указывает, по какому пути фреймворку следует искать статические файлы.
- **:dependencies** определяет дополнительные статические файлы, которые должны быть подключены при сборке страницы. В нашем случае это файл стилей для проекта *weblocks-chat.css*.

Так как сайт у нас всего один, стартовать и останавливаться он будет вместе с сервером приложений *Weblocks*, принимая соединения на порту 18080:

```
(defun start-weblocks-chat ()
  (start-weblocks :port 18080 :debug t)
  (start-webapp 'weblocks-chat))

(defun stop-weblocks-chat ()
  (stop-webapp 'weblocks-chat)
  (stop-weblocks))
```

На этом служебный код заканчивается и далее нам следует приступить к собственно программированию интерфейса и логики проекта.

Что касается бизнес-логики, она в нашем случае тривиальна: пользователь должен залогиниться под уникальным никнеймом и после этого начать общение. В случае, если он был разлогинен по причине длительного бездействия, процесс должен быть повторен заново.

Благодаря веб-фреймворку, основанному на продолжениях, мы имеем возможность ровно в этих терминах и записать алгоритм, не городя никаких второстепенных сущностей, вроде конечных автоматов:

```
(defun init-user-session (root)
  (weblocks:with-flow
    root
    (loop (let ((creds (yield (make-instance 'auth))))
            (yield (apply #'make-ui creds))))))
```

Синтаксис `yield` отмечает в коде конструкции, на которых будет происходить неявный разрыв в ходе выполнения программы: сначала будет отображен виджет `auth`, а затем, когда управление будет им возвращено обратно в блок `with-flow` при помощи `answer`, пользователю будет показываться виджет, созданный вызовом `make-ui`. Выход же из основного интерфейса вызовет очередную итерацию бесконечного цикла, заново передав управление на форму логина.

Если начать разбирать детали реализации `with-flow` с `yield` и `answer`, мы практитески сразу столкнемся с хорошо известными нам продолжениями из *cl-cont*. Вкратце здесь происходит следующее:

- `with-flow` объявляет блок `with-call/cc`.
- `yield` раскрывается в вызов `do-widget`, который, по сути:
  1) выполняет `call/cc` с сохранением в активном виджете текущего продолжения.

2) запускает цикл рендеринга виджета с помощью мультиметода `render-widget`.

- `answer` где-то внутри логики `render-widget` производит «разморозку» сохраненного в предыдущем пункте продолжения, передавая управление дальше в блоке `with-flow`.

Если внимательно посмотреть на `with-flow`, то можно заметить, что в *Weblocks* сопрограммы организованы иерархически: `with-flow` можно определять с любым уровнем вложенности (указывая родительскую сопрограмму), при этом `answer` будет возвращать управление на соответствующий ему блок. В нашем же случае в точке входа `init-user-session` мы организуем одну сопрограмму под корневым виджетом `root`, который попал к нам из параметров.

Давайте перейдем к аутентификации пользователя. Нам нужен один простой виджет, на котором необходимо разместить HTML-форму с единственным полем: никнеймом.

```
(defwidget auth ()
  ((error-msg :initform (make-instance 'flash)
              :reader error-msg)))

(defmethod render-widget-body
    ((widget auth) &key &allow-other-keys)
  (with-html
    (:hr)
    (:h3 (str "Please choose your name"))
    (with-html-form (:get (process-login widget))
      (render-input-field :text :username "")
      (render-input-field :submit :submit "Login")))
  (render-widget (error-msg widget)))
```

Здесь у `auth` в полях определен специальный flash-виджет (всплывающее сообщение, исчезающее через некоторое время), который будет выскакивать, когда надо будет просигнализировать о какой-нибудь ошибке. А в `render-widget-body` используется DSL из модуля *cl-who* и утилитарные функции из *Weblocks*. Обратите внимание, как реализуется реакция на пользовательские действия: это обычное замыкание!

Замыкание, созданное функцией `process-login`, будет вызвано фреймворком в тот момент, когда пользователь нажмет кнопку *Submit*, при этом все значения полей будут переданы в аргументах при вызове. Вообще, это крайне удобное решение в *Weblocks* используется повсеместно: замыкание из кода автоматически преобразуется в т. н. **action** — оно помещается в глобальную таблицу под свежесгенерированным идентификатором, для которого строится URL и соответствующие ему маршруты для *Hunchentoot*. Для программиста все эти процессы совершенно прозрачны, и конечный код практически лишен всяческих побочных сущностей, не относящихся к бизнес-логике. Функциями и замыканиями можно задавать:

- Реакцию на клик по ссылке:

```
(render-link (lambda (&rest args)
               (setf (show-p widget) nil))
             "hide again")
```

- Получение данных из формы:

```
(with-html-form (:get (lambda (&rest args)
                        (do-information
                         (format nil "Hello, ~a"
                                 (getf args
                                       :username)))))
  (render-input-field :text :username "")
  (render-input-field :submit :submit "Login")))
```

- Формирование подсказок по мере набора текста:

```
(render-suggest
 :find
 (lambda (prefix)
   (remove-if-not (lambda (item)
                    (string-starts-with prefix item))
                  *items*)))
```

- Формирование результатов (например, поиска) «на лету» по мере набора текста:

```
(render-isearch
 :find
 (lambda (name query)
   (setf (widget-children my-widget)
         (remove-if-not (curry #'search query)
                        *items*))))
```

И многое другое!

Функция `process-login` осуществляет `login`-запрос к модулю **core** и, в зависимости от его ответа, либо высвечивает ошибку с помощью flash-виджета, либо, при помощи `answer`, возвращается из ближайшего сохраненного продолжения, созданного через `with-flow+yield` (которое у нас было в `init-user-session`).

```
(defun process-login (widget)
  (lambda (&rest args)
    (let ((username (getf args :username)))
      (if (or (not username) (zerop (length username)))
          (flash-message (error-msg widget)
                         (format nil "Username not given"))
          (let ((users-online (try-login username)))
            (if users-online
                (answer widget (cons username users-online))
                (flash-message
                  (error-msg widget)
                  (format nil "Username '~a' is already in use"
                          username)))))))))
```

После того, как клиентом была осуществлена авторизация, мы должны уже иметь имя пользователя и список всех остальных доступных пользователей. Подошло время показывать основной пользовательский интерфейс чата. Макет веб-страницы незамысловатый (см. диаграмму 6.6) и состоит из трех основных виджетов, на которых расположены собственно элементы интерфейса (`online`, `history` и `speak`), и трех вспомогательных, которые нужны только для облегчения верстки при помощи CSS (`ui`, `users-panel` и `chat-panel`).

Давайте посмотрим декларацию всех этих виджетов:

```
(defwidget chat ()
  ((username :initarg :username
             :reader username)))

(defwidget history (chat)
  ((items :initform (make-array 0 :fill-pointer 0)
          :reader items)))

(defwidget online (chat)
```

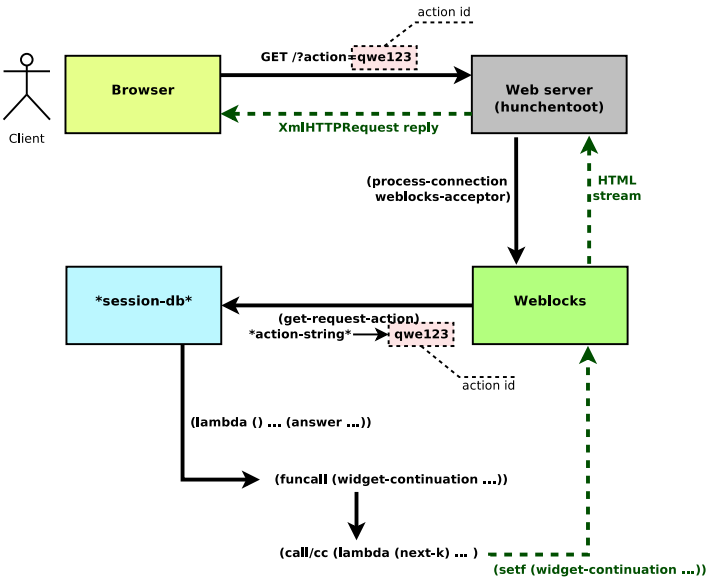


Рис. 6.5. Передача управления при HTTP запросе в Weblocks

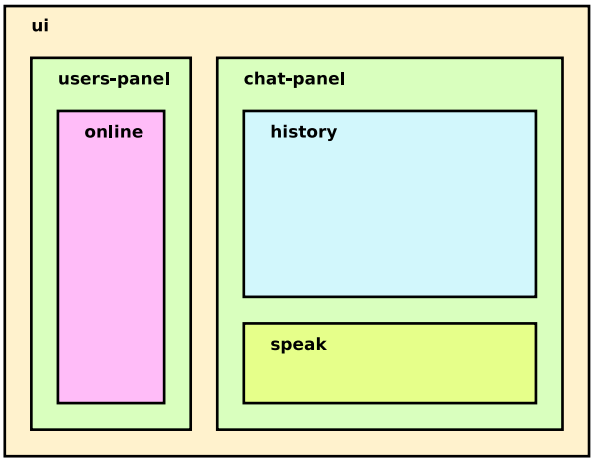


Рис. 6.6. Макет основного интерфейса чата

```
  ((users :initform (make-hash-table :test 'equal)
          :reader users)
   (speak-widget :initarg :speak-widget
                 :reader speak-widget)))

(defwidget speak (chat)
  ((targets :initform (make-hash-table :test 'equal)
            :reader targets)))

(defwidget ui () ())
(defwidget users-panel () ())
(defwidget chat-panel () ())
```

И на функцию `make-ui`, которая собирает макет:

```
(defun make-ui (username &rest users-online)
  (let*
     ((history-wgt
        (make-instance 'history :username username))
      (speak-wgt
        (make-instance 'speak :username username))
      (online-wgt
        (make-instance 'online
                       :username username
                       :speak-wgt speak-wgt))
      (ui
        (make-instance
         'ui
         :children
           (list
             (make-instance 'users-panel
                            :children (list online-wgt))
             (make-instance 'chat-panel
                            :children (list history-wgt
                                            speak-wgt))))))
    (map nil
         (curry #'user-online online-wgt history-wgt)
         users-online)
    (send-script
      (make-comet-script
        (lambda ()
          (comet-handler username
                         history-wgt
                         online-wgt
                         speak-wgt
                         (lambda () (answer ui))))))
    ui))
```

Как видно из кода, виджеты `ui`, `users-panel` и `chat-panel` используются только для группировки и в дальнейшем нужны лишь для того, чтобы соответствующий `<div>` на веб-странице получил отдельный CSS-класс, используя который, можно настроить вид элемента и его размещение на экране. Основные же элементы интерфейса имеют следующие свойства:

- Все они должны знать никнейм пользователя.
- Виджет **history** поддерживает список всех произнесенных в чате фраз. Каждая фраза — это тоже виджет, его класс будет описан чуть ниже.
- Виджет **online**, помимо собственно таблицы текущих залогиненых пользователей, имеет ссылку на виджет **speak** — это ему нужно для того, чтобы иметь возможность выбирать из контакт-листа собеседников для *multicast*-общения.

- Виджет **speak** поддерживает список текущих собеседников. Если он пуст, то считается, что фразы произносятся для всех пользователей чата (*broadcast*).

Что касается произнесенных в чате фраз, то они бывают нескольких типов, и отображаться должны по-разному, чтобы пользователь мог их отличать друг от друга в окне `history`:

- Фразы, произнесенные самим пользователем.
- Широковещательные фразы для всех: *broadcast*.
- Приватные фразы: *multicast*.
- Системные сообщения, например, о новых пользователях в чате.

```
(defwidget history-item ()
  ((date :initform (get-universal-time)
         :reader date)
   (from :initarg :from
         :reader from)
   (text :initarg :text
         :reader text)))

(defwidget history-item-mine (history-item) ())
(defwidget history-item-private (history-item) ())
(defwidget history-item-shared (history-item) ())
(defwidget history-item-system (history-item) ())
```

Как видно из кода, наследники `history-item` не несут никакой дополнительной информации и не имеют отличающегося функционала (специализаций методов). Их основной смысл заключается в том, чтобы *Weblocks* при генерации `<div>`-ов отметил их отдельным CSS-классом, при помощи которого можно настроить отображение конкретного типа фразы.

Давайте ознакомимся с правилами рендеринга каждого из виджетов.

```
(defmethod render-widget-body
    ((widget history) &key &allow-other-keys)
  (iter (for item in-vector (items widget))
        (render-widget item)))
```

Отображение виджета для журнала сообщений устроено предельно просто: метод в цикле отображает последовательно все фразы из массива `items`, каждая из которых, в свою очередь, является самостоятельным виджетом `history-item` и имеет свою собственную специализацию метода `render-widget-body`:

```
(defmethod render-widget-body
    ((widget history-item) &key &allow-other-keys)
  (with-html
    (:pre
      (str (format-date "%d.%m.%y %H:%M:%S" (date widget)))
      (:b (str (format nil " &lt;~a&gt;: " (from widget))))
      (str (text widget)))))
```

Метод отрисовывает все значимые поля класса `history-item`: дату, автора и текст фразы. Все наследники класса `history-item`: `-mine`, `-shared`, `-private` и `-system` пользуются этой специализацией метода (так как не имеют своих), однако отрисовываемые контейнеры `<div>` получают разные CSS-классы, что позволяет индивидуально настроить их отображение в браузере.

```
(defmethod render-widget-body
    ((widget online) &key &allow-other-keys)
  (with-html
    (:h1 (str (format nil "Currently online: ~a"
                      (hash-table-count (users widget)))))
    (render-list
      (iter (for (user v) in-hashtable (users widget))
            (collect user))
      :render-fn
         (lambda (user)
           (render-link
             (lambda (&rest args)
               (declare (ignore args))
               (setf
                 (gethash user
                          (targets (speak-widget widget)))
                 t)
               (mark-dirty (speak-widget widget)))
             user)))))
```

Рендеринг виджета контакт-листа `online` немного посложнее: помимо отображения списка пользователей онлайн следует предоставить возможность текущему пользователю выбирать себе собеседников при помощи клика по никнейму из списка. Соответственно, каждый никнейм представляет собой HTML-ссылку, обработчик которой на стороне *Weblocks* добавляет выбранного собесебника в список, который поддерживается виджетом `speak`.

```
(defmethod render-widget-body
    ((widget speak) &key &allow-other-keys)
  (with-html
    (:h1 (str "Speak"))
    (:h2 (str "to: ")
         (if (zerop (hash-table-count (targets widget)))
             (str "all")
             (progn
               (str (format nil "~{~a~^, ~} "
                    (iter
                      (for (target v)
                           in-hashtable (targets widget))
                      (collect target))))
               (render-link (lambda (&rest args)
                              (declare (ignore args))
                              (clrhash (targets widget))
                              (mark-dirty widget))
                            "[clear]"))))
    (let ((subscribes
           (if (zerop (hash-table-count (targets widget)))
               (list "msg.all.")
               (iter (for (target v)
                          in-hashtable (targets widget))
                     (collect
                       (format nil "msg.~a." target))))))
      (with-html-form
          (:get (lambda (&rest args)
                  (broadcast
                    (username widget)
                    subscribes
                    (strip-tags (getf args :message)))
                  (mark-dirty widget)))
        (render-textarea :message "" 4 60)
        (:br)
        (render-input-field :submit :submit "Speak")))))
```

Реализация метода `render-widget-body` для виджета `speak` оказалась самой сложной среди всех трех основных

виджетов интерфейса. Помимо отображения текущего списка собеседников и формы для ввода сообщения, виджет отрисовывает служебную ссылку [**clear**], которая очищает список получателей сообщения (переводит из *multicast* в режим *broadcast*), а так же формирует список каналов для публикации сообщения через *ZeroMQ*.

В принципе, весь интерфейс готов:

- Виджет `online` отображает контакт-лист и позволяет выбрать собеседников для разговора.
- Виджет `history` отрисовывает всю историю переговоров, включая приватные сообщения и системные нотификации.
- Виджет `speak` содержит форму для отправки очередного сообщения в чат в режиме *broadcast* или *multicast*.

Однако, остался еще один не рассмотренный аспект проекта: интерактивная часть чата. Каким образом обновляется журнал сообщений, отслеживаются залогиненые и разлогиненные пользователи и т. д.

В проекте используется технология *Comet*[19] на основе длительного XHR-запроса. Браузер клиента осуществляет фоновый AJAX запрос, ответ на который происходит либо в тот момент, когда у сервера есть какие-то новости, либо по истечении таймаута. Технически же мы просто вручную регистрируем *Weblocks*-`action`, который конструируем для некоторого замыкания-обработчика:

```
(defun make-comet-script (proc)
  (let ((script
         (format nil "initiateAction(\"~A\", \"~A\");"
                 (make-action (lambda (&rest args)
                                (declare (ignore args))
                                (funcall proc)))
                 (session-name-string-pair))))
    script))
```

Напомню, что эта функция первоначально создает и отсылает клиенту скрипт в функции `make-ui` для инициализации *Comet*-сессии.

Давайте рассмотрим обработчик сообщений поближе:

```
(defun comet-handler (username
                      history-widget
                      online-widget
                      speak-widget
                      end-proc)
 (let ((logged-out-p nil)
       (private-channel (format nil "msg.~a." username)))
   (with-zmq-layout
       ((msg-all zmq:sub
                 :connect *zmq-pub-addr*
                 :option (zmq:subscribe "msg.all."))
        (msg-prv zmq:sub
                 :connect *zmq-pub-addr*
                 :option (zmq:subscribe private-channel))
        (sys-lgn zmq:sub
                 :connect *zmq-pub-addr*
                 :option (zmq:subscribe "sys.login."))
        (sys-exp zmq:sub
                 :connect *zmq-pub-addr*
                 :option (zmq:subscribe "sys.expired."))
        (sock-usr zmq:req :connect *zmq-usr-addr*))
       (:context *zmq-context* :reactor t)
     (with-zmq/cc ()
       ;; ping at start and quit after comet
       ;; session timed out
       (with-flow
         (send-parts/cc sock-usr (list username "ping"))
         (if (string= (car (recv-parts/string/cc sock-usr))
                      "success")
             (sleep/cc (* *comet-session-time* 1000000))
             (setf logged-out-p t))
         (break/cc))
       ;; process broadcasts
       (with-flow ;; msg.all.
         (recv-message
           msg-all #'read-from-string
           (lambda (from text)
             (add-history-item history-widget
                               (if (string= from username)
                                   'history-item-mine
                                   'history-item-shared)
                               from
                               text))))
       (with-flow ;; msg.<username>.
         (recv-message msg-prv #'read-from-string
                       (curry #'add-history-item
                              history-widget
                              'history-item-private)))
       (with-flow ;; sys.login.
         (recv-message sys-lgn #'list
                       (curry #'user-online
                              online-widget
                              history-widget)))
       (with-flow ;; sys.expired.
         (recv-message sys-exp #'list
                       (curry #'user-offline
                              online-widget
                              history-widget
                              speak-widget)))))
   (if logged-out-p
       (funcall end-proc)
       (send-script (make-comet-script
                     (lambda ()
                       (comet-handler username
                                      history-widget
                                      online-widget
                                      speak-widget
                                      end-proc)))))))

(defun/cc recv-message (socket processor proc)
  (let ((data (second (recv-parts/string/cc socket))))
    (prog1 (apply proc (funcall processor data))
      (break/cc))))
```

Алгоритм работы с сообщениями опять же основан на *zeromq-cc*.

Используемая в приведенном коде конструкция `defun/cc` – это видоизмененная с помощью `cl-cont` стандартная `defun`, выполняющая *CPS*-трансформацию кода функции. Для досрочного выхода из неявного цикла мультиплексирования в `ZeroMQ` применяется `break/cc`, определенная в библиотеке *zeromq-cc*.

Алгоритм обрабатывает следующие ситуации:

- Во-первых, при старте *Comet*-обработчика отсылается необходимый модулю **Core** пинг.

[19] http://en.wikipedia.org/wiki/Comet_%28programming%29

- Во-вторых, отслеживается таймаут, по истечении которого клиенту предлагается переустановить *Comet*-соединение.
- В-третьих, осуществляется подписка на несколько каналов.
  - **msg.all.**: сообщения из этого канала превращаются в *history-item-shared* и добавляются в пул виджета `history`.
  - **msg.<username>.**: сообщения из этого канала превращаются в *history-item-private* и добавляются в пул виджета `history`.
  - **sys.login.**: сообщения из этого канала превращаются в *history-item-system*, добавляются в пул виджета `history`, а также добавляют нового пользователя в список виджета `online`.
  - **sys.expired.**: сообщения из этого канала превращаются в *history-item-system*, добавляются в пул виджета `history`, а также удаляют пользователя из списка виджета `online`.

Давайте еще раз обратим внимание на то, насколько компактно у нас получилось описать довольно сложную логику благодаря продолжениям и DSL для описания *ZeroMQ*-макета из *zeromq-tools*.

Итоги

Давайте подобьем результаты, что у нас в итоге получилось.

- Проект состоит из двух самостоятельных модулей, коммуникация между которыми организована с помощью *ZeroMQ*.
- Модуль **Core** поддерживает текущее состояние системы: список залогиненных пользователей, отслеживает их «протухание» и маршрутизирует сообщения в чате.
- Модуль **Web** организует веб-интерфейс проекта на основе фреймворка *Weblocks*.
  - Бизнес-логика состоит из двух этапов: авторизация клиента в системе и последующая работа с интерфейсом чата. Реализация произведена через продолжения в блоке `with-flow` на корневом виджете `root`.
  - Сразу после отрисовки интерфейса чата браузер пользователя осуществляет фоновый *XmlHTTPRequest*, запускающий на стороне сервера обработчик `comet-handler`.
  - По мере появления новых событий XHR-запрос возвращает клиентскому браузеру информацию об обновившихся виджетах, которые он затем перерисовывает.

Реализация с использованием взаимонезависимых модулей уже предусматривает возможное масштабирование всей системы: уже в существующем виде возможно использовать более одного модуля **Web** с модулем **Core**, балансируя между ними нагрузку, например, с помощью DNS. С использованием несложного *ZeroMQ*-маршрутизатора (или их иерархии) можно масштабировать модуль **Core**, равномерно раскидывая пользователей по нодам.

Продолжения в веб, позволяющие прозрачно протаскивать контекст в последовательной логике, пригодились нам для реализации цикла «login» — «chat» — «logout» в пределах одной пользовательской сессии.

Продолжения, абстрагирующие конечный автомат, пригодились нам для реализации обработки протокола коммуникации между модулями проекта.

В итоге общий объем кода (прилагается к статье) достаточного нетривиального проекта получается немногим менее трех сотен строк.

6.8.3. Основные недостатки и паттерн применения

Опытные программисты должны сразу углядеть два самых больших минуса в программировании веб-проектов с использованием *continuation-based* фреймворков.

Слабопредсказуемый расход ресурсов сервера

В основном, конечно, памяти. Действительно, помимо обычного механизма автоматического управления памятью с GC мы получаем еще один слой: на каждую клиентскую сессию в рантайме развешивается некое дерево замыканий и продолжений, которые держат ссылки на различные объекты, не позволяя мусорщику их подчистить.

Конечно, тот же *Weblocks* вводит свой собственный механизм для устаревания таких сессий с последующей их подчисткой. Но все равно получается достаточно сложно прогнозировать жизненный цикл объектов, часть из которых подвешивается с активной ссылкой в каком-нибудь `action`-е формы, а часть может быть подчищена сразу. Это не позволяет простым образом прогнозировать расход памяти в зависимости от веб-активности проекта: память может начать неожиданно течь в относительно примитивных программах при резком скачке посещаемости, или вдруг практически не расходоваться на больших проектах со сложной логикой.

Затрудненное горизонтальное масштабирование

Здесь все совсем просто: мы не можем нарастить производительность системы добавлением нового сервера с балансировкой веб-запросов в кластере. Потому что все запросы клиента в пределах одной пользовательской сессии должны попадать на ту машину, где она была создана.

Продолжение, созданное, например, в рантайме одного процесса SBCL не может быть перекинуто на другую ноду, потому что физически это замыкание с развесистой сеткой активных ссылок на другие объекты рантайма. Для подобной миграции нужно либо каким-то образом сериализовать замыкание (что крайне нетривиальная задача во всех известных мне LISP-рантаймах за исключением Termite Scheme,[20] где сериализация и замыканий и продолжений специально предусмотрены платформой), либо придумывать какой-нибудь другой вариант.

Есть одна идея в этом направлении (кстати, относительно успешно проверенная мной на практике) — это создание «умного» фронтенда — балансировщика нагрузки перед кластером *Weblocks*-серверов. В отличие от обычного балансировщика, такой фронтенд отслеживает идентификаторы сессий и `action`-ов *Weblocks*, не позволяя двум запросам в пределах

[20] http://code.google.com/p/termite/

одной сессии уходить на разные ноды. Такие фронтенды достаточно просто можно выстраивать в иерархию, получая, тем самым, необходимую для отказоустойчивости избыточность.

Разделение труда

Популярные нынче паттерны разработки вроде *Model/View/Controller* продвигают философию разделения труда: верстальщик самостоятельно верстает, программист программирует, а специалист по БД проектирует хранилища. Весь проект как бы разделяется на слабозависимые, максимально абстрагированные друг от друга модули, которыми потом независимо друг от друга занимаются разные специалисты.

И если в *Weblocks* с абстрагированием *Model* проблем никаких нет, то *View* и *Controller* получаются в большинстве своем монолитными из-за `action`-ов, так как замыкания нельзя выносить в шаблон. В принципе, можно использовать и классическую схему с т. н. *Route*-ами, виджетом `selector` и каким-нибудь шаблонизатором вроде гуглового *Closure Templates* (на самом деле в свежих версиях *Weblocks* для этого предусмотрен специальный виджет: `template-block`). Но тогда *Weblocks* лишается львиной доли своих достоинств.

Однако для полноты картины, следует отметить две вещи:

1) В рамках архитектуры *Weblocks* активно используется CSS для настройки отображения в браузере, при этом все виджеты рендерятся в отдельные `<div>`-ы с аккуратным проставлением классов. Это позволяет свести генерацию разметки на стороне CL-кода к минимуму или вообще избежать.

2) Экономия усилий в разработке проекта настолько существенная, что зачастую отдельные специалисты не нужны вовсе.

Место в пищевой цепочке

Обобщив факты в описании недостатков, можно подводить итоги: подобного рода фреймворки плохо подходят для масштабных проектов с очень высокой нагрузкой. Начиная с некоторого момента (я бы навскидку назвал цифру в 100-200 AJAX-запросов в секунду для *Weblocks* на современном сервере) трудозатраты на масштабирование могут превзойти все сэкономленное на реализации проекта время. Особенно если взрывной рост посещаемости не был предусмотрен заранее.

В принципе, сочетание основных плюсов веб-фреймворков на продолжениях с основными минусами, рассмотренными в разделах выше, достаточно ясно вырисовывает картину идеальной сферы их применения: интранет-системы.

Действительно, именно интранет-системы и автоматизированные рабочие места (АРМ) зачастую отличаются повышенной сложностью интерфейса и высокими требованиями к интерактивности. При этом количество одновременно работающих с системой пользователей достаточно невелико, по сравнению с «диким» интернетом. Тем самым, тот же *Weblocks* отличнейшим образом вписывается в подобную нишу, значительно облегчая труд разработчика и позволяя с невероятной скоростью реализовывать сложнейшую логику в короткие сроки. Кстати, в эту же нишу еще замечательно вписывается коммон-лисповый *code hotswap* — модификация кода прямо «по живому», на работающем сервере.

Помимо интранет-приложений я бы рекомендовал подобного рода веб-фреймворки для всех интернет проектов, содержащих больше динамики и интерактива, нежели стильных и красивых элементов в интерфейсе. К примеру, веб-морды для почты (по аналогии с GMail), чаты, админки, разнообразные игровые проекты и т. п. Как минимум, прототипы такого рода сайтов однозначно можно реализовывать на том же *Weblocks*, а если еще заранее позаботиться о возможности масштабирования (как мы сделали в проекте `weblocks-chat`), то можно сразу реализовывать и *production*.

Разумеется, когда заранее известно, что с большой долей вероятности проект надо будет масштабировать, обязательно нужно предусматривать все необходимые архитектурные решения заранее. Чтобы клиентские запросы к *Weblocks* всегда попадали на ту машину, где была инициирована сессия, необходимо предусмотреть специальный «умный» фронтенд. Еще желательно поддержать в нем т. н. «обратную связь», чтобы (для более ровной балансировки) машины кластера умели уведомлять фронтенд о степени своей загруженности. Для организации избыточности можно параллельно выполнять эквивалентный запросы к более, чем одному *Weblocks*-серверу, чтобы выход из строя одного из них не влиял на работу проекта. *Comet*-запросы лучше мультиплексировать еще до того, как они дойдут до *Hunchentoot*, чтобы не заполонять сервер «подвисшими» тредами. И так далее — подобное масштабирование является нетривиальной задачей и может служить основой для отдельной статьи.

6.9. Как это все работает

6.9.1. Принцип работы

Итак, в самом последнем разделе подошло время вкратце ознакомиться с тем, как же все-таки продолжения физически работают. На самом деле есть два распространенных подхода:

- **C-style**: по аналогии с *Posix* функциями `setjmp` и `longjmp`, через клонирование всего стека, правку регистров и аналогом `goto`. Самый очевидный способ, но требующий неприятного копания «в кишках» операционной системы.
- **CPS**: так называемый *continuation passing style*. Это способ реорганизации кода таким образом, когда функция вместо обычного возврата управления через `return` вызывает полученную в аргументах функцию — *продолжение*.

Давайте рассмотрим поподробней второй способ, как следует производить подобную трансформацию кода:

1) Имеем следующий код:

```
(define (x^2 x)
  (* x x))

(define (a^2+b^2 a b)
  (+ (x^2 a) (x^2 b)))

(display (a^2+b^2 3 4))
```

   А теперь преобразуем его в *CPS*.

2) Шаг первый: добавляем в функции последним аргументом *продолжение* — функцию, которой мы будем передавать результаты вычисления (вместо того, чтобы вернуть их традиционным способом):

```
(define (x^2/cc x k)
  (* x x))

(define (a^2+b^2/cc a b k)
  (+ (x^2 a) (x^2 b)))
```

3) Шаг второй: в простых конструкциях вызываем `k` с результатом вычисления вместо возврата управления:

```
(define (x^2/cc x k)
  (k (* x x)))
```

4) Любые вызовы `/cc`-функций преобразуем таким образом, чтобы каким-то образом отловить результат их выполнения: грубо говоря, весь оставшийся в текущей функции алгоритм нам нужно упаковать в замыкание и передать как продолжение, которое будет получать результат:

```
(define (a^2+b^2/cc a b k)
  (x^2/cc a
          (lambda (a^2)
            (x^2/cc b
                    (lambda (b^2)
                      (k (+ a^2 b^2)))))))

(a^2+b^2/cc 3 4 display)
```

Приглядитесь внимательней к тому, что здесь происходит. Код «вытягивается» в диагональную дорожку замыканий: вызов функции, помимо аргументов, включает в себя замыкание, которое содержит алгоритм для дальнейшего выполнения. А в тех местах, где должен происходить возврат управления с каким-нибудь результатом вычисления, происходит вызов функции-продолжения с этим результатом.

В принципе, на этом месте должны стать понятными следующие моменты:

- Используя такую трансформацию можно реализовать продолжения в любом языке, который поддерживает замыкания.
- Крупная программа или циклический алгоритм, написанные в подобном стиле, могут переполнить стек. Если, конечно же, компилятор не поддерживает *tail call optimization — TCO*.
- Должен проясниться принцип работы `call-with-current-continuation` в Scheme. Действительно, конструкция `call/cc` в коде, написаном в *CPS*-стиле будет эквивалентна следующему:

```
(+ (call/cc (lambda (k) ... 1)) 2)

;; equal to

((lambda (k) ... 1)
 (lambda (value)
   (+ value 2)))
```

Разумеется, выполнение `(k value)`, в отличие от «настоящего» продолжения, в конце-концов, все-таки вернет управление обратно, так как является обычным вызовом функции. Но если **вся** программа преобразована в *CPS*, то на самом «дне» вызовов продолжений окажется явный или неявный `exit(2)`, который завершит приложение еще до того, как выполнится хоть какой-то `return`.

Теперь, обладая этой информацией, можно с комфортом использовать продолжения в любом современном языке программирования. В Scheme продолжения являются частью языка, Common Lisp с помощью своих мощных средств метапрограммирования умеет автоматически трансформировать код в *CPS*, а в Haskell, например, нам частично поможет монада *Cont*. Давайте, ради интереса, проверим наши технические знания на каком-нибудь языке, отличном от лиспов. Например, Perl.

6.9.2. map_cc.pl

Perl предоставляет в наше распоряжение полноценные замыкания, поэтому проблем с преобразованием кода в *CPS* возникнуть не должно.

В качестве демонстрационной давайте возьмем уже известную нам задачу, рассмотренную в разделе «*Конвертация коллбека в итератор*», которую мы реализовали на Scheme.

Начнем с реализации аналога функции `map-tree` на Perl:

```
sub map_tree {
   my ( $proc, $tree ) = @_;

   unless ( defined $tree ) {
      return [];
   }
   elsif (ref $tree eq 'ARRAY') {
      return [ map { map_tree( $proc, $_ )
          } @$tree ];
   }

   return $proc→( $tree );
}
```

По сути, это даже не аналог, это полная копия оригинала, переписаная в перловом синтаксисе. Но есть один тонкий момент: если Scheme поддерживает продолжения на нативном уровне, то в Perl нам придется руками переписать ее в *CPS*:

```
sub map_tree_cc
{
   my ( $proc, $tree, $k ) = @_;

   unless ( defined $tree )
   {
      $k→( [] );
   }
   elsif ( ref $tree eq 'ARRAY' )
   {
      return $k→( [] ) unless @$tree;

      my ( $tree_head, @tree_tail ) =
          @$tree;

      map_tree_cc(
         $proc,
         $tree_head,
         sub {
            my ( $head ) = @_;

            map_tree_cc( $proc,
                     \@tree_tail,
```

```
                    sub {
                       my ( $tail ) = @_;

                       $k→( [ $head,
                           @$tail ] );
                    } )
         } );
   }
   else
   {
      $proc→( $tree, $k );
   }
}
```

Помимо стандартных преобразований, рассмотренных нами в предыдущем разделе, нам пришлось отказаться от функции `map` из стандартной библиотеки и реализовать обход списка вручную, потому что иначе вместо передачи управления продолжению `map` осуществит возврат. В этом заключается минус подобных преобразований в языках, которые не поддерживают продолжения изначально: помимо «перелопаченного» кода нам нужно писать совместимую с *CPS* версию комбинаторов из стандарной библиотеки и из всех возможных *third-party* библиотек. Тоесть стопроцентно эмулировать совершенно полноценные замыкание без явной их поддержки со стороны рантайма у нас все равно не получится.

Теперь, имея необходимый `map_tree_cc`, можно реализовывать аналог `tree-gen` — это очень просто (см. оригинал):

```
sub tree_gen
{
   my ( $tree, $flow ) = @_;

   map_tree_cc(
       sub { my ( $e, $k ) = @_;
            $flow→( $e,
                   sub { my ( $next_flow )
                       = @_;
                    $flow = $next_flow;
                    $k→( $e ) } ) },
       $tree,
       sub { $flow→() } );
}
```

Код получается параллельным аналогу на Scheme, только с учетом ручного *CPS*. Точно так же реализуется и «заменитель» функции `fold-trees`:

```
sub fold_trees
{
   my ( $proc, $seed, $cont_a, $cont_b ) =
       @_;

   $cont_a→( sub {
        my ( $leaf_a, $next_cont_a ) = @_;
        $cont_b→( sub {
             my ( $leaf_b, $next_cont_b )
                 = @_;

             if ( $next_cont_a &&
                 $next_cont_b )
             {
                fold_trees( $proc,
                            $proc→( $leaf_a,
                                   $leaf_b,
                                   $seed ),
                            $next_cont_a,
                            $next_cont_b )
             }
             else
             {
                $seed
             }
        } )
   } )
}
```

Ну и, наконец, целевая функция, `trees-prefix-length`. Вот так она будет записана на Perl:

```
sub common_prefix_length
{
   my ( $tree_a, $tree_b, $return ) = @_;

   $return→( fold_trees(
     sub { my ($a, $b, $len) = @_;
          $a == $b ?
          $len + 1 :
          $return→( $len ) },
     0,
     sub { tree_gen( $tree_a, $_[0] ) },
     sub { tree_gen( $tree_b, $_[0] ) } ) )
}
```

В данной записи у нас получился даже преждевременный выход из `fold_trees` наружу, благодаря продолжению `$return`. И теперь подошло время проверить результат:

```
common_prefix_length(
   [ 1, [[[2], 3], 4], 5 ],
   [ [[[1, 2], 9], 4], 5 ],
   sub { print Dumper $_[0]; exit } );
```

С небольшим хаком: `exit` в конечном продолжении, чтобы не раскручивать обратно накопившийся хвост вызовов в стеке. Ну и, опять же, следует понимать, что на длинных одинаковых деревьях вполне вероятно схлопотать переполнение стека, так как, насколько я знаю, *TCO* (*tail-call optimization*) в перле не реализована.

6.9.3. Вывод

Я не случайно поместил раздел про *continuation passing style* в самый конец статьи. В тех языках, в которых возможно комфортное использование продолжений (Scheme, Smalltalk, Common Lisp, Haskell, ...) знать всю техническую часть необязательно, гораздо важнее освоить паттерн практического их применения. В тех же языках, в которых использование продолжений **возможно** (путем клонирования стека или ручной *CPS*-трансформации, как в разделе выше) — их использование больше запутывает код, нежели приносит какую-то ощутимую пользу.

Что касается использования *CPS*-преобразований в сфере программирования компиляторов, то это отдельная самостоятельная тема, которая не пересекается с материалом в данной статье.

Литература

[1] *Abelson H., Sussman G. J.* Structure and Interpretation of Computer Programs, 2nd Edition. — The MIT Press, 1996. http://mitpress.mit.edu/sicp/.

[2] *Ferguson D., Dwight D.* Call with current continuation patterns. — 2001. — Переведена на русский язык, перевод доступен в Библиотеке переводов журнала ПФП.

[3] *Stone J. D.* Srfi-8: Receive: binding to multiple values. — 1999. http://srfi.schemers.org/srfi-8/srfi-8.html.

[4] *Харольд Абельсон, Джеральд Джей Сассман.* Структура и интерпретация компьютерных программ. — М.: Добросвет, 2006.

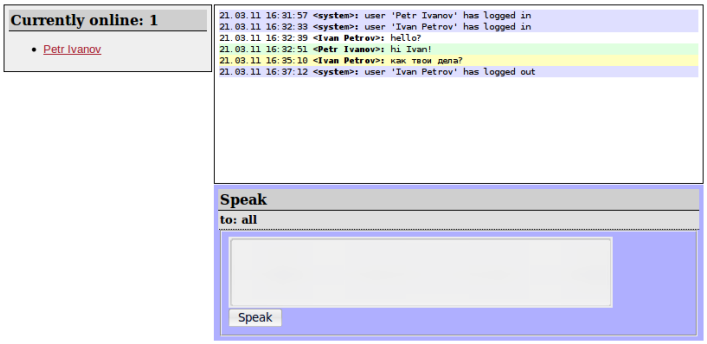


Рис. 6.7. weblocks-chat

Суперкомпиляция: идеи и методы

Илья Ключников
klyuchnikov@fprog.ru

Аннотация

Суперкомпиляция (supervising compilation) — техника преобразования программ, основанная на построении *полной и самодостаточной модели* программы.

В статье описываются основные идеи и методы суперкомпиляции на примере работающего суперкомпилятора SC Mini для простейшего чисто функционального языка.

Supercompilation (supervising compilation) is a program transformation technique based upon constructing a self-sufficient model of the program.

The paper describes the main ideas and methods of supercompilation through a series of examples performed by the simple supercompiler SC Mini.

Обсуждение статьи ведется по адресу:
http://fprog.ru/2011/issue7/ilya-klyuchnikov-supercompilation/discuss/.

7.1. Идея суперкомпиляции

Суперкомпиляция была придумана в СССР в 1970-х годах В. Ф. Турчиным. Сам он объяснял название и идею так [31]:

> A program is seen as a machine. To make sense of it, one must observe its operation. So a supercompiler does not transform the program by steps; it controls and observes (SUPERvises) the machine, let us call it M_1, which is represented by the program. In observing the operation of M_1, the supercompiler COMPILES a program which describes the activities of M_1, but it makes shortcuts and whatever clever tricks it knows, in order to produce the same effect as M_1, but faster. The goal of the supercompiler is to make the definition of this program (machine) M_2, self-sufficient. When this is achieved, it outputs M_2, and simply throws away the (unchanged) machine M_1.
>
> A supercompiler would run M_1 in a general form, with unknown values of variables, and create a graph of states and transitions between possible configurations of the computing system … in terms of which the behavior of the system can be expressed. Thus the new program becomes a self-sufficient model of the old one.[1]

Суммируем кратко:

1) Программе P_1 ставится в соответствие машина M_1, *моделирующая в общем виде* выполнение программы P_1.

2) *Контролируя и наблюдая* (SUPERvises) работу машины M_1, суперкомпилятор *создает* (компилирует, COMPILEs) другую машину M_2, которая полностью описывает M_1.

3) Машина M_2 далее может быть представлена в виде программы P_2.

7.2. Цель данной статьи

Среди людей, занимающихся именно практическим функциональным программированием (да и не только функциональным), отношение к суперкомпиляции (среди тех, кто о ней слышал) очень разное. Наиболее часто встречающаяся реакция — недоверие и даже отторжение по той причине, что реального промышленного суперкомпилятора в природе пока еще не существует. А те, что существуют — экспериментальные. И в большинстве случаев экспериментальные суперкомпиляторы плодотворно используются только их авторами.

Идея суперкомпиляции в некотором смысле универсальна, но полезными и работоспособными оказываются достаточно узко специализированные изделия, использующие идеи суперкомпиляции. А пользователи от универсальной идеи ожидают универсального инструмента. Еще одна причина недоверия — то, что на базе суперкомпиляции нет еще того, что называется killer app.

Достаточно распространено заблуждение, что суперкомпиляция — техника оптимизации программ. Суперкомпиляция — это *метод преобразования программ*. Бывают разные цели преобразований программ. Целью может быть оптимизация программы, а может быть и ее анализ. Методы суперкомпиляции используются и для того, и для другого. Большинство работ посвящены применению суперкомпиляции для оптимизации, но это не значит, что нужно рассматривать суперкомпиляцию однобоко только с этой позиции.

Необходимо различать понятия *суперкомпиляция* и *суперкомпилятор*. Многие статьи описывают различные технические трудности, возникающие при построении конкретного суперкомпилятора, и способы их преодоления, отодвигая основные методы суперкомпиляции на задний план. В большинстве случаев части, из которых состоит конкретный суперкомпилятор, очень тонко и старательно подогнаны друг под друга, из-за чего может возникать ощущение сложной монолитной конструкции.[2]

Основная цель данной статьи — на примере «минимального» суперкомпилятора попытаться четко и обозримо проиллюстрировать приведенную цитату В. Ф. Турчина, описывающую идею суперкомпиляции. Я постараюсь сделать акцент на вычленении и объяснении основных логических частей и показать, как эти части могут быть запрограммированы и соединены достаточно простым и понятным способом.

Как устроена статья и что в ней есть: Главный материал этой статьи — это приложение (pdf-файл) с полным кодом игрушечного[3] суперкомпилятора SC Mini[4] и подробными комментариями. Сам текст статьи стоит рассматривать как подготовку к чтению приложения. В статье вводятся и разбираются основные понятия и на примерах показывается «механика» работы SC Mini. Примеры, приведенные далее, не являются сложными, но и простыми их назвать, наверное, тоже нельзя. Читателю следует подготовиться к некоторому «погружению» в примеры.

Чего нет в данной статье: Даже поверхностное сравнение суперкомпиляции с современными техниками оптимизации и/или анализа программ было бы противоречивым и беспредметным, поэтому я даже не пытаюсь это сравнение сделать.[5]

[1] Приблизительный перевод:

Программа рассматривается в виде некоторой машины. Смысл извлекается из наблюдения за ее действиями. Суперкомпилятор не преобразует программу шаг за шагом; он управляет и наблюдает за машиной (назовем ее M_1), которая представлена в виде программы. Наблюдая за работой машины M_1, суперкомпилятор конструирует другую машину M_2, которая описывает действия машины M_1. При построении M_2 суперкомпилятор пользуется различными трюками, чтобы M_2 работала так же, как и M_1, но быстрее. Цель суперкомпилятора – сделать M_2 самодостаточной. По достижении этой цели суперкомпилятор отбрасывает исходную машину M_1 и выдаёт M_2.

Суперкомпилятор запускает M_1 в общем виде (с неизвестными значениями переменных), строит граф состояний и переходов между различными конфигурациями вычислительной системы. Такой граф полностью описывает поведение системы. Таким образом, новая программа становится самодостаточной моделью исходной программы.

[2] «Если посмотреть, как устроен суперкомпилятор, то видно, что он состоит из нескольких частей, слепленных в один запутанный комок грязи. Когда я попытался по-настоящему понять, как работают суперкомпиляторы, это оказалось очень трудно.» (С. Пейтон Джоунс. Интервью // «ПФП», № 6.)

[3] 250 строк основного кода + 200 строк вспомогательного кода (различные утилиты) на Хаскеле.

[4] https://github.com/ilya-klyuchnikov/sc-mini

[5] Это сравнение в данной статье было бы хоть как-то возможным, если бы в предыдущих выпусках «ПФП» уже были бы обзорные статьи по методам оптимизации и анализа программ — тогда можно было бы сравнить суперкомпиляцию с описанными методами. Нужно вначале прийти к соглашению, какие методы и инструменты мы хотим сравнить. Иначе данная статья будет объектом всевозможных упреков в том, что метод А или Б несправедливо обойден вниманием. Однако автор не уходит от ответственности и готов ответить на конкретные вопросы в дискуссии в ЖЖ.

7.3. Методы суперкомпиляции на примерах

> Достоинства всякого формализованного языка определяются не только тем, сколь он удобен для непосредственного использования человеком, но и тем, в какой степени тексты на этом языке поддаются формальным преобразованиям.
>
> В. Ф. Турчин [43]

Было бы хорошо, если бы все статьи по информатике сопровождались изложением в формальной машинно-читаемой форме (в виде программ или других формальных спецификаций), чтобы не возникала проблема воспроизводимости результатов.

Суперкомпилятор автора SC Mini — формальное изложение основных методов суперкомпиляции, описанных в данной статье. За основу суперкомпилятора SC Mini взят суперкомпилятор, описанный в изумительной магистерской диссертации Мортена Сёренсена [26].[6]

При описании суперкомпиляции так или иначе приходится затрагивать такие базовые, но непростые понятия, как смысл (семантика) программы, результат вычислений, состояние вычислений. Все эти понятия формально описаны (= запрограммированы) в тексте SC Mini, что помогает избежать двусмысленности (по крайней мере, в приложении).

Суперкомпилятор SC Mini преобразует программы, написанные на игрушечном языке SLL.[7] Перефразируя высказывание В. Ф. Турчина, приведенное в качестве эпиграфа, можно сказать, что достоинствами языка SLL являются:

1) Простота описания языка.

2) Простота описания суперкомпилятора SLL-программ.

Несмотря на то, что SC Mini — суперкомпилятор для конкретного языка SLL, методы, на которых он основан, достаточно универсальны и присутствуют в той или иной степени практически во всех суперкомпиляторах.

Далее мы на примерах покажем, что SC Mini способен оптимизировать программы (предварительно формально определив, что такое оптимизатор) и может быть использован для доказательства простых теорем о программах.

Статья содержит замечания, набранные мелким шрифтом, которые при первом прочтении можно пропустить или просто обратить внимание на вопросы, которые там затрагиваются, но к которым можно вернуться после знакомства с текстом SC Mini. Цель замечаний — указать на некоторые очень серьёзные «тонкости», которые обеспечивают корректность преобразований. Именно такие тонкие и сперва неприметные моменты привносят основные сложности в создание суперкомпиляторов.

Следующий раздел, в котором описывается язык SLL, носит формальный характер.

Рис. 7.1. SLL: абстрактный синтаксис

P	::=	$d_1 \dots d_n$	программа
d	::=	$f(v_1, \dots, v_n) = e;$	«безразличная» функция
	|	$g(p_1, v_1, \dots, v_n) = e_1;$	«любопытная» фунция
		...	
		$g(p_m, v_1, \dots, v_n) = e_m;$	
e	::=		выражение
		v	переменная
	|	$C(e_1, \dots, e_n)$	конструктор
	|	$f(e_1, \dots, e_n)$	вызов функции
p	::=	$C(v_1, \dots, v_n)$	образец

Рис. 7.2. `prog1`: работа с числами Пеано

```
add(Z(), y)  = y;
add(S(x), y) = S(add(x), y);

mult(Z(), y)  = Z();
mult(S(x), y) = add(y, mult(x, y));

sqr(x) = mult(x, x);

even(Z())  = True();
even(S(x)) = odd(x);

odd(Z())  = False();
odd(S(x)) = even(x);

add'(Z(), y)  = y;
add'(S(v), y) = add'(v, S(y));
```

7.3.1. Объектный язык SLL

Определить язык — значит описать его синтаксис и семантику. Синтаксис языка традиционно описывается бесконтекстной грамматикой. Описанием семантики может быть реализация языка. То, что дальше описывается словами, гораздо проще и яснее описано в приложении в виде программы на Хаскеле.

Синтаксис

Абстрактный синтаксис SLL приведен на рис. 7.1. Выражения языка SLL:

- переменная,
- конструктор, аргументами которого являются SLL-выражения,
- вызов функции, аргументами которого являются SLL-выражения.

Согласимся, что имена конструкторов начинаются с большой буквы, а имена функций и переменных — с маленькой.

[6] Сёренсен только описал, но не реализовал свой суперкомпилятор.

[7] SLL — Simple Lazy Language. В диссертации Сёренсена рассматриваются языки M_1, $M_{1/2}$, M_0. SLL в точности соответствует языку M_0. Название SLL введено только для удобства речи.

Рис. 7.3. SLL: контекст и редекс

con	::=	$\langle\rangle \mid g(con, \ldots)$	контекст
red	::=	$f(e_1, \ldots, e_n) \mid g(C(e_1, \ldots, e_n), \ldots)$	редекс

SLL-выражение, состоящее только из конструкторов, называется *значением*. SLL-выражение без переменных называется *замкнутым выражением*. SLL-выражение со свободными переменными называется *конфигурацией*.

Программа на языке SLL состоит из определений функций. Функции бывают двух видов — «безразличные» и «любопытные»[8]. Безразличные функции только передают свои аргументы другим функциям и конструкторам: определение безразличной функции — одно предложение. Любопытные функции осуществляют ветвление по первому аргументу: определение любопытной функции состоит из нескольких предложений (по одному предложению на каждый вариант первого аргумента).

Никаких встроенных типов данных (булевых значений, чисел и т. д. в языке SLL нет. Однако их можно определить, например, так:

- константы `True()` и `False()` — логические значения,
- константа `Z()` — 0, `S(Z())` — 1, `S(S(Z()))` — 2 и т. д. Если константа `n` соответствует натуральному числу n, то константа `S(n)` соответствует натуральному числу $n + 1$ — так называемые числа Пеано.

На рис. 7.2 приведена программа, в которой определяются функции для работы с числами Пеано. В этой программе только одна безразличная функция — возведение в квадрат (`sqr`). Все остальные функции разбирают свой первый аргумент и являются любопытными.

Подстановка связывает переменные $v_1, v_2, \ldots, v_n$ с выражениями $e_1, e_2, \ldots, e_n$ и записывается как список пар $\{v_1 := e_1, \ldots, v_n := e_n\}$. Применение подстановки к выражению e определяется стандартным образом и записывается $e/\{v_1 := e_1, \ldots, v_n := e_n\}$.

Упражнение 1 В приложении применение подстановки рассматривается как список пар. Применение подстановки `s` к выражению `e` определяется как `e // s`. Найдите такие `e`, `v1`, `e1`, `v2` и `e2`, что
`e // [(v1, e1), (v2, e2)]` $\neq$ `e // [(v1, e1)] // [(v2, e2)]`.

Семантика

Семантика[9] языка SLL определяется через переписывающий SLL-интерпретатор с нормальным порядком редукции. SLL-интерпретатор $\mathcal{I}_p$ программы p для замкнутого SLL-выражения пошагово вычисляет SLL-значение (или зацикливается)[10] следующим образом.

С точки зрения редукции есть два вида замкнутых выражений:

1) $e = C(e_1, \ldots, c_n)$ — конструктор «выталкивается» наружу, дальше происходит редукция аргументов.
2) $e \neq C(e_1, \ldots, c_n)$ — тогда в выражении находится самый левый редуцируемый вызов функции (редекс, см. рис. 7.3) и редуцируется в соответствии с определением в программе. Вызов функции является редуцируемым, если это 1) вызов безразличной функции или 2) вызов любопытной функции, первым аргументом которого является конструктор.

Формально редукционная семантика SLL-выражений дана на рис. 7.4. Запись $e_1 \stackrel{p}{=} e_2$ означает, что в программе p есть определение $e_1 = e_2$.

Правила вычисления SLL-выражений обладают приятным и важным свойством «композиционности». А именно, если выражение $e_1/\{v := e_2\}$ замкнутое, то:

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

Заметим, что $\mathcal{I}_p$ можно рассматривать, как машину, описывающую программу p.

Упражнение 2 В SC Mini интерпретатор определяется как функция `eval`. Запись $\mathcal{I}_p[\![e]\!]$ соответствует вызову `eval p e`. Докажите, что результатом вычисления
`eval p (e1 // [(v, e2)]) == eval p (e1 // [(v, eval p e2)])`
не может быть `False`.

Пример вычисления выражения `even(sqr(S(Z())))` интерпретатором для программы `prog1` представлен на рис. 7.5. Редексы подчеркнуты.

SLL-задание[11] — пара (e, p) из выражения e и программы p. Вычисление задания (e, p) на аргументах args определяется как:[12]

$$\mathcal{R}_{(e,p)}[\![\text{args}]\!] = \mathcal{I}_p[\![e/\text{ args}]\!]$$

Оптимизатор

Определим, что такое оптимизатор SLL-заданий. SLL-оптимизатор получает задание (e, p) и выдаёт новое другое задание (e', p') такое, что на любых аргументах args результаты вычисления старого и нового заданий совпадают[13]:

$$\mathcal{R}_{(e,p)}[\![\text{args}]\!] = \mathcal{R}_{(e',p')}[\![\text{args}]\!]$$

и при вычислении нового задания интерпретатор делает *не больше* шагов (редукций), чем при вычислении старого.

7.3.2. Обзор устройства суперкомпилятора SC Mini

Суперкомпилятор SC Mini, преобразуя задание (e, p) в задание (e', p'), действует следующим образом:

1) Конструирует машину $\mathcal{M}_p$, которая описывает работу *одного шага* интерпретатора $\mathcal{I}_p$ программы p в «общем виде» — над конфигурациями.
2) Осуществляет конечное (в общем случае, достаточно большое) число запусков машины $\mathcal{M}_p$, и анализирует результаты этого запуска.
3) Строит конечный граф конфигураций, описывающий работу $\mathcal{M}_p$ во время пробных запусков.

[8] Это просто синтаксический трюк, чтобы избежать конструкции `case e ...` и не возиться со связанными переменными в выражениях. Исторически функции первого вида называются f-функциями, а второго — g-функциями.

[9] Со способами описания семантик можно познакомиться, например, по книге [19]; там же описываются понятия редекс и контекст редукции, используемые на рис. 7.3 и 7.4 без объяснений.

[10] Или выдаёт ошибку, но такие случаи мы здесь для простоты не рассматриваем. Их рассмотрение не привносит ничего принципиально нового.

[11] Понятие SLL-задания вводится для упрощения изложения и является в некоторой степени аналогом функции `main`.

[12] В SC Mini этому соответствует `sll_run (e, p) args`.

[13] То есть SLL-система либо выдаёт равные SLL-значения для обоих запусков, либо зацикливается для обоих запусков.

Рис. 7.4. SLL: интерпретатор программы $\mathcal{I}_p$ для программы p

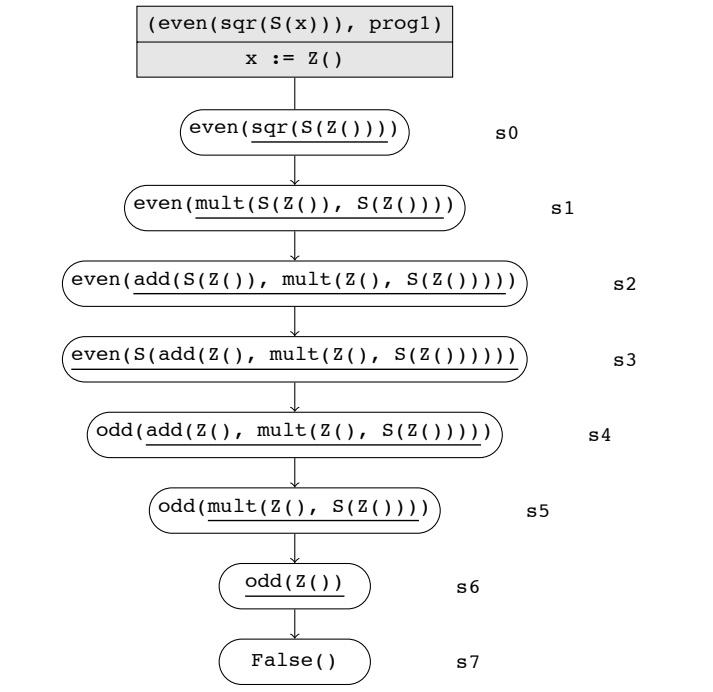
$$\mathcal{I}_p[\![e]\!] \Rightarrow e \quad (I_1)$$
если e – значение

$$\mathcal{I}_p[\![C(e_1, \ldots, e_n)]\!] \Rightarrow C(\mathcal{I}_p[\![e_1]\!], \ldots, \mathcal{I}_p[\![e_n]\!]) \quad (I_2)$$

$$\mathcal{I}_p[\![con\langle f(e_1, \ldots, e_n)\rangle]\!] \Rightarrow \mathcal{I}_p[\![con\langle e/\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]\!] \quad (I_3)$$
if $f(v_1, \ldots, v_n) \stackrel{\text{p}}{=} e$

$$\mathcal{I}_p[\![con\langle g(C(e_1, \ldots, e_m), e_{m+1}, \ldots, e_n)\rangle]\!] \Rightarrow \mathcal{I}_p[\![con\langle e/\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]\!] \quad (I_4)$$
if $g(C(v_1, \ldots, v_m), v_{m+1}, \ldots, v_n) \stackrel{\text{p}}{=} e$

Рис. 7.5. Пример вычисления задания



4) Упрощает граф конфигураций и превращает его в новое задание (e', p').

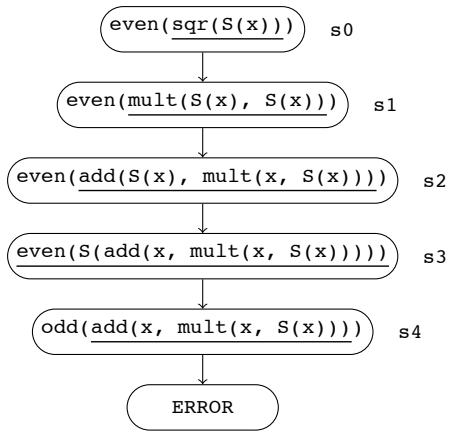
Далее эти шаги разбираются более подробно.

7.3.3. Прогонка

Что будет, если попытаться вычислить следующее задание на пустых аргументах, то есть подать интерпретатору на вход такую конфигурацию?

```
(even(sqr(S(x))), prog1)
```

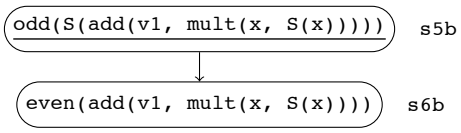
SLL-интерпретатор устроен таким образом, что, в принципе, он «способен» вычислять и выражения со свободными переменными:



Интерпретатор не ожидает переменной в первом аргументе вызова функции `add`. Однако наличие переменной `x` не помешало интерпретатору сделать несколько начальных шагов.[14]

«Расширим» интерпретацию $\mathcal{I}$:[15] вместо аварийного завершения рассмотрим различные варианты дальнейших действий интерпретатора. Программа `prog1` для рассматриваемой ситуации предусматривает два варианта развития: когда `x = Z()` и когда `x = S(x1)` (см. определение функции `add`): рис. 7.6.

Теперь мы можем продвинуться в вычислении правого выражения (и т. д.):



Интерпретатор $\mathcal{I}_p$ рассчитан на пошаговое вычисление замкнутых выражений. Результатом запуска интерпретатора $\mathcal{I}_p[\![e]\!]$ является некоторое (итоговое) SLL-значение.

Машина $\mathcal{M}_p$ будет вычислять конфигурации. Результатом запуска машины $\mathcal{M}_p[\![c]\!]$ будет моделирование шага интерпретатора.

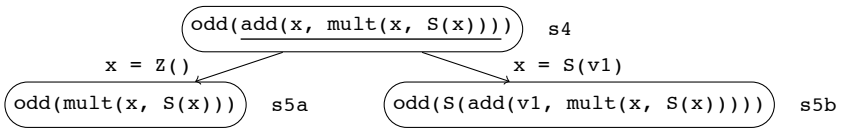
Рассматриваем следующие модели шагов:

1) Транзитный шаг — ближайший шаг интерпретации не зависит от конфигурационных переменных. Пример — переход из состояния `s0` в состояние `s1`.

[14] При передаче параметров по имени (call-by-name), действительно, интерпретатор способен «кое-что» вычислить. При передаче параметров по значению (call-by-value) заставить интерпретатор продвинуться в вычислении выражения со свободными переменными нетривиально. Именно поэтому суперкомпиляцию легче объяснять для языка с передачей параметров по имени.

[15] Многим это напоминает абстрактную интерпретацию [2]. Однако, абстрактная интерпретация (по отношению к обычной) есть сужение области значений. Здесь же никакого сужения области значений не происходит.

Рис. 7.6. SLL: Рассмотрение вариантов при прогонке



2) Остановка. Дальнейшее моделирование невозможно. Возникает, если данное выражение является переменной или константой.

3) Декомпозиция. Часть результата уже известна, например, в выражении `S(sqr(x))` внешний конструктор уже никуда не денется. Переходим к обработке подвыражений.

4) Разбор вариантов. Однозначное моделирование дальше невозможно. Однако можно рассмотреть *все варианты шагов интерпретатора, которые описаны в программе*. Пример — переходы из состояния `s4` в состояния `s5a` и `s5b`.

Такое моделирование в суперкомпиляции называется *прогонкой* (driving). То, что описано выше — шаг прогонки.

Упражнение 3 В SC Mini конструирование машины $\mathcal{M}_p$ есть `driveMachine p`. Покажите, что `driveMachine p` достаточно для вычислений замкнутых выражений. То есть можно написать такую функцию `f`, что

```
f (driveMachine p) e = eval p e
```

7.3.4. Дерево конфигураций

Дерево конфигураций для SLL-задания (e, p) строится следующим образом. Создается машина $\mathcal{M}_p$, моделирующая p. Вначале дерево состоит из одного узла, в который помещена стартовая конфигурация e (цель задания). Затем для каждого листа дерева n, в котором находится некоторая конфигурация c, производится «запуск» $\mathcal{M}_p[\![c]\!]$ (т. е. запуск одного шага интерпретации) и результаты запуска подвешиваются к n в виде дочерних узлов. Затем для них также запускается машина и т. д. Получается дерево конфигураций (в каждом узле дерева находится конфигурация).

В общем случае дерево конфигураций, конструируемое таким образом, будет бесконечным. Но мы всегда можем заглянуть в него на конечную глубину (в предыдущем разделе мы рассмотрели построение верхушки дерева конфигураций для задания `(even(sqr(S(x))), prog1)`).

Упражнение 4 В SC Mini построение дерева конфигураций для задания (e, p) происходит так:

```
buildTree (driveMachine p) e
```

Попробуйте как-то охарактеризовать класс заданий, для которых строятся конечные деревья конфигураций.

Хотя сама концепция дерева конфигураций имеет небольшую практическую ценность,[16] она имеет «важную теоретическую ценность» и помогает понять механизм работы суперкомпилятора.

Предположим на минуту, что мы можем строить (и как-то хранить) бесконечные деревья конфигураций. Тогда, если мы для задания (e, p) с помощью машины $\mathcal{M}_p$ построили дерево конфигураций t, то мы можем отбросить и программу p, и машину $\mathcal{M}_p$, и работать только с деревом t. Дерево t полностью описывает выполнение задания (e, p), абстрагируясь от текста программы p.

Упражнение 5 Чтобы последнее утверждение не было голословным, в приложении приведен «древесный» вычислитель `intTree`, который умеет вычислять задания, используя только дерево конфигураций, и не нуждается в исходной программе. Если `t` — дерево конфигураций для задания `(e, p)`, то `intTree t args` есть результат вычисления задания для аргументов `args`. Убедитесь, что значением выражения

```
eval (e, p) args == intTree (buildTree (driveMachine p) e) args
```

не может быть `False`.

7.3.5. Свертка

> Infinite driving graphs are useful in many ways, but to use a graph as an executable program it must be finite.
>
> В. Ф. Турчин [32]

Мы можем посчитать `even(sqr(S(x)))` для любого `x`, конструируя дерево конфигураций и затем используя «древесный» интерпретатор `intTree`. Однако соответствующее дерево конфигураций получается бесконечным. Цель *свертки* (folding) — превратить бесконечное дерево в конечный объект, из которого при желании восстанавливается исходное бесконечное дерево.

Как осуществляется свертка? Пусть в строящемся дереве конфигураций возникла ветвь, спускаясь по которой, мы встречаем узлы $n_0 \to \ldots \to n_i \to \ldots \to n_j \to \ldots$ Пусть конфигурация в узле n_j является переименованием (отличается только именами переменных) конфигурации в некотором узле n_i $(i < j)$. Тогда делается следующий вывод: поддерево, «растущее» из узла n_j можно получить из поддерева, «растущего» из узла n_i, если последовательно переименовывать переменные в соответствующих конфигурациях.

Рассмотрим дерево конфигураций для задания `(even(sqr(x)), prog1)`, изображенное на рис. 7.7.

Конфигурация в узле n_4 является переименованием конфигурации в узле n_2. Поддеревья (бесконечные!), растущие из n_2 и n_4, отличаются только именами переменных в соответствующих узлах. Поэтому мы можем запомнить, что поддерево n_4 получается из поддерева n_2, и не строить его прямо сейчас, а отложить построение до момента, когда это поддерево действительно понадобится. Важным результатом является то, что для получения поддерева n_4 из поддерева n_2 нам не требуется программа (= машина $\mathcal{M}_p$).

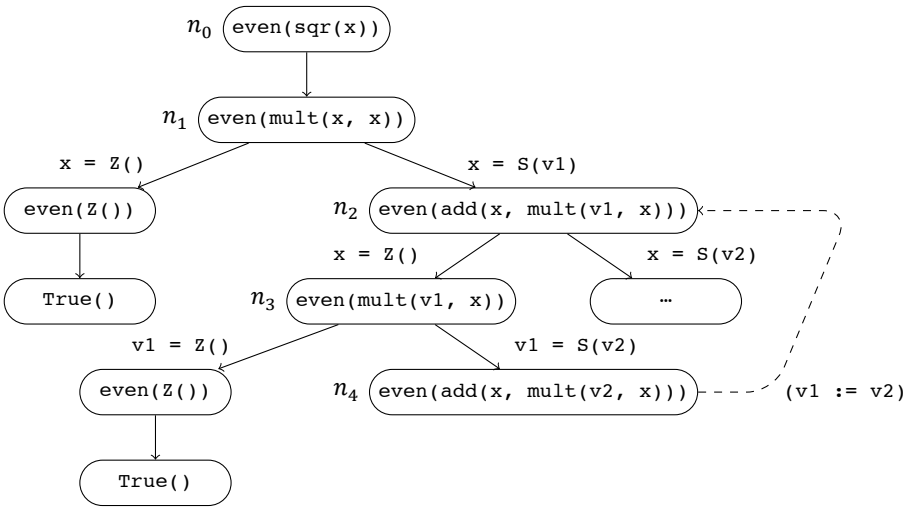
Для обозначения этого факта будем использовать специальную дугу, ведущую из нижнего узла в верхний. В результате дерево конфигураций перестает уже быть деревом и превращается в *граф конфигураций*.

С деревом конфигураций для задания `(even(sqr(x)), prog1)` нам везет — оно сворачивается в граф.[17] Таким обра-

[16] Из-за этого ни дерево конфигураций, ни граф конфигураций (см. дальше) почти не упоминаются в статьях последних лет (особенно западных авторов), обсуждающих создание практического суперкомпилятора.

[17] Граф получается достаточно большим — он приведен полностью в приложении, убедитесь, что для этого примера действительно получается конечный граф без бесконечных ветвей.

Рис. 7.7. Дерево конфигураций для задания `(even(sqr(x)), prog1)`



зом, мы можем превращать некоторые бесконечные деревья в конечные графы. Получившийся граф и будет той самой новой самодостаточной *конечной* машиной для задания. Будем обозначать граф для задания t как $\mathcal{G}_t$.

Упражнение 6 Стоит совсем немного изменить древесный интерпретатор `intTree`, чтобы он умел вычислять задания с помощью соответствующего свернутого графа конфигураций. Посмотрите, как это сделано в SC Mini.

Граф $\mathcal{G}_t$ для задания $t = (e, p)$ можно представить в виде нового SLL-задания (e', p'). Полученная таким образом программа `p'` называется *остаточной (residual)*, а задание $(e',\ p')$ будем по аналогии называть остаточным.

Упражнение 7 В приложении описана функция `residuate`: `residuate g` преобразует граф конфигураций `g` в новое задание `(e', p')`.

7.3.6. Обобщение: превращение дерева в сворачиваемое дерево

В рассмотренном примере нам повезло в том, что дерево конфигураций удалось свернуть в граф. Как было бы замечательно, если бы все деревья конфигураций сворачивались! К сожалению, не все деревья можно превратить в графы. В программе `prog1` есть функция `add'`, определяющая сложение с помощью накапливающего параметра. Дерево конфигураций для задания `(add'(x, y), prog1)` строится, как показано на рис. 7.8.

Это дерево не сворачивается. Действительно, конфигурации могут различаться именами переменных только тогда, когда они (конфигурации) одинакового размера. В данном же примере, идя по «самой правой» ветке, мы встречаем конфигурации, постоянно увеличивающиеся в размере. Однако если бы размер конфигураций в дереве был ограничен, то такое дерево было бы сворачиваемым.[18]

Будем действовать по принципу *divide et impera*. Ограничим размер конфигураций в узлах некоторой константой `sizeBound`, и если некоторая конфигурация превысит допустимый размер — разделим ее на меньшие составляющие, чтобы их можно было рассматривать *независимо*. Однако мы должны так представить конфигурацию, чтобы изучив поведение ее частей, мы могли бы сказать, как ведет себя целое.

Напомню, что вычисление замкнутых SLL-выражение обладает следующим свойством. Если выражение $e_1/\{v := e_2\}$ замкнутое, то:

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

Суперкомпилятор SC Mini ограничивает размер конфигураций в дереве и строит сворачиваемые (в граф) деревья так: если при конструировании дерева возникает конфигурация e, являющаяся вызовом функции,[19] размер которой больше чем `sizeBound`, то эта конфигурация представляется в виде $e = e_1/(v := e_2)$, где e_2 — самое большое по размеру подвыражение конфигурации e, а e_1 — это выражение e, из которого «вынули» e_2 и заменили на переменную v. Для представления такого разделения будем использовать `let`-выражение $let\ v = e_2\ in\ e_1$.[20] Затем конфигурации e_1 и e_2 рассматриваются отдельно.

Конфигурация e_1 является по отношению к исходной конфигурации e более общей. А e по отношению к e_1 является, соответственно, частным случаем (instance). Описанный выше шаг при конструировании дерева конфигураций называется *обобщением* (generalization).

Теперь, если мы ограничим рост конфигураций (по размеру), для любого задания мы можем построить (потенциально бесконечное) дерево конфигураций, которое будет *гарантированно* сворачиваться в *конечный* граф конфигураций.

Например, если мы поставим максимальный размер выражения `sizeBound` равным 5, то в результате обобщения самой правой конфигурации в рассматриваемом примере получим дерево, изображенное на рис. 7.9.

Упражнение 8 Какими свойствами должно обладать задание, чтобы дерево конфигураций было сворачиваемым и без обобщений?

Упражнение 9 Интерпретатор `intTree` очень просто дополняется обработкой конструкции `let`. Посмотрите, как это сделано в приложении.

[18] Рассмотрим переименование как отношение эквивалентности, разбивающее множество выражений на классы эквивалентности. Множество SLL-выражений, в которых присутствуют конструкторы и функции только из программы `p` (= конечная сигнатура) и размер которых ограничен константой, разбивается на *конечное число* классов.

[19] Когда мы встречаем конструктор, то осуществляем декомпозицию, тем самым уменьшая размер конфигураций.

[20] Конструкция `let v = e2 in e1` – синтаксический трюк, чтобы машина $\mathcal{M}_p$ в качестве следующих шагов вернула `e1` и `e2`, а не `e1 / (v := e2)`.

Рис. 7.8. Дерево конфигураций для задания `(add'(x, y), prog1)`

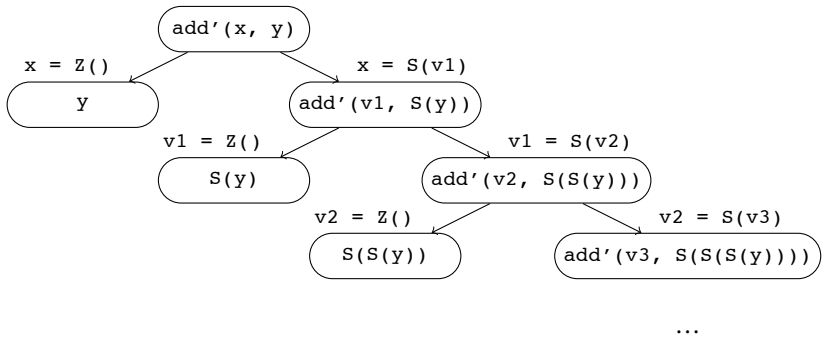
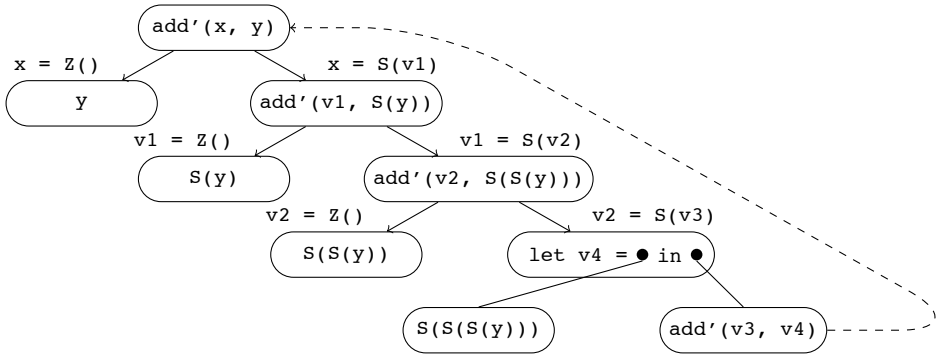


Рис. 7.9. Дерево конфигураций для задания `(add'(x, y), prog1)`



Идея обобщения конфигураций — очень важная составляющая суперкомпиляции. Мы рассмотрели, пожалуй, самый примитивный способ обобщения.

Часть суперкомпилятора, определяющая когда нужно сделать обобщение, на жаргоне суперкомпиляции исторически называется *свистком*. Свистки в большинстве существующих суперкомпиляторов устроены гораздо сложнее, чем рассмотренный здесь свисток.

Термин «свисток» (whistle), несмотря на низкий слог, всем полюбился и стал общепринятым. Свисток является эвристикой, главная задача которой — сигнализировать об опасности появления в дереве конфигураций бесконечных (несворачиваемых) ветвей. Сама задача точного распознавания бесконечных ветвей является алгоритмически неразрешимой (сводится к проблеме останова). Любой свисток лишь делает приближенную оценку и может ошибиться — просвистеть в «нескольких шагах» от хорошего зацикливания.

7.3.7. Прообраз суперкомпилятора

Итак, мы добились того, что для любого задания `(e, p)` мы умеем конструировать конечный граф конфигураций `g`. В графе `g` содержится вся необходимая информация для вычисления задания с *любыми* аргументами. В требованиях к оптимизатору в разделе 7.3.1 мы поставили задачу получить новое задание `(e', p')` такое, что для любых аргументов `s`:

```
sll_run (e, p) s = sll_run (e', p') s
```

В итоге мы конструируем следующий преобразователь программ (см. приложение):

```
transform :: Task → Task
transform (e, p) =
    residuate $ foldTree $ buildFTree
        (driveMachine p) e
```

Функция `buildFTree` строит сворачиваемое (foldable) дерево. И внутри нее, можно сказать, и происходит самое интересное: машина $\mathcal{M}_p$ не просто запускается — результат ее запуска анализируется и, возможно, принимается решение сделать обобщение.

Можно сказать, что данный преобразователь программ формально подходит под определение суперкомпилятора из цитаты (см. раздел 7.1). Хотя этот преобразователь практически ничего не делает, в нем уже есть основа для будущего суперкомпилятора. Будем использовать этот преобразователь как своеобразную точку отсчета, с которым мы сравним описываемые далее преобразователи `deforest` и `supercompile`.

Упражнение 10 Попробуйте показать, что для любого задания `(e, p)` и для любой подстановки `s` вычисление `sll_run (e', p') s` (где `(e', p')` — соответствующее остаточное задание) требует ровно столько же шагов интерпретатора, что и вычисление `sll_run (e, p) s`. То есть преобразователь `transform` не ухудшает, но и не улучшает задание с точки зрения производительности.

Но изменяется ли в ходе данного преобразования программа? Да, вот пример:[21]

$$\texttt{(even(sqr(x)), prog1)} \xrightarrow{transform} \texttt{(f1(x), prog1T)}$$

Где `prog1T`:

[21] Остаточная программа приведена здесь точно в таком виде, как она выдаётся преобразователем `transform`. В качестве имен новых функций используются идущие в некотором порядке идентификаторы. Конечно, было бы замечательно, если бы при такого рода преобразованиях функциям давались понятные имена.

```
f1(x) = g2(x, x);
g2(Z(), x) = f3();
g2(S(v1), x) = g4(x, x, v1);
f3() = True();
g4(Z(), x, v1) = g5(v1, x);
g4(S(v2), x, v1) = f7(v2, v1, x);
g5(Z(), x) = f6();
g5(S(v2), x) = g4(x, x, v2);
f6() = True();
f7(v2, v1, x) = g8(v2, v1, x);
g8(Z(), v1, x) = g9(v1, x);
g8(S(v3), v1, x) = f16(v3, v1, x);
g9(Z(), x) = f10();
g9(S(v3), x) = g11(x, x, v3);
f10() = False();
g11(Z(), x, v3) = g9(v3, x);
g11(S(v4), x, v3) = f12(v4, v3, x);
f12(v4, v3, x) = g13(v4, v3, x);
g13(Z(), v3, x) = g14(v3, x);
g13(S(v5), v3, x) = f7(v5, v3, x);
g14(Z(), x) = f15();
g14(S(v5), x) = g4(x, x, v5);
f15() = True();
f16(v3, v1, x) = g17(v3, v1, x);
g17(Z(), v1, x) = g18(v1, x);
g17(S(v4), v1, x) = f7(v4, v1, x);
g18(Z(), x) = f19();
g18(S(v4), x) = g4(x, x, v4);
f19() = True();
```

Видно, как минимум, одно свойство такого преобразования — в программе теперь нет вложенных вызовов функций: получилась «плоская», более «бесчеловечная» программа.[22]

7.3.8. КМП-тест

В двух следующих разделах мы добавим к преобразователю `transform` два «трюка», которые являются неотъемлемыми составляющими любого суперкомпилятора, и именно они «ускоряют» остаточное задание. Каждый из трюков можно было бы разобрать на отдельных небольших примерах, однако именно вместе они дают «синергетический эффект». И этот эффект сложно прочувствовать, рассматривая совсем игрушечные программы. Так называемый КМП-тест,[23] рассматриваемый далее, является относительно «реалистичной» программой.

Это тест позволит не только ощутить эффект трюков, но и почувствовать «глубину» преобразования, осуществляемого суперкомпилятором. Один из простых методов оценки оптимизирующего преобразования — посмотреть, может ли преобразование породить некоторый общеизвестный эффективный алгоритм из «наивного» и неэффективного. Эффект суперкомпиляции хорошо демонстрируется на примере генерации из наивного алгоритма сопоставления с образцом и зафиксированного образца эффективного алгоритма сопоставления — такого же, как выдаёт алгоритм Кнута — Морриса — Пратта [16] (отсюда и название).[24]

Рис. 7.10. `prog2`: поиск подстроки в строке

```
match(p, s) = m(p, s, p, s);

-- matching routine
-- current pattern is empty, so match succeeds
m("", ss, op, os) = True();
-- proceed to match first symbol of pattern
m(p:pp, ss, op, os) = x(ss, p, pp, op, os);

-- matching of the first symbol
-- current string is empty, so match fails
x("", p, pp, op, os) = False();
-- compare first symbol of pattern with first
-- symbol of string
x(s:ss, p, pp, op, os) =
  if(eq(p, s), m(pp, ss, op, os), n(os, op));

-- failover
-- current string is empty, so match fails
n("", op) = False();
-- trying the rest of the string
n(s:ss, op) = m(op, ss, op, ss);

-- equality routines
eq('A', y) = eqA(y);  eqA('A') = True();
eqB('A') = False();
eq('B', y) = eqB(y);  eqA('B') = False();
eqB('B') = True();
-- if/else
if(True(), x, y) = x;
if(False(), x, y) = y;
```

Рассмотрим программу на рис. 7.10:[25] функция `match(p, s)` проверяет, содержится ли строка `p` в строке `s`. Для упрощения мы рассматриваем только строки над алфавитом из двух символов — `'A'` и `'B'`. Строки представляются списком символов. Далее для экономии места будем использовать стандартные сокращения для записи строк и списков.

Рассмотрим проверку `match(«AAB», s)`, содержится ли подстрока (паттерн) «AAB» в строке `s`. Наша программа будет вычислять такое задание следующим образом: сравнивает `'A'` с первым символом строки `s`, `'A'` — со вторым, `'B'` — с третьим. Если какое-то сравнение заканчивается неудачей, перезапускает серию сравнений для хвоста строки. Однако такая стратегия совсем не оптимальна. Допустим, что строка `s` начинается с «AAA...». Первые два сравнения будут успешными, последнее завершится неудачей. Неэффективно повторять это с хвостом «AA...», так как заранее известно, что первые два сопоставления «AAB» с «AA...» завершаются успехом. Детерминированный конечный автомат, получаемый по алгоритму Кнута — Морриса — Пратта, рассматривает каждый символ строки `s` ровно один раз. В итоге (с помощью преобразователя `supercompile`) мы получим задание, соответствующее такому ДКА.

Наше простое преобразование `transform` дает следующий результат:

[22] Эту программу легче исполнить компьютеру (она в «итеративной» форме) и сложнее понять человеку (вы могли бы догадаться, что эта программа делает?).

[23] Ставший в своем роде классическим примером в суперкомпиляции, так как на этом примере оказалось легче всего сравнивать суперкомпиляцию с другими методами преобразований, такими как дефорестация, частичные вычисления и т. п. [26, 28].

[24] Сам алгоритм КМП в общем случае случае в данном примере не выводится, выводится лишь его экземпляр для конкретного паттерна. То есть происходит эффективная специализация задания.

[25] Автор извиняется за короткие имена функций — они укорочены только для того, чтобы следующие далее графы конфигураций были разумной ширины.

```
(match("AAB", s), prog2)
   transform
  ⟹ (f1(s), prog2T)
```

где `prog2T`:

```
f1(s) = f2(s);
f2(s) = g3(s, s);
g3("", s) = False();
g3(v1:v2, s) = f4(v1, v2, s);
f4(v1, v2, s) = g5(v1, v2, s);
g5('A', v2, s) = f6(v2, s);
g5('B', v2, s) = f22(v2, s);
f6(v2, s) = f7(v2, s);
f7(v2, s) = g8(v2, s);
g8("", s) = False();
g8(v3:v4, s) = f9(v3, v4, s);
f9(v3, v4, s) = g10(v3, v4, s);
g10('A', v4, s) = f11(v4, s);
g10('B', v4, s) = f20(v4, s);
f11(v4, s) = f12(v4, s);
f12(v4, s) = g13(v4, s);
g13("", s) = False();
g13(v5:v6, s) = f14(v5, v6, s);
f14(v5, v6, s) = g15(v5, v6, s);
g15('A', v6, s) = f16(v6, s);
g15('B', v6, s) = f18(v6, s);
f16(v6, s) = g17(s);
g17("") = False();
g17(v7:v8) = f2(v8);
f18(v6, s) = f19(v6, s);
f19(v6, s) = True();
f20(v4, s) = g21(s);
g21("") = False();
g21(v5:v6) = f2(v6);
f22(v2, s) = g23(s);
g23("") = False();
g23(v3:v4) = f2(v4);
```

Данный большой по объему результат приведен только для того, чтобы послужить «точкой отсчета». Отметим, что в преобразованной программе нет вложенных вызовов функций, но символы из `s`, как и в исходном задании, рассматриваются несколько раз.

7.3.9. Устранение транзитных шагов

Очень часто в графах конфигураций есть участки следующего вида:

```
... c1 → c2 → c3 ...
```

где шаг `c2 → c3` является транзитным (понятие транзитного шага было определено в разделе 7.3.3). такой участок можно заменить на:[26]

```
... c1 → c3 ...
```

Например, верхняя часть графа, построенного для нашего КМП-теста преобразователем `transform`, выглядит как показано на рис. 7.11.

Удаляемые транзитные шаги показаны зигзагами. Если от них избавиться, то верхушка «подчищенного» графа будет выглядеть как на рис. 7.12.

Преобразователь `deforest` — модификация преобразователя `transform`, удаляющая транзитные дуги и соответствующие узлы из графа конфигураций (см. приложение).

[26] Это корректно для вычислений языка SLL, где нет никаких побочных действий. Если бы на шаге $c_1 \to c_2$ совершался бы, допустим, ввод-вывод, то удаление этого шага было бы некорректным.

Пропустим наш тест через дефорестатор:

```
(match("AAB", s), prog2)
   deforest
  ⟹ (f1(s), prog2D)
```

где `prog2D =`

```
f1(s) = g2(s, s);
g2("", s) = False();
g2(v1:v2, s) = g3(v1, v2, s);
g3('A', v2, s) = g4(v2, s);
g3('B', v2, s) = g10(s);
g4("", s) = False();
g4(v3:v4, s) = g5(v3, v4, s);
g5('A', v4, s) = g6(v4, s);
g5('B', v4, s) = g9(s);
g6("", s) = False();
g6(v5:v6, s) = g7(v5, v6, s);
g7('A', v6, s) = g8(s);
g7('B', v6, s) = True();
g8("") = False();
g8(v7:v8) = f1(v8);
g9("") = False();
g9(v5:v6) = f1(v6);
g10("") = False();
g10(v3:v4) = f1(v4);
```

В дефорестированной программе в 2 раза меньше функций, чем в просто преобразованной (10 вместо 23) – были устранены вызовы «промежуточных» функций (грубо говоря, произошел инлайнинг). Хоть остаточная программа и упростилась значительно, мы пока еще не добились желаемого эффекта.

Удаление транзитных дуг есть упрощение графа конфигураций. Это упрощение — *один из двух* основных механизмов оптимизации с помощью суперкомпиляции. Наиболее эффективным он оказывается, когда применяется вместе со вторым механизмом — распространением информации.

Упражнение 11 Можно устранять транзитные шаги не из графа конфигураций, а сразу из дерева (до свертки). Какие очень нехорошие «подводные камни» есть у этого варианта?

Немного о дефорестации

Есть отдельный метод преобразования программ под названием *дефорестация* [37, 3]. Цель дефорестации: уменьшить создание промежуточных структур данных — списков, деревьев (отсюда и название) и т. д., которые возникают при композиции функций. В работе [3] рассматривается дефорестация для языка SLL.[27] Классическая дефорестация происходит без обобщения — рассматриваются программы в специальной синтаксической форме (которая гарантирует, что дерево «свернется»). Практическое достоинство дефорестации — то, что для многих синтаксических форм есть так называемая сокращенная дефорестация (shortcut deforestation), для которой можно сразу выписать результат преобразований.

7.3.10. Распространение информации

Рассмотрим граф конфигураций для нашего КМП-теста, порождаемый преобразователем `deforest` и изображенный на рис. 7.13.

В нем есть шаги, где разбирается строка `s` (пустая/непустая — подчеркнуто), а дальше — ниже по дереву — строка `s` опять разбирается (дважды подчеркнуто), хотя уже сделано предположение, что список — непустой. Такое двойное тестирование избыточно.

[27] Правда, там ему не дается никакого имени.

Рис. 7.11. Верхняя часть графа КМП-теста, транзитные шаги отмечены зигзагами

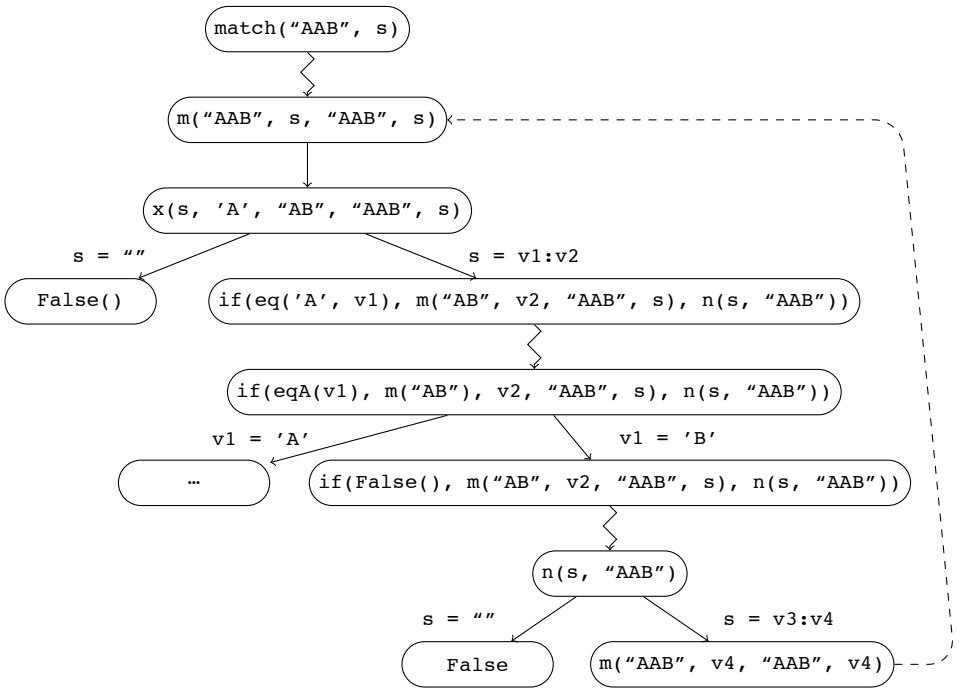


Рис. 7.12. Верхняя часть графа КМП-теста после удаления транзитных шагов

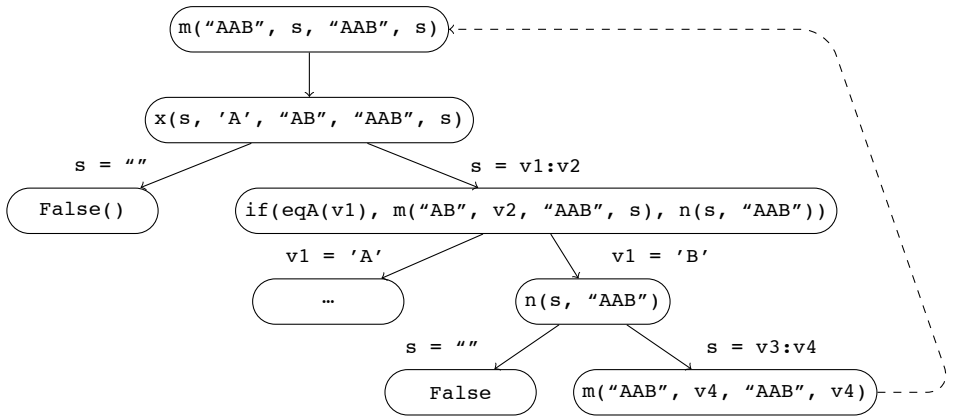
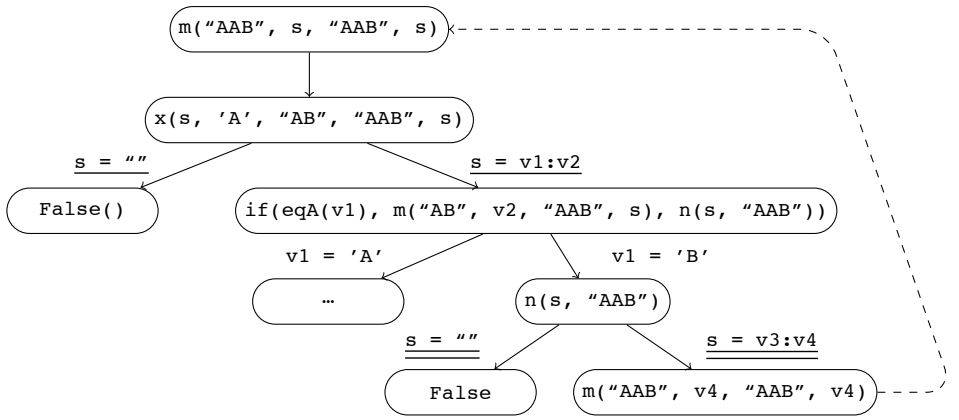


Рис. 7.13. Граф конфигураций КМП-теста после преобразования `deforest`



В суперкомпиляции такая избыточность устраняется так: всякий раз, когда мы разбираем различные условия — тестируем какую-то свободную переменную в выражении, мы распространяем результат тестирования дальше (вниз по соот-

ветствующему поддереву).

Распространим информацию о том, что строка `s` — не пустая: получится граф на рис. 7.14.

Выигрыш не только в том, что мы избавились от лишней проверки, но мы также выявили транзитный переход, который можно затем удалить. В результате получаем граф, верхушка которого показана на рис. 7.15 (граф приведен полностью в приложении).

Преобразователь `supercompile` отличается от преобразователя `deforest` тем, что на каждом шаге прогонки результат тестирования распространяется на все рассматриваемое состояние. В языке SLL предусмотрены проверки только одного вида — что аргумент соответствует некоторому образцу. Соответственно, мы можем распространить информацию только одного вида — что некоторая конфигурационная переменная соответствует образцу. Это называется *распространением позитивной информации*. Соответственно, суперкомпилятор SC Mini — позитивный суперкомпилятор. Если бы в нашем языке были проверки, что аргумент *не соответствует* образцу, мы могли бы распространять и негативную информацию (о несоответствии образцу). Суперкомпилятор, распространяющий и позитивную, и негативную информацию, называется перфектным.[28]

Результат суперкомпиляции нашего КМП-теста таков:

```
(match("AAB", s), prog2)
  ==supercompile==> (f1(s), prog2S)
```

где `prog2S =`

```
f1(s) = g2(s);
g2("") = False();
g2(v1:v2) = g3(v1, v2);
g3('A', v2) = g4(v2);
g3('B', v2) = f1(v2);
g4("") = False();
g4(v3:v4) = g5(v3, v4);
g5('A', v4) = f6(v4);
g5('B', v4) = f1(v4);
f6(v4) = g7(v4);
g7("") = False();
g7(v5:v6) = g8(v5, v6);
g8('A', v6) = f6(v6);
g8('B', v6) = True();
```

Желаемый эффект достигнут — каждый символ из строки `s` теперь рассматривается не более одного раза.

Упражнение 12 Докажите это утверждение.

Упражнение 13 В получившейся программе есть «транзитные» функции – `f1`, `f6`. Можно безопасно осуществить их инлайнинг – везде в программе заменить `f1`, `g2` и `f6` на `g7`, а затем выкинуть `f1` и `f6`. Посмотрите внимательно на граф конфигураций, получившийся при суперкомпиляции КМП-теста, приведенный в приложении. Почему SC Mini не осуществил инлайнинг в этом случае?

Если разрешить в языке SLL использовать вложенные образцы (как это делается в Хаскеле), то получившийся результат можно переписать как `(match(s), prog2S')`, где `prog2S'`:

```
match("") = False();
match('A':s) = matchA(s);
match('B':s) = match(s);
matchA("") = False();
matchA('A':s) = matchAA(s)
matchA('B':s) = match(s);
matchAA("") = False();
matchAA('A':s) = matchAA(s);
matchAA('B':s) = True();
```

По такой программе строится конечный автомат, приведенный на рис. 7.16.

Распространение информации — второй механизм оптимизации с помощью суперкомпиляции (первый — упрощение графа конфигураций через удаление удаление транзитных дуг).

Распространение информации имеет тройной эффект:

1) Переменная не тестируется дважды — очевидная оптимизация.
2) Как следствие, появляются транзитные дуги, которые затем удаляются.
3) В графе конфигураций исчезают ветви, по которым никогда не пойдет вычисление — в остаточной программе удаляется «мертвый» код.

В рассмотренном выше примере в графе, построенном преобразователем `deforest`, есть такой путь:

```
{s = v1:v2}  → ... → {s = ""} →
```

Ясно, что никакое конкретное вычисление не может пройти по этому пути. Однако при дефорестации этот путь переходит в остаточную программу — как следствие, в дефорестированной `prog2d` (см. выше) программе есть функция `g10`, первый аргумент которой никогда не может быть пустой строкой:

```
g10("") = False();
g10(v3:v4) = f1(v4);
```

В результате суперкомпиляции КМП-теста получилась программа, в которой отсутствует мертвый код. Однако в общем случае получить программу без мертвого кода нельзя — это алгоритмически неразрешимая задача.[29]

Упражнение 14 В большинстве работ по суперкомпиляции распространение информации рассматривается как обязательная операция. В SC Mini распространение информации выделено как отдельный логический шаг — модификация машины $\mathcal{M}_p$:

```
addPropagation :: Machine Conf → Machine Conf
```

7.3.11. Квинтэссенция суперкомпилятора SC Mini

Данная глава получилась очень длинной, поэтому будет полезно взглянуть на изложенный в ней материал под несколько иным углом «для закрепления пройденного».

Суперкомпилятор SC Mini является попыткой описания минимального суперкомпилятора для программиста на Хаскеле. Главная цель SC Mini — внятно выделить основные составляющие, присутствующие в любом суперкомпиляторе, и показать, как эти составляющие сочетаются друг с другом.

Давайте проследим, как легким движением руки преобразователь `transform` (не улучшающий, но и не ухудшающий) превращается в `deforest`, который, в свою очередь, превращается в `supercompile`.

Безликий преобразователь `transform`:

[28] Чаще всего встречается промежуточный вариант, когда распространяется только часть негативной информации.

[29] То есть невозможно сконструировать преобразователь, который бы для *любого* SLL-задания выдавал программу без лишнего кода. Лишний код появляется в результате обобщения.

Рис. 7.14. Граф конфигураций КМП-теста после распространения информации о непустоте `s`

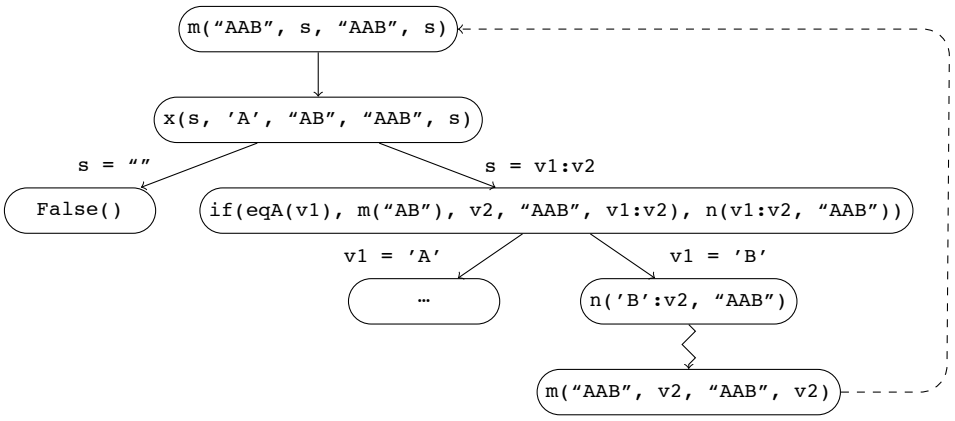


Рис. 7.15. Граф конфигураций КМП-теста после распространения информации о непустоте `s` и удаления транзитного перехода

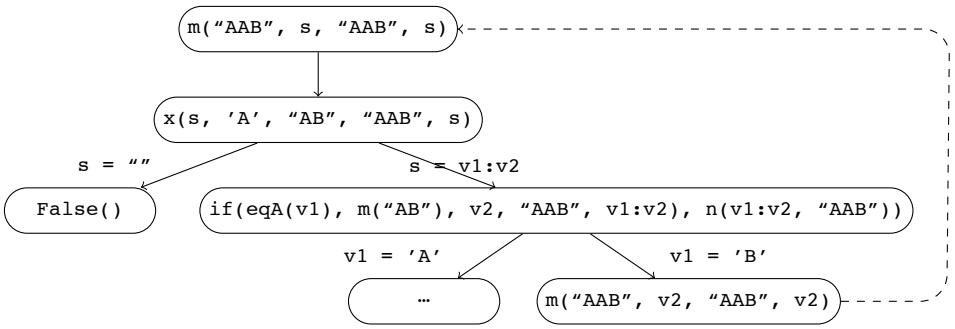
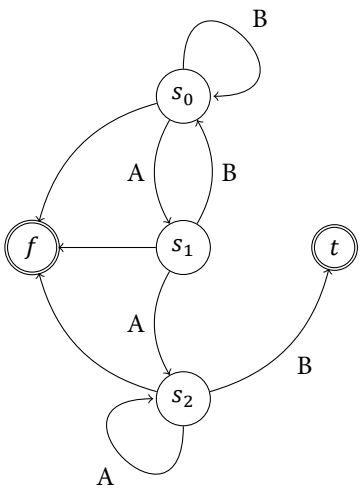


Рис. 7.16. Конечный автомат, соответствующий результату суперкомпиляции КМП-теста




```
transform :: Task → Task
transform (e, p) =
    residuate $ foldTree $
        buildFTree (driveMachine p) e
```

Добавляем упрощение графа конфигураций посредством удаления транзитных шагов:

```
deforest :: Task → Task
deforest (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (driveMachine p) e
```

Добавляем распространение информации:

```
supercompile :: Task → Task
supercompile (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (addPropagation
$ driveMachine p) e
```

За счет распространения результатов тестов на всю конфигурацию и дальнейшего упрощения графа конфигураций наш минимальный суперкомпилятор SC Mini оказался способен свести неоптимальный наивный алгоритм поиска подстроки в строке к хорошо известному эффективному КМП-алгоритму.

7.3.12. Не только оптимизация

Суперкомпилятор SC Mini способен доказывать теоремы о программах. Покажем, что операция `add`, определенная в программе `prog1`, ассоциативна, то есть для любых аргументов `args`:

```
sll_run (add(add(x,y),z), p1) args ==
        sll_run (add(x,add(y,z), p1) args
```

Можно доказать это по индукции (рассматривая все правила операционной семантики SLL). А можно рассмотреть преобразования этих заданий. Суперкомпилятор SC Mini приводит оба эти задания к одной и той же *текстовой* форме:

```
(add(add(x, y), z), p1) ⟹ (g1(x, y, z), prog3S')
(add(x, add(y, z)), p1) ⟹ (g1(x, y, z), prog3S')
```

где `prog3S'`:

```
g1(Z(), y, z) = g2(y, z);
g1(S(v1), y, z) = S(g1(v1, y, z));
g2(Z(), z) = z;
g2(S(v1), z) = S(g2(v1, z));
```

Отсюда сразу следует ассоциативность операции `add`, так как:

```
sll_run (add(add(x, y), z), p1) args ==
  sll_run (g1(x, y, z), prog3S') args ==
    sll_run (add(x, add(y, z)), p1) args
```

7.4. Проблемы

В этом разделе я постараюсь выявить основные недостатки нашего суперкомпилятора с точки зрения промышленного программирования. В той или иной степени все перечисленные моменты присутствуют во всех существующих суперкомпиляторах.

7.4.1. Непредсказуемость результатов

Рассмотренный КМП-тест был на самом деле, что называется, образцово-показательным примером. Нам повезло в том, что свисток не сработал и удалось построить граф конфигураций без обобщений.

Однако в большинстве других случаев размер конфигураций в графе по мере его построения будет увеличиваться, и рано или поздно придется делать обобщение — чтобы гарантировать завершаемость.

Вернемся к ранее рассмотренному заданию:

```
(even(sqr(x)), prog1)
```

При дефорестации этой программы удается построить граф конфигураций, не прибегая к обобщениям. При суперкомпиляции же этого задания без обобщения не обойтись — это плата за распространение информации: распространяя информацию, мы «раздуваем» и усложняем конфигурации. В приложении даны результаты дефорестации и суперкомпиляции этого задания, а также сравнение скорости работы дефорестированных и суперкомпилированных программ. На основе материала, приведенного в приложении, делаются следующие выводы:

1) Размер суперкомпилированной программы намного больше, чем размер дефорестированной программы.

2) Для малых чисел `x` суперкомпилированная программа работает быстрее, чем дефорестированная. Однако для больших чисел `x` суперкомпилированная программа работает медленнее.

Это — антипод КМП-теста: отрицательный пример, показывающий основные слабости суперкомпилятора SC Mini. (Будем для краткости называть его анти-КМП-тест. Такой пример при желании находится для любого из существующих суперкомпиляторов.)

Если с тем, что суперкомпилятор может упускать некоторые возможности оптимизации, мириться можно, то с тем, что суперкомпилятор склонен раздувать остаточную программу, мириться труднее. Опасность *разбухания остаточного кода* (code explosion) — большая проблема суперкомпиляции. В нашем случае мы ограничиваем размер выражений в узлах графа конфигураций — то есть, в какой-то степени ограничиваем рост графа «в ширину» и гарантируем завершаемость. Однако мы не ограничиваем рост «в глубину» — граф (даже после удаления транзитных дуг) получается очень глубоким и, как следствие — раздутая остаточная программа.

Упражнение 15 Попробуйте изменить SC Mini, ограничив рост графа в глубину. Что получится?

Упражнение 16 Можно ли оценить размер графа для конкретной программы?

Самое плохое, что непредсказуемость суперкомпилятора может заметно проявляться даже для небольших программ.

Суперкомпилятор SC Mini всегда выдаёт программу, которая не менее эффективна, чем исходная, если считать эффективность в шагах интерпретатора. Однако эффективность заключается не только в скорости работы, но и в использовании памяти. Практически все суперкомпиляторы могут в некоторых случаях выдавать программы, которые будут быстрее, чем исходные, но будут потреблять намного больше памяти.

7.4.2. Плохая масштабируемость

Суперкомпилятор имитирует поведение программы, стараясь учесть взаимодействие всех частей программы. К сожалению, размер модели — графа конфигураций — очень нелинейно зависит от размера программы. То есть, увеличение размера исходной программы на 30% может привести к тому, что суперкомпилятор будет работать в десятки раз дольше.

Упражнение 17 Попробуйте найти такие примеры для SC Mini.

Более того, у подавляющего большинства суперкомпиляторов время работы нелинейно зависит от размера графа конфигураций.

Упражнение 18 Покажите, что последнее утверждение верно и для суперкомпилятора SC Mini.

А это значит, что суперкомпиляция больших программ — весьма непредсказуемый процесс: в худшем случае суперкомпилятор будет очень долго работать и выдаст огромную программу, которая будет работать ненамного быстрее, чем исходная.

Очевидная область исследований — суперкомпиляция программы «по частям», при которой программа рассматривается как совокупность модулей, которые суперкомпилируются по отдельности. К сожалению, это пока еще никто не исследовал.

7.4.3. А как отлаживаться?

Суперкомпиляция преобразует исходные тексты программ — то, с чем потом работает обычный компилятор (или интерпретатор). Можно ли отлаживать суперкомпилированный код? Конечно, можно, если об этом заранее позаботиться. Другое дело, что чтобы обеспечить возможность такой отладки, потребуется сделать большой объем технической работы. Ясно, что пока нет востребованного суперкомпилятора, мало кто захочет такими исследованиями заниматься.

7.4.4. Детали, детали, детали…

Мы рассмотрели очень маленький, «игрушечный» суперкомпилятор для игрушечного языка, который обладает заметными недостатками. Сконструировать суперкомпилятор для игрушечного языка, лишенный перечисленных недостатков —

большая и серьезная, можно даже сказать, фундаментальная задача, достойная научной степени.[30]

Если говорить о суперкомпиляторах для промышленных языков программирования, то сразу же возникает масса деталей, которые ведут к усложнениям.

Глобальное состояние, побочные эффекты Рассмотренный суперкомпилятор использует предположение о композициональности вычислений, позволяющее корректно делать обобщение:

$$\mathcal{J}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{J}_p[\![e_1/\{v := \mathcal{J}_p[\![e_2]\!]\}]\!]$$

Выражение e_2 можно вычислить вне контекста, в котором оно используется.

Однако если в языке есть глобальное состояние и/или побочные эффекты (а в каком промышленном языке этого нет?), то понятие композициональности в лучшем случае усложняется, а в худшем — теряет всякий смысл. Как с этим бороться? — необходим дополнительный анализ, когда можно распространять информацию, а когда — нет.[31] Еще сложнее становится, когда появляется многопоточность. Однако сейчас наблюдается такая положительная тенденция в современных (прогрессивных) языках программирования, как возможность выделения «функционального кода» (например, это уже сделано в языке D 2.0 и планируется сделать в Scala). Самое распространенное и прямолинейное решение для суперкомпилятора — оставлять «опасные» куски кода без изменений. В таком случае оптимизируется только чистый функциональный код.

Борьба со строгостью и ленивостью Суперкомпилятор SC Mini оказался таким простым изнутри, потому что мы выбрали простую и элегантную (с точки зрения теории) семантику для языка SLL – ленивые вычисления с вызовом по имени (call-by-name). Именно поэтому прогонка (моделирование интерпретатора) оказалась достаточно простой. Однако семантика вызова по имени плоха с практической точки зрения и почти не используется в современных языках программирования. В мире функционального программирования используются семантики call-by-value и call-by-need. Суперкомпилятор, сохраняющий все тонкости вычисления для таких семантик, устроен намного сложнее.

Упражнение 19 Напишите интерпретатор SLL для семантики call-by-value. Найдите пример программы, смысл которой исказится после суперкомпиляции суперкомпилятором SC Mini.

Упражнение 20 Напишите интерпретатор SLL для семантики call-by-need. Насколько он оказался сложнее?

7.4.5. Почему же ей все-таки занимаются?

Букет проблем суперкомпиляции немал, результаты (пока что) непредсказуемы. Почему же тогда интерес к суперкомпиляции в последние три года существенно растет?

Во-первых, потому, что под определенным углом зрения многие методы оптимизации программ являются частным случаем суперкомпиляции. Это и уже упомянутая дефорестация, и частичные вычисления, и сплавка (fusion) разных видов, и инлайнинг, и дефункционализация, и т. д.[32] Это сильно впечатляет, когда простой суперкомпилятор может обрабатывать некоторые программы не хуже, чем сложный узкоспециализированный инструмент.

Во-вторых, из-за концептуальной простоты — идея суперкомпиляции не привязана к какому-то конкретному языку программирования. Она не привязана даже к группе языков, хотя и разработана на нынешний момент лучше всего для функциональных языков.

В-третьих, из-за многогранности идеи — идеи суперкомпиляции применимы не только к оптимизации программ, но и к анализу программ [17, 18, 15, 40]. Очень заманчива перспектива получить с помощью суперкомпиляции «два в одном» — оптимизацию и анализ программ в одном флаконе.

В-четвертых, из-за философии и фигуры автора идеи суперкомпиляции — В. Ф. Турчина.[33] Суперкомпиляция — лишь верхушка большого айсберга, в центре которого находится теория метасистемного перехода. Об этом — кратко в следующем разделе.

7.5. История вопроса

Валентина Федоровича Турчина (1931—2010) — автора идеи суперкомпиляции — я бы кратко охарактеризовал как Программист-Философ.

«В. Ф. Турчин родился в 1931 г. в Москве. Окончил физический факультет МГУ и с 1953 по 1964 г. работал под Москвой в Обнинске в Физико-энергетическом институте, где изучал рассеяние медленных нейтронов в жидкостях и твердых телах и защитил докторскую диссертацию. В 33 года он уже был известным физиком-теоретиком с большими перспективами. И тем не менее в 1964 г. В. Ф. Турчин оставляет физику, переходит в Институт прикладной математики АН СССР (ныне Институт им. М. В. Келдыша) и погружается в информатику. … Он оставил науку ради метанауки.[34]»

«В разделе “Сумасшедшие теории и метанаука” я высказал мысль, что для того, чтобы разрешить трудности в современной теории элементарных частиц, надо разработать методы “метанауки”, т. е. теории о том, как строить теории. Причину я усматривал в том, что основные понятия физики на ранних стадиях ее развития брались из нашей интуиции макроскопического мира. Но для познания законов микромира (а точнее, для построения математических моделей этого мира) наша “макроскопическая” интуиция неадекватна. Если интуиция не дает нам впрямую тех “колесиков”, из которых можно строить модели микромира, то нам нужны какие-то теории о том, как эти колесики выбирать и как модели строить. Это и будет метанаука.[35]»

В 1966 году Турчин придумывает РЕФАЛ[36] — язык, сильно отличающийся от существовавших тогда (да и от существующих ныне тоже) языков программирования и предназначенный в первую очередь для описания и обработки других языков. Не буду вдаваться в детали языка РЕФАЛ — он описан в

[30] Речь, конечно же, не идет о тривиальных суперкомпиляторах, не меняющих программу: $Sc\ p = p$.

[31] В случае языка SLL распространяемая информация не меняется, однако в случае изменяемого состояния необходимо следить, что распространяемая информация не изменилась.

[32] Во многих статьях это утверждение делается авторами искренно, но неформально. Было бы крайне интересно увидеть исследование, где это показывается убедительно — например чью-нибудь дипломную работу.

[33] По моему личному ощущению большинство специалистов по суперкомпиляции поставили бы эту причину в итоге на первое место.

[34] В. С. Штаркман Предисловие редактора к первому русскому изданию книги «Феномен науки» [44].

[35] В. Ф. Турчин. Предисловие к книге «Феномен науки», написанное в 1990 [44].

[36] REFAL = REcursive Functions Algorithmic Language.

отдельной статье в данном номере журнала ПФП. Турчин «заражает» своими идеями молодежь — начинает работать «подпольный» РЕФАЛ-семинар (каждую неделю по вторникам на квартире у С. А. Романенко, вплоть до 77 года).

Следующие 5—6 лет — работа над эффективной реализацией РЕФАЛа. Сначала над интерпретатором, а затем и над компилятором. Компилятор РЕФАЛа — программа на РЕФАЛе, обрабатывающая программы на РЕФАЛе, и выдающая программы на языке сборки. Но ничто не мешает выдавать программы не на языке сборки, а на том же РЕФАЛе. Так в 1970-е годы возникает идея прогонки, которую Турчин описывает как «эквивалентные преобразования функций на языке РЕФАЛ». В 1974 прогонка описывается уже в контексте теории метасистемных переходов.

В 77 году Турчин вынужден покинуть СССР, в том же году в США в переводе на английский публикуется один из главных трудов Турчина — книга «Феномен науки», написанная еще 1970-м году в СССР, сверстанная и уже готовая к печати к 1973-му, но так и не вышедшая в силу политических обстоятельств.

Живет и работает в Нью-Йорке, вначале в Courant Institute, затем в City College. С 1989 года, пока позволяет здоровье, ежегодно приезжает в Россию.

В этом году Валентину Федоровичу исполнилось бы 80 лет.

В истории суперкомпиляции можно условно выделить 3 периода.[37]

1970-е–1990-е. Суперкомпиляция РЕФАЛа С 1979 года Турчин публикует (с соавторами) не один десяток работ по суперкомпиляции и метавычислениям. Как мы сейчас понимаем, в этих работах рассыпано огромное количество интересных и плодотворных идей. Однако многие концепции описаны недостаточно формально, фрагментарно и чаще всего в терминах языка РЕФАЛ, поскольку для Турчина суперкомпиляция была неотделима от языка РЕФАЛ. К сожалению, для неподготовленного человека даже описание прогонки выглядело очень сложно и непонятно. Также, ни в одной работе, посвященной суперкомпиляции Рефала, алгоритм суперкомпиляции не приведен полностью. По этим причинам, несмотря на значительное количество опубликованных работ, к началу 1990-х суперкомпиляция не обрела признания за пределами узкого круга «посвященных». Более того, практически все основные составляющие суперкомпиляции описывались без должной доли формализма — зачастую туманно и расплывчато. Я соглашусь со следующим высказыванием о работах Турчина — оно применимо и к работам по суперкомпиляции: «В. Ф. практически не пишет ничего легкодоступного. Внимательно прочитавших и осмысливших [его] работы, можно и сейчас, наверное, пересчитать по пальцам[38]…» Важные работы этого периода: [43, 30, 31, 33, 34, 36, 35].

1990-е. Суперкомпиляция функциональных языков первого порядка Работа Андрея Климова и Роберта Глюка «Occam's Razor in Metacompuation: the Notion of a Perfect Process Tree» [5] о сущности прогонки — по сути, первая работа, нацеленная на понимание суперкомпиляции как общего метода — без привязки к языку Рефал. В 1994 Мортен Сёренсен сделал очень важную вещь — в своей магистерской диссертации «Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation» [26] пересказал и сформулировал основные идеи суперкомпиляции для простого функционального языка (SLL). Это была первая работа, в которой были описаны все существенные части суперкомпилятора. После этого последовала лавина работ, «объясняющих» суперкомпиляцию, сравнивающих ее с другими методами преобразований программ. Важные работы этого периода: [5, 26, 38, 27, 6, 29, 28, 39].

2000-е. Суперкомпиляция функциональных языков высшего порядка Первая половина нулевых — некоторое затишье. Затем — всплеск интереса к суперкомпиляции функций высших порядков. РЕФАЛ – язык первого порядка, работы девяностых годов также не затрагивали высший порядок. В случае высшего порядка открылось много интересных деталей, возможностей и проблем. Проводятся международные семинары (workshops) META-2008 [4] и META-2010 [25], главной темой которых является суперкомпиляция. Оценить важность работ этого периода можно будет лет через десять.

7.5.1. Текущие направления

Вначале предполагалось, что в данной статье будет два отдельных раздела — «Подходы» и «Текущие направления», где будет дан обзор того, что сейчас происходит в суперкомпиляции. Оказалось, что это достаточно сложно сделать: современный ландшафт суперкомпиляции очень пестр и многие работы посвящены достаточно узко специализированным темам. Деятельность, связанная с суперкомпиляцией, с одной стороны чрезвычайно многогранна. С другой стороны, многие интересные идеи там еще не выкристаллизовались, их объяснение было бы туманным и сложным. Поэтому я ограничусь кратким обзором существующих суперкомпиляторов (см. следующий раздел).

Можно с уверенностью обозначить лишь главный лейтмотив современных исследований — *создание практически полезного суперкомпилятора*.[39]

Любопытным читателям, желающим все-таки узнать полное положение дел в суперкомпиляции, рекомендую почитать соответствующие обзоры в диссертациях по суперкомпиляции: [26, 24, 23, 20, 42, 10, 40, 11].

7.6. Существующие суперкомпиляторы

Все перечисленные в данном разделе суперкомпиляторы являются экспериментальными и очень разными. Заинтересованный читатель может ознакомиться с литературой, ссылки на которую даются ниже.

SCP4[40] Суперкомпилятор SCP4 для языка РЕФАЛ-5 является своего родом итогом работ по суперкомпиляции РЕФАЛа. SCP4 использует свойства языка РЕФАЛ (ассоциативность конкатенации) и, помимо методов суперкомпиляции, включает дополнительные инструменты: распознавание частично рекурсивных константных функций, распознавание частично рекурсивных мономов конкатенации, нахождение и анализ выходных форматов. Основная информация о SCP4 собрана

[37] Следующее далее разделение на периоды и выделение наиболее значимых работ субъективны.

[38] Н. С. Работнов. Давно… В шестидесятые // «Индекс / Досье на цензуру» 1997/2

[39] Как сказали бы раньше, применимого в народном хозяйстве.

[40] http://www.botik.ru/pub/local/scp/refal5/refal5.html

в работе [42] и монографии [41]. Стоит отметить, что суперкомпилятор SCP4 может расширять область определения программы в следующем смысле: если на каких-то входных данных исходная программа завершалась с ошибкой или зацикливалась, то преобразованная программа может завершаться и выдавать некоторое значение.

Суперкомпилятор языка TSG[41] Язык TSG – упрощенный «плоский ЛИСП» (без вложенных вызовов функций). Помимо учебного суперкомпилятора языка TSG, описанного в [39], для языка TSG реализованы методы метавычислений, которые изначально возникли в контексте суперкомпиляции, но которым не уделялось (и, к сожалению, не уделяется сейчас) должного внимания — окрестностный анализ, окрестностное тестирование, инверсное программирование, нестандартные семантики (описаны в [38]).

Jscp[42] Суперкомпилятор для языка Java. Первый суперкомпилятор для нефункционального языка. Один из основных результатов работы: суперкомпилировать нефункциональные языки — сложно. Описан в [12]. Исходные тексты Jscp закрыты.

Суперкомпилятор языка Timber[43] Timber — чистый (pure) объектно-ориентированный язык с передачей параметров по значению. Суперкомпилятор реализован для функциональной части языка Timber. Главная цель проекта — добиться того же эффекта оптимизации, что и в случае суперкомпиляции для языка с передачей параметров по имени, полностью сохраняя семантику программы. Описан в [10, 11].

Supero[44] Суперкомпилятор для подмножества Хаскеля. Первый суперкомпилятор для языка с семантикой call-by-need. Главная цель — сохранить ленивость call-by-need при суперкомпиляции. Описан в [20, 21].

SPSC[45] SPSC — учебный суперкомпилятор для рассмотренного здесь языка SLL. Цель проекта — реализовать суперкомпилятор, соответствующий позитивному суперкомпилятору, описанному (но не реализованному) в работах [29, 28]. Описание SPSC дано в [14].

HOSC[46] HOSC — суперкомпилятор подмножества языка Хаскеля. В отличие от других суперкомпиляторов, главная цель которых — оптимизация программ, основная цель суперкомпилятора HOSC — анализ программ. Помимо обычной суперкомпиляции, суперкомпилятор HOSC выполняет двухуровневую суперкомпиляцию, основанную на выявлении и применении улучшающих лемм. Описан в [40, 13].

Optimusprime[47], **пакет на hackage**[48] Оптимизирует функциональные программы, которые предполагается выполнять на FPGA-программируемом процессоре (Reduceron). Описан в [22].

CHSC[49] Суперкомпилятор подмножества Хаскеля. Цель — встроить суперкомпиляцию в компилятор GHC. Главная особенность — в отличие от всех других суперкомпиляторов, не делает λ-лифтинг вложенных определений. Описан в [1].

Дистиллятор[50] В большинстве суперкомпиляторов конфигурации представлены в виде выражений со свободными переменными. В дистилляции конфигурации представлены графами (то есть возникают такие сложные конструкции как графы, к узлах которых находятся графы): свертка и обобщение выполняются на графах. Дистилляция описана в [7, 8].

7.7. Вместо заключения

Главная цель, которую я перед собой ставил: внятно, обозримо и модульно описать на языке Хаскель основные составляющие минимального суперкомпилятора и показать, как они соединяются вместе. В итоге получилось следующее:

```
supercompile :: Task → Task
supercompile (e, p) =
  residuate $ simplify $ foldTree $
    buildFTree (addPropagation $ driveMachine p) e
```

Данная же статья, по сути, лишь попытка объяснить, что делают эти составляющие, и какой, в итоге , может получиться эффект.

Я надеюсь, что читатель заглянет в приложение к данной статье, где приведен полный код суперкомпилятора SC Mini[51] с разъяснением некоторых технических моментов.

7.7.1. Куда двигаться дальше?

Если вас заинтересовала тема суперкомпиляции и вы не знаете с чего начать, я бы порекомендовал следующее:

1) Видеозаписи лекций[52] Сергея Абрамова по метавычислениям.

2) Блог Сергея Романенко,[53] посвященный суперкомпиляции.

3) В работах [28, 29] доступным языком описываются классические методы суперкомпиляции, оставшиеся за рамками данной статьи — использование в качестве свистка отношения гомеоморфного вложения (*homeomorphic embedding*), тесное обобщение конфигураций (*most specific generalization*).

7.7.2. Приглашение к сотрудничеству

История суперкомпиляции напоминает раскачивание маятника — от простого к сложному, от сложного к простому.

Первый период — суперкомпиляция РЕФАЛа — развитие от простого к сложному: сделать хоть какой-нибудь (сложный) суперкомпилятор, лишь бы он автоматически работал и выдавал разумные результаты. Второй период — суперкомпиляция функциональных языков первого порядка — осмысление полученных результатов, их схематизация и отчуждение. Нынешний период — это опять движение от простого (осмысленного) к сложному (пока еще непонятному). Это что касается практики.

С другой стороны, в области, близкой к суперкомпиляции — частичных вычислениях — очень важные результаты были получены при экспериментах на совсем игрушечных, не имеющих практической ценности, частичных вычислителях для игрушечных языков [9].

[41] http://www.botik.ru/ abram/mca/
[42] http://supercompilers.ru
[43] http://timber-lang.org/
[44] http://community.haskell.org/ ndm/supero/
[45] http://code.google.com/p/spsc/
[46] http://code.google.com/p/hosc/
[48] http://hackage.haskell.org/package/optimusprime
[49] https://github.com/batterseapower/chsc
[50] https://github.com/barnacle/dhc
[51] https://github.com/ilya-klyuchnikov/sc-mini
[52] http://vimeo.com/channels/metacomputation
[53] http://metacomputation-ru.blogspot.com

Одно из крайне интересных для меня направлений для исследований — проведение различного рода экспериментов на простых и понятных суперкомпиляторах. Проблема, что таких суперкомпиляторов пока нет. Данная статья — это попытка движения в этом направлении. Предложенный суперкомпилятор SC Mini невелик, но некоторые части в нем написаны не очень элегантным способом. Буду рад конструктивным предложениям и замечаниям по его улучшению.

Литература

[1] *Bolingbroke M., Peyton Jones S. L.* Supercompilation by evaluation // Haskell 2010 Symposium. — 2010.

[2] *Cousot P., Cousot R.* Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. — 1977.

[3] *Ferguson A., Wadler P.* When Will Deforestation Stop? // 1988 Glasgow Workshop on Functional Programming. — 1988.

[4] First International Workshop on Metacomputation in Russia / Program Systems Institute. — Pereslavl-Zalessky2008. — July 2–5.

[5] *Glück R., Klimov A.* Occam's razor in metacompuation: the notion of a perfect process tree // WSA '93: Proceedings of the Third International Workshop on Static Analysis. — London, UK: Springer-Verlag, 1993. — Pp. 112–123.

[6] *Glück R., Sørensen M. H.* A roadmap to metacomputation by supercompilation // Selected Papers From the International Seminar on Partial Evaluation. — Vol. 1110 of *LNCS*. — 1996. — Pp. 137–160.

[7] *Hamilton G. W.* Distillation: extracting the essence of programs // Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation / ACM Press New York, NY, USA. — 2007. — Pp. 61–70.

[8] *Hamilton G. W.* A graph-based definition of distillation // Second International Workshop on Metacomputation in Russia. — 2010.

[9] *Jones N. D., Gomard C. K., Sestoft P.* Partial evaluation and automatic program generation. — Prentice-Hall, Inc., 1993.

[10] *Jonsson P.* — Positive supercompilation for a higher-order call-by-value language. — Licentiate thesis, Luleå University of Technology, 2008.

[11] *Jonsson P.* Time- and Size-Efficient Supercompilation: Ph.D. thesis / Luleå University of Technology. — 2011.

[12] *Klimov A.* An approach to supercompilation for object-oriented languages: the java supercompiler case study // First International Workshop on Metacomputation in Russia. — 2008.

[13] *Klyuchnikov I.* Towards effective two-level supercompilation: Preprint 81: Keldysh Institute of Applied Mathematics, 2010.

[14] *Klyuchnikov I., Romanenko S.* SPSC: a Simple Supercompiler in Scala // PU'09 (International Workshop on Program Understanding). — 2009.

[15] *Klyuchnikov I., Romanenko S.* Proving the equivalence of higher-order terms by means of supercompilation // Perspectives of Systems Informatics. — Vol. 5947 of *LNCS*. — 2010. — Pp. 193–205.

[16] *Knuth D. E., Morris J. H., Pratt V. R.* Fast pattern matching in strings // *SIAM Journal on Computing*. — 1977. — Vol. 6. — P. 323.

[17] *Lisitsa A., Nemytykh A.* Verification as a parameterized testing (experiments with the scp4 supercompiler) // *Programming and Computer Software*. — 2007. — Vol. 33, no. 1. — Pp. 14–23.

[18] *Lisitsa A., Nemytykh A.* Reachability analysis in verification via supercompilation // *International Journal of Foundations of Computer Science*. — 2008. — Vol. 19, no. 4. — Pp. 953–969.

[19] *Mitchell J. C.* Foundations for programming languages. — MIT Press, 1996. — Русский перевод: Митчелл Дж. Основания языков программирования. – М.-Ижевск: НИЦ «Регулярная и хаотическая динамика», 2010.

[20] *Mitchell N.* Transformation and Analysis of Functional Programs: Ph.D. thesis / University of York. — 2008.

[21] *Mitchell N.* Rethinking supercompilation // ICFP 2010. — 2010.

[22] *Reich J. S., Naylor M., Runciman C.* Supercompilation and the Reduceron // Second International Workshop on Metacomputation in Russia. — 2010.

[23] *Secher J.* Driving-Based program transformation in theory and practice: Ph.D. thesis / Department of Computer Science, Copenhagen University. — 2002.

[24] *Secher J. P.* — Perfect Supercompilation. — Master's thesis, Department of Computer Science, University of Copenhagen, 1999.

[25] Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia / Program Systems Institute. — Pereslavl-Zalessky2010. — July 1-5.

[26] *Sørensen M. H.* — Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation. — Master's thesis, Københavns Universitet, Datalogisk Institut, 1994.

[27] *Sørensen M. H., Glück R.* An algorithm of generalization in positive supercompilation // Logic Programming: The 1995 International Symposium / Ed. by J. W. Lloyd. — 1995. — Pp. 465–479.

[28] *Sørensen M. H., Glück R.* Introduction to supercompilation // Partial Evaluation. Practice and Theory. — Vol. 1706 of *LNCS*. — 1998. — Pp. 246–270.

[29] *Sørensen M. H., Glück R., Jones N. D.* A positive supercompiler. // *Journal of Functional Programming*. — 1996. — Vol. 6, no. 6. — Pp. 811–838.

[30] *Turchin V. F.* The Language Refal: The Theory of Compilation and Metasystem Analysis. — Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

[31] *Turchin V. F.* The concept of a supercompiler // *ACM Transactions on Programming Languages and Systems (TOPLAS)*. — 1986. — Vol. 8, no. 3. — Pp. 292–325.

[32] *Turchin V. F.* Program transformation by supercompilation // Programs as Data Objects. — Vol. 217 of *LNCS*. — Springer, 1986. — Pp. 257–281.

[33] *Turchin V. F.* The algorithm of generalization in the supercompiler // Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop. — 1988.

[34] *Turchin V. F.* Program transformation with metasystem transitions // *Journal of Functional Programming*. — 1993. — Vol. 3, no. 03. — Pp. 283–313.

[35] *Turchin V. F.* Metacomputation: Metasystem transitions plus supercompilation // Partial Evaluation. — Vol. 1110 of *Lecture Notes in Computer Science*. — Springer, 1996. — Pp. 481–509.

[36] *Turchin V. F.* Supercompilation: Techniques and results // Perspectives of System Informatics. — Vol. 1181 of *LNCS*. — Springer, 1996.

[37] *Wadler P.* Deforestation: Transforming programs to eliminate trees // ESOP '88. — Vol. 300 of *LNCS*. — Springer, 1988. — Pp. 344–358.

[38] *Абрамов С.М.* Метавычисления и их применение. — Наука-Физматлит, 1995.

[39] *Абрамов С.М., Пармёнова Л.В.* Метавычисления и их применение. Суперкомпиляция. — Институт программных систем РАН, 2006.

[40] *Ключников И.Г.* Выявление и доказательство свойств функциональных программ методами суперкомпиляции: Кандидатская диссертация / Институт прикладной математики им. М.В. Келдыша РАН. — 2010.

[41] *Немытых А.П.* Суперкомпилятор SCP4. Общая структура. — Москва: ЛКИ, 2007.

[42] *Немытых А.П.* Специализация функциональных программ методами суперкомпиляции: Кандидатская диссертация / Институт программных систем РАН. — 2008.

[43] *Турчин В.Ф.* Эквивалентные преобразования программ на РЕФАЛе: Труды ЦНИПИАСС 6: ЦНИПИАСС, 1974.

[44] *Турчин В.Ф.* Феномен науки: Кибернетический подход к эволюции. — Москва: ЭТС, 2000.

