Практика функционального программирования

Выпуск 5 • Май 2010

ISSN 2075-8456

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

Журнал «Практика функционального программирования»

Авторы статей:	Александр Манзюк, Андрей Сафронов, Владимир Шабанов, Всеволод Дёмкин, Давид Сорокин, Дмитрий Попов, Евгений Лазин, Евгений Мельников, Максим Моисеев, Максим Трескин, Роман Душкин, Сергей Зефиров
Выпускающий редактор:	Дмитрий Астапов
Редактор:	Лев Валкин
Корректор:	Алексей Махоткин
Иллюстрации:	**Обложка** Яна Франк
Шрифты:	**Текст** Minion Pro © Adobe Systems Inc. **Обложка** Days © Александр Калачёв, Алексей Маслов Cuprum © Jovanny Lemonad
Ревизия:	2570 (2010-05-27)
Сайт журнала:	http://fprog.ru/

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

Пожертвования

Мы начинаем традицию публикации текстов, приложенных к пожертвованиям.
Призываем поддержать журнал и вас: fprog.ru/donate.

чашка кофе	WebMoney
За FProg.ru	WebMoney
Viva fprog.ru	WebMoney
Спасибо за прекрасный журнал!	WebMoney
keep up the good work!	WebMoney
Пожертвование от А. Легкого	WebMoney
спасибо	WebMoney
Paypal donation test) Use the XSLT, Luck	PayPal
Текст сообщения	SMS
Индульгенция за ненаписание статьи	SMS
xkrt	SMS
Даша, я тебя люблю! *;??твой Котик. (;	SMS
Итого	**3906.78**

Спасибо за поддержку!

fprog.ru/donate

Оглавление

Результаты конкурса ПФП–2009

В третьем выпуске нашего журнала мы объявили конкурс для программистов. Каждый уважающий себя журнал считает необходимым публиковать конкурсы, викторины и кроссворды с целью собственной популяризации. Естественно, что и мы обязаны были придумать свой собственный конкурс — такой, который бы сумел показать неоспоримое преимущество подхода и инструментальных средств, предлагаемых функциональным программированием, над обычными инструментами императивного программирования.

Детали подготовки к конкурсу

Надо заметить, что наш журнал с первого выпуска пользуется неожиданно высокой популярностью. Согласно статистике уникальных скачиваний PDF-версий журнала, каждый выпуск читают порядка десяти тысяч интересующихся функциональным программированием. Несмотря на популярность журнала у читателей, при создании конкурса у нас не было иллюзий по поводу того, какое количество участников конкурса мы сможем привлечь. Чтобы быть достойными хоть какого-то анализа, конкурсные задания должны быть довольно трудоёмкими.

Почему задания не могли быть слишком простыми? Мы хотели оценить, как использование различных пакетов инструментов влияет на эффективность решения задач. Простые задания зачастую могут решаться парой магических заклинаний в каких-то эзотерических языках, что совершенно ничего не сможет сказать о практике, дневной рутине написания чего-то более стоящего или объемного с помощью этих инструментов. Повышенная, неигрушечная сложность задач не могла не повлиять на желание людей участвовать в конкурсе. Чтобы компенсировать этот эффект, нами были предложены материальные призы и проведена дополнительная работа по популяризации конкурса.

Задания мы собирали «всем миром», опубликовав соответствующие призывы на страницах личных журналов Дмитрия Астапова и Льва Валкина. Слишком математические, слишком хорошо ложащиеся на функциональную парадигму и слишком хорошо решаемые с помощью существующих в мейнстримных языках библиотек задания были отброшены сразу. Оставшиеся несколько десятков заданий были предложены

для оценки нескольким известным специалистам, «представляющим» разные парадигмы программирования.

В результате были выделены три задания, которые можно было начинать более точно формулировать и описывать. В процессе этой работы одно из заданий (синтаксический анализ и валидация CSS) было признано слишком нечётким и отброшено. Оставшиеся два задания — усечение карты и составление плана-графика — были представлены в третьем выпуске журнала вместе с условиями конкурса.

Ожидаемые результаты

Мы приложили силы к тому, чтобы конкурсом заинтересовалось как можно большее количество читателей журнала: были выбраны практические, достаточно повседневные задания, проведена реклама конкурса и обеспечены материальные призы. В каждой задаче было выделено несколько уровней сложности с тем, чтобы простейшее рабочее решение можно было получить буквально за пару часов работы.

Наши ожидания были достаточно скромны: мы рассчитывали на то, что каждый сотый читатель журнала пришлёт нам решение хотя бы одной задачи. Уже это дало бы около сотни решений, среди которых можно было бы увидеть какие-то интересные для анализа закономерности.

Ожидалось, что будут непропорционально представлены два-три объектно-ориентированных языка (C++, C#, Java), а остальные, включая функциональные, будут представлены в гораздо меньшей степени.

Предполагалось, что решения на языках функционального программирования будут отличаться существенно более компактным кодом, и, в среднем, немногим худшим временем выполнения, чем лучшие решения на императивных языках.

Первичная статистика по решениям

Реальность оказалось гораздо более скромной, чем наши консервативные ожидания. Количество участников едва превысило десять человек.

Задание о составлении плана-графика привлекло всего три решения. Задание об усечении карты привлекло девять решений, из которых одно решение было от автора, участвовавшего и в предыдущем задании.

С другой стороны, поразил разброс в языках программирования, использованных для решения задач. Вместо ожидаемого доминирования объектно-ориентированных языков, для задачи об усечении карты мы получили два варианта на OCaml и по одному варианту на языках Haskell, Erlang, Common Lisp, Python, C#, Visual Basic, Ada. Три решения по составлению плана-графика были написаны на Python, Haskell и Clojure.

Нас особенно порадовали варианты решений на Ada и Visual Basic. Это означает, что задания были подобраны корректно, без перекосов в сторону функционального программирования.

Какие же закономерности можно обнаружить в когорте из десяти решений? Давайте посмотрим!

Разбор полётов

Каждое задание оценивалось пятью членами жюри. Жюри оценивало исключительно исходные тексты решений, а функциональное тестирование и оценка производительности производились отдельно. В составе жюри все декларировали знакомство C/C++; с языками Haskell, OCaml, Java и различными диалектами Lisp было знакомо по два человека; а в C#, F#, Erlang и Clojure разбирался только кто-то один. Состав жюри можно условно разделить на двух человек «более функциональных», двух «более императивных» и одного «непримкнувшего».

Члены жюри и редакции журнала в конкурсе участия не принимали.

Для начала, начнём с результатов задания о составлении плана-графика. Анализ результатов облегчается тем, что три присланных результата хорошо отображаются на три доступных места победителей конкурса.

Автор	Язык	LOC	Оценка жюри
Е. Зайцев	Python	1001	3.6
К. Лопухин	Clojure	694	3.4
Р. Салмин	Haskell	1664	2.8

Все решения используют разные подходы к отысканию оптимального плана-графика. В то же время, их объединяет то, что при отсутствии решения они не могут внятно уведомить пользователя о том, в чем же заключается проблема.

При сопоставлении редакцией журнала отчётов жюри по этому заданию выявилась интересная особенность: мнения членов жюри слабо коррелировали друг с другом. Например, вот цитаты из пары отчётов по решению на Haskell:

Эксперт–1	Эксперт–2
«Код в достаточной степени документирован. Разбиение на модули сделано хорошо, реализация функций — небольшая, так что читать код достаточно легко.»	«Код заточен под конкретную задачу, разбиение на модули в некотором роде от фонаря, повторно использовать будет сложновато.»

Теперь перейдём к более интересной задаче, вызвавшей большее количество откликов. Задача по усечению карты была порождена реальной проблемой в использовании существующего инструмента для обрезки карт OpenStreetMap, написанного на Java. Инструмент не справлялся с обрезкой за приемлемое время. Предполагалось, что решение победителей этого конкурса сможет сослужить добрую службу людям, которым регулярно приходится вырезать карты из «мирового атласа» OpenStreetMap.

Соответственно, во главу угла ставились следующие признаки:

- Корректность работы. Решения проверялись на наборе функциональных тестов, призванных отсечь некачественные решения, выдающие некорректный результат.
- Качество кода. Оценка понятности и расширяемости кода производилась редакцией журнала на основе серии экспертных заключений независимых членов жюри.
- Скорость работы решения. Производились замеры скорости усечения реальных карт России и регионов Европы.

Четыре решения были отсеяны сразу по признаку корректности работы. Решение на Python впадало в бесконечный цикл на карте Московской области. Решение на Erlang порождало файл нулевой длины на карте Германии. Решение на Haskell не смогло корректно обработать все случаи рекурсивного включения объектов и оставляло в выходном файле лишние узлы. Решение на языке Ада оставляло в выходном файле ссылки на удаленные узлы и пути.

Язык OCaml был представлен двумя решениями от разных авторов. Одно из них без сбоев прошло все микротесты, кроме проверки на перенос из входного файла в выходной «незнакомых» тэгов. Второе решение не реализовывало режим обрезки «complete objects» и не прошло ряд микротестов. Тем не менее, эти решения не зависали на реальных картах и порождали результирующие ответы, похожие на корректные. Поэтому мы решили включить их в сводную таблицу решений, представляющих интерес.

Язык	**LOC**	**Скорость**	**Оценка жюри**
C#	854	153	3.6
Visual Basic	**225**	341	3.4
Common Lisp	899	**74**	3.1
OCaml–1	449	288	3.3
OCaml–2	370	158	2.8

В таблице жирным шрифтом отмечены лучшие результаты. Под «скоростью» понимается средняя скорость вырезки регионов из нескольких больших файлов карт, в секундах. Так, решение на Common Lisp оказалось самым быстрым, со средним результатом 74 секунды.

Редакция приняла решение разделить третье место между решениями на Common Lisp и OCaml и распределить призовые места следующим образом:

	Автор	**Язык**
1	П. Егоров	C#
2	А. Волков	Visual Basic
3	А. Вознюк	Common Lisp
3	Д. Попов	OCaml

Отмеченная ранее слабая корреляция оценок жюри во всей красе наблюдалась и в этом конкурсе. На рис. 1 наглядно видно, что оценки идут вразнобой. Интересно было бы вывернуть конкурс наизнанку и искать зависимости внутри оценок жюри, а не в результатах выполнения конкурсных заданий!

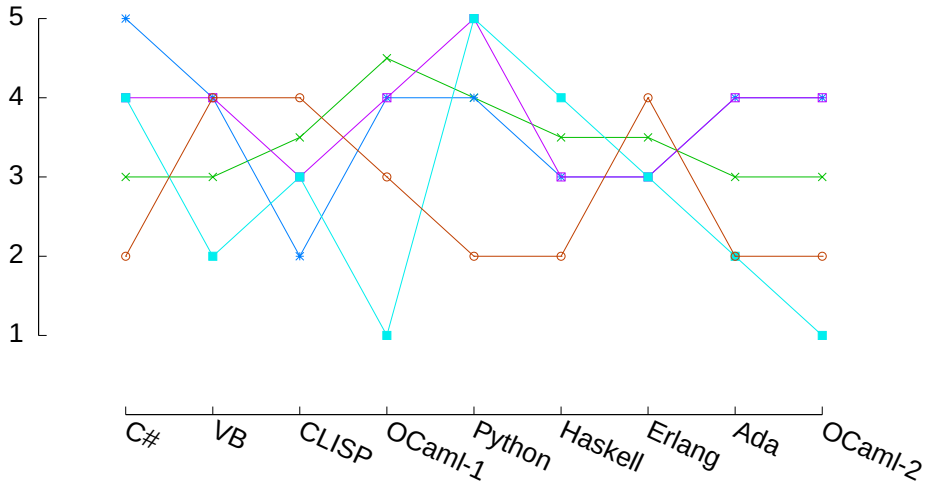


Рис. 1. Разброс индивидуальных оценок членов жюри

Итоги

Поздравляем Павла Егорова и Евгения Зайцева с победой в соответствующих разделах конкурса! Поздравляем Алексея Волкова, Алексея Вознюка, Константина Лопухина, Дмитрия Попова, Романа Салмина с призовыми местами. Спасибо, что нашли время и потратили его с пользой для человечества! Для передачи призов мы свяжемся с вами по электронной почте.

Статистически значимых закономерностей, конечно, в результате конкурса получено не было. Однако, имеющихся данных можно сделать вывод, что более качественные решения были одновременно наименее трудозатратными. Выраженной зависимости между качеством решения и использованным языком программирования мы не наблюдали.

У большинства решений нашлось два общих слабых места. Во-первых, это отсутствие инструментов, облегчающих сборку из исходников, а во-вторых — отсут-

ствие внятных уведомлений об ошибках. Даже те решения, которые сопровождались Makefile-ом или его аналогом, почти всегда требовали «доводки напильником», а вместо внятной диагностики ошибок почти всегда выдавали стандартные сообщения вида «Runtime error 23454 at address 0xdeadbeef».

Мы были приятно удивлены призовому месту решения на Visual Basic. Надо отметить его краткость и простоту кода. Решение «в лоб», без особых изысков в плане алгоритмов и структурирования программы, победило в практичности большинство своих аналогов на языках функционального программирования.

Среди недостатков присланных решений на языках функционального программирования надо отметить излишнее мудрствование: использование нетривиальных решений, отказ от использования библиотек синтаксического анализа XML, метапрограммирование, когда задача решалась более простыми средствами, не слишком теряя при этом в производительности. Сравните, например, решение на Visual Basic с решением OCaml–1: различие в производительности в двадцать процентов было получено ценой двукратного увеличения количества кода. Кстати, в этом выпуске журнала есть статья Дмитрия Попова (одного из участников конкурса) о синтаксическом анализе, содержащая, в частности, сравнительный анализ скорости работы различных библиотек синтаксического анализа XML (см. с. 156).

С другой стороны, в каких-то случаях такое мудрствование может быть оправдано. Вариант на Common Lisp, использующий несколько стадий метакомпиляции, кажется сложным для понимания (впрочем, автор все доходчиво рассказал в своем LiveJournal), но может похвастаться абсолютным рекордом скорости выполнения обрезки карт. Он может служить неплохой иллюстрацией к статье «Лисп — абстракции на стероидах» Виталия Маяцких, опубликованной в предыдущем выпуске нашего журнала.

Интересно отметить, что победившее решение на C# вызвало у жюри более полярные мнения, чем решение на Visual Basic. Вот пара отзывов о победившем решении:

Эксперт–3	**Эксперт–4**
«Программа реализована в лучших традициях продакшен программирования, простой первый прототип и аккуратно реализованный с разбиением на подпроекты и множеством вспомогательных утилит итоговый проект.»	«Декомпозиция: лютый оверди-зайн. Как будто платят за строчки кода. Куча мусора, не относящегося к задаче. Смысл утоплен в деталях реализации.»

Читая отзывы экспертов, становится ясно, что даже крупные задачи не позволяют в должной мере оценить преимущества тех или иных подходов. Конкурс продемонстрировал, что исследования необходимо структурировать так, чтобы иметь возможность проследить больший участок жизненного цикла приложения. Поддержка кода, внесение изменений, влияние подхода на работу в команде — все эти важные аспекты остались за рамками нашего конкурса. Тем, кто будет проводить исследования в этом направлении, необходимо обратить на это внимание.

А всем тем, кто собирался поучаствовать в конкурсе, но по каким-то причинам не стал, желаем в будущем быть более решительными.

Исходные тексты решений, занявших призовые места, можно найти на сайте журнала.

Дмитрий Астапов, adept@fprog.ru

Обсуждение результатов конкурса ведётся в LiveJournal.

Инструменты интроспекции в Erlang/OTP

Максим Трескин
zert@fprog.ru

Аннотация

В статье описаны отладочные механизмы Erlang/OTP: способы трассировки обмена сообщениями между легковесными процессами, отслеживание обращений к функциям и наблюдение за состояниями процессов. Цель статьи — познакомить читателя с одним из главных достоинств платформы, позволяющим увеличить скорость написания рабочего приложения и снизить временные затраты на его отладку.

The article describes debugging tools provided by Erlang platform: the means for tracing message flows, function calls invocations, and inspection of the process state. These features are considered to be the major part of the Erlang platform's advantage, decreasing implementation and debugging time.

Обсуждение статьи ведётся в LiveJournal.

1.1. Введение

Когда говорят о преимуществах применения Erlang/OTP для создания высоконагруженных отказоустойчивых систем, обычно упоминают использование множества легковесных процессов без обращения к разделяемым ресурсам (что избавляет от множества проблем синхронизации и способствует увеличению масштабируемости системы), дерево супервизоров, автоматически перезапускающих процессы в случае их падения, или возможность горячей замены кода прямо в работающем приложении. Менее известными остаются присущие платформе Erlang очень мощные инструменты отладки, позволяющих узнать, что же в данный момент происходит в системе, какими сообщениями обмениваются между собой процессы, какие функции с какими аргументами вызываются.

Можно сказать, что лучше проектировать систему так, чтобы эти инструменты оказались лишними, однако это возможно только теоретически; реальная жизнь же, как правило, сурова и непредсказуема. Например, у заказчика на противоположной стороне земного шара работает сложная, состоящая из многих модулей система, и при превышении некоторой пороговой нагрузки (а то и просто в зависимости от фазы Луны) в ней начинает происходить что-то странное, обусловленное ошибкой в коде или алгоритме (те, кто всегда пишет программы без ошибок — смело бросайте в меня камень покрупнее). Хорошо, если вы предусмотрели ведение логов в критических участках системы — это может вам помочь. Но всего не предусмотришь, и если ошибка затаилась в секции кода, в котором логов отродясь не было, необходимы средства трассировки. Создать систему отслеживания событий в приложении, которая может быть прозрачно использована без серьёзных изменений в коде, проще всего, когда для выполнения программных инструкций используется виртуальная машина.

Erlang/OTP использует виртуальную машину BEAM (Bogdan/Björn's Erlang Abstract Machine), в которой реализованы (в том числе) следующие возможности:

- дешёвое создание легковесных (не системных) процессов (с механизмом отслеживания изменений их состояний);
- лёгкое масштабирование приложения на SMP;
- обмен сообщениями между легковесными процессами (с механизмом их трассировки);
- система оповещения о вызываемых функциях с гибким языком запроса;
- горячая замена кода.

В этой статье мы рассмотрим работу со следующими трассировочными функциями:

- `erlang:trace/3` — настройка списка событий, за которыми будет происходит наблюдение;
- `erlang:trace_pattern/3` — настройка шаблона функций, вызовы которых должны отслеживаться; используется в сочетании с флагом `call` функции `erlang:trace/3`.

Модуль `eintro`, содержащий код функций-обёрток, написан автором статьи и

доступен на сайте журнала.

Запуск ноды Erlang с последующей компиляцией модуля `eintro` выглядит так:

```
[zert@pluto]:erlang-introspection $>> erl
Erlang R13B04 (erts-5.7.5) [source] [...]

Eshell V5.7.5  (abort with ^G)
1> c(eintro).
{ok,eintro}
```

1.2. Процессы

Процесс в Erlang/OTP порождается с помощью встроенной функции (built-in function, BIF) `spawn`, которая принимает в качестве аргументов имя функции и список параметров, с которыми она будет вызвана. Функция spawn возвращает идентификатор процесса (`pid()`), используя который, мы можем осуществлять различные действия над этим процессом.

Под процессом мы будем подразумевать функцию, которая делает вызов `receive`, получает сообщение, и либо завершается возвратом, либо делает следующий `receive`, обычно посредством рекурсии. Во многих случаях эта функция может иметь иной вид и осуществлять возврат без вызова `receive` сразу после каких-либо действий, но мы такие функции рассматривать не будем.

1.2.1. Основные события процессов (создание, завершение, выполнение, приём и передача сообщений)

Итак, создадим первый процесс, используя функцию `eintro:loop/1`, выставив предварительно трассировку для всех новых процессов (параметр `new`). Трассировать будем по флагам `procs` и `running`. Первый флаг даёт возможность наблюдать за порождением нового процесса (метка `spawn`), завершением процесса (`exit`), регистрацией и дерегистрацией под символьным именем (`register`/`unregister`), связыванием процессов и отменой связывания (`link`/`unlink`), получении событий о связывании от другого процесса (`getting_linked`/`getting_unlinked`). Использование второго флага покажет моменты, в которые процесс планируется на выполнение (`in`) и вытесняется другим процессом (`out`):

```
1> T = eintro:tracer(new, [procs, running]).
<0.35.0>
<0.36.0> will run in {erlang,apply,2}
<0.36.0> was running in {io,wait_io_mon_reply,2}
```

В переменной `T` находится значение идентификатора процесса, принимающего трассировочные сообщения — `<0.35.0>`. Идентификатор `<0.36.0>` принадлежит командной оболочке Erlang и нам сейчас не интересен.

```
2> Pid = spawn(eintro, loop, [5]).
<0.36.0> will run in {io,wait_io_mon_reply,2}
<0.36.0> exits with {ok,[{match,1,
                                {var,1,'Pid'},
                                {call,1,
                                      {atom,1,spawn},
                                      [{atom,1,eintro},
                                       {atom,1,loop},
                                  {cons,1,{integer,1,5},{nil,1}}]}}],
                        2}
<0.37.0> will run in {eintro,loop,1}
<0.37.0>
<0.37.0> was running in {eintro,loop,1}
<0.38.0> will run in {erlang,apply,2}
<0.38.0> was running in {io,wait_io_mon_reply,2}
```

Здесь необходимо игнорировать уже два идентификатора: `<0.36.0>` и `<0.38.0>` (в реальных приложениях обеспечивать подобную фильтрацию необходимо либо с помощью сопоставления по шаблону, либо регистрацией нужных процессов). Идентификатор `<0.37.0>` хранится в переменной `Pid`, так как этот процесс нас и интересует. Видно, как процесс помещается планировщиком на выполнение в функции `eintro:loop/1` (`<0.37.0> will run in {eintro,loop,1}`), затем, так как в этой функции сразу же делается вызов `receive`, процесс вытесняется до получения какого-либо сообщения (`<0.37.0> was running in {eintro,loop,1}`).

Теперь завершим старый трассировщик, чтобы он больше не сбивал нас сообщениями о ненужных процессах (нам сейчас известен конкретный `Pid`), и запустим новый трассировочный процесс с новыми флагами:

```
3> exit(T, kill).
true

4> NT = eintro:tracer(Pid, [procs, running, send, 'receive']).
<0.40.0>
```

Флаг `send` отвечает за трассировку отсылки сообщений и порождает два события: метка `send` — непосредственно отсылка сообщения, и метка `send_to_non_existing_process` — посылка сообщения несуществующему процессу. Флаг `receive` включает трассировку приёма сообщений заданным процессом[1] (метка `receive`).

Настал момент послать нашему процессу какое-нибудь сообщение:

```
5> Pid ! pass.
```

[1] Обрамляется одинарными кавычками по той причине, что `receive` является ключевым словом Erlang, а в данном случае необходимо передать в функцию одноимённый атом.

```
pass
<0.37.0> receives pass
<0.37.0> will run in {eintro,loop,1}
<0.37.0> was running in {eintro,loop,1}
```

Трассировщик сообщает, что процесс:

- получил сообщение `pass`;
- планируется на выполнение в функции `eintro:loop/1`;
- вытесняется другим процессом.

И вот наступает финальный момент (по крайней мере, для процесса `<0.37.0>`). Посылаем ему сообщение, получив которое, он завершается:

```
6> Pid ! stop.
<0.37.0> receives stop
Count: 4
stop
<0.37.0> will run in {eintro,loop,1}
<0.37.0> sends {io_request,<0.37.0>,<0.26.0>,
                           {put_chars,unicode,io_lib,format,
                                      ["Count: ~p~n",[4]]}} to <0.26.0>
<0.37.0> was running in {io,wait_io_mon_reply,2}
<0.37.0> receives {io_reply,<0.26.0>,ok}
<0.37.0> will run in {io,wait_io_mon_reply,2}
<0.37.0> receives timeout
<0.37.0> exits with normal
```

Разбираем, что произошло:

- процесс получил сообщение `stop`;
- он планируется на выполнение в функции `eintro:loop/1`;
- посылает сообщение (команду печати символов) системе ввода/вывода;
- вытесняется другим процессом (делает вызов `receive`);
- получает сообщение об успешном исполнении команды от системы ввода/вывода;
- продолжает своё выполнение в функции `io:wait_io_mon_reply/2`;
- получает сообщение `timeout`;
- завершается с причиной `normal`.

Итак, на данном этапе мы ознакомились со следующими событиями трассировки:

- планирование процесса на выполнение (`in`);
- вытеснение процесса (`out`);
- завершение процесса (`exit`);
- посылка сообщения (`send`);
- приём сообщения (`receive`).

1.2.2. Регистрация процесса

Процесс в OTP может адресоваться как по своему уникальному (в пределах ноды) идентификатору, который возвращается при вызове функции `spawn`, так и по его зарегистрированному имени, которое мы можем назначить на своё усмотрение, используя BIF `register/2`.

Вызов `register(RegName, Pid)` связывает имя `RegName` с идентификатором процесса `Pid`. После этого можно посылать сообщения процессу, используя это имя. Так как повлиять на значение идентификатора, возвращаемого вызовом `spawn`, невозможно, его регистрация под фиксированным именем в некоторых случаях может оказаться полезной. BIF `unregister(RegName)` удаляет ассоциацию идентификатора с именем `RegName`. Оба этих события — регистрация и дерегистрация — генерируют событие трассировки.

Создаём процесс, используя уже известную нам функцию `eintro:loop/1`:

```
1> Pid = spawn(eintro, loop, [5]).
<0.35.0>
```

Теперь `Pid` содержит идентификатор этого процесса. Включаем трассировку событий из группы `procs` от этого процесса:

```
2> T = eintro:tracer(Pid, [procs]).
<0.37.0>
```

Вызываем BIF `register/2` для создания ассоциации имени `test_loop` с идентификатором `Pid`:

```
3> register(test_loop, Pid).
true
<0.35.0> registered as test_loop
```

Функция `register/2` вернула значение `true`, и процесс-трассировщик получает сообщение о регистрации `{trace, <0.35.0>, register, test_loop}`, по которому можно судить о том, под каким именем зарегистрирован процесс.

Проводим обратную процедуру — дерегистрацию:

```
4> unregister(test_loop).
<0.35.0> unregistered as test_loop
true
```

Процесс-трассировщик получает сообщение `{trace, <0.35.0>, unregister, test_loop}`. После этого момента попытка послать сообщение, используя имя `test_loop`, будет проваливаться с ошибкой `badarg`.

1.2.3. Связи

Установление связи между процессами позволяет корректно продолжить работу системы в целом в случае аварийного завершения одного из них. Для связывания двух процессов предназначена BIF `link/1`. Например, если процесс A вызовет эту функцию с идентификатором процесса B (`PidB`), то при завершении процесса B с какой-либо причиной, отличной от `normal`, процесс A получит сигнал выхода (`exit`) с такой же причиной и завершится. В случае, если для процесса A выставлен флаг `trap_exit` (`process_flag(trap_exit, true)`), вместо сигнала `exit` он получит сообщение следующего вида:

```
{'EXIT', PidB, Reason}
```

которое можно обработать точно так же, как и любое другое сообщение. Связь может быть разрушена вызовом функции `unlink/1`.[2]

События установления и разрушения связи могут наблюдаться во время трассировки. Для демонстрации этого создадим две функции: `sv_proc` и `sv` (от supervisor). Первая функция устанавливает флаг `trap_exit` для процесса в передаваемое значение и передаёт управление второй функции:

```
sv_proc(TE) ->
    process_flag(trap_exit, TE),
    sv().
```

Вторая функция принимает сообщения, реагирует на них и рекурсивно вызывает себя:

```
sv() ->
    receive
        {link, Pid} ->
            io:format("Link with ~p~n", [Pid]),
            link(Pid);
        {unlink, Pid} ->
            io:format("Unlink from ~p~n", [Pid]),
            unlink(Pid);
        {'EXIT', FromPid, Reason} ->
            io:format("Trap exit from ~p with reason ~p~n",
                [FromPid, Reason]);
        _Other ->
            io:format("Other message: ~p~n", [_Other])
    end,
    sv().
```

Определим три собщения:

[2] В некоторых случаях целесообразнее вместо системы связей использовать мониторы процессов. Они лишены ряда недостатков в сравнении со связями, например, два вызова `link/1` подряд и последующий вызов `unlink/1` разрушат связь. Описание принципов работы системы мониторов выходит за рамки этой статьи.

- `{link, Pid}` — создание связи с процессом `Pid`
- `{unlink, Pid}` — разрушение связи с процессом `Pid`
- `{'EXIT', FromPid, Reason}` — получение сообщения о завершении связанного процесса (в случае с `trap_exit == true`)

Запускаем наш знакомый процесс и трассировщик его событий:

```
1> Pid = spawn(eintro, loop, [5]).
<0.35.0>
2> T = eintro:tracer(Pid, [procs]).
<0.37.0>
```

Запускаем процесс-супервизор и трассировщик с флагом `trap_exit == true`, перехватывающий события от него:

```
3> SV = spawn(eintro, sv_proc, [true]).
<0.39.0>
4> TSV = eintro:tracer(SV, [procs]).
<0.41.0>
```

Посылаем процессу-supervisor запрос на установление связи с процессом `Pid`:

```
5> SV ! {link, Pid}.
Link with <0.35.0>
{link,<0.35.0>}
<0.39.0> links to <0.35.0>
<0.35.0> gets linked to <0.39.0>
```

Видим, что процесс `SV` принял это сообщение, после чего сгенерировал событие `link`, а противоположный процесс сгенерировал событие `getting_linked`.

Останавливаем процесс `Pid`:

```
6> Pid ! stop.
Count: 5
stop
Trap exit from <0.35.0> with reason normal
<0.35.0> exits with normal
<0.39.0> gets unlinked from <0.35.0>
```

После получения сообщения `stop` функция `loop/1` завершается, при этом процесс, который её выполняет, выходит с причиной `normal`. Виден момент, когда процесс `SV`, связанный завершающимся процессом, получает сообщение `'EXIT'` с причиной `normal`.

1.3. Функции

Функции в Erlang делятся на локальные и внешние. Локальные функции определены в том же модуле, где используются, и вызываются без указания модуля.

```
eat(_) -> ok.

let() ->
  eat(bee).
```

Здесь функция `eat/1` используется как локальная. Для того, чтобы она была доступна из других модулей, её экспортируют, используя декларацию `export` перед определениями функций:

```
-module(rod).
-export([perun/1]).

veles() ->
  ok.

mara() ->
  ok.

perun(M) ->
  case M of
    man ->
      veles();
    woman ->
      mara()
  end.
```

При вызове внешней функции явно указывается имя модуля, в котором она определена:

```
-module(olymp).

zeus(M) ->
  rod:perun(M).
```

Но даже если функция указана в декларации `export`, а вызывается в том же модуле, где и определена — без явного указания имени модуля она будет считаться локальной.

Ещё одно важное различие между локальными и внешними функциями в том, что при вызове внешней функции будет использована самая последняя версия модуля, который её реализует. Другими словами, обращение к внешней функции после горячей замены кода вытеснит предыдущую реализацию функции и выполнится свежий код. Напротив, вызов локальной функции будет обработан той же самой версией кода, в которой этот вызов осуществляется, и реализация функции останется неизмен-

ной. Эта особенность позволяет осуществлять замену кода рабочих процессов, написанных в стиле OTP и использующих типичные поведения (behaviors) `gen_server`, `gen_fsm` и т. д.

Определим функцию `fc_loop/0`:

```
fc_loop() ->
    receive
        local ->
            fc_test();
        external ->
            ?MODULE:fc_test_ext();
        lambda ->
            Fun = fun() -> ok end,
            Fun()
    end,
    fc_loop().
```

где функция `fc_test/0` является локальной, а `fc_test_ext/0` — внешней.[3] Для простоты примера обе функции будут всего лишь возвращать атом `ok`:

```
fc_test() ->
    ok.

fc_test_ext() ->
    ok.
```

Запускаем процесс, отлавливающий трассировочные сообщения о вызываемых функциях:

```
1> T = eintro:tracer(new, [call], {{eintro, '_', '_'}, true, [global]}).
<0.35.0>
```

Указываем, что нас интересуют все новые процессы (`new`) в качестве инициаторов вызовов функций (`call`). В третьем параметре определяем шаблон модуля, функций и аргументов (MFA), по которому мы хотим отслеживать вызовы (`{eintro, '_', '_'}`)[4], разрешаем его (`true`) и ставим флаг `global`, который означает, что трассировка будет осуществляться по глобальным вызовам внешних функций.

Запускаем интересующий нас процесс:

```
2> Pid = spawn(eintro, fc_loop, []).
<0.37.0>
<0.37.0> calls {eintro,fc_loop,[]}
```

[3] Несмотря на то, что функция `fc_test_ext/0` определена в этом же модуле, обращение к ней осуществляется как к внешней, через явное указание модуля. Макрос `?MODULE` заменяется препроцессором на имя текущего модуля.

[4] В данном случае трассировка будет осуществляться для вызовов всех функций, определённых в модуле `eintro`, с любыми аргументами.

Видим, что процесс, созданный BIF `spawn`, запустился и вызвал функцию `eintro:fc_loop/0`.

Пошлём процессу сообщение `local`, по которому он должен вызвать локальную функцию:

```
3> Pid ! local.
local
```

Ничего не произошло, трассировочное сообщение не сгенерировалось.

Теперь посылаем сообщение `external`, процесс должен будет глобально вызвать внешнюю функцию:

```
4> Pid ! external.
external
<0.37.0> calls {eintro,fc_test_ext,[]}
```

Как видно из полученного сообщения, это и произошло, была вызвана функция `eintro:fc_text_ext/0`.

Чтобы иметь возможность наблюдать за вызовами не только внешних, но и определённых локально функций, укажем флаг `local` при запуске трассировщика:

```
1> T = eintro:tracer(new, [call], {{eintro, '_', '_'}, true, [local]}).
```

Трассировка внешних функций при этом флаге не изменится, но появится дополнительное сообщение — вызов функции `fc_loop/0` локально:

```
3> Pid ! external.
external
<0.37.0> calls {eintro,fc_test_ext,[]}
<0.37.0> calls {eintro,fc_loop,[]}
```

Вызов локальной функции будет выглядеть так:

```
4> Pid ! local.
local
<0.37.0> calls {eintro,fc_test,[]}
<0.37.0> calls {eintro,fc_loop,[]}
```

Если послать процессу `Pid` сообщение `lambda`, мы увидим следующее:

```
5> Pid ! lambda.
lambda
<0.37.0> calls {eintro,'-fc_loop/0-fun-0-',[]}
<0.37.0> calls {eintro,fc_loop,[]}
```

Функция со странным названием `'-fc_loop/0-fun-0-'/0` является ничем иным, как λ-функцией. Это внутреннее представление в Erlang VM, необходимое для адресации к ней. В пользовательском коде нельзя ссылаться на это имя, так как нет гарантии, что оно не изменится.

1.4. Заключение

Помимо перечисленных способов трассировки событий в Erlang/OTP предусмотрены более высокоуровневые инструменты, такие как:

- Application Monitor
- Erlang Top
- Erlang Debugger
- Process Manager

На этом краткий обзор средств, позволяющих узнать немного больше о том, что происходит в работающем приложении, можно считать завершённым. Более полную информацию можно получить на страницах The Erlang BIFs, Debugger/Interpreter Interface и The Text Based Trace Facility документации Erlang/OTP.

Экономия ошибок

Использование встроенных проблемно-ориентированных языков (DSL) в моделировании электронных схем

С. Зефиров, А. Сафронов, В. Шабанов, Е. Мельников
{zefirov, safronov.a, shabanov.v, melnikov}@prosoft.ru

Аннотация

В статье рассказывается об опыте использования встроенных проблемно-ориентированных языков применительно к моделированию аппаратуры с процессорными ядрами. Главным соображением, давшим старт работам, было желание защититься от рисков, связанных с внесением изменений в будущем. Описана выбранная авторами декомпозиция предметной области (описание команд процессорных ядер различных архитектур). Рассмотрено применение «управляемого типами дизайна» (type-driven design).

The article discusses the use of domain-specific embedded languages in hardware modelling. Authors share their experience in reducing the risks related to the ever-changing requirements via clever decomposition of the subject domain, the use of the specialized DSL and type-driven design.

Обсуждение статьи ведётся в LiveJournal.

2.1. Введение

В компании «ПРОСОФТ», где работает коллектив авторов, разрабатывается система моделирования цифровой аппаратуры для разработки встраиваемой техники. Среди возможностей, предоставляемых системой — моделирование цифровой аппаратуры и процессорных ядер.

Современная аппаратура практически всегда включает в себя хотя бы одно процессорное ядро с выполняемой на нём программой, поскольку изменение кода программы проще и дешевле изменения аппаратуры. Поэтому правильное моделирование поведения программ весьма важно.

Спектр процессорных ядер на рынке чрезвычайно широк, от восьмибитных ядер микроконтроллеров младших классов до 128-битных IBM Cell Broadband Engine. Поскольку поддержать все существующие процессоры практически невозможно, надо попробовать сделать так, чтобы добавление нового класса процессорных ядер было как можно более быстрым и безошибочным.

Само по себе процессорное ядро не очень интересно, интересна возможность его работы в составе какой-либо системы, например, разрабатываемой (или уже существующей) системы на кристалле. Для разработчика разумно использовать модель системы в целом, поскольку так проще контролировать поведение частей системы. Одна часть модели системы, куда будет входить процессорное ядро, может быть создана с помощью языка описания аппаратуры, а другая часть — с помощью редакторов схем. Отсюда возникает требование к гибкости подключения модели ядра к «внешней среде». Такое подключение производится с помощью внешней шины, по которой происходит обмен информацией с другими компонентами системы. Протоколов, по которым могут работать шины, несколько: например, AMBA [2], Wishbone [16] и CoreConnect [6]. Желательно иметь возможность быстро переключаться между различными протоколами, выбирая их под текущие нужды.

Ещё одна степень свободы появляется, если мы хотим получить точность моделирования ценой снижения скорости, и наоборот. Простой пример: за сколько тактов надо выполнять умножение — за один, или как в документации?

Скорость создания программы, в нашем случае модели ядра процессора, зависит от числа ошибок и сложности их исправления. Существенным источником ошибок является необходимость моделировать большое число команд (зачастую более сотни), и это не считая ещё «дополнительных степеней свободы» — дополнительных возможностей применения модели. Контроль за соблюдением правильности выполнения каждой отдельной команды и возможными вариантами общения с шиной хотелось бы поручить формальным средствам. Это подводит нас к необходимости иметь **язык описания моделей процессоров**. Язык может быть внешним, со своим синтаксисом и обычным циклом обработки (лексический разбор, синтаксический разбор, проверка типов, выполнение тестовых участков и кодогенерация), а может быть встроен в язык общего назначения, что позволяет экономить время на написание трёх этапов из пяти.

Во второй главе мы рассмотрим ошибки, которые можно допустить при создании модели ядра; в третьей главе продемонстрируем ядро языка описания, позволяющего отловить большинство из них; в четвёртой главе расскажем о роли тестирования в описании моделей. Пятая глава расскажет о практике применении описанного подхода и о возможных путях его развития. Наконец, шестая глава подведёт итоги.

2.2. Возможные ошибки моделирования ядер процессоров

Какие проблемы стоят перед системами моделирования? Какие возможности требуются от них? Почему бы не воспользоваться существующими средствами — например, доступными в исходных кодах моделями процессоров (QEMU [13] или аналогичными эмуляторами)?

Процессорные ядра обычно хорошо документированы. Более того, обычно их поведение зафиксировано раз и навсегда: выпущенный и широко использующийся процессор 80286 не обретёт внезапно новые команды. Практически все широко распространённые модели процессоров имеют эмуляторы в открытых исходных кодах, по которым можно выяснять спорные детали. В этом отношении наличие документации, опорной реализации и неизменность спецификации очень на руку — ведь этапы сбора требований и общего дизайна, с присущей им высокой стоимостью ошибок, практически отсутствуют.

Напрямую использовать эмуляторы процессорных ядер в открытых исходных кодах невозможно по нескольким причинам. Одна из них — лицензии, под которыми выпущены эти эмуляторы. Обычно это GNU GPL, которая препятствует их использованию в системах моделирования с закрытым кодом. Кроме того, есть две технических проблемы: нам надо, чтобы, во-первых, модель процессора эффективно работала в моделировании схемы в целом и, во-вторых, чтобы мы могли использовать модель процессора в разных окружениях (разные ядра моделирования, разные протоколы шины, разная точность моделирования). Эмуляторы в открытых исходных кодах обычно написаны на языке C, что делает их доработку, как минимум, трудоёмкой задачей. Если же мы планируем использовать параллельное моделирование (с использованием процессов операционной системы или MPI), да ещё под разными целевыми платформами (кодогенерация в C, в Java VM с языком Java, в .Net с C#), то использование языка C становится прямо противопоказанным.

Есть также ещё ряд желательных требований:

1) правильное взаимодействие отдельных частей внутри модели, а также между моделью и внешним миром;

2) гибкость в подключении к внешнему миру;

3) гибкость в размене точности моделирования на скорость моделирования;

4) гибкость в выборе системы моделирования (целевая VM или язык, метод моделирования — параллельный (на нескольких процессорных ядрах или даже на кластере) или последовательный);

5) максимальное повторное использование кода внутри линейки процессоров (снижение комбинаторной сложности с произведения на сумму).

Первая проблема, как показывает опыт, может быть решена простым учётом размеров данных. Обычно внутренний адрес данных имеет размер в одно процессорное слово (32 бита) и адресует данные с точностью до байта, а по шине память адресуется словами или более крупными блоками. Необходимость явного совмещения адресов, задаваемых словами разных размеров, приводит к необходимости продумывать это совмещение везде, где оно встречается. Задание размеров каждой внутренней переменной поможет выявить большинство мест, где они неправильно используются.

Вторая проблема сводится к перечислению обращений к внешнему миру и расстановке задержек и ожиданий, необходимых для отработки протокола шины. Можно также соорудить обобщённый переходник между процессорным ядром и шиной с набором параметров. Заметная часть проблем при этом будет выявлена уже на этапе проверки типов, и для функционального тестирования останется меньше работы.

Третья и четвертая проблема решаются кодогенерацией. Задав максимально точную модель, мы можем упростить её в процессе кодогенерации. Имея формальное описание отдельных команд, мы можем создать максимально эффективный код на любом языке программирования. Более того, для параллельного моделирования можно использовать одни приёмы, а для последовательного — другие.

Пятая проблема, последняя по очередности, но не по значению, решается использование макроязыков (хотя бы препроцессора языка C) и параметризацией процессоров. При использовании встроенного языка описания макроподстановки получаются автоматически, а использование строго типизированного языка программирования обеспечивает проверку правильности типов макроподстановок и точность диагностики ошибок. Параметризация вариантов процессорных ядер покажет нам, когда макроподстановка выполняется неправильно и код какой команды должен быть изменён для достижения корректной работы.

Итак, ядро описания процессорных ядер должно быть встроено в язык программирования высокого уровня. В нём должны присутствовать следующие возможности:

- возможность задавать параметры модели и проверять их на совместимость как можно раньше;
- возможность контроля размеров (и размерности) сущностей, обрабатываемых командой;
- возможность расставлять точки обращения к внешней шине.

Все это позволяет решить пять проблем, обозначенных выше, с минимальными затратами.

На роль языка-носителя для нашего DSL подойдёт любой язык с вычислениями на уровне типов и (хотя бы) возможностью создания (инфиксных) операторов пользователя. Тонким моментом является возможность связывания переменных и значений — можно ли посередине описания ввести в обиход новый идентификатор, или нет. Если такая возможность есть, то использование DSEL становится более удобным.

Рассмотрим пригодность на роль языка-носителя несколько распространённых языков:

- **OCaml** и другие языки семейства ML: позволяют создавать операторы пользователя, однако система типов не позволяет описывать ограничения в виде равенств и неравенств [9]. В этом случае ошибка проверки размеров будет выдана на этапе генерации кода, что затрудняет диагностику ошибок.

- **C++**: система типов C++ достаточно сильна для выражения всех необходимых ограничений, однако она настолько сложна, что для удобного её использования применяют встроенные в Haskell языки спецификаций [3, 7].

- **Haskell**: выразительная система типов, операторы пользователя и возможность использования связывания переменных. Заметным минусом является дублирование кода для значений и типов: нам надо иметь целые числа с операциями сложения и сравнения на уровне типов, а также проекция из чисел на уровне типов в обычные целочисленные значения для этапа кодогенерации и тестирования.

- **Agda** и **Coq**: выразительная система типов, операторы пользователя. Система типов этих двух языков программирования значительно выразительней, чем система типов Haskell и C++: например, обычные значения могут быть параметрами типа. В нашем случае использование этих языков программирования было затруднено из-за отсутствия необходимой экспертизы.

Перед тем, как приступить к объяснению технических деталей описания модели, приведём фрагмент кода, дающий примерное представление о получающемся описании:

```
-- команда ADC — сложение с переносом.
-- сперва идёт название команды с её форматом,
-- потом код команды в микрокомандах.
cADC = command "adc" (rdddddrrrr [0,0,0,1,1,1]) $ do
    -- eFC — выражение, адресующее флаг C.
    -- Оно имеет размер 1 бит и должно быть расширено
    -- нулями до размера регистра.
    --
    -- regRr и regRd — значения регистров по адресам
    -- из переменных Rr и Rd.
    varRes $= regRr + regRd + zeroExt efC
```

```
        setflags
              False False -- вычитание ли, с переносом ли
              regRd regRr regRes -- откуда брать данные
              [FH,FS,FV,FN,FZ,FC] [] -- какие флаги менять
        writeRd regRes
        test_ADD_SUB False False True noVar

-- Тестирование команд сложения, вычитания и прочих.
--
-- AVCCM — монада описания команд AVR, частный вариант
-- монады описания команд.
--
-- Maybe (AVRVar a size) используется для указания переменной
-- процессора с константой (любого размера).
test_ADD_SUB :: (AVRProgMem a, NatInt size)
    ⇒ Bool → Bool → Bool → Maybe (AVRVar a size)
    → AVRCCM a ()
test_ADD_SUB testSub resultIsA withCarry mbReg =
    -- test_binary произведёт вызов кода команды с
    -- разными входными данными и сравнит результаты
    -- прогона с результатами вычисления функции answer.
    test_binary answer withCarry (not resultIsA)
        mbReg [FH,FS,FV,FN,FZ,FC]
    where
        answer a b inputC = (r,oc,hc,v)
            where
            -- здесь вычисляются результат команды r и
            -- всевозможные флаги (oc, hc и v) простейшим
            -- способом (сложением и сдвигами). Алгоритм
            -- отличается от алгоритма в документации, но
            -- смысл сохранён.
            -- Это помогло обнаружить ошибки, допущенные
            -- при редактировании методом copy-paste.
```

В определении команды ADC мы видим:

- Использование «макроподстановок» — «макро»-формата команды `rdddddrrrr`, «макро» `setFlags` вариантов вычисления флагов, макро тестирования команды.
- Контроль размерности — для включения флага C в вычисление нам надо привести его к необходимому типу путём расширения нулями.
- Общий вид описания команды достаточно естественнен. Ясен ход выполнения, он, практически императивен. Правда, действия вместо того, чтобы выполняться, накапливаются, но это дело второе. Здесь главное — внешний вид.

- Тесты команды встроены в определение команды, за счёт передачи параметров они сведены в одну строку текста.

В следующей главе рассказано о технических решениях, позволивших добиться, чтобы проверялось всё, что требуется.

2.3. Ядро языка описания

Ядро состоит из трёх модулей: модуля арифметики на типах данных для задания размерности данных, модуля описания микрокоманд и модуля формирования модели команды. Они напрямую соответствуют предметной области, и строились примерно в перечисленном порядке.

Все операции ядра описания команд параметризованы типом процессора, поэтому мы можем контролировать используемые возможности процессоров и размеры типов данных. Ядро позволяет нам решить первые две проблемы из списка 2.2.

Описание микрокоманд также бьётся на три части: присваиваемые выражения, разрешённые типы микрокоманд и сборка описаний.

2.3.1. Варианты описания

Приведём примеры описаний, когда есть возможность связывания, и её нет. Вот, как будет выглядеть гипотетический встроенный язык без возможности связывания:

```
-- где-то объявлена функция createDescription.
-- vs — это вложенные пары, начиная с ():
-- (), (a,()), (b,(a,()))...
createDescription :: VarList vs ⇒
    (vs → [Stat]) → Description

-- вот, как мы создадим наше описание:
ourDescription = createDescription
    -- изобретением имён заведует
    -- createDescription.
    (λ(x,(y,(z,())))) →
        -- здесь мы можем использовать
        -- данные нам идентификаторы.
        [ input x, y $= f x, load y z
        , y $= g z, ...])
```

С возможностью связывания всё немного проще:

```
-- где-то есть функция createDescription:
createDescription :: DescriptionAction ()
    → Description
```

```
-- вот наше описание:
description = createDescription $ do
    -- здесь и изобретение имени, и
    -- создание части описания одновременно.
    x ← input
    -- изобретение имени «по месту»:
    y ← declareVar
    y $= f x
    z ← declareVar
    load y z
    y $= g z
```

Если обобщить, то первый вариант описывается оператором ≫ из класса Monad, для второго требуется оператор ≫=:

```
class Monad m where
    return :: a → m a
    (≫=) :: m a → (a → m b) → m b
    (≫) :: m a → m b → m b
```

Если все операторы первого примера (`input`, `load`, `(var $= expr)`) имеют одинаковый тип `Stat`, то подойдёт и обычный список.

В нашем случае первый вариант оказался неудобным, поэтому мы выбрали второй.

2.3.2. Арифметика на типах данных и битовые вектора

Идея арифметики на типах данных [9] проста: мы выражаем некоторые вычисления внутри системы типов, контроль типов проводит вычисления во время компиляции и сообщает нам о найденных ошибках. В принципе, это похоже на проверку типов в языках Паскаль или Ада [1], только размеры массивов-векторов не обязаны быть константными, и типы по возможности выводятся автоматически, поэтому от многих объявлений типов можно отказаться.

Технически, простейший вариант основан на арифметике Пеано [17]. Мы создаём два пустых типа, `Z` и `S n` и с помощью семейств типов конструируем «функции» для сложения «численных типов»:

```
data Z                          -- 0
data S n = S {sArgument :: n} -- 1+n

type family Plus a b
--  0+x    ==    x
type instance Plus Z a = a
-- (1+x)+y == 1+(x+y)
type instance Plus (S a) b = S (Plus a b)
```

Недостаток простой арифметики Пеано — очень длинные выражения для размерностей. Число 8 в типах выглядит так: S (S (S (S (S (S (S (S Z))))))). Поэтому для задания размеров придётся завести несколько констант:

```
type ONE = S Z
type TWO = Plus ONE ONE
...
type SIZE12 = Plus SIX SIX
type SIZE16 = Plus EIGHT EIGHT
...
```

В процессе работы нам может понадобиться вычисленное целочисленное значение размера. Его легко получить с помощью класса:

```
class NatInt a where natInt :: a → Int
instance NatInt Z where natInt = const 0
instance NatInt n ⇒ NatInt (S n) where
    natInt (S n) = 1+natInt n
```

Класс NatInt сам по себе служит дополнительной проверкой ограничений во время компиляции — он не даёт задать в качестве параметра-размера произвольный тип данных.

При использовании параметризованных размерами типов мы уменьшаем количество ошибок, связанных с учетом размером данных (см. список в 2.2).

2.3.3. Описание микрокоманд

Ниже в описании микрокоманд часто фигурирует параметр `cpu`. Это тип, определяющий характеристики процессора — его возможности и размерности некоторых операндов. Характеристики процессора и других частей системы задаются как типы данных, чтобы иметь возможность как можно раньше контролировать правильность создаваемого кода интерпретатора.

Параметры определяются ассоциированными типами данных [14] для класса CPU:

```
class CPU cpu where
  type RegDataSize cpu
  type RegAddrSize cpu
  type MemDataSize cpu
  type MemAddrSize cpu
  type Vars cpu
  type Funcs cpu
```

Сперва идут четыре параметра — размеры данных для различных частей системы, регистрового файла и памяти. Они задаются в виде чисел в аксиомах Пеано.

Параметр `type Vars cpu` позволяет уточнить доступные конкретной реализации процессора переменные. Например, 8086 не поддерживал переменную «корень таблицы трансляции адресов», а 80386 и выше её поддерживают. Задав типы переменных процессора, мы можем ограничивать набор команд — команды управления

трансляцией адресов будут требовать расширенный набор переменных и не попадут в модель 8086.

Разные процессоры имеют разные специальные функции, которые не могут быть выражены в виде примитивных команд, что и служит основанием для наличия параметра `type Funcs cpu`. Примером специфичной для AVR функции может служить установка сигнала watchdog, которая не встречается во многих других процессорах.

Параметр `cpu` может быть параметризованным типом. Например, для MIPS32 и MIPS64:

```
data MIPS bits

type MIPS32 = MIPS SIZE32
type MIPS64 = MIPS SIZE64

instance NatInt bits ⇒ CPU (MIPS bits) where
  type RegDataSize (MIPS bits) = bits
  type RegAddrSize (MIPS bits) = FIVE -- 32 regs.
  type MemDataSize (MIPS bits) = bits
  type MemAddrSize (MIPS bits) =
      Sub bits (Log2 (MemDataSize (MIPS bits)))
  type Vars (MIPS bits) = ...
  type Funcs (MIPS bits) = ...
```

Код команд для обоих вариантов MIPS должен уметь работать с любым размером регистров, и проверка типов убедится, что это именно так.

Добавив параметр типа для указания типа шины, мы можем усложнить код — размеры адреса и данных при обращении к памяти, переменные процессора и функции процессора будут зависеть от типа шины. Таким образом, мы можем получить MIPS64, подключенный к 32-битной AXI — `(MIPS SIZE64 (BusAXI SIZE32))`, — или наоборот.

2.3.4. Присваиваемые в микрокомандах выражения

Мы выделили два типа переменных: переменные процессора, содержащие значение типа `Vars cpu`, и пронумерованные временные переменные:

```
data Var cpu size where
  CPUVar :: Vars cpu → Var cpu size
  TempVar :: Int → Var cpu size
```

Переменные процессора соответствуют частям состояния процессора и отражены в документации. Многие из них сохраняются между циклами выполнения различных команд. Временные переменные требуются для хранения значений, которые не потребуются при выполнении следующей команды.

Описание используемых выражений не несёт никаких особенных сюрпризов — это привычный набор из констант, переменных, бинарных и унарных выражений.

Есть даже тернарный оператор `ESelect`, аналог оператора `?:` из языка Си. Исключение составляют взятие значения из регистра EReg[1] и обращение к функциям процессора.

Отметить стоит лишь богатую параметризацию всех операций: размеры результатов и операндов связаны либо через параметр `cpu` (путём использования производных от него параметров-типов), либо через размеры операндов и результата конкретных операций, например, `BinOp`.

```
data Expr cpu size where
  EConst    :: NatInt size ⇒ Integer
            → Expr cpu size
  EVar  :: NatInt size ⇒ Var cpu size
        → Expr cpu size
  EReg  :: Expr cpu (RegAddrSize cpu)
        → Expr cpu (RegDataSize cpu)
  EBinary   :: (NatInt leftsize, NatInt rightsize
              , NatInt resultsize)
            ⇒ BinOp leftsize rightsize resultsize
                → Expr cpu leftsize → Expr cpu rightsize
                → Expr cpu resultsize
    ESelect :: Expr cpu ONE → Expr cpu size
            → Expr cpu size → Expr cpu size
    EUnary  :: (NatInt argsize, NatInt resultsize)
            ⇒ UnOp argsize resultsize
                → Expr cpu argsize → Expr cpu resultsize
    EFunc   :: Show (CPUFuncs cpu resultsize)
          ⇒ CPUFuncs cpu resultsize
          → Expr cpu resultsize
```

Типы `BinOp` и `UnOp` заслуживают пристального внимания[2]:

```
data BinOp leftsize rightsize resultsize where
    Plus, Minus, Mul,
      And, Or, Xor  :: BinOp size size size
    Mul2, Concat
                    :: BinOp size size (Plus size size)
    ShiftRL, ShiftRA,
      ShiftL        :: BinOp size shiftsize size
    Equal, NEqual,
      GetBit        :: BinOp size size ONE

data UnOp argsize resultsize where
    Negate, Complement :: UnOp argsize argsize
```

[1] Авторы считают, что внесение этого действия в набор общих выражений — не самый лучший ход, но из песни слова не выкинешь.

[2] Конструкторы с одним типом перечислены через запятую для уменьшения размера кода, Haskell не позволяет такой синтаксис.

```
    ZeroExt, SignExt   :: UnOp argsize resultsize
```

Тип `BinOp` содержит: а) операции с одинаковым размером операндов и результата операции, размер результата которых равен некоей функции (сумме) размеров операндов, б) операции, один из операндов которых не имеет фиксированного размера[3] и в) операции с фиксированным размером результата.

Тип `UnOp` не столь разнообразен. Он содержит две операции, не изменяющих размер, и две операции, изменяющих размер в произвольную сторону. Операция `ZeroExt` применяется для преобразований между числами разного размера, иногда не совсем безопасно.

Похожие операции есть в каждом языке описания аппаратуры. В VHDL/Verilog производится контроль размерности и используются операции на обычных целых числах или константах. В Bluespec Verilog в систему типов был заведен сорт (kind) Integer, который тоже позволяет использовать (ограниченный, по словам самих создателей Bluespec) набор функций над целыми значениями в типах. Мы же внесли недостающие операции в систему типов, практически не расширяя используемый язык, описанный в 2.3.2.

2.3.5. Типы микрокоманд

Микрокоманды имеют небольшую номенклатуру, буквально самое необходимое:

```
data Stat cpu where
    SAssign     :: NatInt size ⇒ Var cpu size
                → Expr cpu size → Stat cpu
    SLoad       :: CPU cpu ⇒ Var cpu (MemDataSize cpu)
                → Expr cpu (MemAddrSize cpu) → Stat cpu
    SStore      :: CPU cpu ⇒ Expr cpu (MemDataSize cpu)
                → Expr cpu (MemAddrSize cpu) → Stat cpu
    SWriteReg   :: CPU cpu ⇒ Expr cpu (RegAddrSize cpu)
                → Expr cpu (RegDataSize cpu) → Stat cpu
    -- Вызов функции, не требующей присваивания результата.
    --  Возвращается  результат  размером  0,  что  соответствует
тривиальному типу ().
    SStat       :: (CPU cpu, Show (CPUFuncs cpu Z))
           ⇒ CPUFuncs cpu Z → Stat cpu
    -- Разделитель этапов (носит характер указания).
    SStage      :: Stat cpu
```

Здесь упомянуты присваивание, загрузка слов из памяти, сохранение слов в память, запись в регистровый файл,[4] произвольное действие с окружающей средой и разграничение этапов.

[3] А мог бы: `Shift*:: BinOp size (Log2 size) size`, что было бы правильней.

[4] Здесь он один, но во многих архитектурах, даже MIPS [12], бывает и несколько регистровых файлов для сопроцессоров.

Последнее требуется для увеличения точности моделирования: например, общение с внешней памятью не происходит за один такт. Покомандная модель может полностью игнорировать разделение этапов; модель точностью выше может игнорировать только часть, учитывая зависимости; полностью точная модель будет вынуждена выполнить все циклы ожидания.

Количество применений `SStat`, вызывающих специфичные для процессора функции и представляющих особый случай для кодогенератора, как показывает практика, невелико.

2.3.6. Сборка описаний

Описание команд процессора в виде микрокоманд создаётся внутри параметризованной монады состояния (State Monad) поверх монады IO (последняя нужна для отображения результатов тестирования). Ниже приведёны типы, с помощью которых хранится описание процессоров:

```
-- Кортеж описания отдельной команды.
-- Состоит из:
-- имени, битового формата команды, состояния,
-- счётчика временных переменных и флага,
-- была ли команда протестирована.
data Cmd cpu
    = Cmd
      { cmdName :: String
      , cmdMatch :: [Match cpu]
      , cmdStats :: [Stat cpu]
      , cmdTempVarIndex :: Int
      , cmdTested :: Bool
      }

-- Список команд и флаг разрешения прогона тестов при генерации.
data CCMState cpu
    = CCMState
      { ccmStateCommands :: [Cmd cpu]
      , ccmStateTestsEnabled :: Bool
      }
    deriving Show

-- CCM расшифровывается как CPU Construction Monad.
type CCM cpu a = StateT (CCMState cpu) IO a
```

Примитив `command` создаёт начальное описание команды, с заданным форматом команды, нулевым счётчиком временных переменных и пустым списком микрокоманд. Микрокоманды постепенно добавляются в конец списка примитивом `addStat`, поверх которого реализованы все действия выше уровнем — оператор `$=`,

функции `writeRd` и `setFlags` и другие операции описания, не представленные в этой статье.

```
addStat :: Stat cpu → CCM cpu ()
addStat stat = modifyLastCmd $
    λ c → c { cmdStats = cmdStats c ++ [stat] }

modifyLastCmd :: (Cmd cpu → Cmd cpu) → CCM cpu ()
modifyLastCmd f = modify $ λccms → let
        (lastCmd:cmds) = ccmStateCommands ccms
    in ccms { ccmStateCommands = f lastCmd : cmds }

($=) :: NatInt size ⇒ Var cpu size
     → Expr cpu size → CCM cpu ()
v $= e = addStat (SAssign v e)
```

Отдельно стоит упомянуть выделение временных переменных. Размер получаемой переменной не фиксирован:

```
tempVar :: NatInt size ⇒ CCM cpu (Var cpu size)
```

Фиксация размера происходит во время вывода типов, на основе ограничений, полученных из использования переменной. Например, если переменная использовалась, как операнд сложения с 16-битным операндом, её размер автоматически станет 16-битным.

Вывод типов срабатывает не всегда, иногда требуется задать размер явно. У нас таких случаев оказалось всего два: байтовая переменная и переменная в 16 бит. Авторы считают, что это связано с недостаточно хорошо прописанными ограничениями в типах операций. Этот недостаток описания не привел к серьёзным проблемам в процессе реализации проекта.

То, что команды и даже их отдельные части являются объектами первого класса в Haskell, нашем языке-носителе, позволяет нам повторно использовать код без ограничений.

2.4. Тестирование

Тестирование производится ещё в одном преобразователе состояния `StateT`:

```
type TestMonad cpu a =
     StateT (TestMonadState cpu) IO a
```

Микрокоманды из описания поведения переносятся в состояние `TestMonadState` и интерпретируются с разными окружениями, в которых задаются значения индексов регистров-операндов, значения самих регистров-операндов, значения ячеек памяти и т. п.

Параметр `cpu`, помимо возможности задания значений переменных, определяет ещё и карты памяти всевозможных конфигураций. В общем случае, процессор может

обращаться к нескольким адресным пространствам — памяти программ и данных, пространству ввода-вывода, а в некоторых специализированных моделях существуют и другие варианты пространств. В процессе работы в составе системы процессор будет выполнять обращения по разным протоколам. Для тестирования достаточно просто разнести разные по смыслу пространства.

Код тестирования состоит в процедуре сброса окружения, установке его в нужное состояние, вызове интерпретатора микрокоманд и проверке правильности конечного состояния.

Интерпретатор микрокоманд выполняет действия, заданные в описании поведения текущей команды, рапортуя об отсутствующих (не установленных в процессе выполнения или как часть начального окружения команды) значениях регистров или адресов памяти.

2.5. Результаты и обсуждение

2.5.1. Результаты применения

Мы использовали язык описания для создания моделей процессоров семейства Atmel AVR: ATtiny10, ATmega8, ATmega128 и ATmega640. Они отличаются размером счётчика команд (9 бит для ATtiny10, 16 бит для ATmega128) и наличием или отсутствием некоторых команд. Другой особенностью этой линейки является отображение регистров процессора и портов ввода-вывода на пространство адресуемой памяти.

Использовалась внешняя шина APB с немного изменённым протоколом.

Все особенности процессоров были занесены в параметры моделей. Размеры счётчика команд и наличие различных команд были заданы в качестве параметра модели и проверялись статически. Карта памяти, также являвшаяся частью модели, учитывалась при кодогенерации.[5]

Код описания 125 команд содержит 710 полезных строк (5,7 строк на одну команду). Кодогенератор в Java состоит из трёх файлов, содержащих 287 полезных строк. Текст на Java, создаваемый по описанию команд, занимает более 1500 строк.

Работы над описанием команд и кодогенерацией производились тремя программистами: двое хорошо знали Haskell, а третий изучал его в процессе. Основная часть работ по описанию команд и кодогенерации была выполнена нашим коллегой в процессе изучения языка. Средняя скорость описания команд составила приблизительно три команды в день.[6] Один из опытных программистов на Haskell добавил в описания команд тесты, выполняемые перед кодогенерацией, второй улучшил кодогенерацию

[5]Обращение к некоторым регистрам AVR производится с помощью команд `IN`/`OUT`. Для таких адресов выполнять обращение во внешнюю память не требуется, однако требуется сохранить время выполнения команды.

[6]За время от начала работ до их завершения в связи с успешным окончанием тестирования, работы велись практически последовательно.

и подключил модели процессоров к нашей системе моделирования, выполнив на них функциональные тесты.

Поскольку система времени выполнения AVR-gcc использует достаточно большой процент команд AVR, мы использовали 4 функциональных теста на языке Си, основу которых составлял тест с вызовом `sprintf` и вычислениями с плавающей точкой. Прогон функциональных тестов выявил в модели всего 6 ошибок, две из которых относились к неполной документации в описании команд и были поправлены путём поиска дополнительной информации в Интернет, оставшиеся четыре находились в функционале команд условного перехода, трудных для тестирования без функционального теста.[7]

2.5.2. Возможные варианты использования

Поскольку мы имеем достаточно точное описание поведения команд вместе с их форматами, то можем получить реализацию процессора на любом языке описания аппаратуры.

Самый простой и неэффективный вариант описания не будет содержать конвейерного вычисления: команда будут выполняться до своего завершения, затем запустится следующая команда.

Мы можем сформировать и конвейерное выполнение, ведь мы знаем обо всех внутренних переменных процессора, и можем проанализировать команды попарно на предмет наличия межстадийных зависимостей. (Например, если одна команда пишет свой результат в определённую переменную состояния, а другая из неё читает, то требуется внести циклы ожидания во вторую команду до завершения работы первой. Подробнее см. [11].) Эта задача сложнее, но достижима.

Используя интерпретатор микрокоманд, мы можем сделать автоматический генератор кода для кодогенерации, например, для gcc [8] или BURG [4, 10]. В случае BURG кодогенерация идёт путём сравнения текущего дерева выражений программы с набором образцов, заданных в файле описания процессора. У каждого возможного дерева выражений есть определённая семантика. Для создания автоматического генератора кодогенераторов потребуется автоматическое перечисление (всех) возможных образцов дерева выражений программы, а также транслятор семантики образца в семантику изменения регистров и переменных программы. Похожие техники (правда, с ограниченным количеством примитивов) уже использовались в прошлом для устранения ветвлений при трансляции некоторых конструкций языка C [15].

В итоге мы можем получить практичное средство для полного описания новых процессорных архитектур, вплоть до полуавтоматического создания кодогенераторов.

[7] Полный алгоритм команд возврата из процедуры и вызова процедуры в документации указан не был. Эту ошибку без функциональных тестов выявить не удалось.

2.6. Заключение

В статье мы постарались показать, как можно попытаться сэкономить время разработки, планируя реакцию на возможные ошибки. Мы также постарались показать на примерах, какие средства язык программирования Haskell предоставляет для создания встроенных в язык проблемно-ориентированных языков (DSL). В нашей работе мы отталкивались от ошибок, которые можно совершить при создании моделей процессорных ядер. Современная система типов Haskell позволила нам защититься от большинства из них без особых затрат. Похожие рассуждения можно применить и в других предметных областях.

Используя встроенные проблемно-ориентированные языки, можно получить формальное описание задач предметной области, которое открывает возможности, изначально не предусмотренные техническим заданием.

Использование функционального языка не является препятствием для создания описаний процессоров даже при отсутствии соответствующего опыта у программиста. Более того, только в современных функциональных языках программирования система типов достаточно выразительна [14, 9] для указания достаточного количества ограничений.

Если сравнивать наш проект с проектом CGEN [5], открытым проектом для создания средств описания ассемблеров, дизассемблеров и симуляторов для различных архитектур микропроцессоров, то можно отметить следующее: во-первых, наш DSL ориентирован на создание моделей (симуляторов), хотя может быть расширен и информацией о командах для ассемблеров и дизассемблеров; во-вторых, мы используем метапрограммирование на типах для более ранней диагностики возможных ошибок, тогда как CGEN использует макроопределения языка Scheme. CGEN ориентирован на создание симуляторов на языке C, тогда как нашей задачей было получить возможность кодогенерации в разные ЯП для достижения максимально возможного быстродействия всего комплекса моделирования (в частности, с использованием параллельного выполнения разных частей на разных ядрах).

Литература

[1] Ada Reference Manual. — URL: http://www.adaic.org/standards/95lrm/html/RM-TTL.html (дата обращения: 21 мая 2010 г.). — ISO/IEC 8652:1995(E).

[2] AMBA Specification Rev2.0. — URL: http://www.arm.com/products/solutions/AMBA_Spec.html (дата обращения: 21 мая 2010 г.). — Описание семейства шин AMBA.

[3] An experimental domain specific language for template metaprogramming. — URL: http://goo.gl/btlV (дата обращения: 21 мая 2010 г.). — Доклад о применении языка Haskell для программирования в обобщённом стиле на C++/Boost.

[4] *C. W. Fraser R. R. Henry T. A. P.* BURG — fast optimal instruction selection and tree parsing. — 1992. — Описание генератора кодогенераторов на основе сравнения с образцом поверх деревьев внутреннего представления компиляторов.

[5] CGEN, the Cpu tools GENerator. — URL: http://sourceware.org/cgen (дата обращения: 21 мая 2010 г.). — Описание архитектур CPU.

[6] CoreConnect Bus Architecture. — URL: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture (дата обращения: 21 мая 2010 г.). — Шина CoreConnect фирмы IBM.

[7] Generic Libraries in C++ with Concepts from High-Level Domain Descriptions in Haskell. — URL: http://dsl09.blogspot.com/2009/07/here-are-talk-slides-as-pdf.html (дата обращения: 21 мая 2010 г.). — Доклад о применении DSL на Haskell для программирования в обобщённом стиле на C++.

[8] GNU Compiler Collection. — GCC, компилятор C, C++, Ada и многих других языков.

[9] *Kiselyov O.* Number-parameterized types. — 2005. — Целочисленная арифметика внутри системы типов Haskell.

[10] lcc: A portable C compiler. — URL: http://sites.google.com/site/lccretargetablecompiler/ (дата обращения: 21 мая 2010 г.). — Компилятор ANSI C с быстрой настройкой под выбранную архитектуру.

[11] Lecture 6: Pipelining Contd. — URL: http://www.stanford.edu/class/ee282h/handouts/Handout16.pdf (дата обращения: 21 мая 2010 г.). — Часть цикла лекций Стэнфордского университета про реализацию микропроцессорных архитектур.

[12] Microprocessor without Interlocked Pipeline Stages. — URL: http://en.wikipedia.org/wiki/MIPS_architecture (дата обращения: 21 мая 2010 г.). — Один из самых простых RISC-микропроцессоров.

[13] QEMU — a generic and open source machine emulator and virtualizer. — URL: http://wiki.qemu.org/Main_Page (дата обращения: 21 мая 2010 г.). — Эмулятор для большого числа процессоров и виртуальной периферии на большом числе платформ, включает в себя двоичную трансляцию.

[14] *Tom Schrijvers, Simon Peyton-Jones, Manuel M. T. Chakravarty, and Martin Sulzmann.* Type Checking with Open Type Functions. — 2008. — Семейства типов — простые вычисления над типами во время компиляции.

[15] *Torbjorn Granlund R. K.* Eliminating Branches using a Superoptimizer and the GNU C Compiler. — 1992. — Использование супероптимизации для улучшения кодогенерации GCC.

[16] WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. — URL: http://www.opencores.org/downloads/wbspec_b3.pdf (дата обращения: 21 мая 2010 г.). — Описание шины внутрикристального взаимодействия Wishbone.

[17] Арифметика Пеано. — URL: http://ru.wikipedia.org/?oldid=20954203 (дата обращения: 21 мая 2010 г.). — Статья в Wikipedia об аксиомах арифметики Пеано.

Введение в F#

Евгений Лазин, Максим Моисеев, Давид Сорокин
lazin@fprog.ru, moiseev@fprog.ru, dsorokin@fprog.ru

Аннотация

Данная статья призвана познакомить читателя с новым, приобретающим популярность языком программирования F#, который является достойным наследником традиций семейства языков ML. F# поддерживается компанией Microsoft как один из основных .NET языков в их флагманском продукте для разработчиков — Visual Studio 2010. Статья описывает основные возможности F# и ориентирована в основном на .NET программистов, которые хотели бы выйти за рамки уютного C#, погрузиться в функциональную парадигму и изучить новый язык. Кроме того, описание расширенных возможностей языка может показаться интересным для практикующих функциональных программистов.

This article is an introduction into the emerging new programming language F#, derived from ML. F# is supported by the Microsoft in its flagship development environment Visual Studio 2010. Article describes the main features of F#. The target audience of this article is .NET programmers looking for something new outside of the cosy C# world, ready to dive into functional paradigm and learn new languages. Description of the advanced language features would be of interest to seasoned functional programmers.

Обсуждение статьи ведётся в LiveJournal.

3.1. Введение

F# — это мультипарадигменный язык программирования, разработанный в подразделении Microsoft Research и предназначенный для исполнения на платформе Microsoft .NET. Он сочетает в себе выразительность функциональных языков, таких как OCaml и Haskell с возможностями и объектной моделью .NET.

История F# началась в 2002 году, когда команда разработчиков из Microsoft Research под руководством Don Syme решила, что языки семейства ML вполне подходят для реализации функциональной парадигмы на платформе .NET. Идея разработки нового языка появилась во время работы над Generic'ами — реализацией *обобщённого* программирования для Common Language Runtime. Известно, что одно время в качестве прототипа нового языка рассматривался Haskell, но из-за функциональной чистоты и более сложной системы типов потенциальный Haskell.NET не мог бы предоставить разработчикам простого механизма работы с библиотекой классов .NET Framework, а значит, не давал бы каких-то дополнительных преимуществ. Как бы то ни было, за основу был взят OCaml, язык из семейства ML, который не является чисто функциональным и предоставляет возможности для императивного и объектно-ориентированного программирования. Однако заметьте, что Haskell, хоть и не стал непосредственно родителем нового языка, тем не менее, оказал на него некоторое влияние[1]. Например, концепция вычислительных выражений (computation expressions или workflows), играющих важную роль для асинхронного программирования и реализации DSL на F#, позаимствована из монад Haskell.

Следующим шагом в развитии нового языка стало появление в 2005 году его первой версии. С тех пор вокруг F# стало формироваться сообщество. За счёт поддержки функциональной парадигмы язык оказался востребован в научной сфере и финансовых организациях. Во многом благодаря этому Microsoft решила перевести F# из статуса исследовательских проектов в статус поддерживаемых продуктов и поставить его в один ряд с основными языками платформы .NET. И это несмотря на то, что в последнее время всё большую активность проявляют динамические языки, поддержка которых также присутствует в .NET Framework. 12 апреля 2010 года свет увидела новая версия флагманского продукта для разработчиков — Microsoft Visual Studio 2010, которая поддерживает разработку на F# прямо из коробки.

3.2. Начало работы

Прежде чем начать знакомство непосредственно с языком, стоит вкратце рассказать об инструментах, которые могут понадобиться читателю для ознакомления с материалом статьи.

Исполняемый файл `fsi.exe`, входящий в комплект поставки F#, представляет

[1]Стоит учесть тот факт, что в Microsoft Research работает небезызвестный Simon Peyton Jones — один из ведущих разработчиков Haskell.

собой интерактивную консоль, эдакий REPL,[2] в которой можно быстро проверить работоспособность отдельных фрагментов кода на F#.

В состав современных инсталляционных пакетов F# входят также модули интеграции в Visual Studio 2008 и свободно распространяемую Visual Studio 2008 Shell, которые позволяют компилировать участки кода прямо из редактора. Это средство видится авторам наиболее удобным для первоначального знакомства с языком, так как открыв текст программы во встроенном редакторе кода, можно отправлять выделенные участки на исполнение простым нажатием комбинации клавиш Alt+Enter.

Исполняемый файл `fsc.exe` — непосредственно компилятор исходного кода F#, который читатель может использовать совместно со своим любимым текстовым редактором.[3]

Утилиты `fsc.exe` и `fsi.exe` отлично работают и под Mono, открытой реализацией .NET Framework.

Файлы, содержащие код на F#, обычно имеют следующие расширения:

- `.fs` — обычный файл с кодом, который может быть скомпилирован;
- `.fsi` — файл описания публичного интерфейса модуля. Обычно генерируется компилятором на основе кода, а затем редактируется вручную;
- `.fsx` — исполняемый скрипт. Может быть запущен прямо из Windows Explorer при помощи соответствующего пункта всплывающего меню или передан на исполнение в интерактивную консоль `fsi.exe`.

В некоторых источниках можно встретить в начале F# кода директиву `#light on`. Эта директива отключает режим совместимости синтаксиса с OCaml, делая отступы в коде значимыми (как, например в Python или Haskell). В последних версиях облегчённый режим включен по умолчанию, поэтому необходимости в указании директивы `#light` больше нет.

3.3. Основные возможности языка

Любая вводная статья об F# начинается с рассказа о том, как удачно этот язык совмещает в себе качества, присущие разным парадигмам программирования, не будем делать исключения и мы. Итак, как было сказано в определении — F# является мультипарадигменным языком, а это значит, что на нём можно реализовывать функции высших порядков, внутри которых исполнять императивный код и обёртывать всё это в классы для использования клиентами, написанными на других языках на платформе .NET.

[2] Read-eval-print loop — вариант интерактивной среды программирования, когда отдельные команды языка вводятся пользователем, результат исполнения которых тут же вычисляется и выводится на экран.

[3] По следующей ссылке можно найти довольно подробное описание интеграции F# в Emacs: http://samolisov.blogspot.com/2010/04/f-emacs.html.

F# функциональный

Начнём с парадигмы, которая наиболее близка (или, по крайней мере, наиболее интересна) читателям этого журнала — функциональной. Ни для кого не станет сюрпризом, что F#, будучи наследником славных традиций семейства языков ML, предоставляет полный набор инструментов функционального программирования: здесь есть алгебраические типы данных и функции высшего порядка, средства для композиции функций и неизменяемые структуры данных, а также частичное применение на пару с каррированием. Со слов экспертов в OCaml, в F# не хватает функторов. В силу своего практически полного незнания OCaml, авторам остаётся лишь констатировать факт, что функторов тут действительно нет. Но кто знает, может и появятся.

Все функциональные возможности F# реализованы в конечном итоге поверх общей системы типов .NET Framework. Однако этот факт не обеспечивает удобства использования таких конструкций из других языков платформы. При разработке собственных библиотек на F# следует предусмотреть создание объектно-ориентированных обёрток, которые будет проще использовать из C# или Visual Basic .NET.

F# императивный

В случаях, когда богатых функциональных возможностей не хватает, F# предоставляет разработчику возможность использовать в коде прелести изменяемого состояния. Это как непосредственно изменяемые переменные, поддержка полей и свойств объектов стандартной библиотеки, так и явные циклы, а также изменяемые коллекции и типы данных.

F# объектно-ориентированный

Объектно-ориентированные возможности F#, как и многое другое в этом языке, обусловлены необходимостью предоставить разработчикам возможность использовать стандартную библиотеку классов .NET Framework. С поставленной задачей язык вполне справляется: можно как использовать библиотеки классов, реализованные на других .NET языках, так и разрабатывать свои собственные. Следует отметить, однако, что некоторые возможности ОО языков реализованы не самым привычным образом.

И не только…

Подобное смешение парадигм, с одной стороны, может привести к плачевным результатам, а с другой — предоставляет больше гибкости и позволяет писать более простой код. Так, например, внутри стандартной библиотеки F# некоторые функции написаны в императивном стиле в целях оптимизации скорости исполнения.

Томаш Петричек в своём блоге [6] упоминает также о «языко-ориентированном» программировании. Мы тоже коснёмся этой темы ниже, а пока отметим лишь, что

F# отлично подходит как для написания встроенных DSL, то есть имитации языка предметной области средствами основного языка, так и для преобразования F# кода в конструкции, исполняемые другими средствами, например в SQL-выражения или в последовательности инструкций GPU. Кроме того, в комплект поставки F# PowerPack входят утилиты FsYacc и FsLex, являющиеся аналогами таких же утилит для OCaml, и позволяющие генерировать синтаксические и лексические анализаторы, а значит на F# вполне можно разработать свой собственный язык программирования.

3.3.1. Система типов

Каждая переменная, выражение или функция в F# имеет тип. Можно считать, что тип — это некий контракт, поддерживаемый всеми объектами данного типа. К примеру, тип переменной однозначно определяет диапазон значений этой переменной; тип функции говорит о том, какие параметры она принимает и значение какого типа она возвращает; тип объекта определяет то, как этот объект может быть использован.

F# — статически типизированный язык. Это означает, что тип каждого выражения известен на этапе компиляции, и позволяет отслеживать ошибки, связанные с неправильным использованием объектов определенного типа, до запуска программы. Помимо этого, F# — язык программирования со строгой типизацией, а значит, в выражениях отсутствует неявное приведение типов. Попытка использовать целое число там, где компилятор ожидает увидеть число с плавающей точкой, приведёт к ошибке компиляции.

Типы значений

Как и любой другой .NET-язык, F# поддерживает множество примитивных типов .NET, таких как `System.Byte`, `System.Int32` и `System.Char`. Как и в C#, для удобства разработчиков компилятор языка также поддерживает псевдонимы для этих типов: `byte`, `int` и `char` соответственно. Чтобы различать числовые литералы разных типов, используются суффиксы:

- uy — для значений типа `System.Byte`;
- s — для `System.Int16`;
- u — для `System.UInt32`;
- и др., с полным списком можно ознакомиться на странице документации.

Для преобразования значений из одного типа в другой нужно использовать одноимённые встроенные функции `byte`, `sbyte`, `int16`, `int`, `uint` и т. п. Ещё раз отметим, что эти функции следует вызывать явно.

Помимо прочего существует единственное значение типа `unit`, обозначаемое `()`. Оно обозначает отсутствие результата исполнения функции, аналогично `void` в C# или C++.

Типы функций

Функции в F# также имеют тип, обозначение которого выглядит, например, так:

```
string → string → string
```

Такая запись означает функцию, принимающую на входе два параметра типа `string`, и возвращающую `string` в качестве результата.

Типы шаблонных функций записываются аналогично:

```
'a → 'b → 'c
```

Заметьте, что вместо указания конкретных типов значений используются просто метки. Данное описание типа аналогично (но не идентично) следующему делегату из C#:

```
delegate C MyFunction<A, B, C>(A a, B b);
```

Это любая функция двух аргументов, которая возвращает некоторое значение. Типы `A`, `B` и `C` могут быть разными, но могут и совпадать.

Присвоение значений

Сопоставление имени и значения производится при помощи оператора связывания `let`:

```
> let value = 42;;

val value : int = 42
```

В данном контексте `value` — значение, а не переменная. При этом по умолчанию, все значения неизменяемые. Настоящую переменную можно создать, указав перед её именем ключевое слово `mutable`:

```
> let mutable value = 0
value ← 42;;
val mutable value : int = 42
```

Принципиальной разницы между значениями и функциями нет. Функции — это те же значения, а следовательно их точно так же можно использовать в выражениях, передавать в качестве параметров в другие функции и возвращать в качестве результата функции. При вызове функций параметры записываются последовательно и разделяются пробелами, при этом список параметров не надо заключать в круглые скобки, как это делается в Java или C#.

```
> let sqr a = a*a;;
val sqr : int → int
> sqr 2;;
val it : int = 4
```

Для удобства последнее выражение в теле функции автоматически возвращается в качестве результата функции.

Подробнее о функциях

В функциональном программировании есть два близких понятия: каррирование и частичное применение. Для того, чтобы объяснить каррирование, рассмотрим следующие два определения на C#:

```
int ClassicAdd(int a, int b){
  return a + b;
}

Func<int, int> CurriedAdd(int a){
  return delegate(int b){
    return a + b;
  }
}
```

Обе функции делают одно и то же: вычисляют сумму двух целых чисел, но при этом вторая из них является функцией от одного аргумента — первого слагаемого — и возвращает другую функцию от одного аргумента, внутри которой происходит вычисление результата. Вызовы этих функций будут выглядеть следующим образом: `int res = ClassicAdd(40, 2)` и `int res = CurriedAdd(40)(2)`. `CurriedAdd` можно переписать с использованием лямбда выражений следующим образом:

```
Func<int, int> CurriedAdd(int a){
  return b => a + b;
}
```

Функции, подобные `CurriedAdd` из нашего примера, которые берут свои аргументы по одному, каждый раз формируя функцию меньшего числа аргументов, называются каррированными, а преобразование обычных функций в такой вид — каррированием или каррингом. Функции в F# каррированы по умолчанию, а чтобы сделать функцию как в C#, следует объединить параметры в кортеж. Именно так F# и трактует функции стандартной библиотеки .NET Framework.

```
let curriedAdd a b = a + b

let classicAdd (a, b) = a + b
```

Соответствующие вызовы будут выглядеть так: `let res = curriedAdd 40 2` и `let res = classicAdd (40, 2)`. Можно заметить, что вызов некаррированной версии функции совпадает по синтаксису с таким же вызовом в C#.

Вернёмся к определению каррированной функции `CurriedAdd`. Что произойдёт, если вызвать её, передав только один аргумент: `var add40 = CurriedAdd(40)`? В случае с C# ответ вполне очевиден — в теле функции мы явно создаём анонимный метод, который и станет результатом вызова с одним параметром. То есть, `add40` будет функцией одного аргумента, которая будет уметь прибавлять 40 к любому числу, переданному ей в качестве аргумента.

В F# всё точно так же, за тем лишь исключением, что это не так явно следует из синтаксиса. Если написать `let add40 = curriedAdd 40`, то `add40` точно так же будет функцией одного аргумента. Подобная техника как раз и называется частичным применением.

Разница между этими двумя понятиями довольно тонка. Каррирование — это возможность функции принимать аргументы по одному, каждый раз возвращая новую функцию от меньшего числа аргументов. Частичное применение — это техника, позволяющая зафиксировать значение одного или нескольких первых параметров каррированной функции.[4]

Ещё одной важной особенностью функционального программирования, которая поддерживается F#, является композиция функций. В стандартной библиотеке F# определён целый ряд операторов, облегчающих конструирование более сложных функций из простых. Их можно условно разделить на два вида:

- Конвейерные операторы — передают значение, вычисленное одной функцией, на вход второй. Пожалуй, наиболее часто употребляемым оператором из этой группы можно назвать `|>`, определение которого выглядит так:

  ```
  let (|>) x f = f x
  ```

  Казалось бы, ничего сверхъестественного — лишь простая перестановка местами функции и её аргумента. Но это может быть очень удобно в случае, когда необходимо последовательно совершить несколько преобразований над одним исходным значением, например, списком:

  ```
  let list = [1..10]
  list
    |> List.map (fun i → i*i)
    |> List.iter (fun sq → printfn "Square: %d" sq)
  ```

  В данном примере мы сначала сформировали из исходного списка новый, путём возведения всех его элементов в квадрат с помощью библиотечной функции `List.map`, а затем значения из нового списка вывели на экран одно за другим с помощью функции `List.iter`. Возможность комбинировать функции таким образом является функциональным аналогом подхода, известного в объектно-ориентированных языках как «method chaining», который последнее время употребляется рядом с выражением «гибкий интерфейс». Вот пример подобного подхода из стандартной библиотеки .NET:

  ```
  var sb = new StringBuilder()
    .Append("Hello ")
    .Append("World!")
    .AppendLine();
  ```

[4] За более подробной информацией о каррировании и частичном применении читатель может обратиться к разделу 5 статьи «Элементы функциональных языков» [13].

В эту же группу операторов входят аналоги оператора |>, позволяющие работать с функциями более одного аргумента: ||> и |||>, а также их «обратные» варианты: <|, <|| и <|||.

- Операторы композиции — в отличие от конвейерных, не передают значения, а просто формируют новые функции.[5] Объявление одного такого оператора выглядит следующим образом:

```
let (>>) f g x = g (f x)
```

Если передать этому оператору все три аргумента сразу, не произойдёт ничего интересного. Но если вспомнить про частичное применение и опустить аргумент x, то результатом станет новая функция. Для примера реализуем аналог оператора композиции на C#.

```
Func<A, C> Compose<A, B, C>(
    Func<A, B> f,
    Func<B, C> g){
  return (x) => g( f(x) );
}
```

Вполне объяснимо наличие в стандартной библиотеке F# также и «обратного» оператора композиции <<.

Вывод типов

В отличие от большинства других языков программирования,[6] F# не требует явно указывать типы всех значений. Механизм вывода типов позволяет определить недостающие типы значений, глядя на их использование. При этом некоторые значения должны иметь заранее известный тип. В роли таких значений, к примеру, могут выступать числовые литералы, так как их тип однозначно определяется суффиксом. Рассмотрим простой пример:

```
> let add a b = a + b;;
val add : int → int → int
> add 3 5;;
val it : int = 8
```

Функция add возвращает сумму своих параметров и имеет тип int →int →int. Если не смотреть на сигнатуру функции, то можно подумать, что она складывает значения любых типов, но это не так. Попытка вызвать её для аргументов типа float или decimal приведёт к ошибке компиляции. Причина такого поведения кроется в механизме вывода типов. Поскольку оператор + может работать с разными типами

[5] О функциональной композиции можно также прочитать в разделе 6 статьи «Элементы функциональных языков» [13].

[6] Речь идёт, конечно, о промышленных языках, таких как Java и C#, хотя в последнем есть определённые подвижки в правильном направлении.

данных, а никакой дополнительной информации о том, как эта функция будет использоваться, нет, компилятор по умолчанию приводит её к типу `int →int →int`. Это можно проиллюстрировать на примере следующей сессии `FSI`:

```
> let add a b = a + b
  in add 2.0 3.0;;
val add : float → float → float
```

В данном примере функция `add` имеет тип `float →float →float`, так как компилятор может вывести её тип, глядя на её использование. F# также позволяет указывать аннотации для явного указания типов аргументов или возвращаемого значения функции.

```
> let add (a:float) (b:float) = a + b;;
val add : float → float → float
```

В данном примере типы аргументов функции заданы явно с помощью аннотаций. Указывать типы всех аргументов не обязательно: достаточно указать тип любого из них или тип возвращаемого значения, как в следующем примере:

```
> let add a b : float = a + b;;
val add : float → float → float
```

Единицы измерения (units of measure)

Как правило, программы работают не просто с переменными типа `int` или `float`. Например, метод `System.String.Length` возвращает не просто значение типа `int`, а количество символов в строке, и складывать его с другим значением того же типа `int`, содержащим количество секунд, прошедших с 1-го января 1970-го года, бессмысленно. Язык программирования F# позволяет программисту связать с любым значением числового типа произвольную единицу измерения. В дальнейшем, во время компиляции, будет выполняться проверка корректности приложения, на основе анализа размерностей.

Для создания единицы измерения нужно перед объявлением типа указать атрибут `Measure`.[7]

```
> [<Measure>] type m
[<Measure>] type s;;
```

После этого типы `s` и `m` можно использовать в качестве единиц измерения, для чего их следует указать в угловых скобках после соответствующего значения или типа:

```
> let a = 10<m>;;
val a : int<m> = 10
> let b = 2<s>;;
val b : int<s> = 2
```

[7] Рассмотрение атрибутов выходит за рамки данной статьи. Узнать о них подробнее можно по ссылке http://msdn.microsoft.com/en-us/library/dd233179(VS.100).aspx.

Значения a и b имеют одинаковые типы, но разные размерности, что означает, что их нельзя вычитать, складывать и сравнивать. Значения разных размерностей, однако, можно умножать и делить, результат операции в таком случае будет иметь составную размерность.

```
> let c = a + b;;
```

Попытка сложить метры с секундами приводит к ошибке *«error FS0001: The unit of measure 's' does not match the unit of measure 'm'»*.

```
> let v = a/b;;
val v : int<m/s> = 5
```

Значение v имеет составную размерность m/s. В предыдущем примере, единицы измерения m и s являются независимыми, то есть не выражаются друг через друга, однако F# позволяет определять зависимости между разными единицами измерения:

```
> [<Measure>] type Hz = s ^ -1;;
```

или просто:

```
> [<Measure>] type Hz = 1/s;;
```

Значения, имеющие размерность Hz, можно будет использовать в качестве значений с размерностью 1/s.

Обобщённые единицы измерения

Обычные функции, работающие с числовыми значениями разных типов, не будут работать со значениями, имеющими размерность. Это может быть неудобно, так как требует использовать функции приведения типа.

```
> let sqr a = a*a;;
val sqr : int → int

> let a = 2<s>;;
val a : int<s> = 2

> sqr (int a);;
val it : int = 4
```

Чтобы избежать ненужных преобразований, можно не указывать единицу измерения явно. Для этого вместо имени размерности следует использовать символ подчёркивания, в результате чего механизм вывода типов определит используемую размерность автоматически, иначе она будет обобщённой.

```
> let sqr (a : int<_>) = a*a;;
val sqr : int<'u> → int<'u ^ 2>

> sqr a;;
val it : int<s ^ 2> = 4
```

В данном примере, функция `sqr` принимает параметр типа `int`, имеющий любую размерность, либо не имеющий её вовсе.

Следует также отметить, что единицы измерения не вписываются в рамки общей системы типов .NET Framework, поэтому информация о них сохраняется в метаданных и может быть использована только из другой F#-сборки.

Некоторые встроенные типы

Стандартная библиотека F# предоставляет программисту ряд типов данных, предназначенных в первую очередь для создания программ в функциональном стиле.

Кортеж — упорядоченный набор элементов, экземпляр класса `Tuple`. Элементы кортежа могут иметь разные типы. Для создания нового экземпляра кортежа[8] нужно перечислить значения, входящие в него, через запятую:

```
> let tuple = "first element", 2, 3.0;;

val tuple : string * int * float = ("first element", 2, 3.0)
```

В данном примере `string * int * float` — это тип кортежа. Существуют две встроенные функции для работы с кортежами — `fst` и `snd`, возвращающие первый и второй элемент кортежа соответственно. Эти функции определены только для кортежей, состоящих из двух элементов. Для извлечения элементов из кортежа можно использовать оператор связывания. Для этого, в левой части, через запятую, должны быть перечислены идентификаторы, соответствующие элементам кортежа.

```
> let first, second, third = tuple;;

val third : float = 3.0
val second : int = 2
val first : string = "first element"
```

Список может содержать множество элементов одного типа. Для создания нового списка его элементы должны быть перечислены в квадратных скобках через точку с запятой.

```
> let lst = [ 1; 3; 6; 10; 15 ];;
val lst : int list = [1; 3; 6; 10; 15]
```

Чтобы создать новый список, вовсе не обязательно перечислять все его элементы. Существуют другие возможности: создание списка на основе диапазона значений, а также генераторы списков.[9] Для создания списка, содержащего диапазон значений, нужно задать его верхнюю и нижнюю границу:

[8] Следует иметь в виду, что экземпляры класса `Tuple` всегда создаются в куче. В случае, если это недопустимо, нужно использовать структуру.

[9] В англоязычной литературе чаще всего используется термин *list comprehension*, одним из вариантов перевода которого может быть *абстракция списка*.

```
> let a = [1 .. 10];;
val a : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Кроме того можно указать шаг приращения значений, который может быть отрицательным:

```
> let a = [1 .. 2 .. 10];;
val a : int list = [1; 3; 5; 7; 9]
```

В более сложных случаях можно использовать генераторы списков. Генератор списка — это фрагмент кода, заключённый в квадратные скобки, используемый для создания всех элементов списка. Элементы списка добавляются с помощью ключевого слова `yield`. С помощью такого выражения, например, можно получить список чётных чисел:

```
> let evenlst = [
  for i in 1..10 do
    if i % 2 = 0 then
      yield i
]

val evenlst : int list = [2; 4; 6; 8; 10]
```

Внутри генераторов списков[10] можно использовать практически любые выражения F#: `yield`, `for`, `if`, `try` и т.д.

Операции над списками

Списки в F# не являются экземплярами типа `System.Collections.Generic.List<T>`, и в отличие от последних — неизменяемы. Добавить или удалить элемент из списка нельзя, вместо этого можно создать новый список на основе существующего. Вообще, любые операции над списками в F# приводят к созданию нового неизменяемого списка.[11] Для добавления элемента в начало списка служит оператор '`::`', который является псевдонимом функции `Cons` из модуля `List`:

```
> let newlst = 0 :: lst;;
val newlst : int list = [0; 1; 3; 6; 10; 15]
> let newlst = List.Cons(0, lst);;
val newlst : int list = [0; 1; 3; 6; 10; 15]
```

Модуль `List` содержит множество функций, работающих со списками. Два списка можно объединить с помощью оператора конкатенации '`@`':

[10]List comprehensions — это частный случай использования такой возможности языка, как computation expressions, о которой будет рассказано ниже.

[11]Следует отметить, что копирования данных каждый раз не происходит. В новом списке всегда используются ссылки на элементы старого.

```
> let cclst = lst @ [21; 28];;
val cclst : int list = [1; 3; 6; 10; 15; 21; 28]
```

При обработке неизменяемых списков довольно часто требуется получить из исходного списка его первый элемент и список всех остальных элементов. Для этого служат функции `List.head` и `List.tail`, соответственно.[12]

Модуль `List` также содержит несколько важных функций высшего порядка, хорошо известных программистам, знакомым с другими функциональными языками программирования. Эти функции позволяют отказаться от использования циклов и явной рекурсии при обработке списков; кроме того, они упрощают формальное доказательство корректности алгоритма.

`List.map` создает новый список, применяя пользовательскую функцию ко всем элементам исходного списка. Её тип: `('a →'b)→ 'a list →'b list`. То есть, она принимает на вход пользовательскую функцию преобразования и исходный список, и возвращает новый список, с элементами другого типа.

```
> List.map (fun i → i*i) [1; 2; 3; 4];;
val it : int list = [1; 4; 9; 16]
```

В данном примере пользовательская функция возвращает квадрат своего аргумента.[13] В C# аналогичного эффекта можно добиться с помощью `Enumerable.Select` и дальнейшего преобразования результата к списку:

```
var list =
  new List<int>{1, 2, 3, 4};
var result =
  list.Select(i => i*i).ToList();
```

`List.reduce` выполняет операцию свёртки, её тип: `('a →'a →'a)→ 'a list →'a`. Пользовательская функция принимает на вход два аргумента: аккумулятор и следующий элемент списка и должна вернуть новое значение аккумулятора.

```
> List.reduce
    (fun acc i → i + acc) [1; 2; 3; 4; 5];;
val it : int = 15
```

В данном примере с помощью функции `List.reduce`, вычисляется сумма всех элементов списка.[14] При первом вызове функция, переданная пользователем, получает голову списка в качестве аккумулятора.

Функция `List.fold` в целом аналогична функции `List.reduce`, за исключением того, что тип элементов списка и тип аккумулятора могут быть разными. Описанная выше функция `List.reduce` является частным случаем `List.fold`.

[12] О другом способе разбиения списка на голову и хвост будет рассказано в разделе о сопоставлении с образцом.

[13] В примере использована анонимная функция. Анонимные функции аналогичны лямбда-выражениям из C#. Создаются они с помощью ключевого слова `fun`, за которым следует перечисление параметров, '→' и тело функции.

[14] Функция `List.sum` позволяет сделать то же самое намного проще.

Эта функция имеет тип `('a →'b →'a)→ 'a →'b list →'a`. В отличие от `List.reduce`, она содержит один дополнительный аргумент, который выступает в роли начального значения аккумулятора. Это можно проиллюстрировать на примере выражения, преобразующего список чисел в строку:

```
let lst = [1; 2; 3; 4]

> List.fold
    (fun acc i → acc + ", " + (string i))
    (string <| List.head lst)
    (List.tail lst);;

val it : string = "1, 2, 3, 4"
```

Поскольку тип результата отличается от типа элементов списка, использовать функцию `List.reduce` в данном случае нельзя. Итак, первый аргумент `List.fold` — функция с двумя аргументами: строкой и числом. Она прибавляет к строке-аккумулятору запятую, пробел и строковое представление числа — элемента списка. Второй аргумент функции `List.fold` — начальное значение аккумулятора, равное строковому представлению первого элемента списка. Третий и последний аргумент `List.fold` — обрабатываемый список. Так как строковое представление первого элемента списка уже содержится в аккумуляторе, туда передаётся только хвост списка.

Оба варианта функций свёртки списков реализованы в C# с помощью перегруженного метода `Enumerable.Aggregate`, принимающего 2 и 3 аргумента.

3.3.2. Последовательности, ленивость

Обычно выражения в F# вычисляются «энергично». Это означает, что значение выражения будет вычислено независимо от того, используется оно где-либо или нет. Например, если определить список с помощью генератора, то список будет создан в памяти целиком, независимо от дальнейшего его использования.

В противоположность жадному подходу существует стратегия ленивых вычислений, которая позволяет вычислять значение выражения только тогда, когда оно становится необходимо. Преимуществами такого подхода являются:

- производительность, поскольку неиспользуемые значения просто не вычисляются;
- возможность работать с бесконечными или очень большими последовательностями, так как они никогда не загружаются в память полностью;
- декларативность кода. Использование ленивых вычислений избавляет программиста от необходимости следить за порядком вычислений, что делает код проще.

Основной недостаток ленивых вычислений — плохая предсказуемость. В отличие от энергичных вычислений, где очень просто определить пространственную и временную сложность алгоритма, с ленивыми вычислениями всё может быть куда менее

очевидно. F# позволяет программисту самостоятельно решать, что именно должно вычисляться лениво, а что нет. Это в значительной степени устраняет проблему плохой предсказуемости, так как ленивые вычисления применяются только там, где это действительно необходимо, позволяя сочетать лучшее из обоих миров.

Тип данных `Lazy`

С помощью значения данного типа можно представить отложенное вычисление.

```
> let lv = Lazy<_>.Create(
    fun () → printfn "Eval"; 42);;
val lv : System.Lazy<int> = <unevaluated>

> lv.Value;;
Eval
val it : int = 42

> lv.Value;;
val it : int = 42
```

При первом обращении к свойству `Value` происходит вычисление значения. При последующих — используется ранее вычисленное значение. Аналогично использовать ключевое слово `lazy`:

```
> let lv = lazy(printfn "Eval"; 42);;
val lv : Lazy<int> = <unevaluated>
```

Последовательности

Последовательность — всего лишь синоним для .NET-интерфейса `IEnumerable<T>`, поэтому последовательности легко могут быть использованы из сборок, написанных на других .NET-языках.

Последовательности создаются похожим со списками образом. Можно задать последовательность в виде диапазона:

```
> let intseq = seq {1 .. System.Int32.MaxValue};;
val intseq : seq<int>
```

Попытка создать список такого же размера приведёт к ошибке.

Последовательность может быть создана при помощи *генератора последовательности (sequence comprehension)*, аналогично генераторам списка. В отличии от генератора списка, он не создает все элементы последовательности сразу. Выполнение кода прерывается после того, как очередной элемент последовательности был создан с помощью ключевого слова `yield`. Дальнейшее выполнение кода будет продолжено тогда, когда потребуется следующий элемент. Код генератора последовательности записывается в фигурных скобках, расположенных после идентификатора `seq`:

```
let symbols = seq {
  for c in 'A' .. 'Z' do
    yield c
    yield System.Char.ToLower(c)
}
```

Генераторы последовательностей F# аналогичны итераторам, которые появились в C# версии 2. Так, например, приведённый выше код аналогичен следующему:

```
// ABC == "ABC...YZ"
IEnumerable<char> GetLowerABC(){
  for(ch in ABC){
    yield ch
    yield System.Char.ToLower(ch);
  }
}
```

Однако возможности генераторов последовательностей несколько шире. Например, они могут быть вложенными или рекурсивными. Чтобы включить в одну последовательность элементы другой, используется ключевое слово `yield!` (читается «Yield bang»):

```
let rec collatz n = seq {
  yield n
  match n%2 with
  | 0 → yield! collatz (n/2)
  | _ → yield! collatz (3*n + 1)
}
```

В данном примере функция `collatz` возвращает все числа, входящие в последовательность Коллатца, начиная с `n`. Во-первых, функция рекурсивна (поэтому объявлена с помощью ключевого слова `rec`). Во-вторых, функция возвращает последовательность, так как её тело состоит из одного sequence expression. При этом последовательности, возвращаемые рекурсивными вызовами функции `collatz`, объединяются с помощью оператора `yield!` в одну *плоскую* последовательность.

Модуль `Seq` содержит функции для работы с последовательностями. В нём есть как аналоги функций `length`, `filter`, `fold`, `map` модуля `List`, так и специфические для последовательностей функции, такие как `take` и `unfold`.

3.3.3. Сопоставление с образцом (Pattern matching)

Сопоставление с образцом — основной способ работы со структурами данных в F#.

Эта языковая конструкция состоит из ключевого слова `match`, анализируемого выражения, ключевого слова `with` и набора правил. Каждое правило — это пара образец–результат. Всё выражение сопоставления с образцом принимает значение того правила, образец которого соответствует анализируемому выражению. Все правила

сопоставления с образцом должны возвращать значения одного и того же типа. В простейшем случае в качестве образцов могут выступать константы:

```
> let xor x y =
      match x, y with
      | true,  true  → false
      | false, false → false
      | true,  false → true
      | false, true  → true
;;
val xor : bool → bool → bool
```

В правилах сопоставления с образцом можно использовать символ подчеркивания («*wildcard*»), если конкретное значение неважно.

```
> let testAnd x y =
      match x, y with
      | true, true → true
      | _ → false;;

val testAnd : bool → bool → bool

> testAnd false false;;
val it : bool = false
```

Если набор правил сопоставления не покрывает все возможные значения образца, компилятор F# выдаёт предупреждение. На этапе исполнения, если ни одно правило не будет соответствовать образцу, будет сгенерировано исключение `MatchFailureException`.

Помимо констант в правилах сопоставления с образцом можно использовать имена значений. Это нужно, чтобы извлечь данные из образца и связать их с именем.

```
> let printValue x =
      match x with
      | 0 → printfn "is zero"
      | a → printfn "is %d" a;;
```

Конечно, данный пример выглядит надуманным, так как в выражении можно использовать непосредственно значение-образец, а вместо именованного значения — *wildcard*. Однако, это может быть очень удобно в тех случаях, когда мы имеем дело с более сложными данными, например с кортежами или списками. Тогда с помощью правила сопоставления с образцом можно разобрать сложную структуру данных на составные части.

Разные правила могут быть объединены между собой.

```
> let testOr x y =
      match x, y with
      | (_, true) | (true, _) → true
```

```
        | _ → false;;

val testOr : bool → bool → bool
```

В этом примере первое правило сработает в том случае, если один из аргументов равен `true`.

В некоторых случаях, простого сопоставления бывает мало, и требуется использовать в правилах более сложную логику. Тогда можно указать дополнительные ограничения после ключевого слова `when`:

```
> let testOr x y =
      match x, y with
      | _ when x = true → true
      | _ when y = true → true
      | _ → false;;
```

Активные шаблоны (active patterns)

Сопоставление с образцом — намного более выразительный механизм, чем обычные условные выражения, однако их не всегда удобно использовать. К примеру, в правилах сопоставления с образцом нельзя вызывать функции и приходится пользоваться ограничениями `when`. Рассмотрим следующий код:

```
open System.IO

type FileType = Document | Application | Unknown

let fileType filename =
    match filename with
    | _ when Path.GetExtension(filename) = ".doc"
      → Document
    | _ when Path.GetExtension(filename) = ".odf"
      → Document
    | _ when Path.GetExtension(filename) = ".exe"
      → Application
    | _ → Unknown
```

Сопоставление с образцом здесь не даёт каких-либо преимуществ по сравнению с обычным сравнением.

Для восполнения этого и подобных пробелов, служат активные шаблоны — особые функции, которые могут быть использованы в правилах сопоставления с образцом.

Одно-вариантные активные шаблоны[15] позволяют выполнить простое преобразование данных, к примеру, из одного типа в другой. Это может быть полезно тогда, когда данные исходного типа не могут быть использованы в правилах сопоставления

[15]Single-case active patterns.

с образцом, либо когда нам нужно использовать более сложную логику при выполнении сравнения. Активный шаблон этого типа — простая функция одного аргумента, имя которой при объявлении заключено в прямые, а затем в круглые скобки.[16]

Предыдущий пример можно переписать следующим образом:

```
open System.IO

type FileType = Document | Application | Unknown

let (|FileExtension|) filePath =
  Path.GetExtension(filePath)

let fileType filename =
      match filename with
      | FileExtension ".doc"
      | FileExtension ".odf" → Document
      | FileExtension ".exe" → Application
      | _ → Unknown
```

Здесь `FileExtension` — активный шаблон, который просто возвращает расширения файла. За именем шаблона следует его результат. Если результат представлен константой, то результат сравнивается с ней и при совпадении, правило считается выполненным. Если результат — именованное значение, правило считается выполненным, а результат выполнения функции-активного шаблона можно получить по имени. Например:

```
let ext = match "file.txt" with
  | FileExtension ext → ext
```

Значение `ext` будет содержать расширение файла или пустую строку, если файл не имеет расширения.

Частичные активные шаблоны (partial-case active patterns) используются в тех случаях, когда преобразование не всегда возможно. Функция-шаблон должна возвращать значение типа `option<T>`, равное `None`, если преобразование невозможно.[17]

```
type ValueType =
  | Integer of int
  | Float   of float
  | Boolean of bool
  | Unknown

let makeParsePattern tryParseFn str =
  let success, result = tryParseFn(str)
  if success then
```

[16] Banana clips.

[17] Тип данных `option<T>` для простоты можно считать аналогичным `Nullable<T>`, с той разницей, что непустые значения создаются при помощи функции `Some`, а пустые — `None`.

```
      Some result
    else
      None

let (|IsInteger|_|) =
  makeParsePattern System.Int32.TryParse
let (|IsFloat|_|)   =
  makeParsePattern System.Double.TryParse
let (|IsBoolean|_|) =
  makeParsePattern System.Boolean.TryParse

let parseStr str =
  match str with
  | IsInteger x → Integer x
  | IsFloat   x → Float   x
  | IsBoolean x → Boolean x
  | _ → Unknown
```

В данном примере определяется тип `ValueType`, значения которого представляют собой результаты разбора строки.[18] Функция `makeParsePattern` используется для создания активных шаблонов. Далее определяются шаблоны: `IsInteger`, `IsFloat` и `IsBoolean`. Эти шаблоны являются функциями, которые принимают один аргумент строкового типа и возвращают опциональное значение. *Частичный активный шаблон* задаётся с помощью конструкции `(|`«имя»`|_|)`. Как и в случае *single-case* шаблона, на вход функции передается значение-образец. Если функция вернула `None`, считается, что значение не удовлетворяет правилу.

Активные шаблоны могут быть параметризованными. При этом в `match`-выражении указывается сначала имя шаблона, затем параметры шаблона, а потом его результат. В качестве иллюстрации можно переписать предыдущий пример следующим образом:

```
type FileType =
    Document of string
  | Application of string
  | Unknown

let (|FileExtension|_|) fileExt filePath =
  let ext = Path.GetExtension(filePath)
  if ext = fileExt then
    Some <| Path.GetFileNameWithoutExtension(filePath)
  else
    None

let fileType filename =
  match filename with
```

[18] Этот тип является размеченным объединением, о которых будет рассказано чуть позже.

```
    | FileExtension ".doc" name → Document name
    | FileExtension ".odf" name → Document name
    | FileExtension ".exe" name → Application name
    | _ → Unknown
```

Результат шаблона используется здесь для захвата имени файла без расширения, а параметр — для определения типа.

В предыдущих примерах мы классифицировали входные значения, используя активные шаблоны и размеченные объединения. Предполагается, что в дальнейшем значение, которое возвращает функция `parseStr` или `fileType`, будет в свою очередь разобрано с помощью сопоставления с образцом.[19] Можно предположить, что это будет происходить примерно так:

```
let classify a =
  match a with
  | Integer i → printfn "integer %d" i
  | Float   f → printfn "float %f" f
  | Boolean true  → printfn "true"
  | Boolean false → printfn "false"
  | Unknown → printfn "unknown value"

> classify <| parseStr "42";;
integer 42
val it : unit = ()
```

Многовариантные активные шаблоны (multi-case active patterns) позволяют упростить классификацию данных при использовании размеченных объединений. Например:

```
let (|Integer|Float|Boolean|Unknown|) input =
  match Int32.TryParse(input) with
  | true, result → Integer result
  | false, _ →
    match Double.TryParse(input) with
    | true, result → Float result
    | false, _ →
      match Boolean.TryParse(input) with
      | true, result → Boolean result
      | false, _ → Unknown

let classify a =
  match a with
  | Integer i → printfn "integer %d" i
  | Float   f → printfn "float %f" f
  | Boolean true  → printfn "true"
  | Boolean false → printfn "false"
```

[19] Поскольку это наиболее естественный способ работы с размеченными объединениями.

```
    | Unknown → printfn "unknown value"

> classify "42";;
integer 42
val it : unit = ()
```

Нам больше не нужна функция `parseStr`, которая занималась разбором строки и возвращала элемент размеченного объединения.

3.3.4. Пользовательские типы данных

F# поддерживает различные пользовательские типы данных, такие как записи, размеченные объединения и классы. Помимо этого, в полной мере поддерживается параметрический полиморфизм: пользовательские типы могут быть обобщенными, так же как и функции.

В следующем примере создаётся тип-запись с двумя полями.

```
> type Person = { Name : string; Age : int };;
type Person =
  {Name: string;
  Age: int;}

> let mary = { Name = "Mary"; Age = 22 };;
val mary : Person = {Name = "Mary"; Age = 22;}

> let age = mary.Age;;
val age : int = 22
```

Записи F# аналогичны структурам из C# или C++ и могут использоваться для группировки нескольких логически связанных значений в одну сущность, оставляя при этом, в отличие от кортежей, возможность доступа к отдельным элементам по имени. Доступ к полям производится при помощи знакомого синтаксиса: через точку.

Заметьте, мы не указывали тип значения `mary` явно: компилятор смог вывести его самостоятельно на основе того, какие поля использовались при создании экземпляра. Механизм вывода типов работает с пользовательскими типами данных, в большинстве случаев избавляя от необходимости указывать типы явно.

Размеченное объединение — это алгебраический тип данных,[20] такой же как `data` в `Haskell`.[21] Определение такого типа состоит из названия и списка конструкторов с параметрами. Понятие конструктора в данном случае не совсем совпадает с принятым в объектно-ориентированном программировании толкованием, тем не менее, это тоже функция, вызов которой приводит к появлению нового экземпляра. Отличие от конструкторов классов состоит в том, что конструктор размеченного объединения

[20] За подробным описанием теоретической базы, лежащей в основе АТД, рекомендуется обратиться к статье с одноимённым названием [12].

[21] В англоязычной литературе по F# для обозначения данного понятия используется термин Discriminated Union, однако Tagged Union обозначает то же самое.

может использоваться в шаблонах механизма сопоставления с образцом для «разборки» объекта на составные части.

```
type MyDateTime =
  | FromTicks of int
  | FromString of string
```

С объектно-ориентированной точки зрения такой тип выглядит как небольшая иерархия классов, где `MyDateTime` выступает в роли базового класса, а `FromTicks` и `FromString` — классы-наследники. Таким образом, несмотря на то, что размеченное объединение является специфической для F# структурой данных, её можно использовать в публичных API, которые будут вызываться из других .NET языков, хотя такое использование будет далеко не самым удобным.

.NET-перечисления, которые в C# объявляются при помощи ключевого слова `enum`, в F# также объявляются как размеченные объединения. Правда, такое объединение не может иметь параметризованных конструкторов. Кроме того, каждому конструктору должно соответствовать числовое значение.

```
type Color =
  | None  = 0
  | Red   = 1
  | Green = 2
  | Blue  = 3
```

Аналогичный тип можно объявить на C# следующим образом:

```
enum Color {
  None  = 0,
  Red   = 1,
  Green = 2,
  Blue  = 3
}
```

Объекты и классы

F# поддерживает два способа объявления классов: явный и неявный. Первый подходит в тех случаях, когда программисту требуется контроль над тем, как создается объект класса и какие поля он содержит. Второй позволяет переложить большую часть работы на компилятор.

В случае явного объявления класса программист должен объявить поля класса и хотя бы один конструктор:

```
type Book =
  val title : string
  val author : string
  val publishDate : DateTime
```

```
  new (t, a, pd) = {
    title = t
    author = a
    publishDate = pd
  }
```

С помощью ключевого слова `val` объявляются члены-данные класса, а с помощью ключевого слова `new` создаётся конструктор класса. Члены-данные класса обязательно должны инициализироваться в конструкторе, а точнее — внутри *выражения-конструктора (constructor expression)*, которое записывается в фигурных скобках после символа равенства. Если какое-либо поле не будет инициализировано, компилятор выдаст ошибку. Внутри выражения-конструктора можно только инициализировать поля класса, причем каждое поле — только один раз: попытка сделать что-нибудь ещё опять же приведёт к ошибке компиляции.

На первый взгляд это ограничение кажется неразумным, так как иногда всё же требуется выполнить какие-либо действия во время создания экземпляра класса. Например выполнить валидацию параметров конструктора или записать что-нибудь в лог. На самом деле всё не так печально: в конструкторе можно не только инициализировать члены-данные, но и добавить произвольный код, который будет выполняться до инициализации. Он записывается перед *constructor expression*. Также можно написать код, который будет выполняться после инициализации полей, для чего следует использовать блок `then` после соответствующего выражения-конструктора.

```
type Book =

  val title : string
  val author : string
  val publishDate : DateTime

  new (t:string, a:string, pd) =

    if t.Length = 0 then
      failwithf "Book title is empty (%s, %A)" a pd

    if a.Length = 0 then
      failwithf "Book author is empty (%s, %A)" t pd

    {
      title = t
      author = a
      publishDate = pd
    }
    then
      printfn "New book is constructed %s %s %A" t a pd
```

В данном примере перед инициализацией полей класса происходит проверка пере-

данных параметров, а после неё следует вывод сообщения об успешном создании экземпляра. Обратите внимание, что код конструктора не может обращаться к членам-данным объекта иначе, чем через выражение-конструктор.

Все члены-данные по умолчанию неизменяемы. Так же, как и в случае обычных значений, это поведение можно изменить с помощью ключевого слова `mutable`.

Во многих случаях классы можно объявлять намного проще. Для этого нужно сразу после имени класса в круглых скобках перечислить параметры его основного конструктора. Эти параметры будут доступны для использования в методах класса, при этом объявлять их явно с помощью ключевого слова `val` не нужно.

```
type Book(title:string,
          author:string,
          publishDate:DateTime) =
  member this.Title = title
  member this.Author = author
```

В этом случае конструктор класса и его члены-данные создаются неявно. Конструктор класса принимает три параметра, но сам класс будет иметь два поля: `author` и `title`. Поле `publishDate` создано не будет, так как нигде не используется. Попытка использовать `val` в классе, объявленном с использованием неявного синтаксиса приведёт к ошибке компиляции.

Класс, объявленный неявно, может иметь более одного конструктора, которые также объявляются с помощью ключевого слова `new`, за которым следует список параметров и код конструктора. Но, в отличие от явного объявления класса, в этом случае мы не можем использовать *constructor expression*. Вместо этого конструктор обязательно должен вызывать основной конструктор, созданный неявно, и передавать в него соответствующие параметры.

```
type Book(title:string,
          author:string,
          publishDate:DateTime) =

  member this.Title = title
  member this.Author = author
  member this.PublishDate = publishDate

  new (str:string) =
    let parts = str.Split(';')
    let title = parts.[0]
    let author = parts.[1]
    let publishDate = DateTime.Parse(parts.[2])
    new Book(title, author, publishDate)
    then
      printfn "Created from string - %s" str
```

В этом примере определён дополнительный конструктор, который получает на вход строку, разбирает её и вызывает основной конструктор.

Внутри неявного определения класса можно объявлять переменные, используя оператор связывания `let`, а также выполнять произвольный код с помощью ключевого слова `do`.

```
type Book(title:string, author:string, publishDate:DateTime) =

  do if title.Length = 0 then
    failwithf "Book title is empty (%s, %A)" author publishDate
  do if author.Length = 0 then
    failwithf "Book author is empty (%s, %A)" title publishDate

  let mutable publishDate = publishDate

  member this.Title = title
  member this.Author = author
  member this.PublishDate = publishDate
  member this.SetPublishDate(newdate:DateTime) =
    publishDate ← newdate

  new (str:string) =
    let parts = str.Split(';')
    let title = parts.[0]
    let author = parts.[1]
    let publishDate = DateTime.Parse(parts.[2])
    new Book(title, author, publishDate)
    then
      printfn "Created from string %s" str
```

Здесь добавлены проверки для параметров основного конструктора, которые будут выполняться во время создания объекта независимо от того, с помощью какого конструктора объект создаётся. Помимо этого, с помощью оператора связывания здесь объявлена переменная класса `publishDate` и метод класса `SetPublishDate`, с помощью которого можно изменять значение этой переменной.

Классы, как и записи или размеченные объединения, могут быть обобщёнными. Для этого нужно после имени класса в угловых скобках перечислить обобщённые параметры класса:

```
type Point<'a>(x:'a, y:'a) =
member this.X = x
member this.Y = y
```

Создать экземпляр класса можно следующим образом:

```
> let p = Point(1.0, 1.0);;
val p : Point<float>
```

Свойства и методы

В предыдущих примерах методы уже использовались, теперь настало время поговорить о них подробнее. Объявление метода или свойства начинается с ключевого слова `member`, за которым следует имя `self`-идентификатора. Это имя используется для обращения к членам класса внутри метода или свойства. В случае, если метод или свойство статические, `self`-идентификатор указывать не нужно. В отличие от C#, здесь нет неявной переменной `this`, которая передаётся в каждый нестатический метод класса и предоставляет доступ к объекту, для которого вызван этот метод. Вместо этого, разработчик волен сам выбирать удобное имя, под которым будет известен текущий объект.

```
type Point<'a>(x:'a, y:'a) =

let mutable m_x = x
let mutable m_y = y

member this.Reset(other:Point<'a>) =
  m_x ← other.X
  m_y ← other.Y

member this.X
  with get() = m_x
  and  set x = m_x ← x

member this.Y
  with get() = m_y
  and  set y = m_y ← y
```

Здесь класс `Point` имеет два свойства и один метод. Свойства `X` и `Y` доступны для чтения и для записи. Если свойство не предполагает возможности чтения или установки нового значения, соответствующую часть объявления можно опустить.

Видимость методов, свойств, полей и классов можно изменять с помощью ключевых слов `public`, `private` и `internal`. В отличие от других .NET языков, F# не поддерживает модификатор `protected`, однако правильно работает с защищёнными методами и свойствами классов, созданных с использованием других языков программирования.

Ключевые слова `sealed` или `abstract`, которыми можно помечать классы в C#, не поддерживаются в F# на уровне языка. Тем не менее, в стандартной библиотеке есть соответствующие атрибуты: `Sealed` и `AbstractClass`, которые обрабатываются компилятором и приводят к ожидаемому результату.

В F# полностью поддерживается перегрузка методов. Можно определить несколько методов, имеющих одинаковое имя, но отличающихся типом и количеством параметров.

Наследование

Как и любой объектно-ориентированный язык, F# позволяет создать новый класс, наследующий свойства уже существующего.

При явном описании класса-наследника необходимо после ключевого слова `inherit` указать имя базового класса, и вызвать его конструктор в *constructor expression* с помощью того же ключевого слова.

```
type Base =
  val a : int

  new (a:int) = {a = a}

type Derived =
  inherit Base

  val b : int

  new (a:int, b:int) = {
    inherit Base(a)
    b = b
  }
```

При неявном объявлении класса аргументы для вызова конструктора указываются сразу же после объявления базового класса.

```
type Base =
  val a : int

  new (a:int) = {a = a}

type Derived(a:int, b:int) =
  inherit Base(a)

  let b = b
```

Класс-наследник может уточнять либо переопределять поведение базового класса переопределяя его методы. В отличие от C#, все члены класса, которые могут быть переопределены, являются абстрактными и объявляются при помощи ключевого слова `abstract`. Но, опять же в отличие от C#, абстрактный метод может иметь реализацию по умолчанию, которая задаётся с помощью ключевого слова `default`.

```
type Vertex = Point<float>

type Shape =

  new () = {}
```

```
    abstract NumberOfVertexes : int
    default this.NumberOfVertexes =
      failwith "not implemented"

    abstract Item : int → Vertex
    default this.Item ix =
      failwith "not implemented"

type Line(a:Vertex, b:Vertex) =
  inherit Shape()

  override this.NumberOfVertexes = 2

  override this.Item (ix:int) =
    match ix with
    | 0 → a
    | 1 → b
    | _ → failwith "index out of range"
```

В данном примере мы создали класс `Shape`, который является базовым для всех геометрических фигур. Он имеет одно абстрактное свойство и абстрактный метод, которые имеют реализацию по умолчанию и переопределяются в наследнике. Странное объявление типа `Vertex` в начале примера просто создаёт псевдоним для типа `Point<float>`.

```
> let ln = Line(Vertex(0.0, 0.0), Vertex(1.0, 1.0));;
val ln : Line

> ln.[0];;
val it : Vertex = FSI_0145+Point‘1[System.Double] {X = 0.0;
Y = 0.0;}
> ln.[1];;
val it : Vertex = FSI_0145+Point‘1[System.Double] {X = 1.0;
Y = 1.0;}
> ln.NumberOfVertexes;;
val it : int = 2
```

Метод `Item` является методом-индексатором, который позволяет обращаться к объекту с помощью оператора ‘`.[]`’ (обратите внимание на точку перед квадратными скобками, она обязательна). Аналогичные конструкции в C# объявляются при помощи специального синтаксиса индексаторов `this[]`.

Интерфейсы — важная часть платформы .NET. Они позволяют описать контракт, который обязуется выполнить класс, реализуя методы данного интерфейса. Если обычное наследование реализует отношение *«является частным случаем»* и применяется для расширения базового класса, то наследование от интерфейса реализует отношение *«поддерживает»* и применяется для того, что бы определить возможные

точки соприкосновения разных подсистем, которые могут ничего не знать друг о друге, кроме того, что одна из них поддерживает нужный интерфейс.

Интерфейс в F# — это просто чистый абстрактный класс.

```
type ICommand =
  abstract Execute : unit → unit

type DoStuff() =

  interface ICommand with
    override this.Execute () =
      printfn "Do stuff"
```

Реализация интерфейса отличается от наследования класса, что впрочем логично. Нелогичным может показаться то, что нельзя вызывать перегруженные методы интерфейса — это приведет к ошибке компиляции. Чтобы вызвать метод интерфейса, нужно выполнить приведение типа с помощью оператора '**:>**'.

```
> let cmd = DoStuff();;
val cmd : DoStuff

> (cmd :> ICommand).Execute();;
Do stuff
val it : unit = ()
```

Это требование позволяет избавиться от неоднозначности при вызове метода интерфейса. Можно расширить предыдущий пример в качестве иллюстрации:

```
type ICommand =
  abstract Execute : unit → unit

type IExecutable =
  abstract Execute : unit → unit

type DoStuff() =

  interface ICommand with
    override this.Execute () =
      printfn "Command: Do stuff"

  interface IExecutable with
    override this.Execute () =
      printfn "Executable: Do stuff"

  member this.Execute() =
    printfn "Class: Other do stuff"
```

Мы определили три метода `Execute`, один из них — метод класса, два других — методы интерфейсов, реализуемых классом. В F# это не приводит к ошибкам, так как

программист всегда указывает явно и интерфейс, метод которого реализуется, и интерфейс, метод которого он пытается вызвать.

```
> let cmd = DoStuff();;
val cmd : DoStuff

> cmd.Execute();;
Class: Other do stuff
val it : unit = ()
> (cmd :> ICommand).Execute();;
Command: Do stuff
val it : unit = ()
> (cmd :> IExecutable).Execute();;
Executable: Do stuff
val it : unit = ()
```

Объектные выражения

В любом более или менее серьёзном приложении встречаются вспомогательные классы, объекты которых используются только в одном месте, например в качестве предиката сравнения для алгоритма сортировки. Такой класс, как правило, реализует какой-либо интерфейс и служит только одной цели. При этом класс объекта нигде в программе не используется, используются только объекты этого класса. В качестве примера можно привести метод `Sort` контейнера `List`. Этот метод принимает в качестве одного из аргументов объект, реализующий интерфейс `IComparer`, который в дальнейшем используется для сортировки элементов контейнера.

F# позволяет создавать подобные объекты по месту использования с помощью так называемых объектных выражений или *object expressions*. Объектное выражение — это выражение, результатом которого является объект анонимного класса. Записывается оно следующим образом:

```
// prepare a list
let books =
  new List<Book>(
    [|
      Book("Programming F#;Cris Smith;October 2009");
      Book("Foundations of F#;Robert Pickering;May 30, 2007");
      Book("F# for Scientists;Jon Harrop;August 4, 2008")
    |]
  )

books.Sort(
  { new IComparer<_> with
      member this.Compare(left:Book, right:Book) =
        left.PublishDate.CompareTo(right.PublishDate)
  }
```

```
)
```

После выполнения метода `Sort` элементы контейнера будут упорядочены по дате публикации.

Помимо этого, с помощью объектных выражений можно создавать объекты-наследники какого-либо класса. В этом случае, после имени класса в круглых скобках нужно перечислить параметры конструктора:

```
let obj = {
  new SomeType(arg1, arg2, ..argN) with
    member this.Property = 42
}
```

С помощью объектных выражений нельзя добавлять новые методы или свойства.

Методы расширения

Методы расширения — это механизм, аналогичный методам расширения из C#. Он позволяет дополнить любой класс новыми методами и свойствами. Это может быть полезно например в тех случаях, когда исходный код класса недоступен, а наследование неудобно по каким-либо причинам. Следующий код демонстрирует пример создания и использования метода расширения:

```
type System.Int32 with
  member this.Times fn =
    for x = 1 to this do
      fn ()

> (3).Times(fun () → printfn "I <3 F#");;
I <3 F#
I <3 F#
I <3 F#
val it : unit = ()
```

К сожалению, метод расширения, созданный подобным образом можно будет использовать только из другого F#-кода. Чтобы создать метод расширения, доступный другим языкам .NET, необходимо, как и во многих других случаях, воспользоваться магическими атрибутами:

```
open System.Runtime.CompilerServices

[<Extension>]
module IntExtensions =
  [<Extension>]
  let Times(count:int, fn) =
    for x in 1..count do
      fn ()
```

3.4. Расширенные возможности

В предыдущих разделах мы рассмотрели основные возможности языка программирования F#. Конечно, одного факта того, что он привносит некоторые новые функциональные практики в мир .NET, было бы мало для поддержки его как полноправного члена семейства языков Visual Studio. Как было видно из примеров, многие аспекты языка не уникальны и могут быть с той или иной степенью изящности выражены на C# и, возможно, Visual Basic. В следующих разделах будут описаны те возможности языка, которые сложно отнести к базовым. Именно они делают F# полезным с практической точки зрения и могут показаться знакомыми и/или интересными разработчикам на Haskell и Erlang.

3.4.1. Вычислительные выражения

Среди нововведений F# можно особо выделить так называемые *вычислительные выражения (computation expressions или workflows)*. Они являются обобщением генераторов последовательности и, в частности, позволяют встраивать в F# такие вычислительные структуры, как *монады* и *моноиды*. Также они могут быть применены для асинхронного программирования и создания DSL.

Вычислительное выражение имеет форму блока, содержащего некоторый код на F# в фигурных скобках. Этому блоку должен предшествовать специальный объект, который называется еще *построителем (builder)*. Общая форма следующая: *builder* `{` *comp-expr* `}`.

Построитель определяет способ интерпретации того кода, который указан в фигурных скобках. Сам код вычисления внешне почти не отличается от обычного кода на F#, кроме того, что в нём нельзя определять новые типы, а также нельзя использовать изменяемые значения. Вместо таких значений можно использовать ссылки, но делать это следует с большой осторожностью, поскольку вычислительные выражения обычно задают некие отложенные вычисления, а последние не очень любят побочные эффекты.

В зависимости от построителя внутри вычислительного кода можно также использовать особые конструкции `let!`, `use!`, `return`, `return!`, `yield` и `yield!`. Если читатель знаком с языком программирования Haskell, то можно заметить, что `let!` соответствует стрелке из нотации *do*, а `return` имеет тот же смысл, что и в Haskell.

По своей сути вычислительное выражение является синтаксическим сахаром вокруг указанного построителя. Компилятор F# проходится по коду вычисления и заменяет языковые конструкции вызовами соответствующих методов построителя *b* согласно следующей таблице, где кавычки обозначают оператор преобразования:

Конструкция	Форма преобразования
`let!` *pat* `=` *expr* `in` *cexpr*	`b.Bind(`*expr*`,(fun` *pat* → *«cexpr»*`))`
`let` *pat* `=` *expr* `in` *cexpr*	`let` *pat* `=` *expr* `in` *«cexpr»*
`use` *pat* `=` *expr* `in` *cexpr*	`b.Using(`*expr*`,(fun` *pat* → *«cexpr»*`))`
`use!` *pat* `=` *expr* `in` *cexpr*	`b.Bind(`*expr*`,(fun x →` `b.Using(x,fun` *pat* → *«cexpr»*`)))`
`do!` *expr* `in` *cexpr*	`b.Bind(`*expr*`,(fun () →` *«cexpr»*`))`
`do` *expr* `in` *cexpr*	*expr*`;` *«cexpr»*
`for` *pat* `in` *expr* `do` *cexpr*	`b.For(`*expr*`,(fun` *pat* → *«cexpr»*`))`
`while` *expr* `do` *cexpr*	`b.While((fun () →` *expr*`),` `b.Delay(fun () →` *«cexpr»*`))`
`if` *expr* `then` *cexpr1* `else` *cexpr2*	`if` *expr* `then` *«cexpr1»* `else` *«cexpr2»*
`if` *expr* `then` *cexpr*	`if` *expr* `then` *«cexpr»* `else b.Zero()`
`match` *expr* `with` *pat* → *cexpr*	`match` *expr* `with` *pat* → *«cexpr»*
cexpr1	`v.Combine(`*«cexpr1»*`,` `b.Delay(fun () →` *«cexpr2»*`))`
cexpr2	
`return` *expr*	`b.Return(`*expr*`)`
`return!` *expr*	`b.ReturnFrom(`*expr*`)`
`yield` *expr*	`b.Yield(`*expr*`)`
`yield!` *expr*	`b.YieldFrom(`*expr*`)`

Также есть преобразование для конструкции `try`, но оно более длинное. Таким образом, все основные конструкции F# оказываются охвачены.

Основная идея заключается в том, что когда компилятор обрабатывает очередную конструкцию вычислительного выражения, то он пытается вызвать соответствующий метод построителя. Построитель не обязан реализовывать все указанные методы. Если нужного метода нет, то будет сгенерирована ошибка времени компиляции.

Тогда исходное выражение *builder* `{` *comp-expr* `}` будет преобразовано в

```
let b = builder in b.Run(b.Delay(fun () → «comp-expr»))
```

где вызовы методов `Run` и `Delay` обрабатываются особым образом. В случае отсутствия одного из них или сразу обоих ошибка уже не генерируется, просто соответствующая часть опускается. Например, когда оба метода не определены, то все сокращается до преобразованного выражения *«comp-expr»*. Когда метод `Delay` определён, он фактически делает вычисление отложенным, т. е. ленивым.

Для реализации монады достаточно определить методы `Bind`, `Return` и, возможно, обработчики `try`. Через них можно выразить остальные необходимые методы построителя, но часто в случае конкретной монады бывает так, что можно найти более эффективные определения. Это — одна из причин, по которой F#, например, не предоставляет готовых реализаций для методов `While` и `For`.

Для реализации моноида достаточно определить методы `Zero`, `Combine` и, возможно, `For`.

В качестве примера предположим, что у нас имеется построитель `maybe`, который реализует методы `Bind`, `Return` и `Delay`. Тогда следующая функция:

```
let ap m1 m2 = maybe {
  let! x1 = m1
  let! x2 = m2
  return (x1 x2)
}
```

будет преобразована транслятором на этапе компиляции в следующее определение:

```
let ap m1 m2 =
  maybe.Delay (fun () →
    maybe.Bind (m1, fun x1 →
      maybe.Bind (m2, fun x2 →
        maybe.Return (x1 x2))))
```

Эта функция достаточно общая, и она походит для очень многих вычислительных выражений. На практике имеет смысл создать отдельный модуль, назовем его `Maybe`, куда можно и поместить определение функции `ap`. Туда же можно поместить определение комбинатора `(<*>)`, который в данном случае будет совпадать с функцией `ap`. Такое разбиение на модули позволяет различать функции с одинаковыми именами.

```
[<CompilationRepresentation
    (CompilationRepresentationFlags.ModuleSuffix)>]
module Maybe =
  let ap m1 m2 = ... // the function definition
  let (<*>) m1 m2 = ap m1 m2
```

Теперь поставим задачу реализовать такой построитель `maybe`, чтобы мы могли проводить некие вычисления с возможностью их быстрой остановки. Используя вычислительные выражения, мы можем реализовать такой механизм остановки прозрачно. Для этого создадим отдельный класс `MaybeBuilder`, который бы реализовывал необходимые методы.

```
type M<'a> = unit → 'a option

let runMaybe m = m ()
let fail = fun () → None

type MaybeBuilder () =
```

```
  member b.Return (x): M<'a> = fun () → Some x
  member b.ReturnFrom (m): M<'a> = m
  member b.Bind (m: M<'a>, k): M<'b> =
    match runMaybe m with
      | None → fail
      | Some x → k x
  member b.Delay (k: unit → M<'a>): M<'a> =
    fun () → runMaybe (k ())
```

Здесь аннотации типов можно было и опустить. Они использованы лишь чтобы получить на выходе следующую сигнатуру типа:

```
type MaybeBuilder =
  new: unit → MaybeBuilder
  member Return: 'a → M<'a>
  member ReturnFrom: M<'a> → M<'a>
  member Bind: M<'a> → ('a → M<'b>) → M<'b>
  member Delay: (unit → M<'a>) → M<'a>
```

Читатель, знакомый с языком программирования Haskell, сразу увидит в этом определение монады, причём для большего сходства метод `Delay` придаёт вычислению ленивость. В действительности, такое определение — одна из возможных реализаций известной монады `Maybe`.

Собственно сам построитель определяется просто:

```
let maybe = MaybeBuilder ()
```

Теперь мы умеем создавать вычисления, причём они будут отложенными. Чтобы получить результат, необходимо уже применить функцию запуска `runMaybe` к самому вычислению.

Итак, в этом примере основным методом построителя является `Bind`. Он принимает исходное вычисление и его продолжение. Если результатом исходного вычисления является `fail`, то вычисление-продолжение не запускается. В противном случае результат выполнения первого вычисления попадает на вход второго. Методу `Bind` в коде соответствуют конструкции `let!`. Теперь, чтобы прервать вычисление немедленно, в правой части конструкции следует вернуть `fail`.

```
let failIfBig n = maybe {
  if n > 1000 then
    return! fail
  else
    return n
}

let failIfEitherBig (inp1, inp2) = maybe {
  let! n1 = failIfBig inp1
  let! n2 = failIfBig inp2
  return (n1,n2)
```

```
}
```

Подобное поведение вычислительного выражения является лишь частным случаем. Мы можем интерпретировать код вычисления самым разным образом. Если ключевыми шагами вычисления считать места использования конструкций `let!`, `do!` и `use!`, т. е. где вызывается метод `Bind`, то само выражение мы можем рассматривать с позиции встраивания кода между этими шагами, которые уже отвечают за обработку встроенного кода, причём сама обработка происходит прозрачно. Это позволяет меньше думать о деталях и заметно повышает уровень абстракции. Так открывается путь к созданию DSL. Также значительно упрощается написание вычислений, выполняемых асинхронно, о чем мы расскажем далее.

Более того, генератор последовательности (sequence comprehension) является частным случаем вычислительного выражения, только в нём особую роль играют уже конструкции `yield` и `yield!`. Буквально это означает, что средствами самого языка можно создать аналог генератора `seq`, но уже с другим ключевым словом. Также можно создать аналоги других двух стандартных генераторов: списка и массива.

Представляет особый практический интерес то обстоятельство, что вычислительное выражение можно использовать вместе с цитированиями — ещё одной специальной возможностью F#, о которой мы расскажем ниже. Мы можем получить синтаксическое дерево с результатом раскрытия вычислительного выражения. Это позволяет создавать трансляторы таких выражений. В конце статьи приводится практический пример WebSharper, где используется такая возможность.

Так, если взять вышеописанный построитель `maybe` или любой другой, то в F# Interactive можно увидеть результат раскрытия вычислительного выражения, который будет представлен в виде синтаксического дерева:

```
> let a = <@ maybe { return 777 } @>;;
```

Теперь несколько слов об эффективности. Выше было дано общее определение для функции `ap`. Если заменить имя построителя, то определение будет работать со многими другими построителями, реализующими методы `Bind` и `Return`. В языке Haskell аналогичная функция определена с помощью класса типов `Monad`. Там одна и та же функция будет работать с любой монадой. В F# мы должны определять функцию `ap` для каждой монады отдельно, но здесь кроется один важный момент. Для использованного в примере построителя `maybe` мы можем создать более эффективную реализацию функции `ap`: как минимум две лямбды не нужны. Причем, такая ситуация встречается часто: для этого построителя мы можем написать более эффективные реализации методов `While`, `For`, `Delay` и т. д., хотя всё это можно было бы выразить по-прежнему через `Bind` и `Return`. Вырисовывается характерная черта F# в области обработки монад, когда по сравнению с Haskell выбор делается в пользу менее общего, но, как правило, более эффективного кода.

Другим отличительным моментом по сравнению с Haskell является то, что вычислительные выражения могут иметь побочный эффект, не отражённый в системе ти-

пов. Например, наиболее ярко это проявляется при использовании асинхронных вычислений.

3.4.2. Средства для параллельного программирования

Асинхронные потоки операций (async workflows)

Асинхронные потоки операций — это один из самых интересных примеров практического использования вычислительных выражений. Код, выполняющий какие либо неблокирующие операции ввода-вывода, как правило сложен для понимания, поскольку представляет из себя множество `callback`-методов, каждый из которых обрабатывает какой-то промежуточный результат и возможно начинает новую асинхронную операцию. Асинхронные потоки операций позволяют писать асинхронный код последовательно, не определяя `callback`-методы явно. Для создания асинхронного потока операций используется блок `async`:

```
open System.IO
open System.Net
open Microsoft.FSharp.Control.WebExtensions

let getPage (url:string) =
  async {
    let req = WebRequest.Create(url)
    let! res = req.AsyncGetResponse()
    use stream = res.GetResponseStream()
    use reader = new StreamReader(stream)
    let! result = reader.AsyncReadToEnd()
    return result
  }
```

Здесь мы объявили функцию `getPage`, которая должна возвращать содержимое страницы по заданному адресу. Эта функция имеет тип `string →Async<string>` и возвращает асинхронную операцию, которая может быть использована для получения строки с содержимым страницы. Стоит отметить, что классы `WebRequest` и `StreamReader` не имеют методов `AsyncGetResponse` и `AsyncReadToEnd`, это методы расширения.

Построитель асинхронного потока операций, работает следующим образом. Встретив оператор `let!` или `do!`, он начинает выполнять операцию асинхронно, при этом метод, начинающий асинхронную операцию, получит оставшуюся часть блока `async` в качестве функции обратного вызова. После завершения асинхронной операции переданный callback продолжит выполнение асинхронного потока операций, но, возможно, уже в другом потоке операционной системы (для выполнения кода используется пул потоков). В результате код выглядит так, как будто он выполняется последовательно. То, что может быть с легкостью записано внутри блока `async` с использованием циклов и условных операторов, достаточно сложно реализуется с ис-

пользованием обычной техники, требующей описания множества callback-методов и передачей состояния между их вызовами.

Обработка исключений — самый наглядный пример удобства асинхронных потоков операций. Если мы пишем асинхронный код в традиционном стиле, то каждый метод обратного вызова должен обрабатывать исключения самостоятельно. Блок `async` может включать оператор `try`, с помощью которого можно обрабатывать исключения.

```
let getPage (url:string) =
  async {
    try
      let req = WebRequest.Create(url)
      let! res = req.AsyncGetResponse()
      use stream = res.GetResponseStream()
      use reader = new StreamReader(stream)
      let! result = reader.AsyncReadToEnd()
      return Some result
    with
    | _ as e → printfn "error: %s" e.Message
    return None
  }
```

В этом примере поток операций возвращает значение типа `string option`, то есть, либо строку либо пустое значение, чтобы вызывающий код мог обработать ошибку.

Значение типа `Async<_>` нужно передать в один из статических методов класса Async, чтобы начать выполнение соответствующего потока операций. В простейшем случае можно воспользоваться методом `Async.RunSynchronously`, который просто заблокирует вызывающий поток до тех пор, пока все операции не будут выполнены. Такой код:

```
getPage "http://google.com/"
  |> Async.RunSynchronously
```

просто вернёт содержимое веб-страницы по указанному адресу.

Метод `Async.Parallel` — это комбинатор, который позволяет объединить несколько асинхронных потоков операций в один. Этот метод принимает на вход последовательность асинхронных операций, возвращающих значение типа `X`, и возвращает одну асинхронную операцию, возвращающую массив значений типа `X`.

```
let result = ["http://google.com/"; "http://ya.ru/"]
  |> Seq.map getPage
  |> Async.Parallel
  |> Async.RunSynchronously
```

Этот код вернёт массив из двух строк, представляющий результаты выполнения двух асинхронных операций. Помимо этого, класс `Async` содержит методы, позволяющие обрабатывать исключения, прерывать работу асинхронных потоков операций и многое другое.

Возможности асинхронных потоков операций не ограничены вводом/выводом,[22] а также набором стандартных классов. Существует простой способ добавлять произвольные асинхронные операции, однако его рассмотрение выходит за рамки этой статьи.

MailboxProcessor

MailboxProcessor — это класс из стандартной библиотеки F#, реализующий один из паттернов параллельного программирования. MailboxProcessor является агентом, обрабатывающим очередь сообщений, которые поставляются ему извне при помощи метода Post. Вся конкурентность поддерживается реализацией класса, который содержит очередь с возможностью одновременной записи несколькими писателями и чтения одним единственным читателем, которым является сам агент.

```
let agent = MailboxProcessor.Start(fun inbox →
  async {
    while true do
      let! msg = inbox.Receive()
      printfn "message received: '%s'" msg
  })
```

Выше приведена реализация простейшего агента, который при получении сообщения, содержащего строку, выводит его на экран. Послать агенту сообщение, как уже было сказано выше, можно при помощи метода Post:

```
agent.Post("Hello world!")
```

Интересно отметить тип функции, являющейся единственным параметром конструктора агента (и статического метода Start)[23]:

```
static member Start :
  (MailboxProcessor<'Msg> → Async<unit>) →
    MailboxProcessor<'Msg>
```

Из этого определения видно, что основной «рабочей лошадкой» агента является функция, на вход получающая экземпляр самого агента и возвращающая асинхронную операцию, о которых говорилось чуть выше.

Естественно, что прямое соответствие агентов и потоков[24] было бы не очень удобно и крайне неэффективно, потому что сильно ограничивало бы количество одновременно работающих агентов. Благодаря использованию асинхронных потоков операций, агент большую часть времени является просто структурой в памяти, которая

[22] Пример их использования для программирования пользовательского интерфейсам можно посмотреть здесь: http://lorgonblog.spaces.live.com/Blog/cns!701679AD17B6D310!1842.entry.

[23] Для простоты в типе опущен опциональный параметр, отвечающий за отмену операций агента. За более подробной информацией читателю рекомендуется обратиться к документации.

[24] Под потоком здесь, конечно же, понимается thread, а не stream.

содержит некоторое внутреннее состояние и только в те моменты, когда в очередь поступает очередное сообщение, функция обработки регистрируется для исполнения в потоке из системного пула. Функцией обработки как раз и является та, что была передана в конструктор или метод `Start`. Таким образом, всё внутреннее состояние агента поддерживается инфраструктурой, а не ложится тяжким грузом на плечи пользователя. Для подтверждения этого факта можно попробовать создать несколько тысяч агентов, запустить их и начать случайным образом отправлять им сообщения.

Стоит сказать ещё пару слов об инкапсуляции. Несмотря на то, что пользователю недоступно внутреннее состояние самого объекта агента, функция обработки сообщений вовсе не обязана быть чистой, а может иметь побочные эффекты. Вполне работоспособно следующее решение, подсчитывающее количество обработанных сообщений:

```
let agent = MailboxProcessor.Start(fun inbox →
  async {
    let i = ref 0
    while true do
      let! msg = inbox.Receive()
      incr i
      printfn "#%d..." !i
  })
```

Здесь `ref` — это ещё один способ работы с изменяемым состоянием. Это надстройка над простыми `mutable` значениями, описанными в начале статьи. Данный пример показывает, что функция обработки сообщений агента является полноценным хранилищем внутреннего состояния, которое сохраняется между вызовами. Такого же эффекта можно добиться и без использования изменяемого состояния:

```
let agent = MailboxProcessor.Start(fun inbox →
  let rec loop n = async {
    let! msg = inbox.Receive()
    printfn "#%d..." n
    return! loop (n+1)
  }
  loop 0)
```

Оптимизация хвостовой рекурсии не допустит переполнения стека.

Такой довольно простой в использовании инструмент позволяет строить масштабируемые параллельные системы с целыми иерархиями управляющих и управляемых процессов без использования разделяемого состояния и всех проблем, с ним связанных.

Обработка событий

Изначально .NET позволяет обрабатывать события по одному. Обработчиком события является функция, которая вызывается каждый раз с некоторыми аргументами, и если разработчику необходимо хранить какое-то дополнительное состояние

между вызовами событий — это приходится делать самостоятельно. Кроме того, оригинальная модель подписки может приводить к утечкам памяти из-за наличия неявных взаимных ссылок в подписчике и генераторе событий.

F#, конечно, позволяет работать с событиями в классическом понимании. Правда, делается это при помощи немного необычного синтаксиса: вместо использования операторов `+=` и `-=` для регистрации и деактивации обработчика события используется пара методов `Add/Remove`.

```
button.Click.Add(fun args →
  printfn "Button clicked"
)
```

С другой стороны, F# позволяет манипулировать потоками событий, и работать с ними как с последовательностями, используя функции `filter`, `map`, `split` и другие. Например, следующий код фильтрует поток событий нажатия клавиш внутри поля ввода, выбирая только те из них, которые были нажаты в сочетании с Ctrl, после чего, из всех аргументов событий выбираются только значения поля `KeyCode`. Таким образом, значением `keyCodes` будет поток событий, содержащих только коды клавиш, нажатых с удерживаемым Ctrl.

```
let keyCodes = textBox.KeyDown
  |> Event.filter (fun args →
    args.Modifiers = Keys.Control)
  |> Event.map (fun args →
    args.KeyCode)
```

Стоит отметить, что обработка потоков событий позволяет разработчику не заботиться о тонкостях отписки, а просто генерировать новые потоки событий на основе уже существующих, использовать эти потоки в качестве значений, то есть передавать их в качестве аргументов и возвращать из функций.

Использование подобной техники может привести к значительному упрощению кода для реализации, например, функциональности Drag&Drop. Ведь это есть ни что иное, как композиция события нажатия кнопки мыши, перемещения курсора с нажатой кнопкой и затем отпускания. Для примера часть, отвечающую за drag, можно легко перевести с русского на F#:

```
form.MouseDown
  |> Event.merge form.MouseMove
  |> Event.filter (fun args →
    args.Button = MouseButtons.Left)
  |> Event.map (fun args →
    (args.X, args.Y))
  |> Event.add (fun (x, y) →
    printfn "(%d, %d)" x y)
```

Сочетание обработки потоков событий с асинхронными потоками операций позволяет также довольно просто решать печально известную проблему GUI-приложений,

когда обработчик события графического компонента должен исполняться в основном потоке и любое промедление приведёт к «зависанию» интерфейса приложения.[25]

Компания Microsoft разработала библиотеку Reactive Extensions, которая предоставляет разработчикам на других .NET-языках аналогичным образом манипулировать потоками событий.

3.4.3. Цитирования (Quotations)

Цитирования представляют собой интересный инструмент мета-программирования, отчасти похожий на .NET Expression Trees. Цитирования предоставляют разработчику доступ к исходному коду конструкций F# в виде синтаксического дерева. Если Expression Trees позволяют обрабатывать только ограниченное подмножество программных конструкций, то цитирования полностью охватывают все конструкции, возможные в F#. С другой стороны, цитирования можно сравнить с механизмом Reflection, правда в более структурированном виде.

```
<@ 42 @>
// has type Qutations.Expr<int>
// Value(42)
<@ fun i → i * i @>
// has type Quotations.Expr<int → int>
// Lambda (i,
//  Call (None,
//        Int32 op_Multiply[Int32,Int32,Int32](Int32,Int32),
//        [i, i])
```

По существу цитирования — это представление абстрактного синтаксического дерева программы. При желании эта мета-информация может сохраняться в скомпилированной сборке вместе с кодом, которому она соответствует. Цитирования можно использовать по-разному: для получения информации об исходном коде конструкций и генерации, например, запросов к базе данных или для преобразования одних конструкций в другие.

Цитирования бывают типизированные и нетипизированные. Друг от друга они отличаются, как нетрудно догадаться, наличием или отсутствием информации о типах. Нетипизированными цитированиями лучше всего пользоваться при необходимости преобразования AST. В случае использования только для чтения лучше подходят типизированные.

Преобразовать код в цитату можно несколькими способами: используя оператор `<@ @>`, возвращающий типизированное цитирование; оператор `<@@ @@>`, возвращающий нетипизированное; и помечая определения атрибутом `[<ReflectedDefinitionAttribute>]`. В последнем случае мета-информация

[25] Замечательный пример подобного решения можно найти по ссылке http://lorgonblog.spaces.live.com/Blog/cns!701679AD17B6D310!1842.entry?sa=863426610.

будет сохранена в сборке вместе с исполняемым кодом и будет доступна во время исполнения.

Цитирования — это очень интересная тема, довольно слабо освещенная в официальных источниках. Подробнее о ней можно узнать в блоге Томаша Петричека [7], где он приводит примеры преобразования F#-кода в команды графического процессора.

Механизм Quotations активно используется в F# PowerPack для интеграции с механизмом LINQ, который появился в .NET 3.5. С февраля 2010 года F# PowerPack является проектом с открытым исходным кодом и также может стать интересным источником знаний о возможностях цитирований.[26]

3.5. Примеры использования

Существует интересный пример [5] использования quotations в F# как средства мета-программирования. Задача связана с массивной параллельной обработкой данных с помощью специальной библиотеки либо на многоядерной процессорной системе x64, либо на графическом процессоре GPU. Определяется небольшое подмножество F#, код из которого может быть оттранслирован и запущен на целевой платформе. Это позволяет писать «обычные» функции на F#, отлаживать их стандартным образом, а затем с помощью механизма quotations обрабатывать эти функции и запускать на GPU. Более того, программа на F# может создавать «на лету» такие функции, которые уже затем будут оттранслированы и запущены на другой платформе. Примечательно, что при реализации такой трансляции широко используется специальная возможность F# в виде активного сопоставления с образцом (active pattern matching), которая заметно упрощает написание транслятора.

Язык F# может быть удобен для создания DSL, которые становятся частью самого F#, причем такие языки могут быть достаточно краткими и выразительными. Например, в книге Real-World Functional Programming [8] приводится библиотека для описания анимации. Первоначальная идея была реализована в проекте Fran [Elliot, Hudak, 1997] на языке Haskell. Описание идеи ещё можно найти в книге The Haskell School of Expression [3]. Анимация моделируется как зависящая от времени величина. На основе примитивов строятся уже составляющие лексикон предметной области функции, с помощью которых можно описывать достаточно сложные анимации и делать это декларативно. Что касается реализации, то там много общего с монадами.

Вот пример того, как выглядит на языке F# определение анимированной части солнечной системы, которую затем можно визуализировать на экране компьютера:

```
let solarSystem =
  sun
    -- (rotate 80.00f 4.1f mercury)
    -- (rotate 150.0f 1.6f venus)
    -- (rotate 215.0f 1.0f
```

[26] Страница проекта находится по адресу http://fsharppowerpack.codeplex.com/.

```
(earth -- (rotate 20.0f 12.0f moon)))
```

Следующим примером использования F# является коммерческий продукт WebSharper фирмы IntelliFactory [4]. Это платформа для создания клиентских web-приложений. Она позволяет писать клиентский код на F#, который затем будет оттранслирован на JavaScript. Такая трансляция распространяется на достаточное большое подмножество F#, включая функциональное ядро языка, алгебраические типы данных, классы, объекты, исключения и делегаты. Также поддерживается значительная часть стандартной библиотеки F#, включая работу с последовательностями (sequences), событиями и асинхронными вычислительными выражениями (async workflows). Всё может быть автоматически оттранслировано на целевой язык JavaScript. Кроме того, поддерживается некоторая часть стандартных классов самого .NET, и объём поддержки будет расти.

Этот продукт примечателен тем, что здесь используется целый ряд приёмов, характерных для функционального программирования. Так, в WebSharper встроен DSL для задания HTML-кода, в котором широко применяются комбинаторы.

Например, следующий кусок кода на F#:

```
Div [Class "Header"] -< [
  H1 ["Our Website"]
  Br []
  P ["Hello, world!"]
  Img [Src "smile.jpg"]
]
```

будет преобразован на выходе в такой код HTML:

```
<div class="header">
  <h1>Our Website</h1>
  <br />
  <p>Hello, world!</p>
  <img src="smile.jpg" />
</div>
```

Для самой же трансляции в JavaScript используются цитирования. Код для трансляции должен быть помечен атрибутом `JavaScriptAttribute`, унаследованным от стандартного `ReflectedDefinitionAttribute`.

```
[<JavaScriptType>]
type MyControl() =
  inherit IntelliFactory.WebSharper.Web.Control()

  [<JavaScript>]
  override this.Body =
    Div ["Hello, world!"]
```

Коньком WebSharper является работа с HTML-формами. Вводится обобщенный тип `Formlet<'T>`, значения которого извлекают некоторую информацию типа `'T`

из формы. Важно, что этот тип является монадой и имеет соответствующий построитель вычислительных выражений. Это позволяет к значениям типа `Formlet<'T>` применить определенный для данного типа аппликативный комбинатор `(<*>)`. В результате можно объединить и упростить сбор информации из формы.

```
[<JavaScript>]
let AddressFormlet () : Formlet<Address> =
  Formlet.Yield (fun st ct cnt →
    {Street = st; City = ct; Country = cnt})
    <*> Controls.Input "Street"
    <*> Controls.Input "City"
    <*> Controls.Input "Country"
```

Здесь метод `Formlet.Yield` имеет тот же смысл, что и монадическая функция `return`, а методы `Controls.Input` создают значения типа `Formlet<string>`, которые извлекают информацию из текстовых полей `Street`, `City` и `Country`, соответственно. На выходе получаем значение типа `Formlet<Address>`, который уже извлекает всю информацию об адресе.

Некоторый интерес представляет случай, когда пользователь непосредственно использует синтаксис вычислительных выражений для создания значений `Formlet` внутри участков кода, помеченных атрибутом `JavaScript`. Получается, что такое выражение раскрывается, и синтаксическое дерево результата раскрытия сохраняется в бинарной сборке наряду с кодом. Затем WebSharper извлекает это дерево и анализирует, переводя на JavaScript, т. е. транслируется само вычислительное выражение.

3.6. Дополнительные материалы

F#, как уже неоднократно было сказано выше, является наследником языков ML и OCaml. К счастью, эти языки существуют уже довольно давно и для них есть обширная документация. В части изучения теоретических основ F# можно обратиться к русскому переводу лекций Джона Харрисона [14].

Кроме того, сам язык уже успел обзавестись рядом интересных книг, среди которых «Foundations of F#» [9], «Expert F#» [11], «F# for scientists» [1] и «Real World Functional Programming» [8].

Самыми значительными интернет-ресурсами, ориентированными на F#, можно назвать блоги Дона Сайма [10] и Томаша Петричека [7], а также специализированный форум *«hubFS»* [2].

Источником практических знаний о возможностях F# могут служить проекты с открытым исходным кодом [15].

3.7. Заключение

Согласно гипотезе лингвистической относительности, люди, говорящие на разных языках, мыслят и воспринимают окружающую действительность по-разному. Если предположить, что используемый язык программирования влияет на то, как программист видит решение проблемы, то нельзя не признать, что F# будет подталкивать его к поиску более простых и элегантных решений.

В том, что F# найдет широкое применение в финансовой сфере и в области статистики, сомневаться не приходится. Помимо этого, F# заслуживает того, чтобы стать достаточно популярным в других областях, хотя бы потому, что делает многопоточное программирование проще и понятнее.

Следует также отметить, что на данный момент F# является, пожалуй, единственным функциональным языком программирования, который продвигается одним из ведущих производителей в области разработки программного обеспечения. Он позволяет использовать множество уже существующих библиотек, писать приложения для самых разных платформ и что не менее важно — делать всё это в современной IDE.

Литература

[1] *Harrop J.* F# for Scientists. — New York, NY, USA: Wiley-Interscience, 2008.

[2] hubFS: THE place for F#. — Адрес форума, URL: http://cs.hubfs.net/ (дата обращения: 21 мая 2010 г.).

[3] *Hudak P.* The Haskell School of Expression: Learning Functional Programming through Multimedia. — Cambridge University Press, 2000.

[4] IntelliFactory's WebSharper. — Официальная страница, URL: http://www.intellifactory.com/ (дата обращения: 21 мая 2010 г.).

[5] *Petricek T.* Accelerator and F#. — Статья в блоге, URL: http://tomasp.net/blog/accelerator-quotations.aspx (дата обращения: 21 мая 2010 г.).

[6] *Petricek T.* F# Overview — Introduction. — Статья в блоге, URL: http://tomasp.net/articles/fsharp-i-introduction.aspx (дата обращения: 21 мая 2010 г.).

[7] *Petricek T.* Блог Томаша Петричека. — Адрес блога, URL: http://tomasp.net/blog (дата обращения: 21 мая 2010 г.).

[8] *Petricek T., Skeet J.* Real-World Functional Programming with examples in F# and C#. — Manning Publications, 2009.

[9] *Pickering R.* Foundations of F#. — Berkely, CA, USA: Apress, 2007.

[10] *Syme D.* Блог Дона Сайма. — Адрес сайта, URL: http://blogs.msdn.com/dsyme/ (дата обращения: 21 мая 2010 г.).

[11] *Syme D., Granicz A., Cisternino A.* Expert F# (Expert's Voice in .Net).

[12] *Душкин, Роман.* Алгебраические типы данных и их использование в программировании. // *Журнал «Практика функционального программирования».* — 2009. — Т. 2.

[13] *Кирпичёв, Евгений.* Элементы функциональных языков. // *Журнал «Практика функционального программирования».* — 2009. — Т. 3.

[14] Русский перевод курса лекций Джона Харрисона «Introduction to Functional Programming». — Страница проекта, URL: http://code.google.com/p/funprog-ru/ (дата обращения: 21 мая 2010 г.).

[15] Список проектов с открытым исходным кодом, использующих F#. — Адрес сайта, URL: http://stackoverflow.com/questions/383848/f-open-source-projects (дата обращения: 21 мая 2010 г.).

Лисп — философия разработки

Всеволод Дёмкин, Александр Манзюк
vseloved@fprog.ru, manzyuk@fprog.ru

Аннотация

В статье исследуются подходы к разработке, практикуемые в Лисп-среде, на примерах решения различных прикладных задач на языке Common Lisp. Вы узнаете о том, что макросы — это не только синтаксический сахар, но и прекрасный инструмент инкапсуляции, что кроме глобальных и локальных переменных бывают еще специальные, что полиморфные интерфейсы можно создавать без привязки к классам, а также о том, что определять стратегии передачи управления можно не только с помощью монад. Рассмотрены и решения прикладных задач: клиент для хранилища данных Redis, прокси между двумя бизнес-приложениями, внутренний API веб-сервиса, библиотека парсерных комбинаторов.

The article describes the development techniques used in the Lisp environment, illustrated by solving several practical tasks in Common Lisp. You will discover that macros are not only syntactic sugar, but also a way to provide incapsulation; that besides local and global variables there are also special ones; that polymorphic interfaces can be defined without any relation to classes; and that monads are not the only way to model computation control. Examples are drawn from real-world codebases: a client for Redis persistent key-value database, a proxy linking two business applications, and internal API of a web service, and a parser combinator library.

Обсуждение статьи ведется в LiveJournal.

4.1. Кратко о Common Lisp

Common Lisp — это язык, который появился в 80-х годах как попытка объединить многочисленные разрозненные реализации концепции Лиспа, а также задействовать последние на тот момент достижения в области языков программирования. Вероятно, на сегодняшний момент он является наиболее используемым на практике Лиспом.

Его основные характеристики:

- мультипарадигменность;
- динамичность;
- особый синтаксис;
- это «программируемый язык программирования».

Мультипарадигменность подразумевает возможность в рамках одного языка использовать и безболезненно сочетать разные стили и подходы — например, объектно-ориентированный и функциональный.

Динамичность подразумевает динамическую типизацию с возможностью явного объявления типов. Кроме того она означает представление программы в виде живого образа (live image), в котором любой объект, за исключением 25 базовых операторов языка, может быть переопределен в ходе исполнения теми же средствами, что и в момент его создания. Наконец, динамичность поддерживается полноценными возможностями интроспекции.

Синтаксис этого языка — это единообразная префиксная нотация (S-выражения). Вот простейший пример Лисп-кода: `(defun sum (a b)(+ a b))`, эквивалентом которого в языке Алгол-семейства был бы `function sum(a, b){ return a + b;}`. Благодаря этому исходный код гомоморфен синтаксическому дереву, используемому компилятором. Языки, обладающие таким свойством, называются также гомоиконными языками.

Программируемость подразумевает возможность добавлять к языку собственные дополнения, которые будут на равных правах сосуществовать со стандартными языковыми средствами. Это подкреплено тем, что в отличие от большинства других платформ, в Лисп-среде программисту дается возможность влиять (одними и теми же средствами языка) на все три этапа существования программы: синтаксический разбор, компиляцию и исполнение, а не только на самый последний этап — исполнение.

Среди других Лиспов его также характеризуют:

- наличие как лексического, так и динамического диапазонов видимости;
- использование раздельных пространств имен для переменных, функций, типов и т. д. (т. н. свойство Лисп-2 или Лисп-N. Фактически, для Common Lisp N=5).

Исторически Common Lisp развивался в сообществе исследователей искуственного интеллекта, поэтому широко распространен в различных экспертных системах, CAD/CAM системах и в научной среде. Однако, этим отнюдь не ограничивается сфера его применения — скорее наоборот, сейчас Лисп больше развивается в других областях. Примерами удачного использования Common Lisp являются высоконагруженная система расчета цен на авиабилеты QPX, вдохновивший многие Интернет-проекты, один из первых веб-сервисов Viaweb, а также сервис централизованного администрирования компьютеров Paragent.com.

Наконец, нужно сказать, что Лисп всегда был источником, из которого черпались идеи для других языков.[1] Среди этих идей — автоматическое управление памятью, REPL, интроспекция, замыкания и функции высших порядков. На данный момент может показаться, что этот процесс остановился, однако, по моему мнению, это не совсем так: ведь в Лиспе остается еще много уникальных решений. Свидетельством того, что он продолжает вдохновлять разработчиков других языков программирования, являются, например, работа Реджинальда Брайтвайта в рамках Ruby или Ола Бини над Ioke, не говоря уже о новом языке для виртуальной машины Java — Clojure.

Это, конечно, не значит, что миру Лиспа нечему поучиться у других языков — наоборот. И, что как раз демонстрирует язык Clojure, этот процесс впитывания достижений из других областей идет очень активно. Стоит отметить, что благодаря своей гибкости Лисп всегда был отличным выбором для адаптации и тестирования новых идей.

В этой статье мы на практических примерах рассмотрим подходы к разработке, применяемые в рамках Common Lisp, и попытаемся показать их сильные стороны, на которые стоит обратить внимание. В частности будут рассмотрены такие концепции, как метапрограммирование, проектирование от данных, обобщённые функции, сигнальный протокол и другие языковые протоколы.

Первые разделы предполагают минимальное знакомство с языком, но с каждым разделом сложность демонстрируемого кода возрастает, и последние два раздела требуют определенного концептуального понимания, которое можно почерпнуть, например, из книги [7]. Эта часть статьи должна быть прежде всего интересна тем, кто уже имеет существенный опыт программирования на Лиспе или же функциональных языках.

4.2. Код или данные?

Классической моделью создания больших систем в рамках Лиспа является итеративная разработка, ставящая себе целью постепенное «сближение» программы с предметной областью. Она включает в себя такие подходы, как метапрограммирование, т. е. программирования самого языка; проблемно-ориентированное программи-

[1] «Мы целились в программистов на C++, и у нас получилось затащить многих из них примерно примерно на полпути к Лиспу», Guy Steele о Java, http://en.wikiquote.org/wiki/Lisp_programming_language

рование, подразумевающее адаптацию языка к предметной области; и, наконец, т. н. разработка «от данных» (data-driven design).

Широко известна максима Лиспа, выражающая суть метапрограммирования: «код — это данные». Она означает, что саму программу можно рассматривать в виде данных, с которыми можно и нужно работать точно так же, как и с любыми другими.

Обратное также верно: «данные — это код», ведь представление данных играет немаловажную роль в логике программы. В этой формулировке – суть подхода к разработке «от данных».

Посмотрим, как же это все применяется на практике.

4.2.1. Практический пример: клиент для Redis

Redis — это одна из новых баз данных нереляционного типа, которая по сути представляет собой персистентную хеш-таблицу. Кроме того в ней поддерживается работа с данными как со списками, множествами и отсортированными множествами (единственное, что не поддерживается — это вложенность этих структур). Эти базовые структуры данных являются родными и для Лиспа, поэтому Redis выглядит очень естественным дополнением к Лисп-среде. Единственный метод взаимодействия с сервером — низкоуровневый socket API. Соответственно, чтобы связать любой язык и Redis, необходимо реализовать клиент, который бы скрывал эти низкоуровневые аспекты за более абстрактным и удобным фасадом. Попробуем реализовать такой клиент на Лиспе.

С точки зрения пользователя такая библиотека должна давать ему возможность, вооружившись только базовыми знаниями языка и спецификацией команд Redis, взаимодействовать с сервером БД. Кроме того, в идеале, еще желательна возможность управлять в определенных пределах эффективностью такого взаимодействия.

С точки зрения разработчика стоят задачи сделать библиотеку:

1) слабо связанной, чтобы при необходимости можно было вносить изменения в ее отдельные функциональные части, не затрагивая остальных;

2) легко расширяемой. Redis — молодой проект, в котором часто происходят изменения. Например, начальная разработка клиента велась, когда актуальной была версия 0.9, а в текущей на момент написания статьи версии 1.1 по сравнению с ней появился новый тип взаимодействия с сервером, новая группа из десятка команд и две новых самостоятельных команды;

3) легко поддающейся тестированию.

Если рассмотреть предметную область данной задачи, то можно выделить три уровня:

1) нижний уровень управления соединением через socket API;

2) средний уровень реализации протокола взаимодействия;

3) верхний уровень пользовательских команд.

Реализация клиента

Именно так и структурирован исходный код библиотеки.[2]

В файле `connection.lisp` описывается способ установления соединения, реакции на сбои, а также более высокоуровневые конструкции, позволяющие управлять параметрами подключения. Эти конструкции реализованы в принятом в Лиспе with-шаблоне, при котором код, непосредственно выполняющий действие, с помощью макроса оборачивается в шаблонный код, выполняющий служебные задачи, такие как гарантированное освобождение ресурсов или иное восстановление исходного состояния, обработка ошибок, вывод отладочной информации и т. д. Например, следующий фрагмент кода открывает соединение к Redis на локальной машине по нестандартному порту 7000 и выполняет команды `SELECT 1` и `KEYS *` (это и есть типичный клиентский код, который пишут пользователи библиотеки):

```
(with-connection (:port 7000)
  (red-select 1)
  (red-keys "*"))
```

Если посмотреть на него через `macroexpand`, можно увидеть, какие вспомогательные действия добавлены в этот вызов:

```
(LET ((*CONNECTION* (MAKE-INSTANCE 'REDIS-CONNECTION
                                   :HOST #(127 0 0 1)
                                   :PORT 7000
                                   :ENCODING :UTF-8)))
  (UNWIND-PROTECT
      (PROGN (RED-SELECT 1)
             (RED-KEYS "*"))
    (CLOSE-CONNECTION *CONNECTION*)))
```

В файле `redis.lisp` реализуется протокол взаимодействия с сервером. Прочитав спецификацию протокола, можно заключить, что абстракцией, которая хорошо описывает взаимодействие с Redis, является модель запрос-ответ, что характерно для многих (но не всех) приложений, основанных на socket API. При этом синтаксис протокола описывает несколько способов отправить запрос и несколько вариантов форматирования ответа. На этом и построено ядро библиотеки. Методы на обобщённых функциях[3] `TELL` и `EXPECT` реализуют конкретные способы взаимодействия в рамках протокола. Для выбора специфического метода используется его идентификатор: `:inline`, `:bulk`, `:multi`(-bulk), `:integer` и т. д. Пример вызова этих функций может выглядеть таким образом:

```
(tell :bulk 'set "key" "value")
(expect :ok)
```

[2]Полностью он доступен по адресу: http://github.com/vseloved/cl-redis

[3]Generic functions — основной инструмент полиморфизма в Common Lisp. Они представляют собой исполняемый объект, который обеспечивает диспетчеризацию (выбор конкретного метода выполнения) в зависимости от типа или значения входных аргументов.

Дополнительно для функции EXPECT определена операция DEF-EXPECT-METHOD, которая создает новый метод, оборачивая его тело в одинаковый для большинства методов код. Этот код выполняет чтение и первичный разбор строки ответа, формирует очищенную от служебной информации строку в виде локальной переменной reply, а также содержит вызов функций отладки. Создание подобных шаблонов для обёртки методов — еще одна идея Лиспа, взятая на вооружение мейнстримными языками, и получившая в данном контексте название «аспектно-ориентированное программирование».

Итак, определения конкретных методов для TELL и EXPECT выглядят следующим образом.

```
(defmethod tell ((type (eql :bulk)) cmd &rest args)
  (multiple-value-bind (args bulk) (butlast2 args)
    (check-type bulk string)
    (write-redis-line "~A~{ ~A~} ~A" cmd args (byte-length bulk))
    (write-redis-line "~A" bulk)))

(def-expect-method :boolean
  (ecase (char reply 0)
    (#\0 nil)
    (#\1 t)))
```

и его макро-раскрытие:

```
(DEFMETHOD EXPECT ((TYPE (EQL :BOOLEAN)))
  "Receive and process the reply of type BOOLEAN."
  (LET* ((#:G1767 (READ-LINE (REDIS::CONNECTION-STREAM
      *CONNECTION*)))
         (#:G1768 (CHAR #:G1767 0))
         (REDIS::REPLY (SUBSEQ #:G1767 1)))
    (WHEN *ECHO-P* (FORMAT *ECHO-STREAM* "S: ~A~%" #:G1767))
    (CASE #:G1768
      ((#\-) (ERROR 'REDIS-ERROR-REPLY :MESSAGE REDIS::REPLY))
      ((#\+ #\: #\$ #\*) (ECASE (CHAR REPLY 0)
                           (#\0 NIL)
                           (#\1 T)))
      (OTHERWISE
       (ERROR 'REDIS-BAD-REPLY :MESSAGE
              (FORMAT NIL "Received ~C as the initial reply byte."
                      #:G1768))))))
```

Наконец, в файле commands.lisp объявляются команды взаимодействия с сервером. В данном случае они реализованы в виде обычных функций, создание которых происходит программно, а определение — в полностью декларативном стиле на основании минимального набора необходимых данных: списка аргументов, идентификаторов методов отправки запроса и получения ответа, а также документирующей

строки. Таким образом, объявление команды имеет вид, практически повторяющий ее описание в Redis Wiki:

```
(def-cmd SINTER (&rest keys)
  "Return the intersection between the Sets stored at key1, key2,
      ..., keyN."
  :inline :multi)
```

которое раскрывается в:

```
(PROGN
 (DEFUN RED-SINTER (&REST KEYS)
   "Return the intersection between the Sets stored at key1,
       key2, ..., keyN."
   (REDIS::WITH-RECONNECT-RESTART REDIS::*CONNECTION*
     (APPLY #'TELL :INLINE 'SINTER KEYS)
     (EXPECT :MULTI)))
 (EXPORT 'RED-SINTER :REDIS))
```

Возможны также альтернативные варианты реализации команд. Например, можно создать обобщённую функцию `redis-command`, которая бы принимала идентификатор команды и ее аргументы. Преимуществом такого подхода является единая «точка входа» для взамодействия с Redis, а также то, что мы воспользуемся наиболее простым способом расширения, заложенным в язык — обобщёнными функциями. Недостатки же — бо́льшая громоздкость вызовов и меньшая гибкость. Мы выбрали первый подход. Второй подход мы планируем добавить в будущих версиях библиотеки в качестве альтернативы. Пример успешного применения этого подхода также можно будет увидеть в разделе статьи, посвященной созданию интерфейсов.

Анализ предложенного решения

API любой библиотеки с точки зрения клиента можно считать своего рода языком, описывающим какую-то предметную область (DSL). Язык взаимодействия с БД Redis включает типы данных, команды и описание способов соединения. В своем скринкасте «Написание DSL в Лиспе» Райнер Джосвиг утверждает, что оно сводится к тому, чтобы «расставить скобки вокруг спецификации и заставить ее запуститься». Приведенный пример определения команд как раз буквально подтверждает это суждение. Единственная загвоздка заключается во фразе «заставить запуститься». Именно в этой сфере сосредоточена основная работа при написании DSL, и она выполнена в данном случае в модулях `connection.lisp` и `redis.lisp`. Впрочем, от этого никуда не деться в любом случае, а вот возможность сделать спецификацию действительно исполняемой, и, более того, полноценной частью языка, по нашему мнению, довольно ценна. Но еще ценнее то, что при таком подходе мы совершим плавный переход от описания предметной области к ее реализации.

Данная библиотека обладает высокой степенью модульности, в чем мы убедились, когда к работе подключился Александр. Он взялся за преобразование изначально до-

статочно примитивной реализации модуля соединения (которую была выбрана просто для того, чтобы иметь возможность тестировать остальную функциональность, работу над которой мы считали более приоритетной) по образцу более зрелой библиотеки — Postmodern (клиента СУБД PostgreSQL). Ему это удалось без существенных изменений в других частях кода.

Очень важна возможность постепенного совершенствования и детализации программы. Например, оказалось, что команда SORT имеет способ задания аргументов, не полностью совместимый с принятым в Лиспе, поэтому для нее пришлось реализовать особый их разбор в методе TELL. Сделать это, не влияя на способ реализации других команд, позволило то, что обобщённые функции при необходимости могут быть специализированы на любом из своих обязательных аргументов, чем мы и воспользовались, введя дополнительную специализацию по названию именно этой команды. Таким образом, описание метода TELL для команды SORT приобрело такую форму:

```
(defmethod tell ((type (eql :inline)) (cmd (eql 'SORT)) &rest
    args)
  (flet ((send-request (key &key by get desc alpha start end)
           (unless (xor2 start end)
             (error "START and COUNT must be either both NIL or
                 both
 non-NIL."))
           (write-redis-line "SORT ~a~:[~; BY ~:*~a~]~{ GET ~
~a~}~:[~; DESC~]~:[~; ALPHA~]~:[~; LIMIT ~:*~a ~a~]"
                             key by (mklist get) desc alfa start
                                end)))
    (apply #'send-request args)))
```

На данный момент это единственное исключение из общего правила для всех команд, но кто знает, как проект будет развиваться дальше? Когда в новой версии Redis появились новые команды, добавление их поддержки в библиотеку потребовало лишь чтения спецификации, «копирования» ее в файл `commands.lisp` и добавления соответствующего юнит-теста в тестовый набор. Появление нового вида запросов (multi-bulk команд) потребовало аналогичных, незначительных усилий.

Тестирование библиотеки также довольно прямолинейно и не представляет особых проблем благодаря модульности и наличию интерфейсов к каждой отдельной ее части. Соответственно, для нас не представляет труда поддерживать практически полноценный набор тестов для нее.

Таким образом, были достигнуты все цели разработки, сформулированные в начале главы. Путь дальнейшего развития данного проекта — это добавление новых уровней абстракции, управляющих соединениями уже на уровне логических сущностей (отдельных серверов, ферм и т. д.), а также оптимизация работы протокола.

В заключение хочу упомянуть о пути развития самого подхода: интересно было бы посмотреть на его обобщение для любых socket-приложений — ведь модель взаимодействия и сложности, связанные с управлением соединением, везде примерно одинаковы. Не будет ли это дублированием самого socket API? Полагаю, что нет, если

сконцентрироваться на вопросах, которые не покрываются им: на разделении потока символов на отдельные сообщения, на восстановлении соединения в случае сбоев и ошибок и т. п. Также речь идет не о полноценной платформе, которая полностью скрывает сокеты и добавляет дополнительные сервисы и уровни маршализации (вроде CORBA), а лишь о тонком слое, который бы автоматизировал и унифицировал решения стандартных проблем.

4.3. Проектирование «от данных» (data-driven design)

Проектирование, ведомое данными (data-driven design) — это ещё один подход при создании «больших» систем в Лисп.

Начнем с того, что нет единого определения и понимания термина «data-driven design». В разных отраслях он может означать разные вещи. Мы здесь рассматриваем это понятие под тем углом, как его сформулировал Питер Норвиг в своей книге «Парадигмы программирования искусственного интеллекта» [4]. Data-driven design — это проектирование системы таким образом, чтобы выделить максимально общий и в то же время простой алгоритм решения задачи, а часть специфичной логики закодировать в обычных структурах данных языка. Интересные примеры использования этого подхода в разных языках программирования приведены в девятой главе книги «Искусство программирования под Unix» Эрика Реймонда [6][4]. Таким образом, можно уверенно говорить, что проектирование от данных (как, впрочем, и другие описанные в этой статье методы) не является чем-то уникальным для Лиспа и может практиковаться с тем или иным успехом в других средах. Отличие заключается в том, что Лисп поощряет применение этого подхода и предоставляет для этого развитые инструменты.

В результате декомпозиции «от данных» мы получаем программу, которую условно можно разделить на три части:

1) ядро исполняемого кода (в рамках применяемого стиля разработки: императивного, функционального и т. д.);

2) описание специфичной логики (в декларативном стиле): таким образом обретает смысл выражение «данные — это код»;

3) код диспетчеризации, в котором, как правило, приходиться учитывать особенности (нерегулярности) конкретной задачи.

В идеале, таким образом достигаются такие полезные, на наш взгляд, качества программного кода как:

- меньшая взаимозавязанность (decoupling);
- большая степень абстракции (общность);

[4] http://http://www.faqs.org/docs/artu/ch09s01.html

- соблюдение принципа «не повторяйся» (DRY: Don't Repeat Yourself).

В то же время добавляется ещё одна составляющая системы (код диспетчеризации), которая требует дополнительной проработки по сравнению с обычным (монолитным) подходом, в рамках которого каждая отдельная часть функциональности реализуется как индивидуальный фрагмент кода. Впрочем, хорошая новость состоит в том, что эта составляющая может быть реализована в качестве инфраструктурного компонента (фреймворка) в самом языке программирования или же библиотеке. Подробнее об этом рассказано ниже.

4.3.1. Практический пример

Подход «от данных» был применен при решении задачи создания проксирующей прослойки между двумя бизнес-приложениями. Их интерфейсами с одной стороны были вызовы PL/SQL процедур Oracle, а с другой — XML-RPC API. Необходимо было обеспечить двустороннее взаимодействие путем обмена по HTTP XML-сообщениями в рамках заданного протокола. Более детально, модель взаимодействия такова: одна сторона может отправить HTTP-запрос с XML-содержимым для инициации какой-то операции, в результате чего должен быть вызван PL/SQL метод на другой стороне, а его результат отправлен назад в теле HTTP-ответа в виде XML-документа. Другая сторона может выполнить HTTP-запрос к нашему приложению, передавая данные, из которых должно сформироваться XML-содержимое запроса, который будет отправлен с помощью HTTP первой стороне. На первый взгляд, логика каждого из путей взаимодействия различается. Но ведь по сути они одинаковы: необходимо принять одно сообщение, транслировать его в другую форму и отправить другой стороне, а затем обработать специфичный для другой стороны ответ. В обоих случаях используются те же отдельные базовые блоки. В первом случае формат входящего сообщения — XML-документ, исходящего — параметры вызова хранимой процедуры, а результирующего — тоже XML-документ. Во втором: входящее сообщение — POST-параметры, исходящее — XML-документ, результирующее — параметры вызова хранимой процедуры.

В качестве языка реализации был выбран PHP, поскольку он вписывался в среду заказчика (LAMP), а также, в принципе, хорошо адаптирован к решению задач, связанных с технологиями HTTP, XML и взаимодействия с реляционными БД.

Решение задачи с применением проектирования от данных начинается с выбора наиболее удобного формата представления ключевых данных бизнес-логики и описания этих данных.

В данном случае специфику каждого отдельного взаимодействия, описанного в протоколе (который включает 14 типов запросов), можно представить в виде карты соответствия типа XML-запроса, соответствующего имени корневого узла и параметров вызова PL/SQL хранимых процедур. Для одной стороны этого взаимодействия часть этой карты показана ниже:

```
$ACTIONS =
```

```
    array('GetBalance' =>
              array(ORA_FUNC, 'get_user_balance',
                    array('SubscriberID', SQLT_CHR,
                          'AccountID',    SQLT_CHR))
          'AuthorizePurchase' =>
              array(ORA_FUNC, 'authorize_purchase',
                    array('SubscriberID',  SQLT_CHR,
                          'AccountID',     SQLT_CHR,
                          'ApplicationID', SQLT_CHR,
                          'ProductID',     SQLT_CHR,
                          'SubProductID',  SQLT_CHR,
                          'AssetPrice',    SQLT_INT,
                          'AssetTitle',    SQLT_CHR,
                          'ProductName',   SQLT_CHR)),
          'RegisterPurchase' =>
              array(ORA_PROC, 'register_purchase',
                    array('ReferenceID',  SQLT_INT,
                          'SubscriberID', SQLT_CHR)),
          'RollbackPurchase' =>
              array(ORA_PROC, 'rollback_purchase',
                    array('ReferenceID',  SQLT_INT,
                          'SubscriberID', SQLT_CHR)));
```

Также в результате каждого взаимодействия должен быть сформирован XML-ответ, включающий определенный набор узлов, состоящий из стандартного для всех множества (например, `'ResultCode'`), а также дополнительных узлов, которые могут добавляться к ответу на специфичные запросы.

Для другой стороны перечень также задается картой, и если какой-то из типов запросов в ней отсутствует, то ответ для него включает только стандартный перечень узлов. Фрагмент карты выглядит так:

```
$OUTPUT_SPEC =
  array('GetBalance'        => array('Balance'),
        'AuthorizePurchase' => array('ReferenceID',
                                     'AuthCode'),
        'RollbackPurchase'  => array('ReferenceID',
                                     'AuthCode'));
```

В целом, приложение строится из двух файлов, один из которых содержит общие функции, а другой реализует логику взаимодействия конкретного типа. Первичная диспетчеризация происходит по URL запроса, которому в силу специфики PHP соответствует вызываемый файл. Основное содержание обоих специфических файлов составляют приведенные выше карты соответствий, а также код диспетчеризации второго уровня, которая выполняется по типу XML-запроса (для первого варианта) или же по одному из POST-параметров (для второго). Ну и, конечно, служебный код для установки глобальных параметров, ведения логов, отладки и т. п.

Посмотрим на код диспетчеризации для одного из вариантов (для второго выполняется примерно тот же набор действий, только в другой последовательности):

```
$xml = @ simplexml_load_string($HTTP_RAW_POST_DATA);

// пропущен фрагмент обработки ошибок

$action = $xml->getName();

switch ($action) {
case 'AuthorizePurchase':
case 'RollbackPurchase':
  $ref_id = null;
  $out = array($ref_id, SQLT_INT);
  break;
default:
  $out = null;
}

$auth_code = '';
$err = null;

if (DEBUGGING) {
  $rez = plsql_req($ACTIONS[$action][0], $ACTIONS[$action][1],
                   xml_to_plsql_args($xml, $ACTIONS[$action][2]),
                   $out, $err);
} else {  // suppress warnings
  $rez = @ plsql_req($ACTIONS[$action][0], $ACTIONS[$action][1],
                     xml_to_plsql_args($xml,
                         $ACTIONS[$action][2]),
                     $out, $err);
}
```

Из этого кода можно сделать следующие выводы:

1) во-первых, основную диспетчеризацию по типу запроса мы получаем «бесплатно»;

2) во-вторых, для некоторых типов запросов предварительно добавляются дополнительные «выходные» параметры для вызова PL/SQL процедур (если мы хотим получить больше одного результирующего значения). Определение этих параметров также можно было бы произвести в декларативном стиле через карту соответствия, однако для данного простейшего случая в этом нет необходимости.

3) некоторые вещи в PHP делаются просто отвратительно (это про дублирование кода в случае `DEBUGGING`).

Оправдывает ли себя такой подход для системы, которая имеет всего два разных сценария работы? Его преимущество в том, что программа получилась довольно понятной и легко расширяемой: добавление нового типа сообщений выполняется по единому принципу для обоих типов взаимодействия и в базовом случае потребует только добавления нового отношения в обе карты (`$ACTIONS` и `$OUTPUT_SPEC`). Кроме того, как и в предыдущем случае с клиентом для Redis, мы в какой-то мере декларативно (а не императивно) вписали протокол в сам код — хотя в данном варианте в нем также появилась не описанная в протоколе часть, связанная с вызовами PL/SQL. Впрочем, так даже лучше, поскольку теперь мы получили формальную спецификацию интерфейса PL/SQL. Конечно, такой подход не универсален: он хорошо работает в тех случаях, когда проблема состоит из нескольких похожих задач, в которых можно выделить какую-то общую часть.

Приведенный пример довольно прост, однако и на нем уже становятся видны некоторые синтаксические проблемы PHP, связанные с плохой поддержкой декларативного программирования и манипуляций собственным кодом. Также для data-driven подхода очень полезной является возможность оперирования такими структурами данных как первоклассные объекты (наличие синтаксиса литералов, полноценная поддержка в высокоуровневых функциях, итераторах и т. п.), а также хорошие возможности интроспекции.

4.3.2. Диспетчеризация

По сути дела, подход к проектированию от данных задействует полиморфизм, однако без привязки к наследованию и иерархиям классов. Ведь в ОО-языке можно было бы аналогично декомпозировать приведенную задачу, создав отдельные классы и реализовав каждый из методов взаимодействия в виде методов этих классов. Это было бы вполне в духе объектно-ориентированного проектирования. В чем же разница, помимо того, что в данном случае меньше «накладных расходов» как для программиста, так и для среды исполнения в связи с отсутствием необходимости поддержания иерархии классов, создания объектов и т. п.? Разница становится хорошо видна на более масштабных примерах, один из которых будет приведен в следующем разделе статьи.

Вообще говоря, каждая школа разработки рассматривает понятие полиморфизма под собственным углом зрения. Для функциональной школы он прежде всего основывается на типе [8], для объектно-ориентированной — на иерархии классов,[5] а для подхода «от данных» — на любом подходящем для задачи способе. Объяснить это можно таким образом: в очень многих случаях для диспетчеризации, кроме самого идентификатора, больше ничего не нужно, в то время, как в редких случаях могут возникнуть специфические потребности, которые трудно выразить через иерархию

[5]Более того, хотя он и считается одним из трех «китов» ООП, но в чистом виде (без задействования наследования) полиморфизм в обычных ОО-языках не возможен. Это хорошо видно на примере языка JavaScript, который отказался от основанного на классе наследования в пользу прототипной передачи свойств, и таким образом полностью потерял ОО-полиморфизм.

классов или систему типов. Таким образом идентификатором для диспетчиризации может служить: просто строка (или даже интернированная строка, что еще лучше, так как для сравнения таких строк достаточно сравнения адресов памяти), тип/класс объекта или даже значение, получаемое в результате вызова произвольной функции. И все эти варианте неплохо бы предоставлять в рамках языковой среды, в идеале так, чтобы простота применения того или иного подхода коррелировала с частотой его использования.

На практике же можно выделить четыре подхода к диспетчеризации:

- стандартный OO-подход: только по классу первого аргумента;
- примитивный подход: с помощью `CASE`-выражений или же прямо через словари ключ-значение (как это сделано в примере выше);
- на основе сопоставления с шаблоном (pattern matching);
- на основе произвольной функции диспетчеризации.

Недостатком первого варианта, кроме необходимости поддерживать дополнительные (часто не нужные) данные, также являются проблемы с множественной диспетчеризацией, которые упираются в проблему множественного наследования (что, в общем-то, не должно происходить, поскольку это две ортогональные концепции).

Второй подход страдает той же проблемой одномерной диспетчеризации, а также еще и монолитностью, т. е. необходимостью прописывать все варианты в одном месте. А ведь полиморфизм — это механизм расширяемости, поэтому необходимо иметь возможность задавать новые варианты в других частях кода и, что не менее важно, динамически добавлять и удалять их. Последней проблемой также страдает и сопоставление с шаблоном. Кроме того, оба эти подхода плохо приспособлены для выражения нелинейных путей выполнения.

Подходом, лишенным всех этих недостатков, является разделение механизма диспетчеризации и механизма задания специфических методов. Одной из его реализаций является система generic functions (обобщённых функций), встроенная в Common Lisp. Принято считать, что они являются частью Объектной Системы Common Lisp (CLOS, Common Lisp Object System), однако это не совсем так: кроме диспетчеризации по классам аргументов (как в обычном OO-подходе со множественным наследованием), они также поддерживают и выбор по произвольному ключу (EQL-диспетчеризация). Среди возможностей обобщённых функций:

- множественная диспетчеризация;
- отсутствие необходимости объявления методов в одном месте, возможность динамического добавления/удаления методов;
- декораторы `:before`, `:after` и `:around`, позволяющие выборочно дополнять функциональность отдельных методов;

- способы комбинации методов, как стандартные, так и задаваемые пользователем;
- возможность указания порядка просмотра аргументов, отличного от порядка их задания в функции.

Тот же подход взят за основу в механизме мультиметодов языка Clojure, который пошел дальше, позволяя задавать для диспетчеризации произвольную функцию от аргументов. Впрочем, такая возможность есть и для обобщённых функций Common Lisp, только не «из коробки». Приведу пример одно из вариантов ее реализации через механизм комбинации методов, определяемый пользователем:

```
(define-method-combination multi (dispatch-fn)
  ((all *))
  (:arguments &whole args)
  `(or ,@(mapcar (lambda (method)
                   `(when (member (apply ,dispatch-fn ,args)
                                  (method-qualifiers ,method))
                      (call-method ,method)))
                 all)))

CL-USER> (defgeneric foo (collection)
           (:method-combination multi #'length)
           (:documentation "Dispatches on COLLECTION's length"))
#<STANDARD-GENERIC-FUNCTION FOO (0)>
CL-USER> (defmethod foo 3 (x) (format t "~a contains 3 things.~%"
    x))
#<STANDARD-METHOD FOO 3 (T) {B6238A9}>
CL-USER> (defmethod foo 10 (_) (format t "A collection of 10
    elements.~%"))
#<STANDARD-METHOD FOO 10 (T) {B03CBC9}>
CL-USER> (foo "yes")
yes contains 3 things
NIL
CL-USER> (foo (range 10))
A collection of 10 elements.
NIL
```

Еще более детализированный и интегрированный вариант — это библиотека Filtered functions Паскаля Костансы, основанная на использовании мета-объектного протокола Common Lisp.

4.4. Использование обобщённых функций для создания API

В предыдущем разделе обсуждалась диспетчеризация как один из ключевых моментов для обеспечения модульности и определения наиболее обобщённых алгоритмов — т. е. основных принципов проектирования от данных. В Common Lisp для поддержки этого подхода существует механизм обобщённых функций. Посмотрим, как их можно использовать на примере приложения среднего размера и уровня сложности. Это веб-приложение «MindShare»,[6] предоставляющее некоторые дополнительные сервисы для пользователей популярного сервиса микроблоггингового сайта Twitter.com. Twitter предоставляет REST-интерфейс для получения основных данных о пользователях, однако у него довольно жесткие ограничения. Для работы ключевой функции MindShare, выдачи рекомендаций пользователям, необходимо обрабатывать большой объем данных. Их получение напрямую через Twitter API на практике оказывается невозможным из-за необходимости выполнения сотен, а то и тысяч API-запросов для выдачи одной рекомендации: даже не учитывая ограничение по запросам API, время на их выполнение исчисляется минутами и десятками минут в связи с задержкой сети. Поэтому мы выбрали стратегию кеширования данных, получаемых через API, а также их предзагрузки как через API, так и путем получения информации напрямую с web-страниц пользователей (scraping).

Задачей Александра на определенном этапе стало проектирование и разработка внутреннего API, абстрагирующего разные способы получения данных и управляющее их гибким комбинированием. Этот API, в свою очередь, должен использоваться в модуле, обрабатывающем запросы пользователей (frontend).

Итак, у приложения есть три основных способа получения данных:

1) Получение информации о пользователе по его имени или идентификационному номеру посредством Twitter API. Например, GET-запрос по адресу

   `http://api.twitter.com/1/followers/ids.json?screen_name=N`

   возвращает JSON-массив идентификаторов пользователей, следящих за сообщениями пользователя *N*.

2) Получение информации о пользователях без обращения к Twitter API путем извлечения данных из HTML-кода личной страницы пользователя (например, с помощью регулярных выражений).

3) Получения данных из кеша. Кеширование необходимо, поскольку оба приведенных метода получения информации достаточно медленные, и просто не работают, когда Twitter перегружен (что случается достаточно часто). Кроме того, значительная часть информации изменяется относительно редко (например,

[6] http://mshare.tw/

профили пользователей), и ее актуальность не играет решающей роли для нашего приложения, поэтому неразумно каждый раз получать ее заново (хотя периодически обновлять эту информацию все же необходимо; для этих целей в Redis, который мы выбрали в качестве кеша с сохранением данных на диске, весьма удобной оказалась возможность устанавливать для ключей «срок годности», т. е. промежуток времени, по истечению которого ключ и данные, сохраненные под этим ключом, удаляются из базы).

Кстати, создание этого внутреннего API получения данных позволило сделать приложение более модульным. Изначально для кэширования всей информации использовался Redis, однако мы столкнулись с тем, что он не справляется со всем необходимым объемом информации на имеющихся вычислительных ресурсах.[7] Тогда мы решили перенести часть данных в более «тяжеловесное» хранилище. Этот переход мы смогли осуществить относительно легко, практически без изменений в остальных частях кода.

Что должен представлять собой такой API? В качестве прямолинейного варианта можно было бы для каждого свойства пользователя (например, списка последователей), каждого из перечисленных способов получения информации (Twitter API, HTTP или кеш), и каждого типа аргумента (строка или число) ввести отдельную функцию. Кроме того, что при данном подходе пространство имен загрязняется большим количеством неудобоваримых имен, вроде `get-followers-via-api-by-screen-name` (для каждого свойства приходится вводить шесть новых функций, в которых легко запутаться), этот подход обладает и другим серьезным недостатком — он ведет к дублированию кода. Например, код функции для получения списка последователей практически идентичен коду функции для получения списка тех, за кем следит сам пользователь.

С другой стороны, если бы мы попробовали применить объектно-ориентированную декомпозицию, то нам, наверное, пришлось бы создать классы `AccessMethod` с потомками вроде `APIAccessMethod`, `HTTPAcessMethod` и `StoreAccessMethod`, которые бы не имели практически никакого внутреннего состояния, за исключением разве что статических переменных, хранящих служебную информацию, и определить для них методы получения информации типа `GetFollowersByScreen_name` (в общем-то, очень похоже на предыдущий вариант). Впрочем, и из этой ситуации, будь мы экспертами OOD, можно было бы найти выход: например, применив шаблон «визитёр» (Visitor pattern). Довольно сложно для такой на первый взгляд простой задачи.

В Lisp принято решать задачу наиболее простым (но не настолько тупым, как в первом предложенном варианте) способом. Учитывая приведенные в начале соображения, естественнее всего сделать тип извлекаемого атрибута аргументом функции.

[7] В текущей версии (1.2) он не умеет использовать виртуальную память, т. е. весь кеш находится в оперативной памяти и при выходе размера БД за пределы физического объема оперативной памяти работа сервера становится непредсказуемой из-за свопинга.

В Common Lisp для этой цели хорошо подходят ключевые слова — символы, интернированные в специальном пакете `keywords` и доступные в любом пакете без явной квалификации (вместо `keywords:foo` можно писать просто `:foo`). Ключевые слова удобны тем, что (при удачном выборе) имя символа можно включить в строку-шаблон, например, при построении URL, по которому производится запрос (это как раз пример data-driven подхода, когда ключевое слово внутреннего API используется также в качестве идентификатора, по которому осуществляется доступ к внешнему API).

Сделав тип атрибута аргументом функции, мы наталкиваемся на проблему диспетчеризации: поскольку логика извлечения списка последователей отличается от, скажем, логики получения личных данных пользователя, необходимо проводить разбор частных случаев. Делать его в теле функции с помощью оператора `case` — не совсем удачный вариант, так как при этом, как было замечено ранее, страдает модульность. Common Lisp предлагает весьма элегантное решение этой проблемы: обобщённые функции.

У нас есть три метода получения информации, которые мы можем задать ключами `:twitter`, `:http` и `:store`, а также два типа идентификаторов пользователей: имя (строка) и идентификационный номер (число). Для каждой тройки (метод, пользователь, атрибут) имеется алгоритм извлечения данного атрибута для заданного пользователя и заданным методом. Эти алгоритмы можно собрать в трехмерную таблицу, измерения которой соответствуют трем аргументам. Эта таблица и есть обобщённая функция. Применение обобщённой функции к заданным аргументам сводится к вызову другой функции, которая принимает в качестве аргументов саму обобщённую функцию (т. е. таблицу) и ее аргументы, и на их основе проводит диспетчеризацию: находит подходящую функцию в таблице и применяет ее. Более детальное обсуждение этой идеи заинтересованный читатель найдет в [3, раздел 2.5]. Механизм обобщённых функций, реализованный в Common Lisp, несколько сложнее, чем в Scheme, но для наших потребностей приведенного объяснения должно быть достаточно (хоть оно и не до конца точное).

Итак, определим обобщённую функцию `retrieve`.

```
(defgeneric retrieve (method attribute user))
```

Тогда методы, принадлежащие этой обобщённой функции (элементы таблицы) определяются с помощью макроса `defmethod`, например:

```
(defmethod retrieve ((method (eql :api))
                     (attribute (eql :followers))
                     (id integer))
  ;; код метода
  )
```

Из всех методов алгоритм диспетчеризации однозначно выбирает наиболее специфичный. Аргументы, переданные в обобщённую функцию, сверяются в порядке старшинства с типами или значениями, по которым специализированы методы обобщенной функции. По умолчанию, порядок старшинства (precedence order) совпадает

с порядком аргументов в определении функции, но его можно изменить с помощью опции `:argument-precedence-order` в макросе `defgeneric`. На самом деле, для корректной диспетчеризации нам пришлось изменить старшинство аргументов в `retrieve`:

```
(defgeneric retrieve (method attribute user)
  (:argument-precedence-order attribute method user))
```

Этот подход позволил кроме основных методов `:api`, `:http` и `:store` легко добавить также комбинированные методы `:fast` и `:all`. Первый метод сначала ищет заданный атрибут в кеше; если там его нет, делает запрос к Twitter API; и наконец, если это срывается по какой-то причине, пытается извлечь нужную информацию через HTTP.

```
(defmethod retrieve ((method (eql :fast)) attribute user)
  "First try to lookup ATTRIBUTE in Redis; if it is not
      available, try the Twitter API, and finally if this fails
      retrieve through the HTTP."
  (handler-case
      (let ((rez (multiple-value-list (retrieve :store attribute
          user))))
        (if (car (mklist rez))
            (apply #'values rez)
            (handler-case
                (retrieve :api attribute user)
              ((or twi-error http-client-error) ()
                (retrieve :http attribute user)))))
    ((or error sb-ext:timeout) (e)
      (log-message :error "Error, while FAST retrieving ~A ~A: ~A"
                   attribute user e))))
```

Второй метод, наоборот, сначала пытается получить наиболее актуальную информацию посредством запроса к Twitter API; если это не удается, пытается получить данные через HTTP; наконец, ищет в кеше прошлую версию.

Следует отметить, что совсем необязательно специализировать метод по всем аргументам — неспециализированные аргументы на самом деле специализируются по типу `T`, который является корневым в иерархии типов в Common Lisp. В ходе работы над API нам потребовалось определить всего 23 отдельных метода для этой функции, при том, что теоретически возможное число комбинаций составляет $5 \times 7 \times 2 = 70$ вариантов.

Еще одна крайне удобная возможность, которую предоставляет механизм обобщённых функций — это определение методов-декораторов (`:before`, `:after` и `:around`). Например, все методы функции `retrieve` обращаются к Redis для поиска или кеширования данных, поэтому мы хотели бы убедиться, что в случае обрыва соединения с Redis-сервером в процесе выполнения того или иного метода мы автоматически переподключимся к серверу. Для этого тело каждого метода необходимо

завернуть в форму `handler-bind`, которая устанавливает обработчик для ошибок соединения с Redis:

```
(handler-bind ((redis-connection-error #'(invoke-restart
    :reconnect)))
  ;; тело метода
  )
```

Оборачивать явно тело каждого метода в `handler-bind` — это, без сомнения, дублирование кода. К счастью, в Common Lisp есть штатное средство для борьбы с ним — макросы. Мы могли бы определить специальный макрос для добавления методов к функции `retrieve`, который генерировал бы необходимый оберточный код, однако механизм обобщённых функций предоставляет стандартное решение: методы-декораторы. Мы определим `:around`-метод для функции `retrieve`.

```
(defmethod retrieve :around (method attribute id)
  "Ensure that Redis connection persists."
  (handler-bind ((redis-connection-error
                  #'(invoke-restart :reconnect)))
    (call-next-method)))
```

(В данном случае используется собственный макрос чтения `#'`, позволяющий компактно определять анонимные функции.)

Что при этом происходит? Мы добавляем метод, который выполняется до любого другого метода нашей обобщённой функции. Этот метод устанавливает динамический контекст, в котором для ошибок типа `redis-connection-error` установлен обработчик, и вызывает следующий более специфический метод с помощью `(call-next-method)`. В конечном счете мы имеем тот же эффект — тело метода выполняется в динамической среде, в которой виден обработчик для ошибок типа `redis-connection-error`, но при этом мы не пользовались макросами (это сделал за нас компилятор). Таким образом, методы `:before`, `:after` и `:around` позволяют добавлять и изменять функциональность существующих методов, не меняя их собственный код. Кроме того, они могут иметь разный уровень специализации с основными методами: в нашем случае метод задан для всех аргументов, однако, если бы нам понадобилось добавить такой код только при получении данных из кеша, мы бы могли изменить его следующим образом:

```
(defmethod retrieve :around ((method (eql :store)) attribute id)
  ;; код метода
  )
```

Итак, подытожим преимущества, которые дает нам использование обобщённых функций.

- Мы можем отделить политику диспетчеризации от реализации конкретных методов доступа к данным и создавать произвольные комбинации методов, не дублируя при этом программный код.

- Мы имеем единую «точку входа» для API, которую легко запомнить. Ее удобно использовать при интерактивной разработке (например, указав всего одно имя для трассировки командой `(trace retrieve)`, мы будем видеть вызовы всех различных вариантов обращения к API), а также в функциях высших порядков.
- Для диспетчеризации используется наиболее удобный ключ для каждого конкретного измерения: это может быть как просто ключевое слово, так и базовый тип аргумента (как число и строка в нашем случае), или же, если мы станем оперировать сложными объектами — собственно классы таких объектов.[8]
- Мы можем легко использовать декораторы для выборочного изменения поведения некоторых методов, не меняя их основного кода.

Таким образом можно сделать вывод, что механизм обобщённых функций прекрасно подходит для определения различных API. Идеальным вариантом его использования является случай, когда API основывается на большом числе однообразных операций, которые удобно представить в виде многомерной таблицы.

4.5. Использование сигнального протокола вне системы обработки ошибок

Еще одной важной технологией Common Lisp (явно недооцененной, на наш взгляд, в том числе и самими Лисп-разработчиками) является сигнальный протокол. Он лежит в основе системы обработки ошибок, однако может применяться гораздо шире, так как по сути является четко структурированной реализацией синхронной нелокальной передачи управления. Хороший обзор протокола представлен в [7, глава 19]. Именно там сформулирована мысль, что сигнальный протокол может быть использован для реализации многих интересных решений, вплоть до со-процедур.

Нужно сказать, что в Common Lisp невозможно полноценное использование стиля передачи продолжений (continuation-passing style), с помощью которого принято реализовывать со-процедуры в функциональных языках. Это связано с наличием специального оператора `unwind-protect` и, вкратце, может быть выражено афоризмом «непреодолимая сила против непробиваемой брони».

4.5.1. Парсерные комбинаторы

Задача, которую мы будем решать — это создание библиотеки парсерных комбинаторов по образцу библиотеки Parsec в Haskell. Путь ее решения, который приходит в голову сразу же — это портировать основанную на монадах реализацию (именно так устроена, например, библиотека CL-PARSER-COMBINATORS). Однако нам хотелось

[8] Разделение на базовый тип и класс в данном случае условно. Под этим не подразумевается такое разделение, которое есть в Java между примитивными и объектными типами.

понять, можно ли получить столь же прозрачный и удобный интерфейс без использования монад. Таким образом, на наш взгляд, можно уменьшить уровень дополнительной сложности, связанной с решением задачи в силу особенностей реализации (accidental complexity), и, в идеале, довести сложность до минимального уровня, определяемого самой предметной областью.

В рамках этой статьи нет возможности углубиться в теорию синтаксического разбора. Те, кто не знаком с этой областью, могут начать ее изучение с заметки Ивана Веселова, посвященной библиотеке Parsec. Вкратце, парсерные комбинаторы — это набор функций, работающих с потоком символов или других объектов, каждая из которых выполняет разбор по одному правилу. Их ключевое свойство — это возможность комбинировать их стандартными средствами языка вплоть до создания парсеров высших порядков (принимающих на вход другие парсеры). Таким образом парсерные комбинаторы дают возможность декларативного задания правил разбора без необходимости непосредственной работы с потоками символов и прочими низкоуровневыми концепциями.

Рассмотрим простейший пример записи правил разбора для арифметических выражений (без правил приоритета операторов). На «языке» CL-PARSEC[9] он имеет следующий вид:

```
(defparser spaces ()
  (skip-many #\Space))

(defparser expression ()
  (prog1 (many+ #'tok)
    (spaces)
    (eof)))

(defparser tok ()
  (spaces)
  (either #'(parse-integer (coerce (many+ #'(parsecall
      #'digit-char-p))
                                   'string))
          #'(parsecall #\( :lparen)
          #'(parsecall #\) :rparen)
          #'operator))

(defparser operator ()
  (list :op (parsecall '(member (#\- #\+ #\* #\/)))))
```

Здесь `spaces`, `skip-many`, `expression`, `many+`, `tok`, `eof`, `either` и `operator` — это парсеры (в том числе высокоуровневые). `Prog1`, `parse-integer`, `coerce`, `digit-char-p`, `parsecall` и `list` — обычные формы языка.

Вот пример использования этого кода:

[9] Код доступен по адресу: http://github.com/vseloved/cl-parsec.

```
PARSEC-TEST> (with-input-from-string (stream "(1 + 2)/10")
               (parse stream 'expression))
(:LPAREN 1 (:OP #\+) 2 :RPAREN (:OP #\/) 10)
```

Тот же пример на Haskell выглядит так:

```
module Main where

import Text.ParserCombinators.Parsec

data Token = Atom String | Op String | LParen | RParen

parse' :: String → [Token]
parse' s = case (parse expression "expr" s) of
  Left err → error (show err)
  Right x → x

-- operator token
operator :: Parser Token
operator = do
  c ← char '-' <|> char '+' <|> char '*' <|>
char '/' <?> "operator"
  return $ Op (c:[])

-- atom token
atom :: Parser Token
atom = do
  a ← many1 alphaNum <?> "atom"
  return (Atom a)

-- single-char tokens
lparen = char '(' ≫ return LParen
rparen = char ')' ≫ return RParen

-- one token
tok :: Parser Token
tok = do
  t ← atom <|> operator <|> lparen <|> rparen
  skipMany space
  return t

-- the whole expression
expression :: Parser [Token]
expression = do
  skipMany space
  ts ← many1 tok
  eof
```

```
    return ts
```

Как видно, разница в интерфейсе определения парсеров невелика (за исключением внешнего синтаксического различия). Однако реализация разная. Рассмотрим основные идеи CL-PARSEC и роль, которую в ней играет сигнальный протокол.

4.5.2. Реализация CL-PARSEC

Каждый парсер — это обычная функция, которая возвращает результат разбора: это может быть отдельный символ, уже разобранная строка или число, а также значения `NIL` или `T`. Эти значения превращаются в итоговое значение с помощью обычного Лисп-кода без каких-либо дополнительных ограничений,[10] как можно видеть из примера выше.

Наибольшей проблемой для любой библиотеки, связанной с синтасическим анализом, на наш взгляд, являются возвраты (backtracking), особенно в случае таких потоков (streams), как в CL, которые позволяют отмотать (unread) только на один символ назад. Поэтому на каждый парсер накладываются следующие условия:

1) Во-первых, чтобы не передавать между вызовами поток, из которого происходит чтение, он передается неявно в специальной переменной `*stream*`. Более того, даже если бы мы решили передавать этот поток, нам бы все равно пришлось организовывать определенный механизм для просмотра вперед и возврата назад более, чем на 1 символ. Для этого добавлена специальная переменная `*backlog*`, в которую заносятся символы в случае невозможности вернуть их назад во входной поток.

2) Для того, чтобы инкапсулировать механизм чтения из потока или же из бэклога, введен макрос `next-item`, с помощью которого осуществляется получение следующего символа. Таким образом, клиентский код вообще не работает со специальными переменными. Более того, практически никогда ему не придется задействовать и `next-item`, поскольку для большинства случаев этот вызов инкапсулируется еще более высокоуровневым макросом `mkparser`, о котором далее.

3) В случае удачного разбора любой парсер, который вызывает `next-item`, должен просигнализировать `parsec-success` и в этом сигнале передать прочитанный символ. Семантика сигнала такова, что если никто его не обработает, то он просто будет проигнорирован. В то же время, обработать его можно на произвольном числе уровней выше по стеку. Необходимость в этом сигнале продиктована именно потребностью собирать знаки в бэклоге для такого оператора, как `try`, который выходит за пределы простых LL(1)-парсеров.

[10] Ограничения, накладываемые на парсеры в силу особенностей самой предметной области, описаны далее.

4) Наконец, в случае неудачи разбора любой парсер должен сигнализировать ошибку типа `parsec-error`. Соответственно, все парсеры высших порядков должны быть готовы обрабатывать эту ошибку тем или иным образом.

Благодаря последним двум пунктам пользователи при создании парсеров могут сосредоточиться всецело на логике их работы, особо не думая о том, каким образом они будут комбинироваться с другими: всё это берет на себя библиотека, предоставляя несколько макросов.

Начнем с точки входа для выполнения разбора, в которой происходит связывание специальных переменных. После этого их изменения, которые производятся при каждом чтении и возврате в поток, становятся изолированными внутри вложенных форм:

```
(defun parse (stream parser)
  "Parses STREAM with the given PARSER function. Binds specials."
  (let ((*stream* stream)
        *backlog*
        (*source* :stream)
        *current-item*)
    (funcall parser)))
```

Следующий макрос позволяет определять парсеры. Это обертка вокруг формы `defun`, служащей для определения функций, которая, во-первых, добавляет свое имя к стеку ошибок при возникновении `parsec-error`, а, во-вторых, предоставляет в своем теле локальную переменную `_parser-name_`, которую можно использовать в коде определения парсеров. В принципе, все эти задачи являются служебными, так что по сути определение парсера — это просто определение функции.

```
(defmacro defparser (name (&rest args) &body body)
  "DEFUN a parser function with the given NAME and ARGS. BODY is
      wrapped in HANDLER-CASE, that traps PARSEC-ERROR and
      resignals it with NAME added to the error stack. Provides
      internal variable _PARSER-NAME_. Intended for top-level
      use, like DEFUN."
  `(defun ,name (,@args)
     ,(when (stringp (car body))
        (car body))
     (let ((_parser-name_ ',name))
       (handler-case
           (progn ,@body)
         (parsec-error (e) (?! ',name e))))))
```

Однако эти фрагменты кода не дают ответа на вопрос, как же производится чтение и возврат символов. Для описания этой логики используются макрос `mkparser` и функция `parsecall`, которые необходимо использовать в коде, использующем библиотеку.

```
(defun parsecall (test &optional (return :parsed))
  "Funcall TEST as if it was already passed to MKPARSER. RETURN
      semantics repeats MKPARSER's one."
  (funcall (mkparser test return)))

(defmacro mkparser (test &optional (return :parsed))
  "Creates the parser, that implements the following strategy: it
      reads the NEXT-ITEM, checks it with PARSE-TEST (that
      implements the polymorphic logic of testing, depending on
      TEST's type),
 * if test passes, it signals PARSEC-SUCCESS and returns
   - the item (default case)
   - TEST (if return = :test)
   - otherwise RETURN itself
 * otherwise it performs UNREAD-LAST-ITEM and signals
     PARSEC-ERROR."
  (with-gensyms (cur item gtest greturn)
    `(lambda ()
       (let ((,gtest ,test)
             (,greturn ,return)
             (,cur (next-item)))
         (if (parse-test ,gtest ,cur)
             (progn (signal 'parsec-success :result ,cur)
                    (case ,greturn
                      (:parsed   ,cur)
                      (:test     ,gtest)
                      (otherwise ,greturn)))
             (progn (unread-last-item)
                    (?! `(parser ,,gtest))))))))
```

Если убрать весь служебный код, то логика символьного парсера, создаваемого `mkparser` такова:

1) прочитать символ с помощью `next-item`;
2) проверить его с помощью обобщённой функции `parse-test` (это синтаксический сахар для того, чтобы можно было задавать функции принадлежности к классам символов не только непосредственно, как в `(mkparser #'alpahnumericp)`, но и прямо символами `((mkparser #\a)`), а также, например, списками, обозначающими применение функции `((mkparser '(member (#\a #\b)))`);
3) в случае успеха вернуть этот символ, просигнализировав на более высокие уровни с помощью `parsec-success`;
4) в случае неудачи вернуть символ обратно в поток с помощью `unread-last-item` и просигнализировать ошибку `parsec-error` с помощью `?!`.

Осталось рассмотреть, собственно, работу `next-item` и `unread-last-item`. Они полагаются на обобщённые функции `read-item` и `unread-item`, реализующие специфичную логику для чтения/возврата для потока и списка (бэклога).

```
(defmacro next-item ()
  "Sets *CURRENT-ITEM* to newly READ-ITEM from *STREAM* or
      *BACKLOG* and returns it."
  (with-gensyms (cur)
    `(prog1 (setf *source* (if *backlog* :backlog :stream)
                  *current-item* (read-item (or *backlog*
                                                *stream*)))
       (when *echo-p*
         (format *echo-stream* "~:@c" *current-item*)))))

(defmacro unread-last-item ()
  "Unread *CURRENT-ITEM* to current *SOURCE*."
  `(unread-item *current-item*
                (if (eq *source* :stream) *stream*
                    *backlog*)))
```

На этом примере можно видеть, что внутренняя реализация символьного парсера довольно сильно полагается на манипуляцию специальными переменными. Это не создаёт проблем благодаря изоляции таких переменных, обусловленной отличием специальных переменных от глобальных, а также четкой концептуальной границей, создаваемой с помощью макросов. Да, в этом случае возможно преднамеренное нарушение этой логики путем создания и использования парсера, который будет «портить» значения этих переменных. Но, оставив этот вариант в стороне,[11] случайное вмешательство в нее, например, по ошибке, является невозможным. В то же время, за счет того, что в самой основе лежат полиморфные операции `read-item` и `unread-item`, можно легко расширить библиотеку для использования других источников символов для разбора (например, бинарных потоков), не меняя логику передачи значений.

Теперь рассмотрим реализацию различных парсеров в этой парадигме. Простейшие парсеры определены в файле `simple.lisp` и не принимают ничего на вход.

```
(defparser eof ()
  (if-eof (return-from eof t)
    (next-item))
  (?!))
```

Парсеры, параметризуемые классом символов, определены в `item-level.lisp`:

```
(defparser skip (test)
  (parsecall test)
  nil)
```

[11] Если формулировать это как проблему безопасности, то решение ее должно быть глобально для всего кода. В Lisp'е как динамическом языке есть очень много возможностей вмешательства помимо этой: например, переопределения функций.

```
(defparser look-ahead (test)
  "LL(1) lookahead"
  (progn (parsecall test)
         (unread-last-item)
         t))
```

Парсеры высших порядков определены в `higher-order.lisp`:

```
(defparser either (&rest parsers)
  "If either of the PARSERS returns, return its result. Ordering
     matters. LL(1) variant. If it doesn't suit you, use this
     pattern: (EITHER (TRY ...))"
  (dolist (parser parsers)
    (if-err nil
      (return-from either (funcall parser))))
  (?!))

(defparser try (parser)
  "Returns the result of PARSER's application or 'unreads' all
     the read items back."
  (let (backlog)
    (prog1 (intercept-signals cur-signal
             ((parsec-success (push (parsing-result cur-signal)
                                    backlog))
              (try-success (setf backlog (append
                                          (parsing-backlog cur-signal)
                                          backlog))))
             (if-end (progn (setf *backlog* (append (reverse
                                                     backlog) *backlog*)
                                  *source* :backlog)
                            (?!))
               (funcall parser)))
      (signal 'try-success :backlog backlog))))

(defparser maybe (parser)
  "A wrapper around TRY to ignore unsuccessful attempts and
     simply return NIL."
  (if-err nil
    (try parser)))
```

Какие общие черты можно выделить в этих определениях?

- клиенту библиотеки нужно помнить о том, что в случае неудачного разбора, необходимо сигнализировать ошибку, для чего введена функция `?!`. Правда, во многих случаях можно положиться на то, что ошибку сигнализирует `parsecall`;

- используется ряд вспомогательных макросов, типа `if-err`, `if-eof`, `if-end`, которые представляют собой специфичный для этой области язык — это просто кодификация стандартных шаблонов реакции на обычные события: ошибка разбора, конец потока, конец или ошибка и т. п.

Кроме того, в случае `try` используется более низкоуровневый макрос `intercept-signals`. Для приведенного случая он имеет следующее раскрытие:

```
(HANDLER-BIND
 ((PARSEC-SUCCESS (LAMBDA (THIS-SIGNAL)
                    (UNLESS (MEMBER _PARSER-NAME_
                                    (SIGNAL-FEATURES
                                        _THIS-SIGNAL_))
                      (PUSH (PARSING-RESULT THIS-SIGNAL) BACKLOG))
                    (PUSHNEW _PARSER-NAME_
                             (SIGNAL-FEATURES THIS-SIGNAL))))
  (TRY-SUCCESS (LAMBDA (THIS-SIGNAL)
                 (UNLESS (MEMBER _PARSER-NAME_
                                 (SIGNAL-FEATURES THIS-SIGNAL))
                   (SETF BACKLOG (APPEND (PARSING-BACKLOG
                                          THIS-SIGNAL)
                                         BACKLOG)))
                 (PUSHNEW _PARSER-NAME_
                          (SIGNAL-FEATURES THIS-SIGNAL)))))
  (IF-END (PROGN
            (SETF *BACKLOG* (APPEND (REVERSE BACKLOG) *BACKLOG*)
                  *SOURCE* :BACKLOG)
            (?!))
    (FUNCALL PARSER)))
```

Таким образом, в макросе инкапсулируется проверка на повторную обработку одного и того же сигнала в одноименных парсерах выше по стеку. Для случая `try` это необходимо, поскольку каждый такой парсер представляет отдельную ветку разбора, которая будет «свернута» в общее дерево в случае неудачи. Нужно отметить, что такая относительно сложная реализация понадобилась только для данного нестандартного парсера. Специальный сигнал `try-success` понадобился в данном случае для того, чтобы учесть случай вложенных вызовов `try` — когда более низкоуровневый вызов перехватывает часть символов, которые могут затем потеряться в случае отката, происходящего **после** успешного возврата из `try` на том же уровне вложенности.

Наконец, рассмотрим пример применения этих парсеров для описания правил, подобных тем, которые были приведены в заметке Ивана Веселова. В них активно используется наш собственный макрос чтения `#'` для компактного задания анонимных функций от нуля или одной переменной, который позволяет существенно сократить функциональный код и сделать его более единообразным (с записью `#'`).

```
(defparser spaces ()
```

```
  (skip-many #\Space))

(defparser expression ()
  (prog1 (subexpression)
    (spaces)
    (eof)))

(defparser subexpression ()
  (spaces)
  (many+ #'element))

(defparser element ()
  (spaces)
  (either #`(try #'func)
          #'token
          #'operator))

(defparser token ()
  (list :atom
        (coerce (many+ #`(parsecall #'alphanumericp))
                'string)))

(defparser operator ()
  (list :op
        (parsecall '(member (#\- #\+ #\* #\/ #\^ #\.)))))

(defparser func-arg ()
  (prog1 (subexpression)
    (spaces)
    (either #`(look-ahead #\))
            #`(parsecall #\,))))

(defparser func-end ()
  (spaces)
  (parsecall #\)))

(defparser func ()
  (append (list :func
                (second (prog1 (token)
                               (spaces)
                               (parsecall #\()))
          (prog1 (many #'func-arg)
            (func-end))))
```

Например, по ним может разбираться следующее выражение:

```
PARSEC-TEST> (with-input-from-string (in "1 + 2-10a ^ ac1(bd+f(),
```

```
    z)")
                (parse in 'expression))
((:ATOM "1") (:OP #\+) (:ATOM "2") (:OP #\-) (:ATOM "10a") (:OP
    #\^)
 (:FUNC "ac1" ((:ATOM "bd") (:OP #\+) (:FUNC "f"))
              ((:ATOM "z"))))
```

Также интерес представляет случай небольшого изменения входной строки: уберем из нее последний аргумент функции, z, оставив в тексте запятую. В данном варианте, поскольку мы оперируем в рамках контекстно-свободной грамматики, и не можем в правилах задать проверку на висящую запятую, разбор даст следующий результат:

```
((:ATOM "1") (:OP #\+) (:ATOM "2") (:OP #\-) (:ATOM "10a") (:OP
    #\^) (:FUNC "ac1" ((:ATOM "bd") (:OP #\+) (:FUNC "f"))))
```

Впрочем, если необходимо, мы легко можем спуститься на уровень Лиспа и задействовать его возможности, например, те же специальные переменные для перехода в контекстно-зависимую грамматику. Это достигается следующей модификацией приведенных правил:

```
(defvar *more-func-args* nil)

(defparser func-arg ()
  (prog1 (subexpression)
    (setf *more-func-args* nil)  ; еще один нашелся
    (spaces)
    (either #'(look-ahead #\))
            #'((parsecall #\,)  ; должны быть еще
               (setf *more-func-args* t)))))

(defparser func ()
  (nconc (list :func
               (second (prog1 (token)
                         (spaces)
                         (parsecall #\())))
         (let (*more-func-args*)
           (prog1 (many #'func-arg)
             (when *more-func-args*
               ;; закончили разбор аргументов, а должен быть еще
                   один
               (?!))
             (func-end)))))
```

Теперь вызов

```
PARSEC-TEST> (with-input-from-string (in "1 + 2-10a ^ ac1(bd+f(),
    )")
                (parse in 'expression))
```

вернЕт ошибку:

```
Parsing error. Error stack: ((EXPRESSION)(EOF))
```

4.5.3. Выводы

В данном примере, как и в предыдущих, был задействован целый ряд технологий языка:

- специальные переменные, благодаря использованию которых удалось убрать дублирование кода, работающего с потоком;
- обобщённые функции, которые, как всегда, использовались там, где необходимо было обеспечить полиморфное поведение;
- макросы, применяемые для частичной кодификации ограничений, налагаемых на парсеры, а также для создания проблемно-ориентированных абстракций;
- макросы чтения, позволяющие точечно увеличить читаемость кода;
- сигнальный протокол, с помощью которого удалось четко разделить два потока управления (то, для чего в Haskell-реализации используются монады).

Все вместе они позволяют решить задачу создания высокоуровневого декларативного языка описания парсеров, которые являются полноценными объектами хост-языка (обычными функциями), в то же время предоставляя возможность при необходимости влиять на детали внутренней реализации на любом необходимом уровне.

Последнее, на чем мы хотели бы остановиться — это возможности по отладке этого решения, которые использовались нами в Лисп-среде. Часто отладка кода такого рода, включающего высокоуровневые правила и две параллельных нити принятия решений о передаче управления, требует особого подхода. Однако в данном случае нам вполне хватило механизма трассировки, представленного стандартной функцией `trace` (а также пары `print`-выражений).

Ниже приведена трассировка разбора простейшего токена для правил, приведенных выше. На ней хорошо видны вызовы отдельных парсеров и их возвращаемые значения, а также случаи, когда выполнение прерывалось с помощью исключения (в листинге это отсутствие возврата из функции). Читаемость этого примера можно дополнительно повысить, если передавать не анонимные функции, создаваемые `mkparser`, а именованные. Наконец, последней возможностью является выборочный показ только тех вызовов, которые нас интересуют (`trace` обладает и такой возможностью).

```
PARSEC-TEST> (with-input-from-string (in "f")
               (parse in 'expression))
  0: (SUBEXPRESSION)
    1: (SKIP-MANY #\ )
      2: (UNREAD-ITEM #\f #<SB-IMPL::STRING-INPUT-STREAM
          {ACC6309}>)
```

```
      2: UNREAD-ITEM returned NIL
    1: SKIP-MANY returned NIL
    1: (MANY+ #<FUNCTION ELEMENT>)
      2: (SKIP-MANY #\ )
        3: (UNREAD-ITEM #\f #<SB-IMPL::STRING-INPUT-STREAM
            {ACC6309}>)
        3: UNREAD-ITEM returned NIL
      2: SKIP-MANY returned NIL
      2: (EITHER #<FUNCTION (LAMBDA #) {A92F355}>
                 #<CLOSURE SB-IMPL::ENCAPSULATION {A92F40D}>
                 #<CLOSURE SB-IMPL::ENCAPSULATION {A930F8D}>)
        3: (TRY #<FUNCTION FUNC>)
          4: (TOKEN)
            5: (MANY+ #<FUNCTION (LAMBDA #) {A92081D}>)
              6: (MANY #<FUNCTION (LAMBDA #) {A92081D}>)
              6: MANY returned NIL
            5: MANY+ returned (#\f)
          4: TOKEN returned (:ATOM "f")
          4: (SKIP-MANY #\ )
          4: SKIP-MANY returned NIL
        3: (TOKEN)
          4: (MANY+ #<FUNCTION (LAMBDA #) {A92081D}>)
            5: (MANY #<FUNCTION (LAMBDA #) {A92081D}>)
            5: MANY returned NIL
          4: MANY+ returned (#\f)
        3: TOKEN returned (:ATOM "f")
      2: EITHER returned (:ATOM "f")
      2: (MANY #<FUNCTION ELEMENT>)
        3: (SKIP-MANY #\ )
        3: SKIP-MANY returned NIL
        3: (EITHER #<FUNCTION (LAMBDA #) {A92F355}>
                   #<CLOSURE SB-IMPL::ENCAPSULATION {A92F40D}>
                   #<CLOSURE SB-IMPL::ENCAPSULATION {A930F8D}>)
          4: (TRY #<FUNCTION FUNC>)
            5: (TOKEN)
              6: (MANY+ #<FUNCTION (LAMBDA #) {A92081D}>)
          4: (TOKEN)
            5: (MANY+ #<FUNCTION (LAMBDA #) {A92081D}>)
      2: MANY returned NIL
    1: MANY+ returned ((:ATOM "f"))
  0: SUBEXPRESSION returned ((:ATOM "f"))
  0: (SKIP-MANY #\ )
  0: SKIP-MANY returned NIL
((:ATOM "f"))
```

Примечание: как видно из этой трассировки, представленная версия библиотеки реализует простейший подход и поэтому имеет некоторые недостатки: например, ал-

горитм имеет экспоненциальную сложность, а также сообщения об ошибках не всегда достаточно информативны. Эти аспекты вполне поддаются доработке, но ее рассмотрение выходит за рамки этой статьи.

4.6. Роль протоколов в языке

В предыдущих разделах статьи не раз упоминался термин «протокол», в основном в контексте внешних для языковой среды процессов: передаче данных по сети, способам взаимодействия с сервером БД и т. п. Однако эта концепция находит плодотворное применение и в рамках собственно языка.

Как говорит Википедия, протокол — это набор правил, используемых компьютерами при взаимодействии. Кент Питман, один из авторов стандарта Common Lisp и разработчик его системы обработки исключений, пишет [5], что:

> Установление протоколов — это определенная заблаговременная страховка от дилеммы узника (prisoner's dilemma), которая дает очевидный способ структурирования независимо разрабатываемого кода для людей, которые не общаются напрямую, так, чтобы при последующем комбинировании этого кода он оставался согласованным.

В объектно-ориентированном программировании понятие протокола часто объединяют с концепцией интерфейса.[12] Следует отметить, что это несколько ограничивает его: «набор правил», составляющих протокол, не обязательно должен относиться к одному классу, более того он может быть выражен в языке разными способами, примеры которых будут приведены ниже. В общем случае, протокол — это больше чем отдельный интерфейс.

Протоколы являются инфраструктурными компонентами языка, на основе которых могут строиться его «прикладные» модули. Таким образом в Common Lisp реализована объектная система CLOS (включая и описанный ранее механизм обобщённых функций), в основе которой лежит мета-объектный протокол (MOP), спецификации которого посвящена целая книга [2]. В основе системы обработки исключений — сигнальный протокол. Да и само ядро Лиспа, в определенном приближении, построено на протоколе манипуляции списками (во всяком случае это касалось первых реализаций языка). Также в Common Lisp можно выделить и другие протоколы, такие как, например: протокол функций по работе с последовательностями (sequences) и протокол множественных возвращаемых значений.

Протокол функций по работе с последовательностями представляет из себя набор именованных параметров, присутствующих во всех таких функциях, а именно: параметры `key`, `test`, `test-not`, `from-end`, `start`, `end`, а также неформальные правила их использования. Таким образом определяется удобный фреймворк для объединения последовательностей и функций высших порядков. Это очень простой пример, который даже не описан формально.

[12] http://en.wikipedia.org/wiki/Protocol_(object-oriented_programming)

Протокол множественных возвращаемых значений — это набор форм языка, создающих инфраструктуру для того, чтобы функция могла вернуть больше одного значения, а вызывающая сторона могла распорядиться ими по своему усмотрению или же, вообще ничего не зная об этом, работать с основным значением. Таким образом, в том числе, устраняется многолетнее противоречие по поводу того, как возвращать ошибки из функции: с помощью кодов ошибок или исключений,[13] ведь в таком варианте можно вернуть как значение, так и код ошибки. Этот протокол составляют формы `values` и `values-list`, и несколько макросов (`multiple-value-*` (`-bind`, `-list`, …), а также интеграция возможности возвращения нескольких значений в специальном операторе `progn` (который повсеместно используется в языке). Это пример протокола, поддержка которого требует вмешательства в ядро языка.

Как видите, эти протоколы не требуют для своей спецификации целой книги или даже статьи, более того, они определены неформально. Однако, продуманные протоколы приносят пользу как благодаря тому, что могут служить основой для построения более высокоуровневых абстракций, так и в качестве «лучших практик» реализации подобных вещей (это особенно применимо к протоколу работы с последовательностями). Можно привести достаточно примеров инфраструктурных библиотек для Common Lisp, использующих MOP: CLSQL, Elephant или же ранее упомянутая Filtered Functions). На основе сигнального протокола можно создать много интересных решений, об одном из которых мы рассмотрели ранее. Это одна из тех сфер в Лиспе, которые все еще остаются малоизученными и не задействованными полноценно.

«Протокольный» подход как основа построения схем межсистемного взаимодействия распространен не только в Лиспе. Тот же Redis использует эту концепцию для описания способа взаимодействия с клиентом; интересным «экстремальным» примером является рассмотрение системы контроля версий git как своеобразного протокола, на основе которого можно строить рабочий процесс разработки. Как было видно в примере из первой части статьи, Лисп дает возможность реализовать (продублировать) подобные протоколы.

Таким образом, протоколы — это не столько формальный прием или шаблон, сколько определенный взгляд на проектирование. Они могут быть реализованы в языке разными способами, вплоть до включения непосредственно в его ядро. В примере, который будет приведен далее, реализация протокола по сути мало чем отличается от описания интерфейса `ISeq` в языке Java (единственное, чего нельзя сделать в Java — это задействовать макросы для создания многоуровневых абстракций). Как и многие другие подходы, описанные в статье, протоколы — это прием, в той или иной степени доступный для использования в любом языке.

4.6.1. Практический пример: протоколы для последовательностей

Нельзя сказать, что в Common Lisp концепция протоколов учтена и реализована сполна. Несмотря на наличие определенного «протокола» работы с последовательно-

[13] http://www.joelonsoftware.com/items/2003/10/13.html

стями, он не проработан до конца. Вот что пишет Питер Норвиг [4]:

> Common Lisp находится в переходном положении между Лиспами прошлого и будущего. Нигде это так не заметно, как в функциях работы с последовательностями. Первые Лиспы имели дело только с символами, числами и списками. [...] Современные Лиспы добавили поддержку векторов, строк и других типов данных, и ввели понятие последовательности.
>
> С новыми типами данных пришла проблема названий функций, которые ими оперируют. В некоторых случаях Common Lisp решил развить существующую функцию (`length`). В других старые имена были сохранены специально для функций манипуляции списками (`mapcar`) и для обобщённых функций над последовательностями (`map`) были придуманы новые имена. В третьих случаях новые функции были придуманы для операций над специфическими типами данных (`aref`).

Еще большей проблемой оказалось то, что в определенном плане CLOS не был до конца интегрирован в язык, из-за чего существует разделение между стандартными классами (такими как типы последовательностей) и определяемыми пользователем. Выражаясь в объектно-ориентированных терминах, первые можно сравнить с финальными классами в Java (от которых нельзя унаследовать), а вторые — с открытыми классами в Ruby (с которыми можно сделать в общем-то все, что угодно). Это принято обосновывать необходимостью дать создателям реализаций языка определенное пространство для маневра для эффективной реализации ядра. Однако в Лиспе в отличие от некоторых других языков (вспомнить хотя бы примитивные типы в JVM или же Python, в котором как основа для наследования созданы специальные классы-обертки, типа UserDict для словарей или UserList для списков) подобные компромиссы с единообразностью не поощряются сообществом.

Описывая свою мотивацию для реализации языка Clojure в тесной привязке к JVM-платформе, его создатель Рич Хики написал следующее[14]:

> Когда мы смотрим на CL, мы видим подобную рефлексивную, полиморфную расширяемость только на уровне CLOS. CLOS реализован на более низкоуровневом языке (Lisp без CLOS, а этот Lisp в свою очередь на C или чем-то подобном), который не предоставляет полиморфной расширяемости. Clojure написан на языке, который предоставляет оную. Так что, например, `first`/`rest` в Clojure (и вещи, построенные на них) могут быть полиморфно расширяемы пользователями (в Clojure или Java), тогда как `car` и `cdr` во всех Лиспах, с которыми мы знакомы, не могут.

Справедливости ради нужно сказать, что проблема «неповсеместного» использования CLOS в качестве механизма для полиморфизма не является неразрешимой (так что вряд ли это можно признать основной причиной, почему Clojure появился на JVM — скорее, тут дело в перспективах этой платформы в индустрии).

[14] http://groups.google.com/group/comp.lang.lisp/msg/570cc2eaadea36af

Во-первых, вполне практичным представляется использование описанного выше Python-подхода. Во-вторых, все же Lisp остается языком, в котором создаваемые пользователем конструкции почти всегда являются первоклассными объектами, существующими на равных правах с тем, что предоставляет сам язык. Поэтому есть и яркий практический контр-пример в этом случае — это FSet, Common Lisp-библиотека теоретико-множественных персистентных коллекций, эквивалентная (а в некоторых аспектах, даже более широкая) по своим возможностям библиотеке функциональных структур Clojure. При ее реализации активно использовался механизм затенения имен (shadowing), который можно применять в том числе и к стандартной библиотеке языка. Таким образом автор смог создать полноценный протокол манипуляции такими структурами данных, используя необходимые ему имена и абстракции (библиотека основанна на обобщённых функциях) в рамках языка.

Еще одним примером расширяемости, о котором говорит Хики, является реализованный в Clojure SEQ-протокол итерации над абстрактными последовательностями (те самые полиморфные `first`/`rest`). В его основе идея об отказе от привязки концепции «списка» к конкретным аспектами реализации на основе cons-ячеек, а рассмотрение его как абстрактного интерфейса. Используя такой подход, можно программировать обобщённые алгоритмы, которые смогут работать с любыми конкретными структурами, реализующими SEQ-интерфейс.

Впрочем, SEQ-протокол нетрудно реализовать и в Common Lisp, используя механизмы расширения, которые предоставляет язык. Пример такой реализации приводится ниже. Для экономии места она представлена только для двух конкретных структур данных: последовательностей (к которым в CL относятся список и разные варианты векторов) и хеш-таблиц, — как для принципиально разных типов коллекций. Протокол основывается на классе-обертке `seq`, который создается вызовом обобщённой функции `seq` на экземпляре конкретной или абстрактной (SEQ) коллекции, а также обобщённых функциях `head`, `tail` и `next`, которые возвращают первый элемент коллекции, ее хвост, а также следующий элемент при итерации, удаляя его из коллекции (аналог `pop` для списков). Кроме того, определены такие вспомогательные функции, как `seqp`, отвечающая на вопрос, является ли объект экземпляром класса `seq`, `into` для преобразования из конкретной коллекции в абстрактную и обратно, а также `elm` для получения элемента коллекции по ключу (для последовательностей ключ — это порядковый номер элемента).

Ниже для примера приведены некоторые классы и функции ядра подобного SEQ-протокола (в общей сложности его реализация занимает 100 строк кода).

```
(defclass seq ()
  ((ref :initarg :ref :reader ref
        :documentation "reference to the underlying concrete
            data-structure")
   (pos :initarg :pos :initform 0 :accessor pos
        :documentation
        "current postion in REF, used for copying by reference")
   (itr :initarg :itr :initform (error "ITR should be provided")
```

```
          :documentation
          "iterator closure, which returns 2 values: value and key
              (for simple sequences key is number); when end is
              reached should just return nil (no errors)")
     (kv? :initarg :kv? :initform nil
          :documentation "wheather keys are meaningful for the
              collection (e.g. for hash-tables they are)"))
    (:documentation "A wrapper around a concrete data structure for
        use in SEQ protocol."))

(defgeneric head (coll)
  (:documentation "Return first element of a collection COLL. 2nd
      value: it's key (for sequences key is the number)")
  (:method ((coll sequence))
    (values (first coll)
            0))
  (:method ((coll hash-table))
    (with-hash-table-iterator (gen-fn coll)
      (multiple-value-bind (valid key val) (gen-fn)
        (when valid
          (values val
                  key)))))
  (:method ((coll seq))
    (with-slots (pos itr) coll
      (funcall itr pos))))

(defgeneric next (coll)
  (:documentation "The analog of POP.  Return the first element
      of COLL and remove it from the top of COLL.  For SEQ it's
      done in functional manner, while for concrete types the
      removal may be destructive.")
  (:method ((coll list))
    (values (pop coll)
            0))
  (:method ((coll hash-table))
    (with-hash-table-iterator (gen-fn coll)
      (multiple-value-bind (valid key val) (gen-fn)
        (when valid
          (remhash key coll)
          (values val
                  key)))))
  (:method ((coll seq))
    (multiple-value-prog1 (head coll)
      (incf (pos coll)))))
```

На этой основе далее можно создавать различные высокоуровневые конструкции для итерации. Например `doseq` — это простой макрос в духе `dolist`:

```
(defmacro doseq ((var-form coll &optional result) &body body)
  "The analog of DOLIST for any COLLection, which can be put in a
      SEQ.

VAR-FORM can be either ATOM, in which case this name is bound to
    the value (1st), returned by NEXT, or LIST, in which case the
    first symbol will be bound to the 1st value (value), returned
    by NEXT, and the 2nd - to the 2nd (key).

If RESULT form is provided, its value will be returned."
  (with-gensyms (gseq key val)
    `(let ((,gseq (seq ,coll)))
       (loop
          ;; first we obtain VAL and KEY separately
          (multiple-value-bind (,val ,key) (next ,gseq)
            ;; and then bind them to the VAR-FORM
            (multiple-value-bind ,(mklist var-form) (values ,val
                ,key)
              (if ,key (progn ,@body)
                  (return ,result)))))))))
```

Пример его применения:

```
CL-USER> (doseq (v (seq '(1 2)))
           (print v))
1
2
```

Стоит отметить, что основная задача SEQ-протокола — это дать возможность единообразной итерации на разных конкретных коллекциях. Но он не призван решать проблему аккумуляции значений в процессе итерации. Точнее, такая проблема просто не стоит: можно спокойно использовать список как самую простую и эффективную структуру для этой цели, а затем преобразовывать его в требуемый конкретный тип при наличии такой необходимости (например, с помощью операции `into`). Вполне можно определить аналог операции `cons` в семантике Clojure, но ее использование будет достаточно неэффективным: на каждой итерации прийдется выполнять generic-диспечеризацию (в Clojure это неизбежно, а вот в Common Lisp у нас есть выбор). Для формализации использования предложенного выше подхода с аккумуляцией в список был создан макрос `with-acc`:

```
(defmacro with-acc ((var-form acc-name coll &optional result-type)
                    &body body)
  "Iterate over COLL with accumulation to list and conversion
      INTO either RESULT-TYPE or TYPE-OF COLL itself.

VAR-FORM can be either ATOM, in which case this name is bound to
    the value (1st), returned by NEXT, or LIST: the first symbol
```

```
    will be bound to the 1st value (value), returned by NEXT, and
    the 2nd - to the 2nd (key).  If RESULT form is provided, it's
    value will be returned."
  (with-gensyms (gseq)
    `(let ((,gseq (seq ,coll)))
       (into (or ,result-type (ref ,gseq))
             (with-output-to-list (,acc-name)
               (doseq (,var-form ,gseq)
                 ,@body))))))
```

Далее с его помощью можно создать функции еще более высокого уровня абстракции, например `filter`.

```
(defun filter (fn coll)
  "Sequentially apply FN to all elements of COLL and return the
     collection of the same type, holding only non-null results"
  (let ((seq (seq coll)))
    (with-slots (kv?) seq
      (ref (with-acc ((v k) acc seq)
             (when-it (funcall fn v)
               (when kv?
                 (push k acc))
               (push it acc)))))))
```

На примере реализации SEQ-протокола снова хорошо видно, каким образом в Лиспе принято строить высокоуровневые абстракции: в их основе лежит ортогональный набор базовых операций и структур данных, на которые затем уровень за уровнем наслаиваются новые абстракции. Именно так реализован и сам язык [1].

Впрочем, это не последний вариант протокола, созданного вокруг последовательностей и итерации: на самом деле, в CL их еще несколько! Одной из более общих альтернатив является библиотека ITERATE. Ее основная цель — исправить недостатки, присущие другому, более древнему, квази-протоколу итерации LOOP, а именно:

- отсутствие скобок, а точнее, разбиения на подформы (что приводит к трудностям при использовании в макросах и выравнивании в редакторах);
- слишком сложный (и незапоминающийся) синтаксис для итерации по некоторым коллекциям (например, хеш-таблицам);
- отсутствие стандартного механизма расширения (хотя некоторые реализации, в частности SBCL, предоставляют свой механизм).

В отличие от LOOP, ITERATE, как и любой полноценный протокол, предоставляет **механизм пользовательского расширения** — это макросы `defclause` и `defdriver-clause`. Попробуем использовать второй из них для интеграции нашего SEQ-протокола с ITERATE-макросом — это весьма несложно:

```
#+:iter
(defdriver-clause (:FOR var :SEQ coll)
  "Iterate over COLL as a SEQ."
  (with-gensyms (gseq k v)
    '(progn (:with ,gseq := (seq ,coll))
            (:for ,var :next (multiple-value-bind (,v ,k)
                                 (next ,gseq)
                               (if ,k ,v
                                   (:terminate)))))))
```

И в качестве примера использования этого нового драйвера можно привести такой простейший код:

```
CL-USER> (iter (:for item :seq #{:a 1 :b 2})
               (print item))
1
2
```

Вот мы и пришли к протокольно-ориентированному программированию! Кстати, кроме шуток: как раз в Clojure недавно была предложена такая концепция, как `defprotocol`, позволяющая формулировать группы интерфейсов (вроде SEQ) как протоколы, в общем случае без привязки к конкретному классу. Вот так и происходит развязывание полиморфизма и наследования.[15]

Примечание: в последних фрагментах кода использованы некоторые синтаксические расширения Common Lisp, которые мы используем в своей работе, и которые для себя может произвести любой программист. В частности, это анафорический макрос `when-it`, литеральный синтаксис для чтения хеш-таблиц `#{}` и анонимных функций `#'`. Кроме того, продемонстрирована возможность Common Lisp, позволяющая одновременно использовать класс, функцию и переменную с одним и тем же именем `seq`;[16] и использован стандартный макрос чтения `#+`, который позволяет выполнять код условно в зависимости от наличия определенных ключевых слов в списке возможностей (features) данного программного образа. Поскольку библиотека ITERATE является инфраструктурной, она добавляет индикатор своего присутствия, чтобы об этом «знали» другие библиотеки и могли задействовать её через механизм `#+`. Таким же образом объявляет себя конкретная реализация (например, `:sbcl`), указывается, загружены ли некоторые важные компоненты среды (например, `:sb-thread`), какая у нас операционная система (например, `:win32` или `:posix`), вплоть до того, что это среда конкретного программиста (например, `:vseloved`). Кстати, механизм `#+` позволяет проверять не только на наличие отдельных свойств, но и на булевые предикаты от них.

[15] Можно также провести параллель между `defprotocol` и классами типов в Haskell.

[16] То, что во введении названо свойством «Лисп-N».

4.7. Заключение: с расширяемостью в уме

Итак, в статье были разобраны решения весьма различных практических задач в рамках Лисп-философии, в основе которой лежит концепция расширяемости.

Если рассмотреть каждый отдельный язык, то, как правило, можно найти его объединяющую идею. На ее основе принимаются ключевые решения на всех уровнях языка: от высокоуровневой архитектуры до подхода к реализации конкретных технологий. Для Common Lisp объединяющей идеей на данный момент является как раз расширяемость. Это, кстати, ключевое различие между Common Lisp и Scheme, которое на данный момент разделяет эти два направления развития изначальной Лисп-идеи. Это не значит, что Scheme — не расширяемый язык, наоборот его возможности в этом плане хорошо иллюстрирует [3]. Но это не является на данный момент ключевым свойством, на который делается упор при развитии языка. Также это тот критерий, по которому язык Clojure, хотя он и взял некоторые черты от Scheme, относится все-таки к Лисп-направлению.

Еще одним языком, который в основе своей поставил расширяемость, является Java. И в этом смысле интересно сравнить эти два направления развития. Java известна своей многословностью и церемониальностью, она не имеет практически никаких средств для синтаксической абстракции. В то же время у нее есть и сильная сторона, а именно пронизывающая весь язык модель семантической расширяемости через интерфейсы. Так что в общем-то не удивительно, что язык Clojure появился именно на Java-платформе, так же как многие другие языки в свое время вышли из Lisp-платформы (самый яркий пример тут, пожалуй, Smalltalk).

На самом верхнем уровне вопрос расширяемости тесно связан с одной из «священных коров» индустрии программирования — повторным использованием кода. Объектно-ориентированный подход обещал, что базовыми компонентами повторного использования будут классы и их наборы. Однако, как показывают провалы многих технологий, таких как JavaBeans, это тупиковое направление.[17] Ведь отдельные классы не могут быть независимыми компонентами, потому что в этом случае они рискуют превратиться либо в пространства имен, либо в монолитные монстры, противореча самой идее декомпозиции, основанной на классах. Сегодня ясно, что основным механизмом повторного использования являются библиотеки функций (приходилось встречать даже утверждения о «библиотечно-ориентированном» программировании). Необходимость безусловного задействования классов, диктуемая семантикой некоторых языков, подчас только добавляет ненужное усложнение в эту модель. Главным достижением ОО-подхода, по нашему мнению, является концепция интерфейса, т. е. спецификации полиморфного набора функций, необходимых для решения какой-то задачи.

Стоит отметить разницу между языками, в которых повторному использованию

[17] *Прим. ред.* — несмотря на то, что маркетинговые обещания JavaBeans действительно никогда не были в полной степени реализованы, эта технология повсеместно используется в Java-окружении и является частью любого сколько-нибудь серьезного фреймворка.

уделяется должное внимание, и всеми остальными. В первых из них все общие прикладные задачи, а часто и инфраструктурные, как правило, решаются на уровне независимых библиотек, которые пишут пользователи в соответствии со своим видением проблемы. При этом структура завимостей между библиотеками представляет собой произвольный граф. В тех же языках, где расширяемость затруднена (либо по причине плохой проработки инструментов для этого, либо из-за особенностей среды исполнения, либо по другим причинам, например лицензионным) есть тенденция к постепенному включению всего необходимого в стандартную библиотеку или же в какие-то крупные фреймворки, каждый из которых зачастую развивает собственный механизм подключения и повторного использования. Здесь структура зависимостей — это дерево или же несколько отдельных слабо пересекающихся деревьев. Если провести параллели с экономикой, то в первом случае имеет место свободная рыночная конкуренция, а во втором олигополия или же монополия.

В этом смысле показателен пример Python, известного своим девизом «Батарейки в комплекте», который подразумевает наличие в стандартной библиотеке (почти) всего, что нужно для разработки. В нём также существует развитая традиция фреймворков: от Zope и Django для веб до Numpy и Scipy в научной среде. Обратная сторона медали тут отмечена в другой цитате: «Пакет, который попадает в стандартную библиотеку, стоит одной ногой в могиле».[18] Также к таким языкам среди тех, с которыми нам приходилось иметь дело, мы бы по разным причинам отнесли C, C++, PHP и JavaScript.

Для создания удобной инфраструктуры библиотек важны такие свойства, как возможность четкого разделения внешнего API и реализации (и его декларативного описания), для описания компонент системы, средства управления зависимостями (в том числе управления версиями, чтобы не попадать в ситуацию типа «DLL hell»), а также учета особенностей прикладной платформы (включающей аппаратную платформу, ОС и конкретную реализацию языковой среды).

В Common Lisp для этого используются следующие технологии:

- динамические пакеты для инкапсуляции особенностей реализации (в отличие от, как правило, статических пространств имен большинства других языков);
- макросы для создания мини-языков для декларативного описания систем, дистрибутивов (в отличие от использования для этих целей внешних языков, таких как XML);
- обобщённые функции для создания расширяемых API (в отличие от обязательного задействования классов);
- механизм свойств (features) для описания прикладной платформы (в отличие от различных препроцессоров).

[18] http://tarekziade.wordpress.com/2010/03/03/the-fate-of-distutils-pycon-summit-packaging-sprint-detailed-report/

В то же время, что касается конкретных средств управления дистрибутивами, то тут есть определенное поле для улучшения. Система ASDF (и его дополнение ASDF-INSTALL) остается доминирующим механизмом на данный момент. Но впоследнее время появилось несколько альтернативных начинаний по созданию нового пакета для этой цели. Сильная сторона ASDF — это декларативный объектно-ориентированный способ описания компонентов системы и зависимостей. К слабым относятся определенный овер-инжиниринг, препятствующий развитию и доработке, и некоторые внешние «организационные» факторы, связанные с отсутствием стандартизованного механизма его задействования между реализациями, а также развитого центрального репозитория Лисп-библиотек, а-ля CPAN. В то же время развитие такого средства, на наш взгляд, должно идти именно с учетом децентрализованной структуры Лисп-среды, а не вопреки ей.

На этом мы хотим подвести итог статьи. Мнения, изложенные в ней, безусловно, представляют только один из возможных подходов к разработке, отличный как от доминирующих на данный момент объектно-ориентированного и скриптингового подходов, так и от широко представленного в этом журнале функционального. Мы надеемся, что для его подкрепления было приведено достаточно основанных на практическом опыте аргументов. В то же время очевидно, что любой подход — это набор компромиссов, основанных на определенной системе ценностей, — так что если выбрать другие приоритеты, то показанные преимущества могут потерять свою актуальность и быть нивелированными другими особенностями.

Литература

[1] *Graham P.* The roots of lisp. — 2001. http://www.paulgraham.com/rootsoflisp.html.

[2] *Gregor Kiczales Jim des Rivieres D. G. B.* The Art of the Metaobject Protocol. — 1991.

[3] *Harold Abelson J. S.* Structure and Interpretation of Computer Programs, second edition. — 1996. http://mitpress.mit.edu/sicp/full-text/book/book.html.

[4] *Norvig P.* Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. — 1992. http://norvig.com/paip.html.

[5] *Pitman K. M.* Condition handling in the lisp language family. — 2001. http://www.nhplace.com/kent/Papers/Condition-Handling-2001.html.

[6] *Raymond E. S.* The Art of Unix Programming. — 2003. http://www.faqs.org/docs/artu/.

[7] *Seibel P.* Practical Common Lisp. — 2006. — Есть русский перевод: http://lisper.ru/pcl/. http://www.gigamonkeys.com/book/.

[8] *Роман Душкин*. Полиморфизм в языке Haskell. — 2009. http://fprog.ru/2009/issue3/roman-dushkin-haskell-polymorphism/.

Оптимизирующие парсер-комбинаторы

Дмитрий Попов
thedeemon@fprog.ru

Аннотация

Статья рассказывает о технике парсер-комбинаторов для построения функций синтаксического анализа текста или других линейных данных. Описываются классические монадные парсер-комбинаторы, работающие со списками, метод их оптимизации с помощью мемоизации, а также оригинальная техника построения оптимизирующих парсеров, основанная на построении и оптимизации конечного автомата аналогичным набором операторов и комбинаторов.

The article describes parser combinators, a technique for creating functions for syntax analysis sequential data, such as text. The classic monadic parser combinators are described, then a way of optimizing them via memoization, and an original approach to creating optimizing parsers by building a finite state machine using a similar set of operators and combinators.

Обсуждение статьи ведётся в LiveJournal.

5.1. Введение

Парсером называется часть программы, которая из линейной последовательности простых данных (символов, лексем, байтов) с учетом некоторой грамматики строит более сложные структуры данных, неявно содержащиеся в исходной последовательности. Это может быть разбор конфигурационных файлов, разбор исходного кода на каком-либо языке программирования, разбор проблемно-ориентированных языков (DSL), чтение сложных и не очень форматов данных (XML, PostScript), разбор запросов и ответов сетевых протоколов, вроде HTTP, и т. п. Здесь и далее слова «парсинг» и «разбор» считаются синонимами.

Комбинаторы — это функции высшего порядка, которые из одних функций строят другие. Возможность принимать функции в качестве аргумента и возвращать их в качестве результата — отличительная черта функциональных языков программирования, в которых комбинаторы являются обычными функциями.

Парсер-комбинаторы[1] — это широко известная в узких кругах техника создания парсеров, использующая возможности функциональных языков программирования. На менее выразительных языках задача парсинга обычно решается кодогенерацией: по описанию грамматики, сформулированному на некотором внешнем DSL, специальная утилита генерирует исходный текст парсера на нужном языке программирования. Самые известные примеры таких генераторов — yacc (и его прямой аналог bison), ANTLR, Coco/R, JavaCC. Функциональные же языки позволяют построить парсер динамически, используя простейшие функции и комбинаторы для синтеза по правилам грамматики сложных парсеров из простых. При этом и сама грамматика, и семантические действия (выполняющиеся при успешном разборе того или иного элемента грамматики) формулируется на одном языке, а парсер-комбинаторы выступают в качестве DSEL (встроенного предметно-ориентированного языка)[2]. Думаю, для каждого адепта функционального программирования написание своей библиотеки парсер-комбинаторов — такая же обязательная программа, как для новичка в Haskell написание туториала по монадам.

5.2. Классические парсер-комбинаторы

Классический вариант парсер-комбинаторов на языке OCaml может выглядеть следующим образом.

Парсером будет функция, которая получает на вход список неразобранных символов и в случае успеха возвращает разобранное значение и неразобранный остаток списка, а в случае неудачи возвращает признак неудачи.

Тип результата будет выглядеть так:

[1] http://en.wikipedia.org/wiki/Parser_combinator

[2] На нефункциональных языках возможна похожая техника, но результатами комбинаторов там являются не сразу функции-парсеры, а структуры данных, которые интерпретируются отдельной функцией. Примеры: Boost.Spirit для C++ и LPEG для Lua.

```
type 'a parse_result = Parsed of 'a * char list | Failed
```

Здесь 'a — параметр типа, делающий тип `parse_result` полиморфным, благодаря чему парсеры могут возвращать значения разных типов.

Парсеры, созданные генераторами вроде yacc, обычно работают с последовательностью лексем, полученных от лексического анализатора, также сгенерированного внешней утилитой, например, lex. В парсер-комбинаторах мы можем сами выбирать на каком уровне работать — на уровне лексем или на уровне отдельных символов. Можно определить более общий случай, когда на входе может быть не список символов, а список чего угодно, например, список лексем, полученных от лексического анализатора, созданного с помощью ocamllex (аналога генератора лексеров lex). Для общего случая определение типа будет таким:

```
type ('a, 'b) parse_result = Parsed of 'a * 'b list | Failed
```

В случае успеха возвращенный неразобранный остаток списка может быть использован для дальнейшего разбора — так происходит движение вперед по исходному списку символов. При неудаче никаких списков не возвращается, поэтому «текущая позиция» остается без изменений. Такая схема естественным образом реализует разбор с откатами[3]: если какой-то парсер по мере работы довольно далеко продвинулся по входному списку, но в какой-то момент все же потерпел неудачу, то все его продвижение «не считается» и «текущая позиция» оказывается той же, что и до его вызова.

Сначала опишем примитивный парсер, который умеет разбирать один символ, удовлетворяющий заданному условию.

```
let p_pred p s =
  match s with
  | [] → Failed
  | h::t → if p h then Parsed(h, t) else Failed
```

Функция `p_pred` берет предикат `p` и список символов `s`. Если список пуст, то разбор заканчивается неудачей. Если не пуст, то если символ удовлетворяет предикату, возвращаем символ и остаток списка, иначе опять возвращаем неудачу. С помощью `p_pred` можно получить парсер, умеющий разбирать заданную букву или, например, произвольную цифру.

```
let p_char c = p_pred ((=) c)
let isdigit c = c>='0' && c<='9'
let p_digit = p_pred isdigit
```

Здесь мы используем карринг, чтобы сократить и упростить запись. Тип функции `p_pred` таков:

```
(char → bool) → char list → char parse_result
```

[3] В литературе чаще используется термин «возврат» вместо «отката», но здесь это может создать путаницу с возвратом разобранного значения.

Передав в `p_pred` только первый параметр, мы получаем функцию с типом:

```
char list → char parse_result
```

т. е. как раз функцию-парсер, применение которой к списку символов дает результат парсинга.

Теперь на сцене появляются комбинаторы, которые из примитивных парсеров составляют более сложные. Опишем оператор последовательности (аналог `ab` в регулярных выражениях):

```
let (>>>) a b s =
  match a s with
  | Parsed(_, s2) → b s2
  | Failed → Failed
```

В языке OCaml запись имени в скобках означает определение оператора. Ряд комбинаторов удобно использовать именно в виде операторов. Парсер `a >>> b` отрабатывает успешно, если сначала успешно отработает `a`, а потом `b`. Если один из них не разобрал свою часть строки, то и вся комбинация возвращает неудачу.

Теперь определим оператор альтернативы (`a|b` в регулярных выражениях):

```
let (|||) a b s =
  match a s with
  | Parsed _ as ok → ok
  | Failed → b s
```

Парсер `a ||| b` отрабатывает успешно, если успешно отработал `a` или `b`. Операторы составлены из трех символов, чтобы не пересекаться с входящими в язык операторами `|`, `||`, `>` и невходящим `>>`, который я обычно использую для композиции функций.

Опциональный парсер (аналог `a?` в регулярных выражениях):

```
let p_opt defval a s =
  match a s with
  | Parsed _ as ok → ok
  | Failed → Parsed(defval, s)
```

Если его аргумент отработал успешно, возвращается его результат, в противном случае все равно возвращается успех, но с заданным значением по умолчанию и исходным списком символов.

Комбинатор повторения (аналог `a*`):

```
let p_many a  =
  let rec loop s =
    match a s with
    | Parsed(_, s') → loop s'
    | Failed → Parsed((), s) in
  loop
```

Применяет парсер-аргумент столько раз, сколько сможет (0 или более). Если парсер-аргумент не сработал ни разу, это тоже считается успехом, а результатом разбора в любом случае будет `()` — значение типа `unit`, аналога `void` в Си. Таким образом, результат разбора каждой итерации выкидывается. Если результат нужен, то вот вариант с функцией накопления результата:

```
let p_manyf a f v0 =
  let rec loop v s =
    match a s with
    | Parsed(x, s') → loop (f v x) s'
    | Failed → Parsed(v, s) in
  loop v0
```

Здесь `v0` — начальное значение, которое трансформируется функцией `f` на каждой успешной итерации с использованием результата разбора на этой итерации. Например, вот так можно разобрать целое положительное число:

```
let mkInt v x = v * 10 + int_of_char x - 48
let p_int = p_manyf p_digit mkInt 0
```

Оператор последовательности выкидывал результат первого парсера и возвращал результат второго. Но что делать, если результат первого тоже важен? Тут можно задействовать монады. Создадим вспомогательный парсер:

```
let return x s = Parsed(x, s)
```

`return x` — парсер, возвращающий значение `x`, игнорируя переданный ему список символов и возвращая его в качестве неразобранного остатка. Теперь опишем оператор монадного сочленения парсеров:

```
let (>>=) p1 f s =
  match p1 s with
  | Parsed(x, s2) → f x s2
  | Failed → Failed
```

Его правый аргумент `f` — не парсер, а функция, которая получает значение из успешного результата отработки парсера `p1` и возвращает новый парсер, который тут же и применяется. Таким образом получается последовательное применение двух парсеров, где поведение и результат второго зависят от значения, разобранного первым. Например, разбор знакового целого числа может быть сделан так:

```
let p_int =
  p_opt 1 (p_char '-' >>> return (-1)) >>= fun sign →
  p_manyf p_digit mkInt 0 >>= fun n → return (sign * n)
```

Если полученная на входе строка начинается на минус, то успешно отрабатывает `p_char '-'`, его результат (символ '`-`') заменяется на `-1` и возвращается в качестве результата `p_opt`, а если минуса не было, то возвращается значение по умолчанию: `1`.

Данное значение запоминается под именем `sign`. Потом разбираются цифры, из которых получается целое число `n`. Результат разбора цифр умножается на `sign` и полученное значение считается результатом всего разбора `p_int`.

А вот так выглядит разбор вещественного числа вида `-123.4567`:

```
let mkFloat (fv,fr) c =
  fv +. float_of_int (int_of_char c - 48) *. fr , fr *. 0.1

let p_float =
  p_opt 1.0 (p_char '-' >>> return (-1.0)) >>= fun sign →
  p_manyf p_digit mkInt 0 >>= fun n →
  p_char '.' >>>
  p_manyf p_digit mkFloat (0.0, 0.1) >>= fun (fv, _) →
  return (sign *. (float_of_int n +. fv))
```

Что интересно, этот парсер работает почти вдвое быстрее, чем стандартная окамловская функция `float_of_string`.

Данный набор комбинаторов позволяет описывать довольно сложные грамматики и одновременно действия по разбору. Например, я его успешно использовал для разбора сообщений от сервера в соревновании Sapka'09,[4] а также в утилите для раскраски исходного кода на OCaml.[5]

5.3. Классические комбинаторы и Packrat

У описанных выше классических парсер-комбинаторов есть одна проблема — экспоненциальная сложность по отношению к грамматике. В простых случаях этого эффекта не заметно, но когда грамматика становится достаточно сложной и ветвистой (например, при разборе языка программирования), то скорость падает очень сильно. Дело в том, что схема работы парсер-комбинаторов есть рекурсивный спуск с откатами. Допустим, мы разбираем некий язык программирования с арифметическими операциями над числами, переменными и элементами массивов. Тогда, чтобы разобрать выражение «42», парсер будет перебирать варианты: исходная строка-выражение — это или а1) сумма подвыражений, или б1) разность подвыражений, или в1) просто подвыражение, где каждое подвыражение есть или а2) произведение атомов, или б2) отношение атомов, или в2) деление с остатком атомов, или г2) просто атом, где каждый атом есть или а3) число, или б3) переменная, или в3) элемент массива, где индекс — тоже выражение. Парсер сначала будет пытаться разобрать выражение как сумму, но не найдя знака '+', начнет пытаться разобрать выражение как разность, и не найдя знака '–', будет разбирать выражение как просто подвыражение. При этом подвыражение будет разобрано трижды, каждый раз по очереди пытаясь интерпретировать его как произведение, как отношение и т. д. Количество перебираемых подвариантов перемножается, а если наше исходное выражение — часть более сложного,

[4] http://about.thedeemon.com/texts/sapka09/
[5] http://ocolor.thedeemon.com/

то оно тоже будет разобрано много раз, соответственно числу вариантов ветвей грамматики, включающей выражение, т. е. умножаем еще, отсюда и экспонента.

Основная проблема тут — что одна и та же работа делается многократно, при каждом откате результат разбора теряется. Логичным решением этой проблемы является мемоизация результатов. Техника подобного разбора с мемоизацией всех результатов носит имя Packrat[6], что дословно означает «земляная крыса», но также имеет переносный смысл «человек, привыкший ничего не выбрасывать и хранить все подряд, даже ненужное». На практике нет необходимости мемоизировать абсолютно все — достаточно мемоизации нескольких самых ветвистых (с большим количеством альтернатив) правил грамматики. Когда я применял парсер-комбинаторы для разбора собственного языка программирования, мемоизации четырех правил оказалось достаточно, чтобы от «очень медленно» перейти к «мгновенно».

Для мемоизации результатов парсинга отлично подходит механизм ленивых вычислений, доступный в OCaml. Допустим, мы разбираем вышеприведенный пример с выражениями, состоящими из переменных, чисел и арифметических операций. Правило грамматики для подвыражения выглядит так:

```
subexp =
  | atom '*' atom
  | atom '/' atom
  | atom '%' atom
  | atom
```

Ему соответствует парсер вида

```
let p_subexp =
  (p_atom >>= fun a1 → p_char '*' >>>
   p_atom >>= fun a2 → return Mul(a1, a2))
  ||| (p_atom >>= fun a1 → p_char '/' >>>
       p_atom >>= fun a2 → return Div(a1, a2))
  ||| (p_atom >>= fun a1 → p_char '%' >>>
       p_atom >>= fun a2 → return Mod(a1, a2))
  ||| p_atom
```

который возвращает результат типа `expr`. Чтобы не делать кучу работы по многу раз, мы мемоизируем результаты работы парсеров `p_subexp` и `p_atom`. Для этого вместо списка символов наши парсеры будут работать со списком значений, содержащих не только исходный символ, но и ленивые вызовы функций разбора, начиная с этого символа:

```
type mem_char = {
  ch : char;
  subexp :  (expr, mem_char) parse_result Lazy.t;
  atom :  (expr, mem_char) parse_result Lazy.t
}
```

[6] http://pdos.csail.mit.edu/~baford/packrat/thesis/

```
let prepare char_list =
  List.fold_right (fun ch res →
    let rec v = { ch=ch;
      subexp = lazy(p_subexp_r vlst); atom = lazy(p_atom_r vlst) }
    and vlst = v::res in
    vlst) char_list []

let p_subexp = function [] → Failed | h::t → Lazy.force h.subexp
let p_atom   = function [] → Failed | h::t → Lazy.force h.atom
```

Заменим `p_subexp` и `p_atom` на форсирование ленивых вычислений, а исходные функции переименуем в `p_subexp_r` и `p_atom_r`. Функция `prepare` берет исходный список символов и превращает его в список `mem_char`'ов, содержащих эти символы и пока еще не вызванные вызовы разбора подвыражений и атомов. Текст, который мы хотим разобрать, мы должны сперва превратить в список символов, который функцией `prepare` превратить в список `mem_char`'ов, который уже подавать на вход парсер-комбинаторам. Когда измененный таким образом наш парсер языка программирования работает, он вызывает функции для разбора атомов, подвыражений и пр., которые вызывают сохраненные в нашем списке ленивые функции разбора, а они работают ровно один раз — при первом обращении. Все следующие обращения к ним возвращают сохраненный результат. В итоге, если надо разобрать выражение «`42`», то когда парсер будет пытаться разобрать его как сумму, начинающуюся с подвыражения, и разберет подвыражение, он запомнит результат и в мыслях «а уж не разность ли это или, может, просто подвыражение» будет использовать этот результат вместо повторения проверок.

У такого подхода есть еще один большой плюс. Если в какой-то момент, расширяя грамматику языка, мы случайно создадим левую рекурсию (например, выражение состоит из произведений, те из чего-то, что состоит из простых выражений, одно из них может быть оператором, а один из операторов может быть выражением), то обычные парсер-комбинаторы на таком просто виснут — зацикливаются. А при мемоизации через ленивые вызовы получится, что при форсировании ленивого значения идет обращение к нему самому. Механизм реализации ленивых вычислений в языке OCaml такую ситуацию отлавливает и бросает исключение `Lazy.Undefined`, источник которого указывается в back trace. В результате можно сразу увидеть, где есть такой цикл, и исправить его, убрав левую рекурсию из грамматики.

Очевидным минусом явной мемоизации является необходимость для каждого мемоизируемого парсера менять тип `mem_char` и функцию `prepare`. Впрочем, изменения эти однотипные и много времени не отнимают.

Альтернативным подходом к решению проблемы многократного разбора является левая факторизация, когда грамматика преобразуется к виду, где у правил разные альтернативы начинаются по-разному, что минимизирует необходимость в откатах. Например, данный фрагмент грамматики можно преобразовать к виду

```
op = '*' atom
```

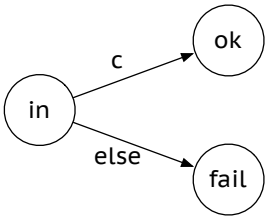
```
      | '%' atom
      | '/' atom
      | empty

subexp = atom op
```

Тогда каждый атом будет разобран один раз. Заметим, что когда парсеры строятся по естественному виду грамматики, как в первом примере, код построения разобранного значения получается максимально простым и наглядным: для выражения `atom * atom` строится значение `Mul(a1, a2)`, где составляющие `a1` и `a2` получаются и используются локально. Но при использовании левой факторизации построение разобранного значения сильно усложняется, т. к. теряется эта локальность. Придется из парсера `op` возвращать значение специального типа с результатом его разбора (пара из операции и значения второго атома либо признак их отсутствия), а в парсере `subexp` использовать сопоставление с образцом для выбора нужного варианта и по-разному строить результат. Это сильно усложняет и загромождает код, поэтому левая факторизация для ускорения парсер-комбинаторов — не лучшее решение.

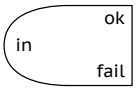
5.4. Оптимизирующие комбинаторы

Процесс работы классических парсер-комбинаторов состоит из конструирования замыканий, вызовов функций и выделений памяти под промежуточные результаты, что создает нагрузку на сборщик мусора и вообще не способствует хорошей скорости. Кроме того, исходные данные должны быть представлены в виде списка символов. В отличие от Haskell, где строки так и определены, в OCaml строки хранятся в виде непрерывных массивов символов, а конвертирование строки в список может занимать вдвое больше времени, чем сам парсинг. Можно заменить список парой «строка – позиция», но это не дает существенного выигрыша в скорости работы (конкретные цифры замеров приведены в таблице 5.1).

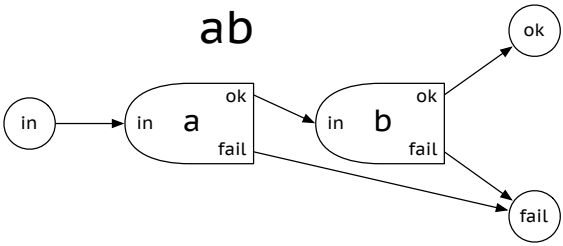
Поэтому можно подойти к реализации парсер-комбинаторов с другой стороны. Если мы посмотрим на самый простой из них — парсер, разбирающий один символ, то увидим, что его можно описать простым конечным автоматом (КА) с таким графом состояний:



Сначала он находится в состоянии `in`, и если символ подходящий, то переходит в состояние `ok`, возвращая успешный результат и остаток списка, а если символ не подходит, то автомат переходит в состояние `fail`, не изменяя переданного ему списка. И вообще, любой парсер имеет такой общий вид: у него есть одно входное состояние и два выходных — успех и неудача.

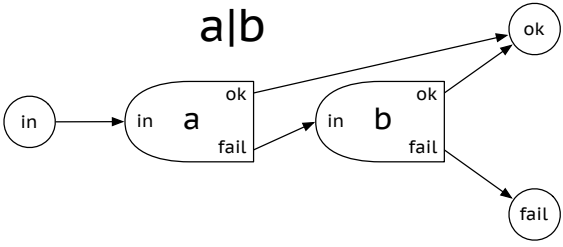


Графы состояний автоматов парсер-комбинаторов можно получить, соединяя разным образом графы автоматов их аргументов. Результат тоже всегда имеет одно входное состояние и два выходных, поэтому подходит для дальнейшего манипулирования другими комбинаторами. Вот так, например, выглядит автомат для последовательности двух парсеров (мы его описывали оператором `>>>`):

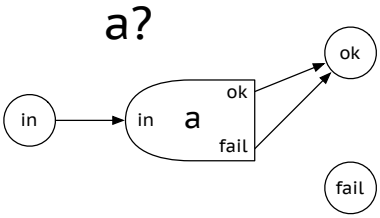


В точном соответствии с логикой оператора последовательности, он возвращает успех (приходит в состояние `ok`), если успешно по очереди отработали оба парсера, из которых строится последовательность. Если же какой-то из них не смог разобрать свою часть, то возвращается неудача (происходит переход в состояние `fail`).

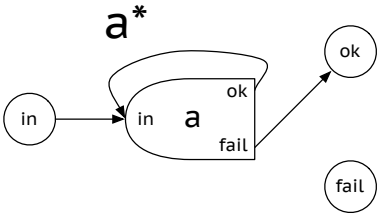
Аналогично описываются комбинаторы альтернативы,



опциональности,



и повторения.



Комбинируя таким образом автоматы, можно получить те же парсеры, что мы могли построить с помощью классических парсер-комбинаторов, кроме некоторых рекурсивных.

Подобно тому, как в классических парсер-комбинаторах мы использовали несколько операторов и функций для построения сложных парсеров из простых, здесь мы опишем набор аналогичных операторов и функций, которые будут строить граф конечного автомата из других более простых графов. Затем построенный граф может быть упрощен и превращен в таблицу, по которой функция-парсер уже сможет работать без лишних накладных расходов. Единственное затруднение — как описать семантические действия по созданию разобранных значений. В классическом подходе использовались замыкания, и разобранные ранее значения можно было использовать позже для синтеза более сложных. Однако результатом комбинаторов была монолитная функция, которую преобразовать и упростить уже нельзя. Возможно, у проблемы есть элегантное решение, но я пока поступил просто: семантические действия передают друг другу данные через изменяемые переменные, доступные им всем.

Граф можно описать набором вершин, каждая из которых может иметь несколько переходов из себя в другие вершины. В графе КА парсера есть три вида переходов: а) встречен подходящий символ, б) ветка `else` и в) безусловный переход. Каждый переход может сопровождаться каким-нибудь действием. Действия на переходах по символу получают код этого символа в качестве аргумента, другие действия ничего не получают и работают исключительно с описанным снаружи состоянием.[7]

[7]Можно было бы избавиться от глобальных переменных, если снабдить функцией-действием каждый

Переход по узнанному символу меняет текущую позицию в разбираемом тексте. Если какой-то парсер не смог разобрать свою часть, то, как и в классическом варианте, текущая позиция должна остаться той же, что была до начала его работы. Для этого мы должны уметь ее запоминать и при необходимости восстанавливать. Так как сложные парсеры обычно состоят из вложенных друг в друга более простых, нам потребуется стек для хранения позиций. При входе в сложный парсер текущая позиция будет запоминаться в стеке, при неудачном исходе она будет из него извлекаться и использоваться для отката, а при удачном исходе будет выкидываться из стека как более не нужная.

Определим возможные операции со стеком позиций:

```
type pos_oper = Push | Pop | Rollback | No
```

`Push` — запомнить текущую позицию на стеке, `Pop` — убрать позицию с вершины стека, `Rollback` — взять со стека позицию и сделать ее текущей (откатиться назад), `No` — ничего не делать.

Тогда тип переходов можно описать так:

```
type move_kind =
  | Symb of char * char * (int → unit) option * int * pos_oper
  | Else of (unit → unit) option * int * pos_oper
  | Goto of (unit → unit) option * int * pos_oper
```

Тут используются пары значений типа `char`, чтобы сразу задавать не один символ, а набор символов от одного до другого, например 'a' – 'z' или '0' – '9'. Параметр типа `int` у перехода — номер вершины в графе, куда осуществляется переход. `Else` и `Goto` отличаются лишь тем, что если из вершины есть безусловный переход `Goto`, то это единственный переход из этой вершины. Этот факт используется дальше при упрощении графа.

Храня наборы переходов и вершин в массивах, элементарные графы, разбирающие один символ, можно описать так:

```
let ok = 1
let fail = 2

let p_range c1 c2 =
  [| [| Symb(c1,c2, None, ok, No);
        Else(None, fail, No) |];
     [||]; [||] |]
let p_char c = p_range c c

let f_range c1 c2 f =
```

переход в графе, чтобы эти функции принимали на вход состояние и возвращали измененное состояние. Но это негативно отразится на скорости, т. к. во-первых, вызовом функции будет сопровождаться каждый переход в графе, а не только необходимые, а во-вторых, будет сложно заниматься упрощением графа, потому что цепочку переходов без действий можно сократить до одного перехода, выкинув лишние, а с переходами с действиями так сделать нельзя.

```
    [| [| Symb(c1,c2, Some f, ok, No);
          Else(None, fail, No) |];
       [||]; [||] |]
let f_char c f = f_range c c f
```

`p_char c` успешно разбирает символ `c` и возвращает неудачу, если на входе любой другой символ. `p_range` описывает допустимый интервал символов, например цифры или буквы. Их аналоги, начинающиеся с `f_`, делают все то же, но при этом содержат некие семантические действия, которые будут выполнены при успешном разборе символа. Эти графы имеют три вершины — `in`, `ok` и `fail` — и два перехода из нулевой (`in`) в другие. Поскольку все КА парсеров имеют эти три состояния, можно условиться, что первые три вершины любого графа у нас будут обозначать `in`, `ok` и `fail`. Каждый синтезируемый из других граф будет иметь эту внешнюю рамку из таких трех состояний, причем из нового состояния `in` будет безусловный переход на начальное состояние первого подграфа. Описанная ниже функция `frame` создает такую рамку. Функция `relocate` делает из заданного графа подграф, сдвигая номера упоминаемых в нем вершин. Функция `connect` создает подграф и соединяет его выходные состояния `ok` и `fail` с заданными состояниями нового графа.

```
let frame in_move = [| [| in_move |]; [||]; [||] |]

let relocate parser delta =
  Array.map (Array.map (function
    | Symb(c1, c2, f, idx, op) → Symb(c1, c2, f, idx + delta, op)
    | Else(f, idx, op) → Else(f, idx + delta, op)
    | Goto(f, idx, op) → Goto(f, idx + delta, op))) parser

let connect parser delta ok_move fail_move =
  let res = relocate parser delta in
  res.(ok)   <- [| ok_move |];
  res.(fail) <- [| fail_move |];
  res
```

Используя эти вспомогательные функции, опишем построение графа для последовательности двух парсеров.

```
let (>>>) a b =
  let b_start = 3 + Array.length a in
  Array.concat [frame (Goto(None, 3, Push));
    connect (norb a) 3
      (Goto(None, b_start, No)) (Goto(None, fail, Rollback));
    connect (norb b) b_start
      (Goto(None, ok, Pop)) (Goto(None, fail, Rollback))]
```

В соответствии с приведенной выше схемой, из начального состояния в нем делается переход на начало первого парсера, при этом текущая позиция кладется на стек. Если первый парсер отработал успешно, то идет переход на начало второго, иначе переход в состояние `fail` с откатом позиции. Если второй парсер отработал успешно,

то переходим в состояние ok, убирая с верхушки стека сохраненную позицию. Если же второй парсер выдал неудачу, то точно так же переходим в fail, восстанавливая сохраненную позицию. Поскольку сохранением и восстановлением текущей позиции занимается верхний «слой» графа (переходы из начального состояния и переходы в конечные), то подграфы могут сами этим не заниматься. Для этого они перед добавлением в граф пропускаются через вспомогательную функцию norb, которая убирает из них соответствующие действия (если они там были). Эта необязательная оптимизация, код которой здесь не приводится, позволяет заметно сократить число переходов в графе.

Аналогично строятся и остальные комбинаторы — опциональности, альтернативы, повторения и т. д. Их аналоги с префиксом f_ запоминают в графе семантические действия.

```
let p_opt a =
  Array.append (frame (Goto(None, 3, No)))
    (connect a 3 (Goto(None, ok, No)) (Goto(None, ok, No)))

let f_opt a f =
  Array.append (frame (Goto(None, 3, No)))
    (connect a 3 (Goto(None, ok, No)) (Goto(Some f, ok, No)))

let (|||) a b =
  let b_start = 3 + Array.length a in
  Array.concat [frame (Goto(None, 3, No));
    connect a 3 (Goto(None, ok, No)) (Goto(None, b_start, No));
    connect b (b_start) (Goto(None, ok, No)) (Goto(None, fail,
        No))]

let p_many a =
  Array.append (frame (Goto(None, 3, No)))
    (connect a 3 (Goto(None, 0, No)) (Goto(None, ok, No)))

let f_many initf a =
  Array.concat [frame (Goto(Some initf, 3, No));
    connect a 3 (Goto(None, 3, No)) (Goto(None, ok, No))]

let p_plus p = p >>> p_many p

let (>>=) p f =
  Array.append (frame (Goto(None, 3, No)))
    (connect p 3 (Goto(Some f, ok, No)) (Goto(None, fail, No)))
```

Имея такие комбинаторы, можно строить графы конечных автоматов для сложных парсеров.

В результате получается довольно развесистый граф. Однако, его можно сильно упростить, убрав все состояния, содержащие только безусловный переход в другое.

```
let rec simplify p =
 let skip_first e = Enum.junk e; e in
 Array.enum p |> skip_first |> Enum.mapi (fun srcn node →
  Array.enum node |> Enum.filter_map
   (function Goto(None, dstn, No)→ Some(srcn+1, dstn)| _→None))
  |> Enum.concat |> Enum.get |> Option.map_default
   (fun (srcn, dstn) →
    let dstn' = if dstn < srcn then dstn else dstn-1 in
    let adjust_idx idx =
     if idx < srcn then idx else
     if idx=srcn then dstn' else idx-1 in
    let adjust_move = function
     | Symb(c1, c2, f, idx, op) → Symb(c1, c2, f, adjust_idx
         idx, op)
     | Else(f, idx, op) → Else(f, adjust_idx idx, op)
     | Goto(f, idx, op) → Goto(f, adjust_idx idx, op) in
    let p' = Array.enum p |> Enum.mapi (fun i node→ (i,node))
     |> Enum.filter_map (fun (i,node) →
      if i = srcn then None else Some(Array.map adjust_move node))
     |> Array.of_enum in
    simplify p') p
```

Тут используются ленивые последовательности из модуля Enum, входящего в библиотеки ExtLib и Batteries, дополняющие стандартную библиотеку OCaml.

Дальше граф можно превратить в таблицу, где в каждом состоянии для каждого возможного входного символа будет указано действие — принять символ (перейти дальше по строке) или не трогать, выполнить какое-то семантическое действие или не выполнять — номер состояния, в которое следует перейти из данного, и сопутствующая операция со стеком позиций.

```
type move_action =
  | Consume | Keep
  | ConsumeA of (int → unit) | KeepA of (unit → unit)

let tabulate p =
  p |> Array.map (fun node→
    let tmp = Array.make 257 (Keep, -1, No) in
    let default_move = Array.fold_left (fun def kind →
      match kind with
      | Symb(c1, c2, fo, idx, op) →
          let action =
            Option.map_default (fun f → ConsumeA f) Consume fo in
          let m = (action, idx, op) in
          for i = int_of_char c1 to int_of_char c2 do
            tmp.(i)<-m
          done;
          def
```

```
      | Else(fo, idx, op)
      | Goto(fo, idx, op) →
          Option.map_default (fun f → KeepA f) Keep fo, idx, op
      ) (Keep, fail, No) node  in
    tmp |> Array.map (fun ((k, idx, op) as x) →
      if idx >=0 then x else default_move))
```

Эту таблицу можно еще оптимизировать. Если при встрече какого-то символа не совершается никаких действий, и просто идет переход в другое состояние, можно проследить по таблице, какое же действие в конце концов будет выполнено для этого символа и это действие поставить сразу в рассматриваемую ячейку, таким образом сэкономив на переходах. Помимо этого, некоторые действия можно объединить. В итоге ни одна смена состояния не будет совершаться вхолостую, все они будут сопровождаться теми или иными действиями.

```
let optimize tbl =
  let rec optimove ch move =
    match move with
    | _, 1, _
    | _, 2, _ → move
    | Keep, next_state, No → optimove ch tbl.(next_state).(ch)
    | Keep, next_state, (Push as op)
    | Keep, next_state, (Pop as op) →
        (match optimove ch tbl.(next_state).(ch) with
        | Consume, nxt2, No → Consume, nxt2, op
        | _ → move)
    | KeepA f, next_state, No →
        (match optimove ch tbl.(next_state).(ch) with
        | Keep, nxt2, op → KeepA f, nxt2, op
        | KeepA f2, nxt2, op → KeepA (f >> f2), nxt2, op
        | _ → move)
    | _ → move in
  Array.mapi (fun state tab →
    if state!=1 && state!=2 then Array.mapi optimove tab else
      tab) tbl
```

Оптимизация таблицы и упрощение графа — очень похожие операции, и если не делать упрощения графа, а только оптимизацию таблицы, то количество совершаемых при работе парсера переходов будет тем же, однако количество состояний в таблице будет значительно больше, отчего ей будет сложнее поместиться в кэш процессора.

Имея такую оптимизированную таблицу, можно уже получить готовый быстрый парсер, который будет действовать по этой таблице и в конце концов выполнять либо действие, соответствующее успеху всего парсера, либо соответствующее неудаче. Конец строки представим особым символом с номером 256, чтобы реакция на него тоже была в таблице действий.

```
let execute success fail tbl str =
```

```
  let len = String.length str in
  let stack = Array.make (Array.length tbl) 2 in
  let rec run state i sp =
   match state with
   | 1 → success ()
   | 2 → fail ()
   | _ →
    let ch = if i<len then int_of_char str.[i] else 256 in
    match tbl.(state).(ch) with
    | Consume, next_state, No → run next_state (i+1) sp
    | Consume, next_state, Push →
       stack.(sp)<-i; run next_state (i+1) (sp+1)
    | Consume, next_state, Pop → run next_state (i+1) (sp-1)
    | Keep, next_state, Push → stack.(sp)<-i; run next_state i
        (sp+1)
    | Keep, next_state, Pop → run next_state i (sp-1)
    | Keep, next_state, Rollback → run next_state stack.(sp-1)
        (sp-1)
    | ConsumeA f, next_state, No → f ch; run next_state (i+1) sp
    | KeepA f, next_state, No → f (); run next_state i sp
    | KeepA f, next_state, Push →
       f (); stack.(sp)<-i; run next_state i (sp+1)
    | KeepA f, next_state, Pop → f (); run next_state i (sp-1)
    | KeepA f, next_state, Rollback →
       f (); run next_state stack.(sp-1) (sp-1)
    | _ → failwith "unexpected move" in
  run 0 0 0

let prepare success fail parser =
  parser |> simplify |> tabulate |> optimize |> execute success
      fail
```

Функция `execute` содержит «сердце» быстрого парсера — цикл с тремя переменными: номером текущего состояния, индексом текущей позиции в разбираемой строке и индексом верхушки стека позиций. Напомним, что состояния с номерами 1 и 2 — это конечные состояния успеха и неудачи. Попадая туда, функция выполняет соответствующее им действие и завершает работу. В остальных же состояниях, подобно машине Тьюринга, она берет код символа в текущей позиции и по таблице, индексируемой номером состояния и кодом символа, определяет следующее действие и состояние.

Функция `prepare` берет исходный граф, построенный нашими комбинаторами, упрощает его, превращает в таблицу, оптимизирует ее и создает функцию-парсер, принимающую на вход строку и в зависимости от результата разбора выполняющую одно из заданных действий. Построив с помощью описанных выше комбинаторов граф парсера, достаточно передать его функции `prepare`, сопроводив двумя действиями — что делать в случае успеха и в случае неудачи, чтобы получить функцию,

принимающую строку и разбирающую ее. Тип результата такой функции определяется типом параметров-действий `success` и `fail`.

5.5. Сравнение с другими методами парсинга

Чтобы оценить скорость работы описанных методов и сравнить ее с альтернативными подходами, мы применили различные способы парсинга на одной задаче, не очень сложной, но достаточно практической. Суть задачи в том, чтобы прочитать файл карты в формате OpenStreetMap и определить ее реальные границы — минимальные и максимальные значения координат имеющихся на карте точек. Формат карты включает в себя поле `bounds` с похожими по смыслу значениями, но обычно это границы карты «с запасом», так что точки карты этих границ не касаются. Формат карты основан на XML, но в данной задаче полностью разбирать все детали XML не нужно, достаточно уметь извлечь нужную информацию из интересующих нас тэгов и полей.

Решение задачи с использованием оптимизирующих парсер-комбинаторов умещается в 50 строк:

```
let minlat = ref 1000.0
let maxlat = ref (-1000.0)
let minlon = ref 1000.0
let maxlon = ref (-1000.0)

let p_latlon name =
  p_str (name ^ "=\"") >>> p_float >>> p_char '"'

let low_letter c = c >= 'a' && c <='z'

let p_param =
  p_plus (p_pred low_letter) >>> p_str "=\""
  >>>  p_many (p_pred ((<>)'"')) >>> p_char '"'

let p_node_param =
     (p_latlon "lat" >>= fun () → let lat =
         !Fsm_parsing.float_res in
        minlat := min !minlat lat; maxlat := max !maxlat lat)
 ||| (p_latlon "lon" >>= fun () → let lon =
     !Fsm_parsing.float_res in
         minlon := min !minlon lon; maxlon := max !maxlon lon)
 ||| p_param

let p_ws = p_many (p_pred ((>=) ' '))

let p_endnode =
  (p_str "/>") ||| (p_all_until "</node>")
```

```
let p_node =
  p_str "<node" >>>
  p_many (p_ws >>> p_node_param) >>> p_endnode

let p_tag =
  p_char '<' >>> p_many (p_pred ((<>) '>')) >>> p_char '>'

let p_osm =
  p_many ((p_node ||| p_tag) >>> p_ws)

let process osm =
 let ok_action () =
  Printf.printf "ok: lat=%f..%f lon=%f..%f\n"
    !minlat !maxlat !minlon !maxlon in
 let parse = prepare ok_action (fun () → print_string "fail")
     p_osm in
 parse osm
```

Смысл его в том, что карта формата OSM представляется последовательностью тэгов, между которыми могут быть разделители, вроде пробелов, символов конца строки и т. д. Тэг может быть либо тэгом `node`, содержащим информацию о точке, либо каким-то другим тэгом. Все тэги, отличные от `node`, нас не интересуют, поэтому их представим как произвольную последовательность символов, заключенную в угловые скобки. Что касается тэга `node`, то он состоит из заголовка (строки «`<node>`»), последовательности параметров тэга, и окончания, состоящего либо из строки «`/>`» (закрытия тэга), либо из вложенных тэгов, за которыми следует строка «`</node>`». Координаты точек содержатся среди параметров тэга `node`. Это параметры `lat` для широты и `lon` для долготы. Они состоят из имени параметра, знака «=» и вещественного числа в кавычках, которое мы разбираем и тут же используем для обновления переменных, содержащих текущие минимальные и максимальные значения. Другие же параметры тэгов мы игнорируем, описывая их как произвольный набор букв, за которым идет знак «=» и некая строка в кавычках. Такого простого описания грамматики достаточно, чтобы решить исходную задачу.

Ее решение на классических парсер-комбинаторах выглядит почти идентично. Кроме них, были сделаны решения на базе генератора парсеров ocamlyacc, библиотеки парсер-комбинаторов на Хаскелле Parsec 2 (GHC 6.12.1, Parsec 2.1.0.1), еще одной библиотеки парсер-комбинаторов attoparsec 0.8.0.2, на C++ с использованием библиотеки парсер-комбинаторов Boost.Spirit (версия 2.2 из Boost 1.42, компилировалось в VS2005), на Lua 5.1.4 с использованием библиотеки LPEG, и на языке C# 2.0 с использованием стандартного средства для работы с XML — библиотеки System.Xml. Код на OCaml компилировался версией компилятора 3.10.2 в варианте, использующем ассемблер и линкер из MSVC.

Все решения были протестированы на карте Сингапура[8] на ноутбуке с процессором Intel Core 2 Duo 2.33 MHz и ОС Windows Vista. Там, где это было возможно, скорость чтения файла не входила в замер, учитывалась только скорость разбора уже прочитанного в память файла.

Haskell / Parsec 2	3.5 МБ/с
Haskell / attoparsec	8.2 МБ/с
Классические парсер-комбинаторы на OCaml, если включать время на преобразование из строки в список символов.	6.6 МБ/с
Они же, если считать только парсинг списка.	18.9 МБ/с
Эти же комбинаторы были реализованы в варианте, где вместо списка использовалась пара строка-позиция. Такой вариант не требовал перевода из строки в список.	8.3 МБ/с
Парсер, сгенерированный парой ocamlyacc и ocamllex.	12.9 МБ/с
Вариант на C++ с использованием библиотеки парсер-комбинаторов Boost Spirit 2.	18.1 МБ/с
Lua / LPEG (на языке Lua описывается грамматика, а разбор по ней осуществляет модуль LPEG на Си).	20.2 МБ/с
Оптимизирующие парсер-комбинаторы на OCaml, включая время на построение графа, его оптимизацию, построение таблицы и т. д.	42.6 МБ/с
Наконец, вариант C# с System.Xml.	46.0 МБ/с

Таблица 5.1. Скорость полученных решений

Исходные тексты всех вариантов доступны на официальном сайте журнала.

Можно видеть, что применение предлагаемого подхода к построению оптимизирующих парсер-комбинаторов позволяет сохранить преимущества классических парсер-комбинаторов (выразительность, отсутствие отдельной фазы кодогенерации) и получить в несколько раз большую скорость работы.

5.6. Заключение

Описанные в этой статье парсер-комбинаторы очень просты и в таком виде не могут сравниться по возможностям с такими развитыми библиотеками, как Parsec и Spirit. Зато они легки в понимании, реализации, использовании и исполнении. Во

[8] http://downloads.cloudmade.com/asia/singapore/singapore.osm.bz2

многих задачах их возможностей оказывается вполне достаточно, поэтому они могут служить практичным компромиссом по гибкости и скорости.

Модель типизации Хиндли — Милнера и пример её реализации на языке Haskell

Роман Душкин
darkus@fprog.ru

Аннотация

Статья описывает алгоритм Хиндли — Милнера, используемый для автоматического вывода типов выражений. Рассматриваются дополнения данного алгоритма, используемые в функциональном языке программирования Haskell в связи с наличием в этом языке ограниченного полиморфизма. Приводится пример реализации функции для автоматического вывода типов, реализованной на языке Haskell, для чего, в том числе, даются полезные примеры применения библиотеки синтаксического анализа.

The article describes the Hindley–Milner type inference algorithm. Extended version capable of handling bounded polymorphism is shown. Implementation of algorithm in Haskell is provided, complete with a small parser for lambda-calculus and a top-level for exploration of the type inference.

Обсуждение статьи ведётся в LiveJournal.

6.1. Введение

Настоящая статья продолжает цикл публикаций, посвящённый типизации в функциональном программировании и в языке программирования Haskell в частности. Предыдущие статьи данного цикла были опубликованы в номерах 2–4 журнала [10, 12, 11].

В перечисленных статьях упоминалось, что для некоторых видов полиморфизма проблема вывода типов неразрешима. В настоящей статье рассматривается механизм автоматического вывода типов, основанный на алгоритме Хиндли — Милнера (иногда в специальной литературе называемый алгоритмом Дамаса — Милнера или алгоритмом Хиндли — Милнера — Дамаса), который позволяет вывести тип в рамках большого числа формальных систем типов, в том числе и для полиморфных типов ранга 2 (определение и о способах применения параметрического полиморфизма высших рангов см. [12, 11]).

Если разработчик не указал явно тип выражений, используемых им в исходных кодах, алгоритм Хиндли — Милнера позволяет получать *наиболее общие типы* этих выражений. Наиболее общим типом называется такой тип заданного выражения, к которому можно привести любой другой тип, который может быть приписан этому выражению. Настоящая статья как раз описывает один из способов получения наиболее общего типа, а также процедуру приведения к нему — *унификацию*. В следующем разделе будет дано формальное определение наиболее общего типа.

Особенностью и положительным эффектом от использования алгоритма является то, что он позволяет разработчику программных средств не задумываться о типизации выражений в статически типизированных языках программирования — механизм вывода типов автоматически построит все недостающие определения, а компилятор языка проверит их на корректность.

Суть системы типизации Хиндли — Милнера заключается в том, что на уровне синтаксиса программы можно провести формальные преобразования таким образом, чтобы автоматически определить типы выражений, используемых в этой программе. Это избавляет разрабочика от необходимости явно указывать типы выражений, что в свою очередь влечёт за собой лаконичность и высокую степень читаемости кода программ. В таком языке программирования, как Haskell, декларации типов выражений обычно вносятся в исходные коды только для выражений самого верхнего уровня в целях документирования кода.

Алгоритм автоматического вывода типов основан на работах Хаскеля Карри и Роберта Фейса по типизированному λ-исчислению и был впервые представлен Роджером Хиндли в 1969 году. Последний доказал, что его алгоритм выводит наиболее общий тип выражения. Независимо от работы Хиндли в 1978 году Робин Милнер представил свой вариант алгоритма, который он назвал «алгоритм W» [6]. Наконец, в 1982 году Луис Дамас доказал, что алгоритм Милнера полон и позволяет выводить типы в полиморфных системах [2]. Алгоритмы, разработанные Р. Милнером и Р. Хиндли в свою очередь опираются на принцип резолюции, который был впервые предложен

Дж. А. Робинзоном [8].

Независимо от этих исследователей алгоритм унификации и типизации были предложены Дж. Моррисом (1968 год), К. Мередитом (в 1950-х годах) и, предположительно, А. Тарским (вообще в 1920-х годах). По этому поводу Роджер Хиндли заметил: «Вероятно, кому-то не помешало бы научиться читать, или кому-то другому — писать» [15].

В алгоритме используются следующие обозначения и понятия:

- *Типами* τ являются базовые типы из некоторого зафиксированного множества B, типовые переменные с идентификаторами из другого множества V, а также типы, полученные при помощи применения конструктора типов $\boxed{\rightarrow}$:

$$\tau ::= T \mid v \mid \tau \rightarrow \tau$$

  здесь $T \in B$ — типы из зафиксированного множества базовых типов, $v \in V$ — типовые переменные, идентификаторы которых берутся из некоторого множества V.

  Базовые типы представляют собой обычные для разработчиков программного обеспечения конструкции — такие типы как `Int`, `Bool` и т. д. Теория λ-исчисления требует только наличия некоторого зафиксированного множества B, а состав этого множества не регламентируется.

  В качестве идентификаторов типовых переменных обычно используются строчные буквы греческого алфавита с его начала: α, β, γ и т. д. В качестве соглашения, принимаемого для упрощения дальнейшего изложения, будет считаться, что строчные греческие буквы с начала алфавита — это типовые переменные. Буквы σ и τ (возможно, с индексами) — это обозначение типов вообще.

  Конструктор типов $\rightarrow$ является правоассоциативным, так что можно опускать скобки следующим образом: $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \equiv \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. Тип может быть представлен в виде двоичного дерева, в узлах которого находятся конструкторы типов, а в листьевых вершинах — базовые типы.

- *Выражениями* e являются константы из некоторого зафиксированного множества C, переменные, абстракции и аппликации (применения):

$$e ::= c \mid x \mid \lambda x : \tau . e \mid (e\ e)$$

  здесь $c \in C$ — константы, имеющие базовые типы. Константами, являются конкретные значения из базовых типов. Например, `True`, `False` $\in$ `Bool`, `0`, `1` $\in$ `Int` и т. д.

  Запись $\lambda x : \tau . e$ обозначает типизированную абстракцию — связанная переменная x имеет тип τ.

- *Контексты типизации* обозначаются полужирными заглавными буквами греческого алфавита: $\mathbf{\Gamma}$, $\mathbf{\Delta}$ и т. д. и представляют собой множества *предположений о типизации* вида $x : \tau$, то есть предположений, что «переменная x имеет тип τ»:

$$\mathbf{\Gamma} = \{x_i : \tau_i\}_{i=1}^{n}$$

- *Отношение типизации*, обозначаемое как $\mathbf{\Gamma} \vdash e : \tau$, обозначает, что выражение e имеет тип τ в контексте $\mathbf{\Gamma}$ и, таким образом, корректно типизировано. Конкретизации отношений типизации называются *суждениями о типизации*.

- Корректность суждений о типизации должна быть подтверждена при помощи *правил вывода типов*, которых имеется четыре:

  1) Константы корректно типизированы и имеют либо определённый базовый тип, либо тип, представляющий собой типовую переменную:

$$\frac{c : (T \in B) \in C}{\mathbf{\Gamma} \vdash c : T} \vee \frac{c : (v \in V) \in C}{\mathbf{\Gamma} \vdash c : v} \tag{6.1}$$

  Типовые переменные используются для обобщения типовых выражений. При непосредственном использовании выражения, в типе которого имеются типовые переменные, все они должны быть конкретизированны либо базовыми типами, либо «стрелками»,[1] в вершинах которых находятся базовые типы (речь, конечно, идёт о параметрическом полиморфизма первого ранга).

  2) Если переменная x имеет тип τ в некотором контексте, то x^{τ} есть отношение типизации в этом контексте и x корректно типизирована.

$$\frac{x : \tau \in \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash x : \tau} \tag{6.2}$$

  3) Если в определённом контексте $\mathbf{\Gamma}$ вместе с предположением $x : \tau$ корректно типизировано выражение e типа σ, то в том же самом контексте без предположений о типе переменной x можно корректно типизировать абстракцию $\lambda x : \tau.e$:

$$\frac{\mathbf{\Gamma} \cup \{x : \tau\} \vdash e : \sigma}{\mathbf{\Gamma} \vdash (\lambda x : \tau.e) : \tau \to \sigma} \tag{6.3}$$

[1] Здесь и далее термином «стрелка» будет пониматься функциональный тип вида $\sigma \to \tau$.

4) Если в некотором контексте выражение e_1 корректно типизировано типом $\tau \to \sigma$, а выражение e_2 корректно типизировано типом τ, то в этом же контексте можно корректно типизировать аппликацию $(e_1 e_2)$:

$$\frac{\mathbf{\Gamma} \vdash e_1 : \tau \to \sigma, e_2 : \tau}{\mathbf{\Gamma} \vdash (e_1 e_2) : \sigma} \tag{6.4}$$

Правила вывода представляют собой продукции, записанные в форме дробей. «Числитель» дроби представляет собой посылку правила (*антецедент*), «знаменатель», соответственно, — заключение правила (*консеквент*). Если антецедент правила истиннен, то по этому правилу можно заключить, что истиннен и консеквент.

Теперь всё готово для рассмотрения самого алгоритма. Далее в статье в разделе 1 рассматривается сам алгоритм Хиндли — Милнера в «классической постановке». В разделе 2 описывается адаптация этого алгоритма для языка программирования Haskell. В разделе 3 приводятся определения функций на языке Haskell, реализующие алгоритм. В качестве дополнительной литературы по проблеме автоматического вывода типов можно порекомендовать [14, 7].

6.2. Описание алгоритма Хиндли — Милнера

Алгоритм автоматического вывода типов строит систему уравнений, неизвестными в которой являются типы, после чего решает эту систему, находя неизвестные значения. Базовый вариант алгоритма достаточно прост, поэтому он неоднократно переоткрывался независимо друг от друга различными исследователями в области информатики и прикладной математики, как уже было показано во введении. Базовый вариант оперирует только теми сущностями, которые используются в простом типизированном λ-исчислении — хотя непосредственно в процессе вывода используются типовые переменные, в результате работы алгоритма для λ-термов получаются только базовые типы из известного множества.

6.2.1. Построение системы уравнений

Чтобы построить систему уравнений вывода типов (*ти*повых уравнений), необходимо последовательно применить два процесса, а именно:

1) На основании правил вывода типов породить систему предположений о типах для самого обрабатываемого λ-выражения и всех его подвыражений вплоть до переменных и констант. Правила вывода применяются без проверки предусловий, и это — особенность алгоритма Хиндли — Милнера.

2) Для предположений о типах всех подвыражений построить равенства вида $\tau_1 = \tau_2$ на основании того, что типы τ_1 и τ_2 приписаны одному и тому же

λ-терму. Весь набор таких равенств и представляет собой систему типовых уравнений.

Система предположений о типах λ-выражений строится посредством применения правил вывода, указанных во введении, к каждому элементу обрабатываемого λ-терма. По сути предположения о типах представляют собой выражения вида $e : \tau$, а сама система предположений — это ничто иное, как несколько расширенное понимание контекста $\mathbf{\Gamma}$. Важными отличиями от определённого ранее термина является то, что, во-первых, при построении системы предположений в неё могут включаться несколько предположений для одного и того же выражения, а во-вторых, в контекст входят лишь типизации переменных, в то время как предположения о типах строятся для каждого подвыражения.

Например, простейшая аппликация двух переменных друг к другу $(x\ y)$ породит систему предположений, состоящую из пяти элементов, а именно:

- $x : \alpha$ — применение правила вывода (6.2) для переменной x.
- $y : \beta$ — применение того же правила вывода уже для переменной y.
- $(x\ y) : \gamma$ — применение правила вывода (6.4) для аппликации переменных.
- $y : \delta$ — применение того же правила вывода для аппликации к переменной y.
- $x : \delta \to \gamma$ — расширение типа переменной x на основании правила вывода (6.4) в стрелку.

На основании этой системы предположений строится система типовых уравнений, состоящая из двух элементов, а именно:

- $\alpha = \delta \to \gamma$;
- $\beta = \delta$.

Поскольку рассматриваемый случай вырожден, полученная системая вывода типов тривиальна — она сама по себе является решением.

Если дополнить рассмотренную аппликацию абстракцией одной из переменных (к примеру, x), то система предположений о типах, а за ней и система типовых уравнений дополнятся. Соответственно, система предположений для λ-терма $\lambda x.(xy)$ будет дополнена следующими предположениями:

- $x : \varepsilon$;
- $(\lambda x.(xy)) : \varepsilon \to \eta$;
- $(xy) : \eta$.

ну а система типовых уравнений будет преобразована к следующему виду (здесь первое уравнение записано в таком виде, чтобы не записывать три уравнения попарного равенства):

- $\alpha = \delta \to \gamma = \varepsilon$;
- $\beta = \delta$;
- $\eta = \gamma$.

В процессе построения системы типовых уравнений на основе предположений о типах можно сразу проводить оптимизацию, не строя уравнения вида $\alpha \to \beta$, то есть уравнения, с каждой стороны которой стоит просто типовая переменная. Если для какого-то λ-терма в контексте $\mathbf{\Gamma}$ уже назначен тип в виде типовой переменной, то второе назначение типовой переменной не осуществляется. Если применить такой оптимизирующий подход, то три вышеприведённых уравнения «схлопнутся» в одно: $\alpha = \beta \to \gamma$.

6.2.2. Унификация

Для решения построенной системы уравнений применяется процедура унификации. Её суть заключается в сравнении типов и попытках сопоставить типы друг с другом.

Как указано во введении, тип представляет собой либо какой-то базовый тип из известного множества, либо типовую переменную, либо стрелку. Понятие типовой переменной вводится для упрощения процедуры вывода типов. В итоге есть три класса типовых сущностей: константа, переменная и стрелка, по обеим сторонам которой могут находиться эти же самые типовые сущности.

Для осуществления унификации необходимо понятие *подстановки типов*. Подстановкой типов называется отображение S из множества типов в него же, такое что вместо заданного типа τ подставляется некоторый другой тип σ. Базовые типы могут подставляться только сами в себя, а вместо типовой переменной можно подставить любой другой тип, который не включает в себя эту типовую переменную. Подстановка для стрелки осуществляется по схеме: $S(\sigma \to \tau) = S(\sigma) \to S(\tau)$.

Если $S(\sigma) = \tau$, то тип τ называется *примером* типа σ. Соответственно, если для некоторого λ-терма M имеет место, что $\vdash M : \tau$, и при этом любой другой тип терма M является примером типа τ, то тип τ называется наиболее общим типом (или *главным типом*).

Унификация производится на паре типов, таким образом имеется $3^2 = 9$ комбинаций для унификации. Также необходимо отметить, что операция унификации некоммутативна. Этот процесс односторонен — результат приведения первого типа ко второму в общем случае не равен результату приведения второго типа к первому. Пример: константу невозможно унифицировать переменной, а переменная унифицируется константой, принимая в качестве конкретного значения её саму.

Из-за чего возникает односторонность операции унификации? Дело в том, что в процессе вывода типа некоторого λ-выражения алгоритм должен пройтись по всем его подвыражениям и занести в контекст предположения о типах этих подвыражений. Подвыражения могут повторяться, а потому новый тип, выводимый для повторного проявления подвыражения должен унифицироваться с тем типом, который уже имеется в контексте.

В таблице 6.1 представлены способы, при помощи которых типы унифицируются друг с другом. Строки соответствуют вариантам типов для исследуемого подвыражения, которые уже находятся в контексте (эти варианты указаны в первом, заго-

	C	v	$\rightarrow$
C	Унифицируется только в случае, если унифицируемые базовые типы совпадают. В противном случае имеет место ошибка несоответствия базовых типов.	Предположение о типовой переменной отвергается, из контекста берётся базовый тип. Осуществляется проверка совместимости этого базового типа с использованием исследуемого выражения.	Унифицируется только в случае, если базовый тип является функциональным типом, и обе стороны применения стрелки также могут быть унифицированы друг с другом.
v	Типовая переменная в контексте принимает конкретное значение в виде унифицируемого базового типа.	В контексте для исследуемого подвыражения оставляется только один идентификатор типовой переменной с обязательной заменой вхождений удаляемого идентификатора.	Типовая переменная в контексте заменяется на стрелку (с подстановкой во всех типах в составе контекста).
$\rightarrow$	Унифицируется только в случае, если унифицируемый базовый тип является функциональным типом, и обе стороны применения стрелки соответствуют друг с другу.	Производится проверка на вхождение типовой переменной в состав стрелки. Если вхождение имеется, то имеет место ошибка типизации (например, как для λ-терма $\lambda x.xx$). В противном случае вместо переменной используется стрелка.	Унифицируется только если друг с другом унифицируются соответствующие правые и левые операнды стрелок.

Таблица 6.1. Унификация типов

ловочном столбце). Столбцы соответствуют вариантам типов, которые необходимо унифицировать с тем, что уже имеется в контексте (соответственно, эти варианты перечислены в заглавной строке).

В данной таблице C обозначает базовый тип из соответствующего множества базовых типов, v обозначает типовую переменную, а $\boxed{\rightarrow}$ обозначает стрелку.

6.2.3. Несколько несложных примеров

В качестве примеров, при помощи которых можно детально изучить алгоритм и закрепить его понимание, можно рассмотреть типизацию нескольких базовых комбинаторов. Например, пусть это будут композитор **B**, пермутатор **C** и дубликатор **W**[2].

Композитор **B** имеет комбинаторную характеристику $\mathbf{B}xyz = x(yz)$. В виде λ-терма его можно представить как $\lambda xyz.x(yz)$ или, записывая явно все синтаксические конструкции, — $\lambda x.\lambda y.\lambda z.(x(yz))$. Типизация первой абстракции, связывающей переменную x, даёт следующие предположения о типах подвыражений:

[2]Традиционные наименования данных комбинаторов происходят от их комбинаторных характеристик. Композитор **B** представляет собой операцию композиции функций, пермутатор **C** переставляет аргументы функции (англ. *permute* — менять порядок), а дубликатор **W** дублирует аргумент функции.

- $x : \alpha$;
- $(\lambda y.\lambda z.(x(yz))) : \beta$;
- $(\lambda x.\lambda y.\lambda z.(x(yz))) : \alpha \to \beta$.

Далее последовательно применяя правила вывода для каждого подвыражения, можно получить следующий набор предположений о типах:

- $y : \gamma$;
- $(\lambda z.(x(yz))) : \delta$;
- $(\lambda y.\lambda z.(x(yz))) : \gamma \to \delta$;
- $z : \varepsilon$;
- $(x(yz)) : \eta$;
- $(\lambda z.(x(yz))) : \varepsilon \to \eta$;
- $x : \theta \to \iota$;
- $(yz) : \theta$;
- $(x(yz)) : \iota$;
- $y : \kappa \to \mu$;
- $z : \kappa$;
- $(yz) : \mu$.

Сопоставляя подвыражения исходного λ-терма для композитора $\mathbf{B}$, выстраивается система типовых уравнений:

- $\alpha = \theta \to \iota$;
- $\beta = \gamma \to \delta$;
- $\gamma = \kappa \to \mu$;
- $\delta = \varepsilon \to \eta$;
- $\varepsilon = \kappa$;
- $\eta = \iota$;
- $\theta = \mu$.

Наконец, собирается общий тип исходного λ-терма, который в контексте выглядит как $\alpha \to \beta$:

$$
\begin{aligned}
& \alpha \to \beta \\
\Rightarrow \quad & (\theta \to \iota) \to (\gamma \to \delta) \\
\Rightarrow \quad & (\theta \to \iota) \to ((\kappa \to \mu) \to (\varepsilon \to \eta)) \\
\Rightarrow \quad & (\theta \to \eta) \to ((\varepsilon \to \theta) \to (\varepsilon \to \eta)) \\
\Rightarrow \quad & (\theta \to \eta) \to (\varepsilon \to \theta) \to \varepsilon \to \eta \\
\overset{\alpha}{\Rightarrow} \quad & (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma
\end{aligned}
$$

На последнем шаге вывода типов была произведена α-конверсия идентификаторов типовых переменных в целях приведения сигнатуры к более традиционному виду. В итоге получается корректно типизированный λ-терм: $\mathbf{B} : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$. Если вспомнить, что в языке Haskell композитор $\mathbf{B}$ соответствует операции композиции функций `.`, то можно сравнить их типы (с помощью интерпретатора GHCi, использовав команду `:t`):

```
> :t (.)
(.) :: (b → c) → (a → b) → a → c
```

Так же типизируется пермутатор **C**, который в виде полного λ-терма записывается как $\lambda x.\lambda y.\lambda z.((xz)y)$. Система уравнений для этого λ-терма выглядит следующим образом (для упрощения изложения перечисленные ниже выражения составлены из предположений о типах и типовых уравний, в связи с чем должны читаться как «для такого-то λ-терма в контексте имеются следующие подлежащие унификации типы», для этого использован символ двойного двоеточия):

$$
\begin{aligned}
x &:: \alpha = \kappa \to \mu.\\
y &:: \gamma = \theta.\\
z &:: \varepsilon = \kappa.\\
\lambda y.\lambda z.((xz)y) &:: \beta = \gamma \to \delta.\\
\lambda z.((xz)y) &:: \delta = \varepsilon \to \eta.\\
((xz)y) &:: \eta = \iota.\\
(xz) &:: \theta \to \iota = \mu.
\end{aligned}
$$

Унификация этих типовых уравнений выглядит примерно так:

$$
\begin{aligned}
& \alpha \to \beta\\
\Rightarrow\quad & (\kappa \to \mu) \to (\gamma \to \delta)\\
\Rightarrow\quad & (\kappa \to \mu) \to (\gamma \to (\varepsilon \to \eta))\\
\Rightarrow\quad & (\kappa \to \mu) \to (\theta \to (\kappa \to \iota))\\
\Rightarrow\quad & (\kappa \to \theta \to \iota) \to \theta \to \kappa \to \iota\\
\overset{\alpha}{\Rightarrow}\quad & (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma
\end{aligned}
$$

Полученное выражение типизации $\mathbf{C} : (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$ опять же полностью соответствует типу функции `flip` в языке Haskell, которая переставляет аргументы для заданной функции:

```
> :t flip
flip :: (a → b → c) → b → a → c
```

Для дубликатора **W** процесс вывода типа абсолютно такой же. Читатель может самостоятельно составить набор предположений, на основании него вывести систему типовых уравнений, после чего провести процесс унификации. Результат можно сравнить со следующей цепочкой унификации:

$$
\begin{aligned}
& \alpha \to \beta\\
\Rightarrow\quad & (\theta \to \iota) \to (\gamma \to \delta)\\
\Rightarrow\quad & (\theta \to (\varepsilon \to \eta)) \to (\gamma \to \delta)\\
\Rightarrow\quad & (\theta \to \theta \to \eta) \to \theta \to \eta\\
\overset{\alpha}{\Rightarrow}\quad & (\alpha \to \alpha \to \beta) \to \alpha \to \beta
\end{aligned}
$$

Результат типизации $\mathbf{W} : (\alpha \to \alpha \to \beta) \to \alpha \to \beta$ можно проверить в GHCi:

```
> :t (λx y → x y y)
(λx y → x y y) :: (a → a → b) → a → b
```

Хорошее краткое описание алгоритма, дополнительные примеры, а также объяснение дополнительных правил вывода дано в презентации Р. Чепляки [15].

6.3. Адаптация алгоритма для языка Haskell

Любой прикладной язык программирования отличается от простого типизированного λ-исчисления наличием дополнительных синтаксических конструкций, ограничений и прочих вещей, которые не выражаются в терминах оного λ-исчисления. Например, комбинаторы неподвижной точки не типизируются в системе Хиндли — Милнера [9], а потому либо должны быть внедрены в ядро языка, либо для вывода типов должна использоваться иная система типизации (например, система Жирара-Рейнольдса, известная как «Система F»).

В языках программирования семейства ML (к которому можно отнести и язык Haskell) используются дополнительные синтаксические конструкции, которые позволяют определять функции (в том числе и рекурсивные). Общей методикой доработки алгоритма типизации является включение в состав базовых типов и правил вывода дополнительных сущностей, которые отражают особенности языка программирования [1].

В качестве примера можно рассмотреть некоторые особенности алгоритма типизации в Haskell. Несмотря на то, что алгоритм для этого языка базируется на системе Жирара-Рейнольдса, в её основе в любом случае лежит первоначальный алгоритм W и система типизации Хиндли — Милнера. Отметим следующие особенности, которые нужно учесть не только при изучении этого языка, но и других функциональных языков программирования:

1) Все типы в языке Haskell имеют в своём составе специальное значение $\perp$, которое используется для обозначения незавершающихся или ошибочных вычислений. Обработка этого значения заложена в системе типизации языка Haskell.

2) Разработчик может определять свои алгебраические типы данных [10] и изоморфные типы, которые должны динамически вноситься в множество базовых типов и использоваться в процессе вывода типов.

3) Система классов типов, используемых в языке Haskell для описания функциональных интерфейсов и ограничений, налагаемых на используемые типы, дополняет алгоритм типизации необходимостью обрабатывать такие ограничения. Возможность использования нескольких ограничений, а также своеобразное «наследование» классов накладывает дополнительные требования на алгоритм типизации.

4) Расширения языка Haskell, связанные с системой типизации, а именно полиморфизм высших рангов [12], экзистенциальные типы [11], многопараметрические классы, классы с зависимыми параметрами и т. д., также дополняют и усложняют алгоритм вывода типов.

Дополнительно о типизации в языке Haskell и способах реализации алгоритма можно ознакомиться в статье [4].

6.4. Пример реализации функций для автоматического вывода типов

Чтобы закрепить полученные сведения об алгоритме вывода типов, имеет смысл реализовать простой вариант данного алгоритма. Пусть в качестве языка программирования выступает язык Haskell, а в качестве формальной системы, для которой будет реализовываться алгоритм типизации, выступает простое типизированное λ-исчисление. Дополнительно вводится ограничение, что в качестве константного типа выступает тип $*$, так что все функциональные типы имеют вид $(* \to *)$, $(* \to * \to *)$, $((* \to *) \to *)$ и т. д.

Далее в данном разделе описываются типы для представления сущностей предметной области, специализированные функции для облегчения работы с разрабатываемой программой, а также основной алгоритм вывода типов по Хиндли — Милнеру.

6.4.1. Типы и связанные с ними определения

Для внутреннего представления структур данных, описывающих λ-термы и их типы, необходимо определить соответствующие типы языка Haskell. Эти типы определяются в полном соответствии с формулами, данными во введении.

Тип `Expr` описывает одно λ-выражение (англ. *expression*), которое может быть переменной (конструктор `Var`), применением терма к другому терму или аппликацией (конструктор `App`) или λ-абстракцией, то есть связыванием переменной с λ-термом (конструктор `Abs`):

```
data Expr = Var String
          | App Expr Expr
          | Abs String Expr
  deriving (Eq, Ord)
```

Тип `Type` описывает типы, которые могут быть сопоставлены λ-термам. Тип может быть константным (конструктор `Const`), типовой переменной (конструктор `TyVar`) и функциональным типом или стрелкой (конструктор `Arrow`):

```
data Type = Const
          | TyVar String
          | Arrow Type Type
  deriving Eq
```

Для того чтобы отображать значения двух вышеприведённых типов, имеет смысл специальным образом реализовать для них экземпляры класса `Show`, а не полагаться на автоматически построенные определения. Это делается тривиально:

```
instance Show Expr where
  show (Var name)      = name
  show (App exp1@(Abs _ _) exp2)
                       = "(" ++ show exp1 ++ ")" ++ show exp2
```

```
  show (App expr (Var name))
                        = show expr ++ name
  show (App exp1 exp2) = show exp1 ++ "(" ++ show exp2 ++ ")"
  show (Abs name expr) = "\\" ++ name ++ "." ++ show expr
```

и

```
instance Show Type where
  show Const               = "*"
  show (TyVar name)        = name
  show (Arrow Const typ2) = "* → " ++ show typ2
  show (Arrow (TyVar name) typ2)
                           = name ++ " → " ++ show typ2
  show (Arrow typ1  typ2) = "(" ++ show typ1 ++ ") → " ++
show typ2
```

Данные экземпляры позволят выводить на экран значения типов `Expr` и `Type` соответственно в «традиционном» виде, учитывая соглашения об опускании скобок.

Так, например, комбинатор $\mathbf{K} \equiv \lambda xy.x$, который во внутреннем представлении имеет вид «`Abs "x"(Abs "y"(Var "x"))`» (и в примерно таком же виде это представление выводилось бы на экран в случае использования автоматически сгенерированного экземпляра класса `Show`), будет преобразован в строку «λ`x.`λ`y.x`».

А тип комбинатора $\mathbf{S} \equiv \lambda xyz.xz(yz)$, который в принятых ограничениях может быть представлен как $(* \to * \to *) \to (* \to *) \to * \to *$ и имеющий следующее внутреннее представление:

```
Arrow (Arrow Const
             (Arrow Const Const))
      (Arrow (Arrow Const Const)
             (Arrow Const Const))
```

будет преобразован в строку «$(* \to * \to *) \to (* \to *) \to * \to *$».

6.4.2. Вспомогательные функции

Для того чтобы не утруждать пользователя разрабатываемой программы вводом описаний простых λ-выражений во внутреннем представлении, имеет смысл реализовать синтаксический анализатор, который будет переводить вводимую строку во внутреннее представление. Это также поможет при разработке цикла интерактивного взаимодействия, когда у пользователя просто не будет возможности вводить данные во внутреннем представлении в строке интерпретатора.

Синтаксический анализатор имеет смысл реализовывать только для простых λ-выражений, поскольку типы пользователь вводить не будет, они будут автоматически выводиться программой. Синтаксический анализатор, по сути, должен преобразовывать строку, сгенерированную при помощи экземпляра класса `Show`, обратно во внутреннее представление. Другими словами, для синтаксического анализатора `parse` должны выполняться следующие правила:

- `parse · show ≡ id :: Expr → Expr`
- `show · parse ≡ id :: String → String`

Для реализации синтаксического анализатора проще всего воспользоваться готовой библиотекой комбинаторов синтаксического анализа, например библиотекой `Parsec`, которая была разработана Д. Лейеном [5]. При помощи этой библиотеки формальная грамматика языка представления простых λ-выражений практически без изменений (с точностью до синтаксиса) записывается на языке Haskell.

Формальная грамматика для описания простых λ-выражений изоморфна определению оных (см. введение):

```
Expr ::= Var | App | Abs

Var  ::= <Id>
App  ::= '(' Expr Expr ')'
Abs  ::= 'λ' <Id> '.' Expr
```

В список задач настоящей статьи не входит обучение читателя основам синтаксического анализа и разработки функций для него. Заинтересованного читателя можно отослать к специализированной литературе на эту тему. Определения же функций для анализа простых типизированных λ-термов имеются в исходных кодах, которые можно получить с сайта журнала. Корневой функцией для анализа является функция `parseExpr` в модуле `TIParser`.

Проверка работоспособности этих функций осуществляется при помощи следующего определения (в нём также показан вариант использования функции `parseExpr`):

```
test :: String → String
test l = case parse parseExpr "" $ filter (not · isSpace) l of
           Left msg → show msg
           Right rs → show rs
```

Если в передаваемой на вход анализатору строке имеется синтаксическая ошибка, парсер вернёт значение `Left`, в котором содержится сообщение об ошибке. Если же синтаксический анализ прошёл успешно, в значении `Right` будет возвращён результат анализа, в рассматриваемом случае — λ-выражение во внутреннем представлении типа `Expr`.

6.4.3. Алгоритм типизации по Хиндли — Милнеру

Всё готово для того, чтобы реализовать функцию вывода типов для простого типизированного λ-исчисления. Правила вывода типов полностью совпадают с теми, что описаны во введении. Это позволяет непосредственно реализовать данные правила на языке Haskell. Поскольку констант в языке простого типизированного λ-исчисления нет, функция для вывода типов состоит из трёх клозов.

Имеет смысл рассмотреть тип функции `inferType`, предназначенной для вывода типов. Он таков:

```
inferType :: Expr
          → [String]
          → Environment
          → Either String (Environment, [String])
```

Первый аргумент представляет собой выражение, тип которого необходимо вычислить. Второй аргумент — это список доступных для использования идентификаторов типовых переменных. Третий аргумент является окружением, в котором хранятся пары вида (Выражение, Тип). Это может быть простой список типа `[(Expr, Type)]`, однако в целях оптимизации этот тип определён как отображение:

```
type Environment = Map Expr Type
```

Результатом выполнения функции `inferType` является пара, первый элемент которой представляет собой новое состояние окружения, в которое включён тип для обработанного выражения (заданного первым аргументом), а второй элемент — это обновлённый перечень доступных для использования идентификаторов типовых переменных. Как видно, результат функции `inferType` обёрнут в монаду `Either String` для обработки ошибочных ситуаций. Кроме того, можно было бы «спрятать» явную передачу окружения и перечня доступных для использования идентификаторов типовых переменных в монаду `State`, однако этого не сделано потому, что одновременное использование для иллюстрации алгоритма нескольких идиом языка Haskell не позволит за ними увидеть суть алгоритма.

Использование типовых переменных в данном случае обусловлено тем, что в процессе вывода необходимо проводить унификацию типов, что крайне затруднительно делать, если в качестве типов непосредственно подставлять принятый константный тип `*`. Реализованный алгоритм вывода типов возвращает общий тип с типовыми переменными, который легко конкретизируется константным типом просто при помощи подстановки в качестве каждой типовой переменной константного типа `*`. Эта функциональность подразумевается, и не будет реализована в рамках данной статьи.

Первый клоз функции `inferType` соответствует схеме типизации переменной. Если переменная уже присутствет в окружении, то ничего менять не надо. Если же переменной ещё в окружении нет, то необходимо добавить в окружение новую пару, взяв в качестве идентификатора типовой переменной первый свободный элемент списка. Таким образом, переменная типа `Expr (Var n)` связывается с типом `TyVar tv`:

```
inferType e@(Var n) t@(tv:tvs) env
  = case Map.lookup e env of
      Nothing → return (Map.insert e (TyVar tv) env, tvs)
      Just _  → return (env, t)
```

Следующий клоз функции `inferType` предназначен для обработки λ-абстракций. Как показано во введении, связывание переменной типа τ с некоторым выражением типа σ даёт λ-терм типа $\tau \to \sigma$. Опять же, это несложно выражается на языке Haskell. В описываемом ниже клозе вводится дополнительное ограничение — нельзя использовать повторяющиеся идентификаторы связанных переменных, поскольку для демонстрации возможностей алгоритма типизации нет необходимости реализовывать α-конверсию и формализм де Брауна [3]. В связи с этим производится проверка на наличие в окружении типа для связанной переменной: если тип уже назначен, то имеет место дублирование переменной, что недопустимо.

Если же тип переменной, связываемой рассматриваемой λ-абстракцией, не назначен, то производится попытка вывести тип выражения (тела λ-абстракции). Если попытка удачна, то в новое окружение записываются три пары — тип для связываемой переменной, тип для тела λ-абстракции (запись производится неявно во время вывода типа для него) и тип для всего λ-терма. При этом надо отметить, что во время вывода типа для тела λ-абстракции уже мог быть выведен и тип связанной переменной (если она входит в тело). Итак, клоз функции `inferType` выглядит следующим образом:

```
inferType e@(Abs n e1) (tv:tvs) env
  = case Map.lookup (Var n) env of
      Nothing → do
        (env', tvs') ← inferType e1 tvs env
        let Just tp' = Map.lookup e1 env'
        case Map.lookup (Var n) env' of
           Nothing → do
             let env'' = Map.insert (Var n) (TyVar tv) env'
             return (Map.insert e (Arrow (TyVar tv) tp') env'', tvs')
           Just tp → return (Map.insert e (Arrow tp tp') env', tvs')
      Just tp → fail ("ERROR: Duplicate bound variable \"" ++
                      n ++ "\" in lambda-abstraction.")
```

Самым интересным и самым непростым является случай вывода типа для аппликации. Здесь необходимо проверять, к чему применяется выражение, поскольку для разных видов λ-термов необходимо применять различные правила вывода типов, которые являются следствием правила вывода для аппликации (опять же см. введение).

Если выражение, к которому применяется другое выражение, является простой переменной, то в зависимости от того, используется эта переменная в применяемом выражении или нет, производится либо добавление новых типов в окружение, либо попытка унификации типов с последующим обновлением окружения в случае, если унификация прошла успешно.

Для аппликации, в которой первым выражением также является аппликация, вывод типов производится при помощи унификации типов, поскольку тип первой аппликации должен быть стрелкой. Если унификация прошла успешно, в окружение записываются новые значения типов.

Наконец, если аппликация производится к λ-абстракции (то есть это просто вызов функции в терминах функционального программирования), то производится сравнение типов на «совместимость», ведь если у λ-терма тип вида $\tau \to \sigma$, то у прикладываемого к нему выражения должен быть тип, унифицируемый типом τ.

В итоге получается следующий достаточно громоздкий клоз определения функции `inferType`. Для его понимания необходимо внимательно проследить изменения окружения, которые производятся при помощи вызова функции `Map.insert`.

```
inferType e@(App e1 e2) t@(tv:tvs) env
  = do (env', tvs') ← inferType e2 tvs env
       let Just tp' = Map.lookup e2 env'
       case e1 of
         Var{} →
           case Map.lookup e1 env' of
             Nothing → do return (Map.insert e
                                    (TyVar tv)
                                    (Map.insert e1
                                       (Arrow tp' (TyVar tv))
                                       env'), tvs')
             Just tp → do env'' ← unifyTypes (Arrow tp' (TyVar tv), tp
                          let Just (Arrow tp'' tp''') =
Map.lookup e1 env''
                          return (Map.insert e tp''' env'', tvs')
         App{} →
           do (env'', tvs'') ← inferType e1 tvs' env'
              let Just tp = Map.lookup e1 env''
              env''' ← unifyTypes (tp, Arrow tp' (TyVar tv)) env''
              let Just (Arrow tp'' tp''') = Map.lookup e1 env'''
              return (Map.insert e tp''' env''', tvs'')
         Abs{} →
           do (env'', tvs'') ← inferType e1 tvs' env'
              let Just (Arrow tp1 tp2) = Map.lookup e1 env''
              if areTypesCompatible tp1 tp'
                then do env''' ← unifyTypes (tp1, tp') env''
                        let Just (Arrow tp1' tp2') =
Map.lookup e1 env'''
                        return (Map.insert e tp2' env''', tvs'')
                else fail ("ERROR: Can't apply \"" ++ show e1 ++
                           "\" to \"" ++ show e2 ++
                           "\". Incompatible types.")
```

Теперь необходимо реализовать функцию для унификации типов `unifyTypes` и предикат для проверки типов на совместимость `areTypesCompatible`.

Первая функция получает на вход два типа — первый необходимо унифицировать вторым. Результат работы этой функции помещается непосредственно в окружение, поэтому она тоже работает в монаде `Either String`. Таким образом, фукнция по-

лучает на вход пару типов, текущее значение окружения, а при успешном завершении возвращает обновлённое состояние окружения. Её определение выглядит следующим образом:

```
unifyTypes :: (Type, Type) → Environment →
Either String Environment
unifyTypes (Const, Const) env = return env
unifyTypes (TyVar n, t) env =
  return $ Map.map (substituteTyVar n t) env
unifyTypes (Arrow t1 t2, Arrow t1' t2') env
  = do env' ← unifyTypes (t1, t1') env
       unifyTypes (t2, t2') env'
unifyTypes (t1, t2) _
  = fail ("ERROR: Can't unify type "
          "(" ++ show t1 ++ ") with "
          "(" ++ show t2 ++ ").")
```

Типовая константа унифицируется с типовой константой, при этом изменений в окружение не вносится. Типовая переменная может быть унифицирована любым типом при помощи подстановки вместо типовой переменной этого типа во всех типовых выражениях, находящихся в окружении. Две стрелки унифицируются, если унифицируются соответствующие компоненты стрелок. Другие варианты унификации типов ошибочны (например, попытка унифицировать тип $\alpha \to \beta$ типом α в λ-терме $\lambda x.xx$), в связи с чем возвращается сообщение об ошибке.

Функция `substituteTyVar` для подстановки типа вместо типовой переменной реализуется несложно:

```
substituteTyVar :: String → Type → Type → Type
substituteTyVar _ _ Const
  = Const
substituteTyVar n t (TyVar n')
  | n' == n   = t
  | otherwise = TyVar n'
substituteTyVar n t (Arrow t1 t2)
  = Arrow (substituteTyVar n t t1)
          (substituteTyVar n t t2)
```

Первым аргументом является идентификатор типовой переменной, вместо которой необходимо подставить тип. Вторым аргументом является тип, который подставляется вместо типовой переменной. Третий аргумент — это тип, в котором производится замена (подстановка). Функция просто обходит дерево типа и заменяет вхождения заданной типовой переменной.

Наконец, предикат `areTypesCompatible` возвращает значение `True` в случае, если типы совместимы с точки зрения процесса унификации (первый тип может быть унифицирован вторым). Определение опять говорит само за себя:

```
areTypesCompatible :: Type → Type → Bool
```

```
areTypesCompatible Const _
  = True
areTypesCompatible TyVar{} _
  = True
areTypesCompatible (Arrow t1 t2) (Arrow t1' t2')
  = areTypesCompatible t1 t1' && areTypesCompatible t2 t2'
areTypesCompatible _ _
  = False
```

6.4.4. Цикл опроса пользователя

Для упрощения способа использования разработанной функции вывода типов `inferType` можно организовать цикл интерактивного общения с пользователем. Это достигается при помощи простых определений:

```
main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  putStrLn "Type checker for simple Lambda-calculus.\n
            Enter lambda-term for typing or :quit to quit.\n"
  runInterpreterCycle

runInterpreterCycle :: IO ()
runInterpreterCycle = do
  putStr "> "
  term ← getLine
  if (map toUpper term) `isPrefixOf` ":QUIT"
    then return ()
    else do
         case parse parseExpr "" (filter (not · isSpace) term) of
          Left msg → putStrLn (show msg ++ "\n")
          Right r  →
            case inferType r generateTyVarNames Map.empty of
              Left msg      → putStrLn (msg ++ "\n")
              Right (e, _) →
                let Just t = Map.lookup r e
                in  putStrLn (show r ++ " :: " ++ show t ++ "\n")
         runInterpreterCycle
```

Функция `runInterpreterCycle` выполняет один шаг цикла опроса пользователя (детали см. в [13]). Если пользователь ввёл что-то похожее на команду «`:QUIT`», то цикл заканчивается и программа завершается. Иначе введённая строка трактуется как λ-терм, который анализируется синтаксическим анализатором `parseExpr` и в случае успеха для этого терма выводится тип. Если тип также выводится успешно, он печатается на экран, а цикл опроса пользователя запускается на новый виток. Ес-

ли на каком-то этапе произошла ошибка, диагностическая информация о ней также выводится на экран, после чего цикл опроса пользователя запускается снова.

Результаты работы написанной программы примерно такие:

```
Type checker for simple Lambda-calculus.
Enter lambda-term for typing or :quit to quit.

> λx.x
λx.x :: b → b

> λx.λy.x
λx.λy.x :: c → b → c

> λx.λy.λz.xz(yz)
λx.λy.λz.xz(yz) :: (f → e → d) → (f → e) → f → d
```

Как уже сказано ранее, здесь не производится подстановка вместо типовых переменных константы $*$, поскольку это тривиальная операция. Вместо этого разработанная программа позволяет изучать типы простых λ-термов в наиболее общем виде, как это доказано Р. Милнером для алгоритма типизации Хиндли — Милнера.

6.5. Заключение

В статье приведено краткое и не претендующее на полноту описание одного из важнейших механизмов, используемых в функциональной парадигме программирования — системы типизации Хиндли — Милнера. Понимание данного механизма поможет разработчикам программного обеспечения более полно осознавать процессы, происходящие в «недрах» трансляторов, что позволит в свою очередь разрабатывать более эффективные алгоритмы. Для углублённого изучения механизмов автоматического вывода типов можно порекомендовать уже упоминавшиеся работы [14] и [7], а также статью [1].

Все рассмотренные в статье исходные коды можно загрузить с сайта журнала.

Литература

[1] *Cardelli L.* Basic polymorphic typechecking // *Science of Computer Programming*. — 1987. — Vol. 8. — Pp. 147–172.

[2] *Damas L., Milner R.* Principal type-schemes for functional programs // POPL-82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM. — Pp. 207–212.

[3] *de Bruijn N. G.* A Combinatorial Problem. — 1946. — P. 758–764.

[4] *Jones M. P.* Typing Haskell in Haskell // Haskell Workshop. — 1999.

[5] *Leijen D.* Parsec, a fast combinator parser. — 2001.

[6] *Milner R.* A theory of type polymorphism in programming // *Journal of Computer and System Sciences*. — 1978. — Vol. 17. — Pp. 348–375.

[7] *Pierce B. C.* Types and Programming Languages. — Cambridge, Massachusetts: The MIT Press, 2002. — По адресу URL: http://newstar.rinet.ru/~goga/tapl/ (дата обращения: 21 мая 2010 г.) опубликован перевод книги на русский язык.

[8] *Robinson A. J.* A machine-oriented logic based on the resolution principle // Journal of the ACM. — Vol. 12. — 1965.

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

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

[11] *Душкин Р. В.* Мономорфизм, полиморфизм и экзистенциальные типы // *Научно-практический журнал «Практика функционального программирования»*. — 2009. — Т. 4, № 4. — С. 79–88.

[12] *Душкин Р. В.* Полиморфизм в языке Haskell // «*Практика функционального программирования*». — 2009. — Т. 3, № 3. — С. 67–81.

[13] *Душкин Р. В.* Справочник по языку Haskell. — М.: ДМК Пресс, 2008. — 544 с.

[14] *Кирпичёв Е. Р.* Элементы функциональных языков // *Научно-практический журнал «Практика функционального программирования»*. — 2009. — Т. 3, № 3. — С. 83–197.

[15] *Чепляка Р.* Вывод типов и полиморфизм. — 2009. — Презентация доклада на LtU@Kiev, URL: http://ro-che.info/docs/2009-05-30-ltu-kiev-type-inference.pdf (дата обращения: 21 мая 2010 г.).