Практика
функционального программирования

Выпуск 2
Сентябрь 2009

ISSN 2075-8456

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

Журнал «Практика функционального программирования»

Авторы статей: Александр Самойлович
Алексей Отт
Влад Балин
Дмитрий Астапов
Дмитрий Зуйков
Роман Душкин
Сергей Зефиров

Редактор: Лев Валкин

Корректор: Ольга Боброва

Иллюстрации: **Обложка**
© UN Photo/Andrea Brizzi
© iStockPhoto/sx70

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

Ревизия: 495 (2009-09-29)

Сайт журнала: http://fprog.ru/

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

Оглавление

От редактора

Спасибо вам за интерес ко второму выпуску журнала! Планируя первый выпуск, мы не имели никакого представления о масштабе интереса к декларативному программированию в общем и к русскоязычному ресурсу о нём в частности. Наши оптимистичные оценки потенциальной аудитории журнала находились в районе полутора тысяч читателей. Каково же было наше удивление, когда оказалось, что только за первые две недели с момента выхода в свет первого номера журнала количество уникальных читателей нашего электронного выпуска зашкалило за десять тысяч!

Вот бы ещё иметь возможность узнать, кто сумел дочитать выпуск до конца…

Мы получили от вас большое количество отзывов и идей. Конечно, было получено и множество противоречивых откликов, от «в первом выпуске статьи — для самых маленьких» до «слишком запутанно, умер на двадцатой странице». На последнее можно заметить, что практические задачи и способы их решения у всех авторов разные. Поэтому ожидайте статей самого разного уровня, от начального до теории категорий. Если одна статья «не идёт», за ней есть следующая, другого автора, и так далее.

Что же касается *недостаточного* уровня статей, тут тоже всё просто. Одной из главных задач нашего журнала видится популяризация — распространение информации об иных, часто более удобных, подходах к пониманию и трансформированию реальности. Мы хотим сделать так, чтобы большее количество людей заинтересовалось альтернативными подходами к программированию, начало читать соответствующие учебники и расширило свой набор используемых приёмов и инструментов.

Мы не хотим делать журнал для сотни-другой человек, понимающих, что такое комонада и прочий «матан». Эти люди, как правило, знают английский и интересуются темой достаточно глубоко, чтобы читать первоисточники (так сложилось, что практически все первоисточники в настоящее время — на английском). Рассчитывать журнал исключительно на них[1] — значит тратить огромное количество усилий ради исчезающе малого эффекта. Поэтому и в этом выпуске мы снова…

[1]Если вы узнали в этом портрете себя, лучше станьте автором или рецензентом. Напишите в редакцию: `editor@fprog.ru`. Материалы рецензируются и перед публикацией проходят корректуру, так что вы ничем не рискуете, пробуя себя в этом амплуа впервые.

Займёмся популяризацией

Центральная тема второго выпуска журнала — демонстрация применения функционального программирования в реальных, а не академических проектах.

Первые четыре статьи — Дмитрия Зуйкова, Дмитрия Астапова, Сергея Зефирова в соавторстве с Владиславом Балиным, и Алексея Отта — вытаскивают на поверхность «кухню» нескольких компаний. Статьи демонстрируют, что функциональные языки находят применение в промышленном программировании в самых разных нишах. Конечно, использование «нестандартных» языков накладывает на проекты некоторые сложно оценимые риски, и далеко не все из них рассмотрены в статьях. Но если статьи этого номера позволят развеять хоть часть сомнений, мифов и предрассудков и поднять дискуссию о применимости функциональных языков в промышленном программировании на новый уровень, мы будем считать свою задачу выполненной.

Статья Александра Самойловича рассматривает создание на языке Erlang игрушечного, но практичного проекта — рекурсивного многопоточного обходчика сайтов. К третьему выпуску журнала мы планируем подготовить ещё несколько статей про Erlang.

Завершающая статья Романа Душкина в большей степени ориентирована на теорию: она познакомит вас с концепцией алгебраических типов данных (АТД) в Haskell и других функциональных языках.

Языки разные, языки прекрасные

Те, кто уже успел прикоснуться к функциональному или логическому программированию в эпоху их экстенсивного роста (например, занимались Lisp-ом или Prolog-ом в восьмидесятые), иногда относятся к практической применимости функционального подхода и инструментария со скепсисом. И недаром: компиляторы тогда были наивными, компьютеры — маломощными, а аппаратные Lisp-машины[2] — редкими и дорогими. В то время «практическое применение» функционального инструментария естественным образом ограничивалось исследовательскими задачами.

Сейчас эти детские болезни по большей части уже в прошлом. С практической стороны, усовершенствования в техниках компиляции и интерпретации программ позволили ускорить функциональные языки так, что становится неочевидным, кто победит по скорости в той или иной задаче по обработке данных. Одними из «самых быстрых» языков программирования, часто обгоняющими результаты компиляции C и C++, признаются языки Stalin [2] и ATS [1] — диалекты функциональных языков Lisp и ML, соответственно. Даже такие распространённые реализации функциональных языков, как OCaml или GHC (Haskell), иной раз показывают *ускорение* относительно эквивалентных программ на C/C++ в десять и более раз[3], и редко отстают более чем в

[2] Lisp-машины, реализованные «в железе», были первопроходцами, сделавшими коммерчески доступными лазерную печать, оконные системы, компьютерную мышь, растровую графику высокого разрешения и другие инновации. Таких машин было произведено всего несколько тысяч штук.

[3] За счёт более простых (быстрых) алгоритмов управления памятью.

два-три раза.

Но не быстродействие тут главное. Быстродействие всегда можно получить, переписав критические участки кода на языках, в большей степени приближенных к аппаратуре, и оптимизировав их вручную. Настоящий вопрос — как убрать излишнюю сложность из программ, привносимую, в частности, бесконтрольным использованием программистами возможностей изменения состояния [3].

В отношении безопасности программ теория языков за последние двадцать лет продвинулась далеко вперёд. Фокус научных исследований сместился с Lisp-а и его реализаций на статически типизируемые языки, такие как Haskell и Epigram, а также на системы доказательства теорем Agda и Coq. Современные исследования преимущественно нацелены на создание и теоретическое обоснование таких *систем статической типизации*, которые позволяли бы по максимуму вооружить компилятор возможностью раннего обнаружения ошибок в программах, сохранив при этом максимальную гибкость для программиста. Настороженное отношение к декларативным языкам программирования, сформировавшееся под влиянием негативного опыта восьмидесятых, в настоящее время требует переосмысления.

Профессиональные программисты часто имеют устоявшиеся представления о градациях языков программирования, выражающиеся в терминах «лучше-хуже» и дополняемые неким практическим контекстом — «лучше для веба», «лучше для системного программирования» и т. д. Такая двухмерная матрица оценки языков довольно полезна (несмотря на субъективность), но я рискну ввести ещё один ортогональный критерий — дидактичность языка, возможность с использованием языка научиться максимуму интересных концепций.[4] Дидактичности можно противопоставить практичность, то есть полезность языка в качестве инструментального средства, определяемую как степень удобства разработки на нём промышленных систем. Так язык Haskell обладает большей дидактичностью, чем OCaml,[5] из-за ленивости, более развитой системы типов и большего количества принятых в нём интересных идиом. С другой стороны, OCaml может оказаться чуть более практичным инструментальным средством для промышленного программирования, благодаря возможности «срезать углы» и использовать элементы императивного стиля, а также часто существенно более шустрым результатам (и процессу!) компиляции.

В результате, мне представляется — конечно же, субъективно — следующая картинка для наиболее распространённых языков функционального программирования (см. таблицу 1).

С учётом этого, старые, взращенные на Lisp-ах, инстинкты по поводу использования (или неиспользования) функционального программирования должны быть переосмыслены: Lisp уже давно не является фронтиром, «лицом» функциональной парадигмы, мы должны иметь смелость с ним попрощаться. А если вы совсем не испорчены функциональным программированием, могу порекомендовать начинать сразу с языка Haskell.

[4] И применять их в любых других языках программирования, см. [4].

[5] И Haskell, и OCaml являются наследниками языка ML.

Дидактичность	Практичность	Лёгкость освоения
Haskell	OCaml	Erlang
Lisp	Erlang	OCaml
Erlang	Haskell	Lisp
OCaml	Lisp	Haskell

Таблица 1. Некоторые характеристики функциональных языков

Впрочем, есть и другие мнения. Читайте статью Алексея Отта *«Использование Scheme в разработке семейства продуктов „Дозор-Джет“»*, в которой описывается использование диалекта языка Lisp, зарекомендовавшего себя с хорошей стороны на решаемых в «Дозор-Джет» задачах.

Ну и вообще — читайте!

Лев Валкин, `vlm@fprog.ru`

P. S. Мы хотим, по возможности, продолжать распространять журнал бесплатно. Но *создавать* его бесплатно у нас совершенно не получается. Поэтому мы будем признательны всем, кто имеет возможность материально помочь журналу: на странице fprog.ru/donate вы можете найти детали перевода через системы WebMoney и Яндекс.Деньги. Даже сто рублей — эквивалент чашки кофе — имеет шанс сделать наши материалы ещё чуточку качественнее.

Также мы обращаемся к организациям и частным лицам, которые хотели бы видеть свои услуги или товары, интересные программистам и менеджерам софтверных проектов, в нашем журнале. Свободное место у нас имеется, добро пожаловать!

Литература

[1] ATS (programming language). — Статья в Википедии: URL: `http://en.wikipedia.org/wiki/ATS_(programming_language)` (дата обращения: 28 сентября 2009 г.).

[2] Stalin (Scheme Implementation). — Статья в Википедии: URL: `http://en.wikipedia.org/wiki/Stalin_(Scheme_implementation)` (дата обращения: 28 сентября 2009 г.).

[3] *Кирпичёв Е.* Изменяемое состояние: опасности и борьба с ними. // *Журнал «Практика функционального программирования»*. — 2009. — № 1. `http://fprog.ru/2009/issue01/`.

[4] *Кирпичёв Е.* Что дает ФП мне лично на практике. — Запись в дневнике: URL: `http://antilamer.livejournal.com/288854.html` (дата обращения: 28 сентября 2009 г.). — 2009.

История разработки одного компилятора

Дмитрий Зуйков
dmz@fprog.ru

Аннотация

Данная статья не является учебным курсом по написанию компиляторов, не ставит задачи подробно описать алгоритмы выведения типов или оптимизации. Это просто история о том, как внешне большая, сложная и страшная задача оказывается небольшой и не очень сложной, если использовать правильные инструменты для её решения. Тем не менее, ожидается, что читатель владеет хотя бы основными представлениями о компиляторах.

This article is not a tutorial on how to write a compiler and does not aim to describe the type inference algorithms in detail. This is just a history which tells how big, complex and scary task turns out to be small and quite simple, if proper instruments are utilized. Nevertheless, it is expected that the reader knows at least some basics about compilers.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/1605.html.

1.1. Предпосылки

Приоритетное направление деятельности нашей компании — разработка и внедрение решений, связанных с системами глобального позиционирования и навигации. Основным продуктом является сервисная система для предоставления услуг мониторинга автотранспорта, ориентированная, в первую очередь, на операторов сотовой связи.

В настоящий момент сервис на базе данной системы находится в стадии запуска в северо-западном регионе РФ в качестве совместного проекта с одним из операторов большой тройки, планируется развёртывание и в других регионах. Помимо этого, разрабатываются совместные проекты с несколькими государственными структурами.

Основные компоненты сервисной системы:

Инфраструктурные сервисы: данный слой обеспечивает взаимодействие системы с операторами связи и централизованное управление *мобильными терминалами*. Для реализации используется платформа Erlang/OTP, а в качестве СУБД — Mnesia.

Прикладные приложения: данный слой реализует различные приложения для пользователей системы; в настоящий момент это решения, предназначенные для контроля личного автотранспорта, а также планирования и мониторинга грузов (логистики). Данные приложения предоставляют веб-интерфейс, но его использование необязательно — может использоваться как «толстый клиент», так и доступ с мобильного телефона посредством IVR. Для реализации приложений также используется Erlang.

Мобильные терминалы: GSM/GPS и GSM/GLONASS трекеры — автономные модули на базе микроконтроллеров, GSM модема и приемника системы глобального позиционирования. Данные устройства устанавливаются на контролируемых системой объектах и могут взаимодействовать с ней посредством SMS или GPRS. Устройства имеют различные варианты исполнения и могут дистанционно перепрограммироваться в зависимости от решаемых задач.

Применяемые в системе мобильные терминалы должны обладать возможностью очень гибкой удалённой настройки, так как они могут быть использованы в самых различных приложениях, условиях эксплуатации и географических регионах: использование устройств на личных автомобилях с питанием от бортовой сети на дорогах в пределах Московской области, практически полностью покрытой сетями GSM, имеет совершенно иную специфику, чем использование модулей, располагающих только собственным источником питания, перемещающихся вместе с железнодорожными контейнерами через Урал в условиях длительного отсутствия связи и получающих техническое обслуживание (включая зарядку и замену батарей) не чаще, чем раз в полгода.

Для некоторых приложений необходимо иметь возможность динамического изменения поведения устройств в зависимости от меняющейся ситуации. Например, со временем может меняться покрытие сети GSM или используемый в данной местности оператор связи, может возникнуть необходимость обрабатывать меняющиеся расписания.

Чтобы обеспечить все требуемые режимы работы, необходимы трекеры с очень большим количеством настроек. Первоначально именно такими устройствами мы и располагали — их инициализация осуществлялась путем установки значений приблизительно двухсот регистров, определяющих режимы работы в различных географических зонах.

Установка регистров осуществлялась посредством сообщений SMS сети GSM, для инициализации трекера (достаточно часто происходящий процесс) требовалось послать более сотни сообщений, порядок которых был иногда критически важен.

Стоит ли упоминать, что сеть GSM не гарантирует порядок доставки SMS, и задержавшееся и повторно посланное шлюзом сообщение могло сломать весь и без того небыстрый процесс инициализации.

Совокупность этих факторов привела к тому, что от системы, состоящей из серверной части, сети GSM и примитивных автономных устройств добиться надежной устойчивой работы было невозможно. Требовался в корне другой подход.

Решение задачи, как наделить устройства дистанционно атомарно изменяемым поведением, варьируемым в зависимости от внешних факторов и внутренних событий системы, было достаточно очевидно — устройствам требовалась возможность загружать скрипты с сервера. Оставалось выяснить, что это должны быть за скрипты:

- Они должны быть очень компактными.
- Они должны с разумной скоростью выполняться на восьмибитных микроконтроллерах (мы начинали с устройств на базе микроконтроллера семейства PIC18), с небольшим объемом RAM (1–3 КБ), ограниченным объемом перезаписываемой постоянной памяти, доступ к которой может быть достаточно дорогим (например, по шине I^2C с частотой максимум 400 кГц).
- Среда исполнения для них должна обходиться минимальным объемом памяти.
- Они должны быть безопасными: никакой загруженный извне скрипт не должен приводить к падению системы с потерей связи с ней.

Было рассмотрено достаточное число готовых реализаций различных языков: форты, лиспы, Joy, Cat, Raven, Tcl, Staapl, Squeak, Lua, Haxe, Python, Java и другие. Был рассмотрен вариант написания своего рантайма для существующего компилятора.

Также был исследован вопрос плотности кода, после чего интерпретаторы сразу же отпали: по критерию плотности кода они серьезно уступают байткоду, да и по трудоемкости реализации рантайма тоже, учитывая платформу, для которой требовалось их реализовать.

По плотности байткода в финал вышли форты и Java (на самом деле squeak, но он отсеялся по другим причинам), но спецификация JVM весьма сложна, и были серьезные сомнения, что за разумное время удастся реализовать её для нашей платформы. При этом пришлось бы научиться дорабатывать компилятор Java самостоятельно, а это сводило преимущества использования чужого решения к нулю. Разрабатывать свой компилятор Java в планы точно не входило.

В результате было принято решение реализовать скриптовый язык и его рантайм самостоятельно.

1.2. Первое приближение: Форт

Итак, в качестве скриптового языка в первом приближении был выбран Форт. Его преимущества очевидны: он крайне прост в реализации, но при этом довольно мощен. Программа на Форте представляет собой поток токенов, разделённых пробельными символами — таким образом, транслятор Форта не требует парсера, нужен лишь лексический анализатор, он же токенайзер.

В качестве языка реализации транслятора был выбран Python по причине его широкого, на тот момент, применения в проекте.

Рантайм представлял собой типичную двухстековую Форт-машину, без какого-либо управления динамической памятью, реализованную по мотивам F21.[1]

Инструкции и литералы (константы) имеют разрядность 8 бит. Используется token threaded code, то есть код, представленный потоком чисел, каждое из которых соответствует смещению в таблице обработчиков.[2]

Транслятор был реализован на Python достаточно быстро, в императивном стиле, и занял в районе двух тысяч строк кода. Останавливаться на его дизайне или реализации, равно как и на подробном описании Форта, выходит за рамки данной статьи: Python — весьма распространенный императивный язык программирования, по Форту тоже доступна масса информации.

При всей красоте Форта как идеи, язык все-таки имеет ряд особенностей, которые делают его использование не таким уж и простым.

Главная проблема — отсутствие типизации. Ошибка типов может привести к падению всей прошивки устройства, что нарушает одно из важнейших требований к нашему скриптовому языку — безопасность.

Еще одна проблема — наиболее компактный код на Форте получается, когда все вычисления проводятся на стеке. Но доступная глубина стека для вычислений в общем случае (не рассматривая Форты с командами типа pickn) ограничена приблизительно четырьмя его верхними ячейками. Более того, циклы со счетчиком организуются либо путем организации третьего «программного» стека для переменных цик-

[1] Известный, даже можно сказать, культовый в Форт-среде процессор, см. например: http://www.ultratechnology.com/f21data.pdf.

[2] На самом деле, конкретный способ обработки может варьироваться в зависимости от реализации виртуальной машины.

лов, что разрушает всю изящность языка, либо хранением переменной цикла в стеке адресов, что приводит к различным казусам в случае попытки возврата из *слова*, вызываемого внутри цикла. Данная проблема приводит к нарушению двух важных требований сразу: безопасности и компактности байткода. Если для реализации некоторого алгоритма не хватает двух-трех переменных, то приходится прибегать к различным ухищрениям: использованию стека адресов для промежуточного хранения данных, неуправляемой памяти — для хранения переменных, а также примитивов типа *rot*, *swap* или *over*. Это раздувает код и, в случае использования стека адресов, может приводить к непредсказуемым ошибкам времени исполнения, а также весьма затрудняет написание и прочтение кода. Как правило, чтобы понять некий алгоритм, реализованный на Форте, требуется его мысленно выполнить, что удаётся не всем.

Теперь можно сказать пару слов о применимости Python в качестве инструмента для реализации трансляторов. Особенности Python — динамическая типизация, неявное введение переменных и «утиная типизация»[3]. Все это означает, что многие ошибки будут обнаружены только во время исполнения. Разработка сколько-нибудь нетривиального кода зачастую требует многократного рефакторинга и попросту переписывания кода, с частыми откатами назад. Большое количество юнит-тестов в такой ситуации — это не просто признак культуры разработки, а вопрос выживания проекта; при этом тестами мы пытаемся поймать не только функциональные дефекты, но и элементарные ошибки и опечатки, которых при рефакторинге возникает масса. В случае разработки компилятора, придумывать юнит-тесты (в отличие от прочих видов тестов) может быть очень непросто, и этот фактор со временем сильно сокращает преимущество в производительности от применения высокоуровневого языка программирования.

Несмотря на озвученные проблемы, решение на основе Форта получилось вполне работоспособным и было применено. Для того, чтобы гарантировать рантайм от критических ошибок, пришлось разработать эмулятор, который имитировал основные периферийные устройства трекера (модем, приемник GPS, таймеры) и поток порождаемых ими данных и событий. Перед тем, как скелет скрипта начинал использоваться в устройствах, он тестировался на эмуляторе с целью обнаружения ошибок типов и переполнения стеков, а также прочих проблем, которые могли вызвать отказ трекеров.

Итак, поставленные цели были в основном достигнуты, но как результат, так и процесс разработки оставляли впечатление, что всё могло бы быть гораздо лучше.

1.3. Второе приближение — Бип /Веер/

Несмотря на то, что нами была разработана версия прошивки с реализацией Форт-машины для сторонних GPS-трекеров, применить её в деле не удалось из-за постоянных проблем аппаратного характера, помноженных на особенности ведения бизнеса

[3] Duck typing.

компанией-разработчиком.

Чтобы избавиться от подобных рисков в дальнейшем, было принято решение о самостоятельной разработке устройств.

Для их реализации был выбран микроконтроллер семейства MSP430, который выгодно отличается от PIC простой и удобной архитектурой, а также является 16-разрядным.

Предыдущий рантайм был реализован на ассемблере для PIC18 и не подлежал портированию, так что предстояло реализовывать его с нуля. Учитывая описанные выше недостатки Форта как прикладного языка, а также бóльшие возможности MSP430, было решено попробовать разработать более удобный, безопасный и доступный скриптовый язык, который бы давал возможность прямого программирования устройств их конечными пользователями.

Данный язык получил название Бип (англ. *Beep*) в честь звука, которым заглушают нецензурные выражения в средствах массовой информации, так как реалистичность подобной разработки была совсем неочевидна (а еще потому, что у автора есть правило — не тратить на придумывание названий более десяти минут).

1.3.1. Требования и дизайн языка

Дизайн языка определялся следующими первоначальными требованиями:

«Скриптовость», трактуемая как:

- Простота использования.
- Отсутствие необходимости явной декларации типов.
- Быстрая компиляция скрипта в байткод и его запуск.
- Управление памятью. Для сколько-нибудь серьёзных применений управление памятью совершенно необходимо, а отсутствие встроенного менеджера памяти приводит к тому, что его приходится каждый раз реализовывать с нуля.

Высокоуровневость: Поддержка в языке структур данных, применимых для прикладного программирования: пар, списков, структур, массивов и строк.

Универсальность: Так как границы области применения языка заранее не ясны, то язык должен быть универсальным, использование DSL нецелесообразно.

Императивность: По предыдущему опыту, типичные реализуемые алгоритмы выглядели императивными, так что логично делать язык императивным.

Простой синтаксис: Чтобы язык можно было быстро изучить, и чтобы его можно было быстро реализовать.

Расширяемость: Поскольку скорость исполнения даже скомпилированного в байт-код скрипта на порядок ниже скорости исполнения машинного кода, необходимо иметь возможность реализовывать критичные участки в виде функций на Си или ассемблере и вызывать их из скрипта.

Типизация: Отсутствие типизации делает невозможным простую разработку скриптов, так как даже незначительная ошибка может привести к критическим для устройства последствиям — разрушению памяти и стеков, краху прошивки и последующей недоступности устройства.

Вопрос заключался в выборе вида типизации: статическая или динамическая. Другими словами, необходимо было выбрать между сложным компилятором и простым рантаймом и более простым компилятором и сложным рантаймом.

Концептуально реализация динамической типизации выглядит весьма просто: каждая операция должна некоторым образом проверять типы своих операндов; в случае, если операнды имеют подходящий тип, операция должна выполняться, в противном случае должно порождаться исключение.

В исключениях и заключается главная проблема: в условиях автономно функционирующего устройства обрабатывать их каким-то разумным способом не представляется возможным.

Фактически мы опять приходим к варианту с разработкой эмулятора устройства и отсутствием возможности самостоятельного программирования устройств конечными пользователями, так как неправильно написанный скрипт может привести устройство в состояние, когда оно будет недоступно извне.

Вторая проблема динамической типизации заключается в том, что каждое значение в программе, где бы оно ни хранилось, должно содержать информацию о типе. Принимая во внимание выравнивание, это означает, что если каждое значение занимает *слово*, то и информация о типе занимает *слово*. Следовательно, доступная для скрипта память, которой и так немного (в начальной конфигурации — 512 *слов* кучи и 128 *слов* стека), сразу сокращается вдвое.

В итоге выбор здесь практически отсутствует — типизация для наших целей может быть только статической.

Таким образом, разрабатываемый язык должен обладать следующими свойствами:

- императивность,
- простой, привычный синтаксис,
- автоматически управляемая динамическая память (сборщик мусора),
- статическая типизация,
- автоматический вывод типов,
- типобезопасность и
- встроенные типы данных.

1.3.2. Выбор инструмента разработки

В качестве инструментов разработки рассматривались только высокоуровневые языки с сильной статической типизацией. К этому моменту уже присутствовало понимание того, что такой класс задач удобнее решать с использованием функционального подхода.

Рассматривались два варианта — Haskell и OCaml. Оба языка весьма зрелые, имеют давнюю историю и большое сообщество пользователей, а также большое количество библиотек и различных вспомогательных средств.

Несмотря на то, что Haskell выглядел более выигрышно: богатая, хорошо организованная и единообразная встроенная библиотека, имеется много большее количество сторонних решений, больше доступных онлайн руководств, примеров и книг, начать получать результаты оказалось проще с OCaml. Он и был выбран в итоге.

Существует большое количество примеров компиляторов, реализованных на OCaml, с которыми оказалось интересно ознакомиться перед тем, как начать разрабатывать свой. Вот некоторые из них:

Haxe Высокоуровневый компилируемый язык со статической типизацией и выводом типов. Компилируется в код для виртуальной машины NekoVM, которая является весьма интересным проектом сама по себе.

MinCaml Подмножество ML, компилируется в оптимизированный машинный код для SPARC, обгоняющий на некоторых тестах gcc. Являясь частью учебного курса японского Tohoku University, интересен как пример генерации кода и реализации различных оптимизаций.

The Programming Language Zoo Учебное пособие по курсу разработки трансляторов от Andrej Bauer, включающее реализацию интерпретаторов нескольких языков программирования. Хороший пример базовых техник, применяемых при разработке трансляторов, демонстрирующий использование *ocamlyacc* и *ocamllex*.

1.3.3. Инфраструктура проекта

После того, как было принято решение использовать OCaml, требовалось определиться со средством генерации парсеров и системой сборки проекта. Наиболее простым и хорошо документированным генератором парсеров и лексеров для OCaml является комплект *ocamlyacc* и *ocamllex*. Как правило, они распространяются в одном пакете с компилятором, так что фактически их можно считать частью языка.

Для сборки удобно использовать *ocamlbuild* — эта утилита в простейшем случае вообще не требует конфигурирования и собирает проект в текущем каталоге, самостоятельно определяя зависимости. Кроме того, она понимает входные файлы *ocamlyacc* и *ocamllex*, и примеры из *Language Zoo* используют именно её.

1.3.4. Дизайн компилятора

Процесс компиляции состоит из следующих фаз:

- лексический анализ,
- синтаксический разбор и построение AST[4],
- раскрытие макроподстановок,
- валидация AST,
- построение словаря,
- вывод и проверка типов,
- оптимизация на уровне AST,
- генерация промежуточного кода,
- оптимизация на уровне кода,
- генерация выходных файлов и
- генерация стабов.

Часть этих фаз вполне тривиальны, как, например, лексический анализ, который осуществляется автоматически сгенерированным из описания грамматики кодом, какие-то фазы достаточно интересны, чтобы рассказать о них более подробно.

Синтаксический разбор и построение AST

Разработка AST оказалась наиболее ответственной задачей, так как работать с ним приходится практически на всех фазах компиляции, и изменения в его структуре могут привести к переписыванию всего компилятора.

При проектировании типа AST следует учитывать способы его обработки и возможность последующей модификации. Типичный пример такой модификации — это добавление в него информации, необходимой компилятору для обработки ошибок.

При наивном конструировании синтаксического дерева и прямом вызове конструкторов типа в правилах разбора пришлось столкнуться с ситуацией, когда модификация типа AST привела к необходимости доработки и повторного тестирования синтаксических правил.

Важным уроком здесь стало то, что конструкторы алгебраических типов данных не являются функциями, и их поведение невозможно определять произвольно. Следовательно, было бы правильнее обернуть их в соответствующие функции и использовать эти функции при генерации AST. В этом случае модификация типа AST затронула бы только этот тип данных и тонкую прослойку функций-конструкторов.

Распространенным способом обработки синтаксического дерева в функциональных языках является рекурсивный обход в сочетании с сопоставлением с образцом; это вообще одна из часто встречающихся при разработке на этих языка идиом.

Оказалось полезным разрабатывать AST так, чтобы его тип был как можно более удобен для подобной обработки. Наивная реализация на рекурсивных типах, которую любят приводить в различных учебных курсах и примерах, привела к очень запутанным клозам сопоставления с образцом и сложностям при рекурсивном обходе, что в

[4] Abstract syntax tree, «абстрактное синтаксическое дерево».

свою очередь повлекло за собой несколько тяжелых рефакторингов, пока структура AST не пришла к виду, достаточно удобному для обхода и перестроения.

Разумеется, для успешной разработки AST необходимо определить, из чего состоит сам язык и как он устроен.

Бип на текущий момент состоит из следующих основных элементов:

Модули , каждый из которых представляет собой список определений.

Определения , которые могут быть декларациями функций, внешних функций, типов или макроопределений.

Операторы — основные примитивы языка. Они определяют синтаксические конструкции языка и не возвращают значений.

Блоки представляют собой последовательности операторов, разделённые символом «**;**» (точка с запятой).

Выражения Выражение есть некая операция, возвращающая значение и, следовательно, имеющая тип.

Макроопределения Макроопределение есть идентифицированный блок кода. На этапе раскрытия макроподстановок идентификатор кода в AST заменяется самим кодом. На текущий момент таким образом реализуются только именованные константы, но инфраструктура для макроопределений вполне настоящая, более сложные макроопределения оставлены на следующие фазы развития языка.

Исходный код описания AST:

```
type ast_top   = Module of mod_props * parser_ctx
and block      = Block of blk_props * parser_ctx
and definition =
    | FuncDef of func_props * parser_ctx
    | ExternFunc of (name * beep_type)
    | TypeDef of (name * beep_type) * parser_ctx
    | MacroDef of macro * parser_ctx
and statement  =
    | StEmpty of parser_ctx
    | StLocal of (name * beep_type) * parser_ctx
    | StArg of (name * beep_type) * parser_ctx
    | StAssign of expression * expression * parser_ctx
    | StWhile of (expression * block) * parser_ctx
    | StIf of if_props * parser_ctx
    | StBranch of (expression * block) * parser_ctx
    | StBranchElse of block * parser_ctx
    | StCall of expression * parser_ctx
```

```
    | StRet of expression * parser_ctx
    | StBreak of parser_ctx
    | StContinue of parser_ctx
    | StEmit of Opcodes.opcode list
and expression =
    | ELiteral of literal * parser_ctx
    | EIdent of name * parser_ctx
    | ECall of (expression * expression list) * parser_ctx
    | EAriphBin of operation * (expression * expression) * parser_ctx
    | EAriphUn  of operation * expression * parser_ctx
    | ECmp      of operation * (expression * expression) * parser_ctx
    | EBoolBin  of operation * (expression * expression) * parser_ctx
    | EBoolUn   of operation * expression * parser_ctx
    | EListNil  of parser_ctx
    | EList     of (expression * expression ) * parser_ctx
    | EPair     of (expression * expression ) * parser_ctx
    | ERecord   of (name * expression list) * parser_ctx
    | ERecordFieldInit of (name * expression ) * parser_ctx
    | ERecordField of rec_desc * rec_field
    | ENothing
    | EVoid of expression
    | EQuot of name * parser_ctx
and lvalue     = Named of name * parser_ctx
and rec_desc   = Rec of expression
and rec_field  = RecField of name * parser_ctx
and operation  =   Plus | Minus | Mul | Div | Mod | BAnd | BOr | BXor | BShl |
    BShr | BInv
| Less | More | LessEq | MoreEq | Equal | Unequal | And | Or | Not
and literal    = | LInt of int | LBool of bool | LString of string
and mod_props  = { mod_defs:definition list }
and func_props = { func_name:name; func_type:beep_type; func_code:block }
and blk_props  = { blk_code:statement list; }
and if_props   = { if_then:statement; if_elif:statement list; if_else:statement
    }

and macro      = MacroLiteral of (name * expression)
```

Валидация AST

После того, как осуществлен разбор исходного кода, требуется проверить корректность построенного AST с точки зрения семантики.

На самом деле, «после» — не совсем правильное слово. Бип устроен таким образом, что подавляющее большинство проверок осуществляется при разборе исходного текста. Это возможно благодаря специальным усилиям при дизайне AST, подходу к описанию синтаксиса и минимализму языка.

Те конструкции, которые неудобно проверять таким образом, проверяются путем обычного сопоставления с образцом при генерации промежуточного кода из AST.

Типичные примеры валидации: проверка корректности управляющих конструкций (например, того, что операторы continue и break находятся внутри циклов) и проверка корректности выражений (например, того, что в левой части оператора присваивания находится lvalue, т.е. значение, которому можно что-либо присваивать).

В качестве примера можно рассмотреть следующий код, генерирующий оператор присваивания.

```
and assignment e1 e2 ct =
    match e1 with
    | EIdent(n,c) → ct |> add_expr e2
            |> add_code (store (get_var ct (n, id_of c)))
    | ERecordField(Rec(re),RecField(fn,c2)) →
    let fi = typeof_name (dot fn,id_of c2) ctx
    in let rt = field_rec_type fi
    in let off = rec_field_offset rt fn
    in ct |> add_expr re
          |> add_expr e2
          |> add_code (op (TBCWD(off)) ~comment:(dot fn) :: [])
    | other → raise (Type_error("lvalue required"))
```

Оператор |> можно назвать «конвейерным оператором», который определен как

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

Это очень часто встречающаяся в программах на OCaml идиома, которая позволяет представить последовательность действий в виде конвейера, где результаты предыдущего этапа передаются на вход текущему, аналогично тому, как это можно делать в командной строке при помощи оператора | («пайп»):

```
find . -name '*.c' | xargs cat | wc -lc
```

В нашем случае, запись

```
st |> push_loop e |> nest
```

можно интерпретировать как последовательность операций: «взять начальный контекст, добавить цикл, добавить вложенный контекст».

Приведённая функция генерирует промежуточный код для оператора присваивания, принимая два узла AST типа «выражение» и контекст. В левой части присваивания в Бипе в настоящий момент может быть только выражение типа «идентификатор» или «ссылка на поле структуры», остальные типы выражений приведут к генерации ошибки.

Пример использования данной функции при генерации кода (и весь верхний уровень генератора кода):

```
in let code_fold st code =
    match code with
    | StArg((n,t),c)       → st |> add_arg n
    | StLocal((n,t),c)     → st |> add_loc n
    | StAssign(e1,e2,c)    → st |> assignment e1 e2
    | StWhile((e,_),_)     → st |> push_loop e |> nest
    | StCall(e,_)          → st |> add_expr (EVoid(e))
    | StIf({if_elif=ef},_) → st |> push_if |> nest
    | StBranch((e,_),c)    → st |> branch (Some(e)) CBr (id_of c) |> nest
    | StBranchElse(_,c)    → st |> branch None CBrElse (id_of c)  |> nest
    | StRet(e, _)          → st |> ret e
    | StContinue _         → st |> continue
    | StBreak _            → st |> break
    | StEmpty _            → st
    | StEmit(l)            → st |> add_emit l
```

Можно заметить, что генератор кода ориентирован на стековую машину.

Делать какие-либо дополнительные проверки необходимости не возникло.

Очень часто упоминается сложность отладки парсеров, генерируемых автоматическими построителями. Могу сказать, что, благодаря сильной статической типизации и дизайну типа AST, удалось практически совсем избежать отладки. Во всяком случае, на анализ логов парсера ушло пренебрежимо мало времени. После разработки сходного проекта на динамическом языке такой результат кажется неправдоподобным.

Построение словаря

Бип обладает следующими областями видимости:

Модуль Имена модуля (функции, типы, макроопределения).

Функция Формальные аргументы функции и переменные верхнего блока.

Блок Каждый блок может содержать объявления переменных.

Подход к областям видимости совершенно традиционен и похож на Си, Java и другие подобные им языки. Любая область видимости может содержать объявления, перекрывающие имена вышестоящих областей видимости.

Для того, чтобы различать одинаковые имена, принадлежащие разным областям видимости, пришлось добавить уникальные идентификаторы на уровне AST, которые генерируются во время разбора. Таким образом, словарь состоит из пар вида:

((имя, идентификатор), дескриптор)

где *дескриптор* содержит необходимую компилятору информацию об идентификаторе.

Вывод и проверка типов

Для вывода типов в Бипе используется алгоритм Хиндли–Милнера, как наиболее простой и хорошо описанный в литературе и сети. Существует большое количество примеров его реализации на различных языках.

Очень поверхностно идею данного алгоритма можно описать так: каждой переменной, тип которой следует определить, мы сначала присваиваем некий уникальный тип, затем собираем ограничивающие условия, и получаем, фактически, систему уравнений, которую решаем методом *унификации*.

Структура типов данных в языке описывается при помощи алгебраических типов OCaml. Например, атомарные типы данных:

```
| TInt | TString | TBool
```

или составные типы данных:

```
| TPair of beep_type * beep_type
| TList of beep_type
| TVect of beep_type
```

или тип «полиморфный параметр»:

```
| TAny of int
```

или тип «неизвестный тип данных»:

```
| TVar of int
```

Типы `TVar` и `TAny` очень похожи, наличие обоих типов в реализации типизации обусловлено необходимостью отличать автоматически введённые переменные типов от параметров полиморфных функций.

Ограничения представляют собой равенства вида:

$$x_i \quad = \quad A,$$

где слева находится автоматически введённый «неопределённый тип данных», а справа — условие, полученное из анализа выражений, в которых данный тип участвует, и явных деклараций типов, которые язык также допускает.

Операторы и выражения накладывают определённые ограничения на типы операндов. Арифметические и битовые операции подразумевают, что их операнды и возвращаемые значения являются целыми числами. Логические операции подразумевают булев тип данных, операции и функции над списками требуют аргумента типа список (и являются полиморфными, так как список — составной тип данных). Циклы и оператор ветвления требуют булева типа в условии, а оператор присваивания декларирует, что типы переменных в левой и правой частях присваивания одинаковы.

Процесс решения системы уравнений заключается в выводе всех типов `TVar` и `TAny` через типы, не содержащие значений неизвестных и полиморфных типов.

Для того, чтобы алгоритм унификации мог обрабатывать наши произвольные типы, для каждого типа требуется определить *правило унификации* и *правило проверки*

вхождения одного типа в другой, а также способ замены типов в определении каждого составного типа.

Система уравнений может быть решена не всегда, как например, в случае типов, определения которых содержат циклические ссылки (таких как рекурсивные типы данных) либо противоречивые условия:

$$\begin{aligned} x_1 &= A \\ x_2 &= x_1 \\ x_2 &= B \end{aligned}$$

Проверка типов осуществляется в два основных этапа. Алгоритм унификации обнаруживает наличие ошибок типов и, в случае, если система уравнений не решаема, порождает исключение.

Второй класс ошибок — отсутствие достаточной информации для выведения типов, т.е. случаи, когда не удаётся устранить все типы `TVar` и `TAny`.

Данные ошибки обнаруживаются на этапе формирования промежуточного представления кода, так как генератор кода может произвести только конструкции для известных ему типов, за исключением нескольких случаев, где полное определение неважно (например, функциям для работы со списками не существенно, списки чего именно имеются ввиду). Если при генерации кода встречается значение неизвестного или недоопределённого типа, порождается исключение.

Никаких других специальных проверок в нашем случае не требуется. Это важный момент: реализовав вывод типов, проверку типов мы получили бесплатно. У такого способа типизации есть и свои недостатки, наиболее очевидные из них следующие:

Жёсткая типизация операций, например, арифметических. Если у нас есть целочисленные операции

```
+ - * /
```

то для того, чтобы пользоваться такими операциями для чисел с плавающей точкой, нам придётся либо ввести для них другие обозначения, например:

```
+. -. *. /.
```

либо сделать их полиморфными. При этом они станут бесполезны для выведения типов своих операндов, что сократит количество случаев успешного автоматического выведения типов и потребует большего количества аннотаций. Можно ввести *классы типов* и применять другой, более сложный алгоритм выведения. С подобной проблемой сталкиваются и «большие» языки. При этом OCaml, например, выбирает наиболее простое решение — использование разных обозначений для таких операций. Тот же путь приемлем и для нашего языка.

Трудность реализации рекурсивных типов данных. При буквальной реализации алгоритма Хиндли–Милнера невозможно реализовать рекурсивные типы данных, так как в этом алгоритме существует проверка на циклические ссылки, т.е. определение типов через самих себя. Поддержка таких типов требует их распознавания и отдельной обработки. Реализация данной функциональности отнесена к следующим фазам разработки языка.

Подробно теория систем типов изложена в книге Benjamin C. Pierce Types and Programming Languages. Примеры из этой книги исключительно полезны для понимания различных аспектов типизации.

Оптимизация на уровне AST

Многие оптимизации удобно проводить на уровне AST в случае, если дерево сохраняет семантику языка, и её не требуется реконструировать.

Например, на этом уровне очень несложными видятся оптимизация хвостовых вызовов (фактически на уровне клозов сопоставления с образцом) и вычисление константных выражений, которое можно рассматривать как интерпретацию AST с заменой вычислимых во время компиляции выражений на их значения.

В настоящее время компилятор языка Бип поддерживает лишь небольшое количество простых оптимизаций — только те, реализация которых не требовала большого количества времени. Все они производятся на этапе генерации промежуточного кода из AST, например:

- Замена последовательности операций сложения с единицей и присваивания на инкремент.
- Замена последовательности вычитания единицы и присваивания на декремент.
- Замена целого литерала '0' на команду VM 'FALSE' размером один байт.
- Замена целого литерала '1' на команду VM 'TRUE' (аналогична предыдущей).

Отметим удобство механизма сопоставления с образцом в данном случае: он позволяет сначала реализовать общую схему генерации кода, добавляя обработку частных случаев по мере необходимости.

Типичные примеры — замена сложения инкрементом и вычитания декрементом:

```
| EAriphBin(Plus, (_,ELiteral(LInt(1),_)),_) → repl_inc st
| EAriphBin(Plus, _, _)     → st @ [op ADD]

| EAriphBin(Minus,(_,ELiteral(LInt(1),_)),_) → repl_dec st
| EAriphBin(Minus,_, _)     → st @ [op SUB]
```

операция '@' в OCaml — конкатенация списков

Генерация промежуточного кода

В нашем случае промежуточным кодом является представление байткода виртуальной машины в виде структур данных языка реализации, то есть элементов алгебраического типа данных.

На этом этапе генерируется представление конструкций языка в виде команд виртуальной машины, происходит вычисление адресов условных и безусловных переходов, генерируются данные для константных строк и вычисляются их смещения в сегменте байткода.

Как уже упоминалось выше, на этом же этапе происходят проверки определённости типов данных, а также те семантические проверки, которые невозможно реализовать на уровне парсера.

Код генерируется только для конструкций с корректной семантикой и операций с полностью определёнными типами (за исключением нескольких частных случаев), в остальных случаях порождаются исключения.

Оптимизация на уровне кода

Полученное представление кода тоже требует оптимизации, включая устранение артефактов генерации кода, таких как лишние команды NOP, устранение бессмысленного кода, такого как сочетание команд вида:

```
JMP X
JMP Y
```

а также оптимизации, которые просто удобнее делать на этом уровне. Например, на этапе генерации кода удобнее трактовать все вызовы как вызовы по переменному адресу, который помещается на вершину стека. Поскольку виртуальная машина поддерживает вызов по константному адресу одной командой, для увеличения производительности и уменьшения объема кода лучше преобразовать последовательность команд:

```
    LIT X
    CALLT

B:

    CALL X
```

Генерация выходных файлов

Сгенерированная модель кода преобразуется в бинарный вид с учетом особенностей целевой платформы, например endianness.

Генерация стабов

Поскольку язык имеет возможности FFI[5] для связи с функциями на низкоуровневых языках порождаются различные стабы, предназначенные для линковки с кодом виртуальной машины — обертки функций, обертки структур (осуществляющие отображение структур Бипа на структуры Си) и таблицы опкодов виртуальной машины.

1.3.5. Производительность и оптимизация

Поддержанию производительности компиляции на приемлемом уровне уделялось достаточно много внимания. Главной особенностью являлось периодически появлявшееся экспоненциальное поведение в самых разных местах компилятора. Оно проявлялось и при построении словаря, являясь просто алгоритмической ошибкой, но особенно много головной боли доставила система вывода типов.

Унификация — не самый дешевый в смысле производительности алгоритм, так что важно поддерживать количество данных, участвующих в унификации, минимальным и не допускать повторных вычислений того, что уже вычислено.

Этот момент стоит отметить особо, так как большую проблему с производительностью создала реализация полиморфных функций. Дело в том, что если мы полностью вычисляем типы для каждого выражения каждый раз, как оно встречается, то реализация полиморфных функций тривиальна — мы просто используем выведенные значения типов параметров и возврата полиморфной функции в том выражении, где мы её используем, не сохраняя значений для вычисленных типов. Функция действительно получается полиморфной, её тип различен в разных контекстах. Но перевычисление всех типов для каждого выражения приводит к абсолютно неприемлемой вычислительной сложности. Чтобы этого избежать, пришлось пойти на достаточно нетривиальные меры.

В остальном обычный профайлинг и не очень агрессивная мемоизация позволили удержать производительность на уровне, неформально определённом нами как «менее секунды на любом файле, который в состоянии написать человек». Это означает, что как только на каком-либо скрипте время компиляции превышает секунду, берётся в руки профайлер, и эта проблема устраняется. До сих пор это получалось.

Стоит упомянуть, что разбор исходного файла парсером, сгенерированным ocamlyacc, происходит очень быстро, затраты времени на него примерно на два порядка ниже, чем на последующую обработку AST, и парсер еще ни разу не стал объектом оптимизации.

Достаточно интересен и тот факт, что не возникло необходимости использования деструктивных алгоритмов и структур данных в целях оптимизации. Имевшие место попытки не принесли заметных результатов. Несмотря на использование хешей, которые в OCaml не являются чистыми и реализуют деструктивные присваивания,

[5]Foreign Function Interface, интерфейс для вызова функций, реализованных на других языках, в нашем случае на C или ассемблере.

программа остается чистой, так как хеши используются иммутабельным образом — один раз создаются и далее применяются только для поиска.

В целом производительность компилятора следует признать удовлетворительной, хотя до идеала далеко — в частности, компилятор OCaml работает существенно быстрее.

1.3.6. Дизайн рантайма

Рантайм представляет собой двухстековую виртуальную машину, оптимизированную для шестнадцатибитной архитектуры. Текущей платформой является микроконтроллер MSP430 от Texas Instruments, но существует и версия, запускающаяся на PC, используемая в настоящий момент для прототипирования и отладки логики скриптов.

Стек A представляет собой стек данных, стек R — стек для адресов возврата и служебных данных. В отличие от Форта, в Бипе прямой доступ к стеку R невозможен, виртуальная машина управляет им сама.

Стек A разделен на фреймы. В начале каждого фрейма находятся ячейки, зарезервированные для локальных переменных и формальных параметров функции, вершина стека используется для вычислений. Каждый фрейм создаётся соответствующими инструкциями в прологе вызова функции, после возврата из функции восстанавливается предыдущий фрейм.

Существуют отдельные инструкции для доступа к зарезервированным ячейкам стека. Восемь первых ячеек выделены, и инструкции для доступа к ним не имеют литералов, то есть имеют размер в один байт. Эти инструкции приводят к копированию содержимого значений локальных переменных или параметров на вершину стека A, либо к записи вершины стека в ячейки локальных переменных.

На текущий момент виртуальная машина насчитывает 73 команды.

Опкоды имеют разрядность 8 бит, литералы — 16 бит.

Память адресуется исключительно словами. Помимо расширения пространства доступной памяти для адресов, ограниченных шестнадцатью битами, это решение ускоряет и упрощает сборку мусора.

VM имеет не фоннеймановскую архитектуру — куча, стеки и код находятся в разных виртуальных адресных пространствах. Такой дизайн вызван особенностями микроконтроллеров, для которых, в первую очередь, предназначен Бип. В то же время, это решение никак не затрагивает язык и компилятор — это деталь реализации, скрытая от пользователя.

Данные, умещающиеся в слово, размещаются на стеке A, а данные, превосходящие в размере слово, размещаются в динамической памяти.

Динамическая память (куча) управляется автоматически: реализован консервативный сборщик мусора, важной особенностью которого является отсутствие затрат памяти на обход графа объектов.

Не используются ни стек, ни дополнительное поле заголовка для применения техники *reversed pointers*. Использование такого алгоритма ухудшает асимптотику алго-

Параметр	Размер	Источник ограничения
Code memory	≈3 килобайта	реализация VM
Системный стек + RAM	< 100 байт	реализация VM
Стек A	128 слов	определяется пользователем
Стек R	64 слова	определяется пользователем
Куча	4096 слов	определяется пользователем

Таблица 1.1. Приблизительные требования виртуальной машины к ресурсам

ритма обхода, но, ввиду отсутствия глобальных переменных, периметр сборки мусора определяется только текущей глубиной стека A, и в некоторые моменты память может быть очищена за гарантированное время O(1).

Куча организована в виде связного списка свободных и занятых блоков, накладные расходы на управление памятью — одно слово на блок. Максимальный размер блока — 8191 слово, что превышает количество RAM на типичном представителе целевой архитектуры.

Микроконтроллер	Частота	Code memory	RAM	… в т. ч. куча
MSP430F1612	7.3728 МГц	55 КБ	5 КБ	1 КБ
MSP430F5418	7.3728 МГц	41 КБ[6]	16 КБ	до 14 КБ

Таблица 1.2. Типичные используемые конфигурации

Производительность виртуальной машины можно характеризовать как «достаточно приличную». Оценивать её имеет смысл только в сравнении с другими, но на целевой платформе сравнивать не с чем, а сравнение с PC будет происходить в слишком неравных условиях, так как рантайм Бипа ориентирован на очень маленькое количество оперативной памяти, и её экономия имеет больший приоритет, чем производительность. И сборка мусора, и выделение памяти ориентированы на минимизацию накладных расходов по памяти, а не на максимальную производительность.

Тем не менее, сравнение работы приблизительно одинакового алгоритма синтаксического разбора строки NMEA[7] на Lua, Python и Веер показало, что даже в условиях, когда в VM Веер за цикл работы скрипта GC вызывается более десяти тысяч раз,[8] Бип оказывается в три — пять раз быстрее Python, и в полтора — два раза быстрее Lua. Отнести такие результаты можно на счет статической типизации, при которой отсутствуют накладные расходы на проверку типов во время исполнения.

[6]Расширенная память не используется.

[7]Стандарт на обмен данными для GPS устройств.

[8]При ограничении кучи в 1 КБ.

1.4. Окончательный вид языка

В результате нескольких итераций разработки компилятора и рантайма, внёсших свои коррективы в понимание и дизайн, язык был приведен к стабильному состоянию. Интересно, что язык не получалось стабилизировать до тех пор, пока он не включил в себя некий минимум, присущий языкам подобного класса.

Типы данных

Язык поддерживает следующие встроенные типы данных:

Int Целочисленный тип шириной в *слово*. Может быть знаковым и беззнаковым. Символы и байты также представляются числом.

String Строка, интерпретируемая как последовательность символов (чисел). В текущей реализации предполагается, что символ не превышает в разрядности один байт, строка хранится в упакованном виде (каждое слово содержит два символа). Строки являются иммутабельными, одинаковые константные строки в программе являются ссылками на одну и ту же строку.[9]

Bool Логический тип данных. Управляющие конструкции и логические операторы оперируют значениями этого типа.

Pair Кортеж элементов разных типов размерностью 2.

List Список однородных элементов.

Vector Массив однородных элементов с произвольным доступом.

Fun Функция.

Record «Запись», аналог структуры в языках C, C++.

Литералы

Поддерживаются численные шестнадцатеричные, десятичные и строковые литералы, а также литералы «символ», транслируемые в **`Int`**.

Переменные

Переменные объявляются оператором local и могут содержать спецификации типов. Переменные могут объявляться в любом месте блока и являются только локальными. Глобальных переменных в Бипе не существует. При объявлении каждая переменная должна быть инициализирована значением, попытка объявить переменную

[9] Константы размещаются в памяти кода, не занимая память кучи.

без её инициализации приводит к ошибке компиляции. В Бипе также не существует значений, подобных *null*, *None* или *undefined* в других языках.

Операции

Бип обладает достаточно стереотипным набором операций: арифметические, битовые и логические, операции присваивания, декларирования переменной и доступа к полю записи.

Все операции типизированы, например, логические операции принимают и возвращают значения типа Bool.

Оператор сравнения полиморфен и может принимать значения типов **`Int`** и **`String`**. В последнем случае компилятор генерирует вызов встроенной функции `strcmp`, которую можно вызвать и напрямую.

Можно также отметить операцию конструирования списка **`::`**, аналогичную соответствующей конструкции языка OCaml, которая создает новый список из указанных головы и хвоста.

Встроенные функции

Бип содержит некоторое количество встроенных функций, входящих, если можно так выразиться, в стандарт языка. Встроенными функциями реализуются работа со строками, списками, парами и массивами.

Управляющие конструкции

Язык включает минимум управляющих конструкций: цикл с условием `while`, условный оператор `if-elif-else` и операторы `break` и `continue`, прерывающий цикл и переходящий к следующей итерации цикла, соответственно.

Условный оператор и оператор цикла требуют в качестве условия выражение типа Bool.

Блоки

Блоки являются последовательностями операторов, разделённых символом «**`;`**» (точка с запятой).

Функции

Функции объявляются ключевым словом `def` и вызываются при помощи оператора **`()`**.

Функции в Бипе являются первоклассным типом,[10] могут присваиваться переменным, передаваться как параметры функций и возвращаться из них, помещаться в списки и массивы и так далее.

При объявлении функции можно специфицировать типы её аргументов и тип возвращаемого значения, в противном случае эти типы будут выведены.

Макрокоманды

На текущий момент язык поддерживает макроопределение только для литералов.

FFI

Компилятор языка умеет автоматически генерировать необходимые стабы для вызова внешних функций, а также обертки для записей в виде структур языка Си.

Обработка ошибок

Бип является языком с сильной статической типизацией, вследствие чего скрипты на нем гарантированно корректны в отношении типов. Исключения на текущий момент отсутствуют, ошибки времени исполнения, такие как переполнения стеков, обрабатываются на уровне API рантайма.

1.5. Примеры скриптов

Hello, world!

```
def main() {
    putsn(”Hello, world!”);
}
```

Создадим и распечатаем список пар строк

Ради разнообразия здесь мы декларируем типы явно.

```
def print(val:(string,string)):void {
    puts(fst(val));
    puts(snd(val));
}

def main() {
    local l = (”B”,”E”)::(”E”,”P”)
            ::(”R”,”U”)::(”L”,”Z”)::[];
```

[10] Насколько это возможно без реализации замыканий.

```
    local t = l;
    while !nil(t) {
        print(head(t));
        t = tail(t);
    }
}
```

Функции — почти совсем первоклассные граждане

```
def print_smth() {
    putsn(”BEEP RULZ!”);
}

def print(x: fun(void):void ) {
    x();
}

def main() {
    local l:[fun(void):void] = print_smth
                             :: print_smth
                             :: print_smth :: [];
    local t = l;
    while !nil(t) {
        print(head(t));
        t = tail(t);
    }
}
```

Работаем с приемником GPS

Чтобы предоставить читателю возможность проникнуться атмосферой продукта, приведу часть реального боевого скрипта:

```
# FFI - declaring external functions
# stubs are generated automatically

@extern gps_power(bool):void;
@extern nmea_read() : string;
@extern seconds(void):int;

type gps_data {
     gps_utc:string
    ,gps_fx:int
    ,gps_sat:int
    ,gps_hdop:string
    ,gps_lat:string
```

```
    ,gps_lats:int
    ,gps_lon:string
    ,gps_lons:int
}

def str_ntok(s,seps,n) {
    local len  = strlen(s);
    local len2 = vect_len(seps);
    local off = 0, size = 0;

    if n == 0 then {
        off  = 0;
        size = vect_get(seps,0);
    }
    elif n >= len2 then {
        n   = len2;
        off  = vect_get(seps, len2-1) + 1;
        size = len - off + 1;
    }
    else {
        off  = vect_get(seps,n-1) + 1;
        size = vect_get(seps,n) - off;
    }

    ret strsub(s, off, size);
}

def collect_gps_data() {
    local fx   = 0;
    local utc  = "";
    local sat  = 0;
    local hdop = "";
    local lat  = "";
    local lats = "";
    local lon  = "";
    local lons = "";
    local i = 0;
    local timeout = 30;
    local t1 = seconds(), dt =0;
    while i < 10 && timeout != 0 {
        dt = seconds() - t1;
        t1 = seconds();
        if timeout >= dt then timeout = timeout - dt;
                         else timeout = 0;
        local s = nmea_read();
        if s != "" then {
```

```
            #putsn(s);
            local sep  = strfindall(s,',');
            if startswith(s, "$GPGGA") then {
                fx    = strtoul(str_ntok(s,sep,6),16);
                sat   = strtoul(str_ntok(s,sep,7),16);
                utc   = strsub(str_ntok(s,sep,1),0,6);
                lat   = str_ntok(s,sep,2);
                lats  = str_ntok(s,sep,3);
                lon   = str_ntok(s,sep,4);
                lons  = str_ntok(s,sep,5);
                hdop  = str_ntok(s,sep,8);
                i = i + 1;
            }
        }
        if fx > 0 then break;
    }
    local ss = strnth(lats, 0);
    ret { gps_data:
             gps_utc  = utc
            ,gps_fx   = fx
            ,gps_sat  = sat
            ,gps_hdop = hdop
            ,gps_lat  = lat
            ,gps_lats = strnth(lats,0)
            ,gps_lon  = lon
            ,gps_lons = strnth(lons,0)
        };
}

def main() {
    putsn("GPS ON");
    gps_power(true);

    while true {
        local nmea = collect_gps_data();
        if nmea.fx > 0  then {
            putsn("Coords fixed")
            puts("Satellites: ");
            putsn(utoa(nmea.gps_sat, 16));
            puts("Latitude: ");
            putsn(nmea.gps_lat);
            puts("Longitude: ");
            putsn(nmea.gps_lat);
        }
    }
}
```

Модем (и макросы)

И напоследок, поработаем с модемом и немного с макросами:

```
@extern  gps_power(bool) : void;
@extern  modem_power(bool) : void;
@extern  modem_power_check(void) : bool;
@extern  modem_init(int) : bool;
@extern  modem_ussd(string, int) : string;
@extern  modem_sms_send(string, string) : bool;
@extern  nmea_read() : string;
@extern  seconds(void) : int;

@literal INITIAL 0;
@literal DIGIT   1;
@literal NOPINCODE 0xFFFF;

def parse_account(s) {
    local i = 0, begin =0, end = 0;
    local len  = strlen(s);
    local state = `INITIAL;
    while i < len {
      local c = strnth(s, i);
      if state == `INITIAL &&  c >= '0' && c <= '9'
         then { state = `DIGIT; begin = i; }
      if state == `DIGIT   &&  c == '.'
         then { end = i; break; }
      if state == `DIGIT   &&  c < '0' || c > '9'
         then state = `INITIAL;
      i = i + 1;
    }
    if begin >= end then ret (false,0);
    ret (true,strtoul(strsub(s,begin,end-begin),10));
}

def main() {
    gps_power(true);
    modem_power(true);
    putsn("MODEM ON");
    local i = 0;
    while i < 20 {
        puts(".");
        sleep_ms(1000);
        i = i + 1;
    }
    modem_init(`NOPINCODE);
    sleep_ms(2000);
```

```
    local money = snd(parse_account(
                        modem_ussd("#102#",32)));
    puts("MONEY: ");
    put_int(money);
    sleep_ms(2000);
    modem_sms_send("+71234567890"
                   "PRIVED, KRIVEDKO!");
}
```

1.6. Итоги

Практическое применение языка

Разумеется, Бип был разработан вовсе не как фан-проект для обучения написанию компиляторов. И его предтеча, Форт, и он сам применялись в разрабатываемых системах, даже будучи не совсем стабильными, развиваясь параллельно с основными системами.

Основные цели, которые ставились при разработке Бипа, были достигнуты, а по отдельным параметрам он даже превзошел связанные с ним ожидания.

Разрабатывать скрипты на нем оказалось гораздо быстрее и проще, чем, например, писать код на Си, что привело к тому, что часть логики работы устройства, которая изначально планировалась к реализации в прошивке, реализуется в скрипте. Это может быть не слишком хорошо с точки зрения архитектуры, но зато ощутимо экономит время.

Еще один немаловажный аспект: байткод Бипа плотнее, чем машинный код, генерируемый компилятором Си. В то время, когда ресурсы code memory, отведённые под прошивку (около 32 КБ flash), практически исчерпаны, 8 КБ сегмента, отведённого под байткод, еще имеют резервы. В связи с этим нехватка функциональности прошивки вполне может быть компенсирована за счет скрипта.

Применение Бипа позволило добиться таких характеристик системы, как надежное удалённое изменение поведения устройств и отсутствие необходимости в фиксированных протоколах работы: протокол реализуется скриптом и может быть любым, удобным для конкретного применения устройства.

В некоторых случаях устройствам, несущим на борту Бип, даже не требуется никакой специальной серверной инфраструктуры. В случае, если их в системе не очень много, и им достаточно работы по GPRS/HTTP, трекеры могут работать на общих основаниях с остальными пользователями веб-приложения, что помогает значительно упростить интеграцию уже существующими системами.

HTTP-клиент, разумеется, также реализован на Бипе; ничто не мешает реализовать SOAP или XMLRPC, если возникнет такая необходимость.

Обновление скрипта может происходить различными способами, наиболее простой из них — HTTP с докачкой.

Удалённое обновление скрипта уже многократно использовалось при пользовательском тестировании, позволяя устранять различные проблемы на лету, практически между двумя событиями трекинга, менее чем за 30 секунд. Пользователи даже не замечали, что произошло что-то особенное.

Стоит упомянуть, что принятые в дизайне языка и рантайма решения — строгая статическая типизация, отсутствие неинициализированных переменных, отсутствие рантайм-исключений[11] — полностью окупились: за полгода пользовательского тестирования не зафиксировано ни одного падения прошивки устройств, вызванного дефектами дизайна рантайма или компилятора.[12] При этом не потребовалось разработки эмуляторов и длительного тестирования скриптов на них — скрипты пишутся сразу и тестируются непосредственно на самих устройствах. Благодаря типизации есть уверенность, что скомпилировавшийся скрипт будет нормально работать и не приведет к потере связи с устройством, что не может быть гарантировано при использовании, например, Форта или гипотетических динамических языков.

OCaml в качестве языка разработки

Выбранный инструментарий безусловно оправдал себя. OCaml — очень удачный выбор для тех, кто только начинает применять функциональные языки программирования. Он крайне прост в освоении и позволяет очень быстро начать получать результаты, не углубляясь в дебри и не теряя темпа разработки.

В рамках данной задачи именно OCaml оказался идеальным инструментом — остальные альтернативы неизбежно привели бы к сильному проигрышу в сроках, что в нашем случае было эквивалентно смерти проекта, так как он мог выжить только в случае очень быстрого появления хотя бы proof-of-concept реализации, которая показала бы, что язык с требуемыми характеристиками вообще возможно разработать за реалистичное время имеющимися силами.

Количество кода живых проектов имеет тенденцию постоянно увеличиваться. Чтобы этот рост контролировать, необходим рефакторинг, который, в свою очередь, требует плотного покрытия кода тестами, требующими времени на написание и поддержку.

Невозможно переоценить, насколько упрощается задача рефакторинга в случае применения языка с сильной типизацией.

Рост размера кода, увеличение его структурной сложности, количества предположений, соглашений и взаимосвязей, уменьшение его понятности — очень существенные негативные факторы, которые нельзя игнорировать.

[11] Управляемые исключения не противоречат концепции системы, здесь имеются ввиду исключения вида NullPointerException, исключения при ошибках типов и тому подобные, которыми «радует» своих пользователей, например, Java, и которые были бы фатальны для устройства. Декларируемые, управляемые исключения с гарантированной обработкой компилятором вполне допустимы и не были реализованы исключительно из-за недостатка времени.

[12] Разумеется, падения из-за ошибок реализации присутствовали и устранялись в процессе разработки.

Можно привести такой пример: существуют довольно интересные языки программирования Нахе и Boo. Они во многом похожи, например, строгой статической типизацией и выводом типов. Первый язык реализован преимущественно на OCaml, второй — на C#.

Реализация системы типов в Boo, написанном на C#, занимает более семнадцати тысяч строк кода (более половины мегабайта кода на высокоуровневом языке), размещающихся в ста двадцати двух файлах. При этом представлявший наибольший интерес алгоритм вывода типов надежно декомпозирован на слои и «*шаблоны проектирования*» и код, который его реализует является вполне идиоматичным для этого класса языков. Задача идентифицировать основной код алгоритма и понять его не выглядела решаемой за разумное время и была оставлена.

Ни одной понятной реализации системы типов и алгоритма выведения на императивных языках найдено не было (были рассмотрены варианты на C#, C++ и даже Perl).

Реализация системы типов в языке Нахе, занимает 4088 строк (127624 байт кода) на OCaml и размещается в трех файлах, при этом интересующий алгоритм идентифицируется сразу же и представляет собой, в основном, свертку списков с применением сопоставления с образцом. Прочитать и понять его было достаточно легко, несмотря на то, что на тот момент опыт разработки и чтения кода, написанного на императивных языках, сильно превосходил аналогичный опыт для функциональных.

Даже со всеми возможными оговорками разница в функциональности систем типов Boo и Нахе гораздо меньше, чем разница в количестве кода, эту функциональность реализующего: 100072 строки (2955595 байт) кода текущей реализации Boo убивали всякую надежду, что подобный проект может вообще быть реализован самостоятельно.

Для сравнения, текущие значения для Бипа: 2592 строки (109780 байт) кода, а время, затраченное на реализацию вместе с рантаймом, не превышает трех человекомесяцев. Это маленький проект, и сделать его маленьким помог именно выбор OCaml в качестве языка разработки.

Количество строк — достаточно спорный критерий оценки, но это единственная метрика, которую можно получить из исходных текстов, затратив разумные усилия. В данном случае её использование правомерно, поскольку сравниваются частные и достаточно близкие по смыслу вещи. Реализуемые языки отличаются, но системы типов у них весьма похожи, а алгоритм выведения типов и вовсе один.

Не было никаких оснований полагать, что выбор низкоуровневого статического языка для реализации Бипа приведет к результатам, которые ближе к тем, что получились у авторов Нахе, чем к показателям авторов Boo.

Использование динамических языков, таких как Python или Ruby,[13] наверняка бы привело к разрастанию количества юнит-тестов и отказу от продолжения разработки проекта на этапе еще второго рефакторинга. Эта тенденция четко прослеживалась на

[13] Ruby не демонстрирует особенных преимуществ перед Python, но имеет ряд очень существенных недостатков, так что реально его рассматривать было бессмысленно.

нашем опыте использования Python в качестве основного языка разработки в различных проектах, и разработка транслятора Форта, речь о котором шла в начале статьи, её только подтвердила. Исследование возможности применения проекта PyPy (реализация транслятора Python на Python, а также инфраструктура для построения компиляторов на этом языке) тоже не убедило в обратном.

Никакое количество тестов не может гарантировать отсутствие ошибок. Между тем, сильная статическая типизация именно гарантирует отсутствие ошибок определённого класса.

Опыт применения Python выявил еще одну интересную тенденцию: переписывание сложных или проблемных участков кода в функциональном, иммутабельном стиле зачастую приводило не только к правильному функционированию кода, чего не получалось добиться от его императивного аналога, но и к сокращению его размеров.

В таком случае, зачем пытаться использовать функциональный подход в языках, которые его не поощряют, если есть функциональные языки? Языки, которые являются более безопасными в силу типизации, имеют дополнительные возможности и полноценные оптимизирующие компиляторы, генерирующие код, минимум на порядок превосходящий по производительности упомянутые динамические языки.

Последний фактор оказался весьма важен, так как скорость компиляции — это заметный фактор, влияющий на использование языка.

Принимая во внимание усилия, которые пришлось (и еще придется в дальнейшем) приложить для достижения приемлемого для работы времени компиляции, можно констатировать правильность этого выбора.

При выборе сыграла свою роль и оценка сложности развертывания приложений: в одном случае это установка большого количества зависимостей, начиная с рантайма динамического языка, в другом — возможность получить единственный самодостаточный исполняемый файл.

Попытки использовать Haskell также предпринимались, но он был с сожалением отложен, несмотря на все его возможности, которых недостаёт в OCaml. Довольно быстро стало ясно, что использование Haskell неподготовленным человеком может привести к затягиванию сроков, а кроме того, было неочевидно влияние его ленивости на разработку.

В заключение хотелось бы сказать, что не стоит рассматривать OCaml и Haskell как языки, предназначенные исключительно для разработки компиляторов — это отличные инструменты, подходящие для обширного круга задач. Эти языки предлагают удачный набор абстракций, который позволяет концентрироваться на решении задачи, не размениваясь на второстепенные цели типа обслуживания инфраструктуры[14] или то, что обычно называют строительством велосипедов. Они обладают развитым инструментарием[15] и набором библиотек, при этом позволяют генерировать код, по производительности не сильно уступающий, а иногда превосходящий код более распространённых низкоуровневых языков.

[14]Пресловутые «шаблоны проектирования».

[15]Если не понимать под ним исключительно IDE.

Использование Haskell при поддержке критически важной для бизнеса информационной системы

Дмитрий Астапов
adept@fprog.ru

Аннотация

Статья рассказывает о том, как язык функционального программирования Haskell использовался автором в качестве инструментального средства для решения прикладных задач, возникавших в процессе развития и поддержки критически важной для бизнеса информационной системы в рамках крупной телекоммуникационной компании.

The article describes how Haskell functional language proved instrumental in solving practical tasks arising during development and maintenance of a certain business-critical information system in a large telecom company.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/1985.html.

2.1. Обзор системы управления услугами

Рассказ стоит начать с краткого описания контекста, в рамках которого существовала некая критически важная для бизнеса информационная система и связанные с ней проблемы.

Описываемые события происходили восемь лет назад, в 2001 году. В то время я работал в одном из крупнейших в Украине операторов мобильной связи, и мне было поручено отвечать за технические аспекты внедрения в компании промышленной системы управления услугами. После того, как проект внедрения был завершен, я должен был единолично отвечать за «вторую линию» поддержки и развитие системы.

Система управления услугами[1] отвечает за претворение в жизнь высокоуровневых команд на управление услугами, таких как: «подключить нового абонента», «приостановить обслуживание абонента за неуплату», «активировать услугу MMS», и так далее.

Эти высокоуровневые команды должны быть преобразованы в набор низкоуровневых инструкций для телекоммуникационного оборудования (например, «активировать услугу GSM Data» или «дать абоненту доступ к GPRS APN 2»), после чего инструкции должны быть выполнены в определенном порядке нужными экземплярами коммуникационного оборудования. Система должна учитывать, что одни и те же функции в рамках сети оператора могут выполняться на разнотипном оборудовании нескольких поставщиков — например, в сети могут присутствовать коммутаторы нескольких поколений от двух поставщиков. Соответственно, система должна выбирать правильные протоколы для подключения к оборудованию, правильный набор команд для формирования инструкций, обрабатывать всевозможные исключительные ситуации и вообще всячески скрывать от других информационных систем детали и подробности процесса управления услугами.

Являясь критически важной для бизнеса, система использовалась круглосуточно. В часы пик в нее поступало от 6 до 10 тысяч входящих запросов в час. Каждый запрос преобразовывался в 5—15 низкоуровневых задач, каждая из которых, в свою очередь, состояла из нескольких команд для конкретного экземпляра оборудования. Система имела дюжину различных интерфейсов к нескольким десяткам телекоммуникационных платформ.

Ошибки в обработке запросов немедленно приводили к недополучению услуг абонентами, что означало финансовые потери для компании и клиентов. Соответственно, процесс обработки запросов должен был быть отлажен до мелочей, и все изменения в нем должны были производиться со всевозможным тщанием.

[1] Речь идет о продукте Comptel MDS/SAS, ныне известном как Comptel InstantLink.

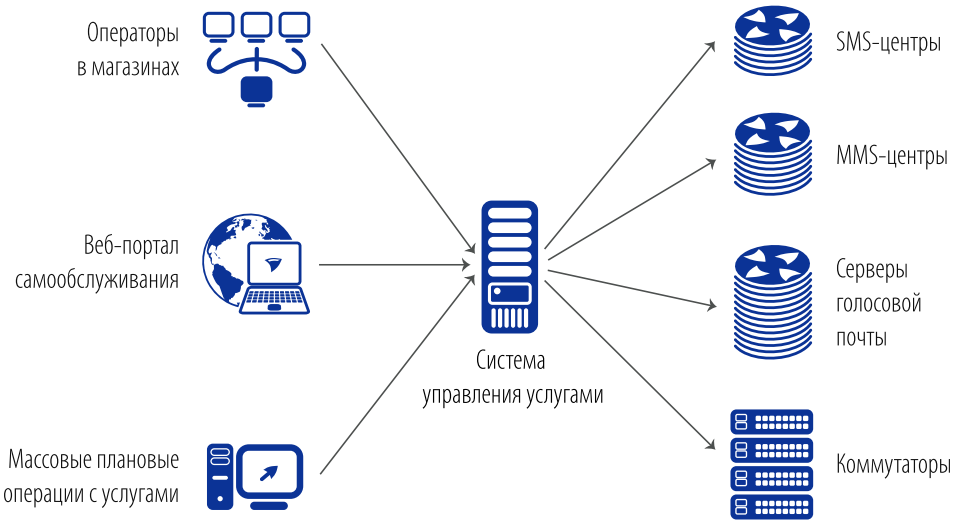


Рис. 2.1. Система управления услугами

2.2. Используемый в системе язык программирования и связанные с ним проблемы

Входящие запросы отличаются двумя основными свойствами: во-первых, все необходимые для обработки запроса данные содержатся в нем самом; во-вторых, запрос как правило имеет декларативную суть. То есть, в нем можно выделить несколько независимых друг от друга частей, которые могут быть содержательно проинтерпретированы независимо друг от друга в произвольном порядке.

Например, в рамках запроса «активировать для указанного абонента услуги SMS, MMS, GPRS, CSD, WAP-over-GPRS, WAP-over-CSD» можно выделить часть, описывающую абонента (его номер телефона, номер SIM-карты и т. п.), и части, описывающие параметры всех перечисленных услуг.

Производители системы решили, что лучше всего процесс обработки запросов организовать в виде, представляющем по сути интерпретатор императивного языка:

Входящий запрос представляет собой список пар «имя=значение». Все переменные, упомянутые в начальном запросе, составляют стартовое *окружение*. Далее на каждом шаге обработки проверяется, выполняется ли условие, сформулированное в терминах обычных операций сравнения над переменными из окружения. Если условное выражение `conditionX` истинно, то выполняются связанные с ним команды `actionX`, в противном случае происходит переход к следующему блоку «условие + команды», и так до конца списка.

Таким образом, обработка запроса сводилась к анализу переданных в запросе пе-

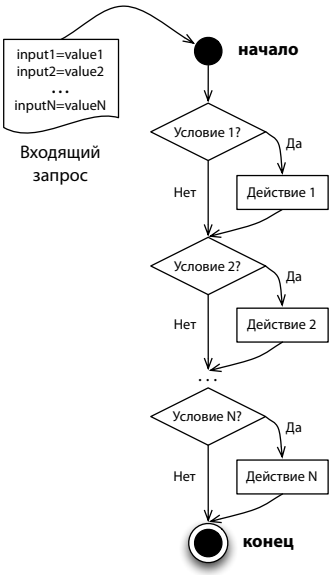


Рис. 2.2. Блок-схема работы интерпретатора

ременных, порождению на их основе новых, и, по окончании анализа, порождению команд на основании всего множества переменных. Запрос вида «удалить указанного абонента» мог быть обработан таким образом:

- В запросе присутствует переменная `$IMSI`? Значит, речь идет об абоненте сети GSM, выполняем `MARKET="GSM"`.
- Если `($MARKET=="GSM")` и в списке услуг абонента есть MMS, то надо удалить его inbox на MMS-платформе. Выполняем `MMS_ACTION="DELETE"`.
- …и так далее для всех прочих услуг, которые требуют отдельного удаления учетных записей.
- Если `($MARKET=="GSM") && ($MMS_ACTION<>"")`, то вычисляем ID абонента на MMS-платформе по его номеру телефона и SIM-карты.
- …и тому подобное для всех прочих услуг.
- Для каждого запланированного действия находим его приоритет в справочной таблице.
- Преобразуем все действия в команды для оборудования и выполняем в порядке возрастания приоритета.

Подобная архитектура позволяла системе иметь относительно простое ядро: для каждого запроса нужно было хранить относительно небольшой контекст выполнения, отсутствовал стек вызовов или его аналог и т. д. Обработка запросов хорошо распараллеливалась на несколько процессов в рамках одного сервера или на несколько независимых серверов. Система могла не бояться достаточно серьезных сбоев инфраструктуры — регулярно сохраняя текущий контекст для всех запросов, можно было в случае сбоя легко восстановить их обработку, в том числе и на другом сервере.

К сожалению, команд в языке было всего пять:

- Присвоение константного значения новой или существующей переменной «окружения»: `Assign(var, constant)`.
- Присвоение переменной значения, полученного конкатенацией значений других переменных и констант: `Concat(destination, $foo, $bar, "baz")`.
- Сопоставление значения переменной с регулярным выражением и присвоение результата сопоставления всего выражения и входящих в него групп другим переменным: `Regexp($var1, regexp, full_match, group`$_1$`_match, group`$_2$`_match, ...)`.
- Извлечение значения из внешнего «словаря» (базы данных), используя значение переменной в качестве «ключа»: `Lookup("datasource", $key, value)`.
- Отправка на исполнение устройству X команды, составленной из шаблона T, заполненного значениями переменных S1, S2, S3…: `Command("X", $T, $S1, $S2, $S3, ...)`.

Другими словами, для описания процесса обработки входящих запросов в системе использовался свой собственный проблемно-ориентированный язык (domain-specific language, DSL), код на котором выглядел примерно так:

```
10 Comment : 'NMT Convert MAIN_DIRNUM →  NMT_PRIMARY_RID'
Cond : Equals{$MARKET,"NMT"}
Oper : Regexp{$MAIN_DIRNUM=(.....)(...)(.*),$MTX_DUO=/2/,
              $TEMP_REMAIN=/3/}
  AND Concat{$NMT_MTX_NUMBER,$MTX_DUO,$TEMP_REMAIN}
  AND Regexp{$NMT_MTX_NUMBER=(.)(.*),
             $NMT_EDI_MSISDN_11_1=/2/}
  AND Lookup{mtxridlookup,$MTX_DUO,$PORT_DUO}
  AND Concat{$NMT_PRIMARY_RID,$PORT_DUO,$TEMP_REMAIN}

20 Comment : 'RID2 conversions'
Cond : Equals{$MARKET,"NMT"}
Oper : Regexp{$EDI2_NEW_PORT=(......)(.*),$NMT_RID2=/2/}

30 Comment : 'NMT Convert MAIN_DIRNUM →  NMT_VMS_NUMBER'
Cond : Equals{$MARKET,"NMT"}
```

```
Oper : Regexp{$MAIN_DIRNUM=(.....)(...)(.*),
              $TEMP_DIRNUM=/2/,$TEMP_REMAIN=/3/}
  AND Lookup{vmslookup,$TEMP_DIRNUM,$VMS_TRIPLET}
  AND Concat{$NMT_VMS_NUMBER,$VMS_TRIPLET,$TEMP_REMAIN}
```

Можно заметить, что во всех трех блоках сопоставление с регулярным выражением используется для того, чтобы реализовать отсечение нескольких первых символов от значения переменной. Учитывая ограниченность синтаксиса, программа просто изобиловала подобными ухищрениями.

Приведенный выше текст программы — это выдержка из системного отчета, который выводил всю DSL-программу в красивом читаемом текстовом виде. В самой же системе редактирование текста DSL-программы (или *бизнес-логики*, как я буду называть её в этой статье) осуществлялось с помощью графического интерфейса.

Интерфейс был типичным для всех нишевых продуктов, а в телекоммуникациях таких продуктов — большинство. Производители программного продукта в первую очередь фокусируют усилия на создании хорошего «ядра» продукта. При этом авторы программ не уделяют интерфейсам пользователя должного внимания, так как зачастую сами ими не пользуются. В описываемой системе текст бизнес-логики можно было редактировать частями, по одному выражению за раз, выбирая имена операторов из выпадающих списков. О какой-либо поддержке процесса разработки не было и речи — интерфейс не предоставлял даже функции поиска с заменой, не говоря уже о чем-то более сложном.[2]

Естественно, что сколько-нибудь существенная модификация текста бизнес-логики с помощью этого пользовательского интерфейса почти наверняка вносила глупые ошибки, выявить которые можно было только с помощью тестирования.

Тут крылась следующая проблема: для нужд тестирования имелся второй экземпляр системы. Тестирование заключалось в том, что тестировщик отправлял на исполнение в тестовом экземпляре системы пачку запросов, а потом вручную исследовал команды, которые были отправлены на оборудование, и результаты их работы. Проблема заключалась в том, что генерация команд даже для дюжины запросов требовала от тестировщика значительного объема ручной работы. Один проход тестирования даже небольшого изменения занимал несколько часов, требовал большого напряжения внимания и не ловил ошибки, случайно внесенные в те ветви бизнес-логики, которые не должны были изменяться и, соответственно, не тестировались.

После окончания тестирования необходимо было перенести изменения из тестовой системы в промышленную. Никаких специальных инструментов для этого не существовало, перенос изменений по задумке авторов системы выполнялся вручную. Рядом располагались два интерфейсных окна: одно от тестовой системы, второе — от промышленной, и изменения вдумчиво переписывались от руки. Естественно, вероятность что-то при этом пропустить или ненамеренно изменить была весьма высока.

[2]Тут хотелось бы заметить, что подобное наплевательское отношение к инженерам, обслуживающим системы, не является прерогативой какой-то одной компании. Большинство программных и программно-аппаратных продуктов, с которыми мне довелось иметь дело за время работы в телекоммуникациях, имели пользовательские интерфейсы, на которые нельзя было смотреть без слез.

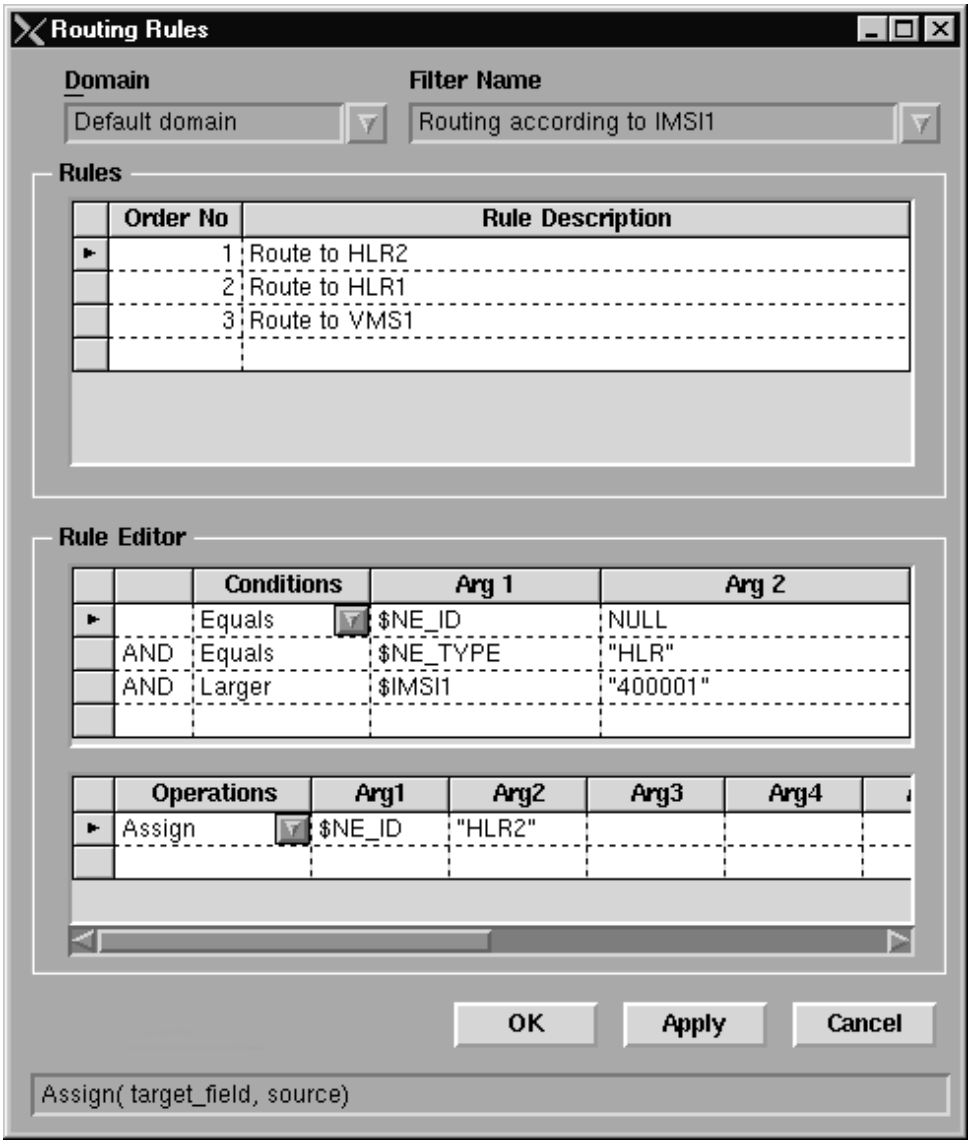


Рис. 2.3. Снимок экрана графического интерфейса системы

Производитель, естественно, обещал золотые горы и новый, радикально измененный интерфейс пользователя буквально в следующей версии системы, которая выйдет буквально через год-два. До тех пор предлагалось довольствоваться тем, что есть.

И все было бы ничего, если бы в реализации системного DSL не обнаружились сложновоспроизводимые ошибки. Например, все переменные по умолчанию имели специальное значение `NULL`, которое, в принципе, могло быть присвоено другой переменной. Однако, присвоение значения `NULL` *иногда* приводило к тому, что интерпретатор без всякой диагностики пропускал остаток программного блока после такого присваивания. Такое поведение проявлялось только под большой нагрузкой, и нам стоило больших трудов докопаться до первопричины. До тех пор, пока производитель не устранит ошибку в своем коде, необходимо было найти способ как-то обходить эту ошибку. Логично было бы не допускать присваивания `NULL` вообще, но как это отследить?

2.3. Постановка задачи

Из вышесказанного становится ясно, что для успешной поддержки этой критически важной для бизнеса системы требовалось радикально изменить все этапы стандартного жизненного цикла поддержки и развития системы.

Если бы информационная система была разработана «под заказ», естественным решением было бы заказать её доработку. Если бы она была доступна в исходных кодах, можно было бы потенциально думать над тем, чтобы произвести все необходимые изменения самостоятельно. Однако система представляла собой коробочный программный продукт, в связи с чем сравнительно быстрого и/или недорого способа получить требуемую функциональность не предвиделось.

Кроме того, поскольку я отвечал за поддержку системы самостоятельно, без помощников, и это была далеко не единственная моя обязанность, рассчитывать на внедрение практики парного программирования, рецензирования кода или иные подобные организационные методы не приходилось.

В результате я решил разработать отдельный набор инструментов, которые позволили бы:

- Облегчить разработку. Получить, по меньшей мере, возможность использовать копирование/вставку текста и поиск с заменой.
- Облегчить тестирование новых версий бизнес-логики. Необходимо было, как минимум, получить возможность легко проводить регрессионное тестирование: то есть, проверять, что внесенные изменения имеют локальный эффект и не затрагивают сторонние ветки кода. На случай обнаружения ошибок были необходимы какие-то средства пошаговой отладки или аналог отладочной печати.
- Обеспечить перенос новых версий кода из тестовой системы в промышленную без участия человека, чтобы исключить вероятность внесения изменений в код после его тестирования.

- Обеспечить контроль за версиями бизнес-логики, получить возможность параллельно вести разработку нескольких альтернативных вариантов кода.

2.4. Написание инструментальных средств на Haskell

Поскольку система предоставляла возможность экспортировать полный текст бизнес-логики в текстовый файл, вопрос с контролем версий был частично решен путем регулярного размещения этого текста в корпоративной системе контроля версий.

В самой системе текст бизнес-логики хранился в обработанном виде в нескольких таблицах в СУБД Oracle. После изучения схемы базы на языке Haskell был написан компилятор, который выполнял синтаксический разбор текстового файла с бизнес-логикой и преобразование его в набор SQL-выражений, замещающих код бизнес-логики в системе новой версией.

В результате появилась возможность не только экспортировать бизнес-логику из системы в текстовый файл, но и импортировать подобный текстовый файл обратно. После этого вся разработка начала вестись в нормальном текстовом редакторе.[3] Кроме того, появилась возможность автоматически переносить изменения из тестовой системы в промышленную без вмешательства человека.

Далее при помощи модуля синтаксического анализатора из вышеупомянутого компилятора был построен интерпретатор бизнес-логики, который принимал на вход пакет файлов с входящими запросами и эмулировал работу ядра системы.

На выходе получались протоколы, в которых было указано, какие команды для какого оборудования были сформированы, и в каком порядке они будут исполняться. Кроме того, при необходимости интерпретатор мог выдать полную «трассу» исполнения бизнес-логики с указанием, какие переменные были модифицированы на каждом шаге, с каким результатом были вычислены все части условного выражения в рамках каждого программного блока, и полным перечислением состояния переменных после каждого блока. Файлы-протоколы формировались в виде, облегчающем их обработку стандартными утилитами `grep`, `diff` и т. п. В результате появилась возможность наладить нормальное тестирование при внесении изменений в бизнес-логику.

После этого на базе интерпретатора был сделан «детектор багов», который для данной программы бизнес-логики и большого массива запросов проверял, не возникают ли в ходе их обработки условия, в которых могут срабатывать известные ошибки в системе. В частности, идентифицировались все случаи присваивания `NULL` и для них выдавалась диагностика о том, в каком месте программы это произошло, при обработке каких переменных, и как выглядит полная трасса исполнения до этого момента.

Кроме этого, на базе интерпретатора был сделан инструмент, позволяющий автоматически формировать репрезентативную выборку входящих запросов. Запросы, прошедшие через систему за какой-то достаточно большой период (месяц или квартал), разбивались на классы эквивалентности по следующему критерию: все запросы,

[3] Использовался emacs, для которого был сделан свой модуль подсветки синтаксиса.

которые генерировали похожую трассу исполнения и порождали один и тот же набор команд (с точностью до значений переменных типа «номер телефона», «номер SIM-карты» и т. п.), считались эквивалентными. Из каждого класса эквивалентности в репрезентативную выборку попадал только один запрос. Полученный набор запросов использовался для регрессионного тестирования и поиска программных блоков, которые по каким-либо причинам ни разу не были использованы при обработке всех этих запросов.

Также на базе интерпретатора был сделан аналог утилиты «sdiff»[4] для результатов работы бизнес-логики. На вход ей подавались две версии бизнес-логики и набор входных запросов для тестирования, а утилита генерировала отчет о том, на каких входных запросах поведение программ различается и в чем именно заключаются эти различия. Отчет включал в себя подробный перечень того, чем отличаются трассы исполнения обеих версий программы. На примере этой утилиты можно проиллюстрировать, как выглядел код разрабатываемых инструментальных средств:

```
diffBusinessLogic oldLogic newLogic request =
  let context = mkInitContext request
      oldLogicTrace = runAndTrim context oldLogic
      newLogicTrace = runAndTrim context newLogic
      in
      if newLogicTrace == oldLogicTrace
       then return ()
       else do printSectionHeader
               printRequest request
               printAligned $
                 suppressEquals oldLogicTrace
                                newLogicTrace
  where
    runAndTrim context logic =
      trimVolatileVars $ run context logic
```

Функция `run`, предоставляемая интерпретатором, преобразует запрос и бизнес-логику в трассу исполнения. Функция `trimVolatileVars`, взятая из генератора репрезентативной выборки, удаляет из трассы исполнения все упоминания переменных, которые обязательно разнятся от запроса к запросу (порядковый номер запроса и тому подобные служебные переменные). Если обработанные таким образом трассы исполнения различаются, то они выводятся в два столбца (при помощи функции `printAligned`), при этом в выводе подавляются (функцией `suppressEquals`) упоминания переменных и выражений, которые в обеих трассах имеют одинаковые значения.

Именно с помощью этой утилиты и проводилось тестирование новых версий бизнес-логики перед передачей в промышленную эксплуатацию. В частности, если

[4]Утилита «sdiff» выводит текст сравниваемых файлов в две колонки, обозначая вставки, удаления и правки в сравниваемых текстах.

было известно, что новая версия бизнес-логики отличается от старой только обработкой запросов, включающих в себя входную переменную `FOO`, то репрезентативная выборка запросов при помощи `grep` разделялась на две части: запросы, содержащие переменную `FOO`, и запросы, её не включающие. После чего с помощью «sdiff»-подобной утилиты проверялось, что старая и новая версии обрабатывают все запросы из первой группы по-разному, а все запросы из второй группы - одинаково.

Наконец, на базе библиотеки `QuickCheck` был сделан генератор случайных (но не произвольных!) входных запросов. В частности, было известно, что номера телефонов абонентов могут принадлежать только некоторому определенному диапазону, номера IMSI тоже определенным образом ограничены, перечень услуг известен и конечен и так далее. Используя эти знания, можно было генерировать запросы, корректные по структуре, но не обязательно непротиворечивые и правильные по содержанию. Они использовались для тестирования бизнес-логики на «дуракоустойчивость».

Полный перечень разработанных инструментальных средств и отношения между ними можно увидеть на рисунке 2.4.

Рис. 2.4. Полный перечень разработанных инструментальных средств

2.5. Достигнутые результаты

Главный результат заключался в том, что удалось уйти от использования убогого графического интерфейса и ручной работы по переносу кода из тестовой системы в промышленную. Удалось наладить полноценный контроль версий разрабатываемого кода.

Удалось устранить падения системы под нагрузкой из-за ошибок в системном интерпретаторе бизнес-логики. Сам производитель окончательно устранил все трудно обнаруживаемые ошибки в области исполнения бизнес-логики только спустя полтора года после появления на свет нашего «детектора багов».

Удалось радикально повысить качество разработки новых версий бизнес-логики: большинство изменений проходили приемку тестировщиками с первого раза и не имели проблем в ходе промышленной эксплуатации. Работу, которая раньше занимала неделю календарного времени, теперь реально было сделать в течение одного рабочего дня.

Общий объем написанного мною кода — примерно 1600 строк на Haskell с обширными многострочными комментариями. В том числе:

Программный модуль	Количество строк
Типы данных абстрактного синтаксического дерева	80
Парсеры бизнес-логики и запросов	190
Компилятор в SQL	100
Интерпретатор бизнес-логики	280
Утилита «sdiff»	100
Построитель «репрезентативной выборки»	300
Генератор запросов на QuickCheck	300

Таблица 2.1. Объем кода программных модулей, в строках

Общее время разработки оценить тяжело, так как между реализацией отдельных модулей были значительные перерывы, но, судя по записям в системе контроля версий, интерпретатор был написан примерно за неделю.

Почему же был выбран именно Haskell и в чем же, в ретроспективе, оказались его преимущества?

Поскольку довольное долгое время я занимался поддержкой системы в одиночку, причем это не было моим основным занятием, то моей главной целью было как можно скорее достигнуть практических результатов с минимальными затратами времени и сил. В таких условиях я мог позволить себе взять любое инструментальное средство.

Я выбрал Haskell, который уже тогда привлекал меня простотой и скоростью разработки. Кроме того, мне импонировала возможность положиться на механизм вывода типов и писать код, который по возможности будет «правильным по построению».

Ведь я замахивался на то, чтобы своим интерпретатором находить и устранять ошибки в другом интерпретаторе, и мне вряд ли удалось бы это, будь в моем интерпретаторе свои уникальные ошибки.

В числе прочих преимуществ Haskell можно назвать:

- Возможность «встроить» код парсера непосредственно в код основной программы при помощи библиотеки комбинаторов парсеров Parsec.
- Богатый набор контейнерных типов, предоставляемых языком и стандартными библиотеками (списки, массивы, хэш-таблицы, отображения **`map`**).
- Широкий арсенал средств для написания кода на высоком уровне абстракции и комбинирования решения из частей: функции высших порядков, монады `Reader`, `Writer` и `State`.
- Возможность тестировать свой код на псевдо-случайных входных данных, автоматически генерируемых на основании типов тестируемых функций библиотекой `QuickCheck`.

Все это в сумме приводило к тому, что в большинстве случаев код, в соответствии с расхожей присказкой про Haskell, правильно работал после первой же успешной компиляции. Благодаря относительно небольшому объему всего проекта и наличию тестов, можно было смело и радикально переписывать любые части проекта. В результате большая часть времени посвящалась решению прикладной задачи и алгоритмическим оптимизациям, а не низкоуровневому программированию и борьбе с ограничениями языка.

Таким образом, считающийся «академическим» язык был с большим успехом применен для решения практических повседневных задач в критичной для бизнеса области.

2.6. Постскриптум

Я делал доклад о применении описанных в этой статье инструментальных средств на ежегодном собрании пользователей продуктов Comptel в 2003 году. В 2005 году в очередной версии системы проявился нормальный графический интерфейс пользователя, инструменты для миграции кода между тестовыми и промышленными экземплярами системы, а также инструменты для отладки, функциональность которых во многом повторяла мои персональные наработки.

Я считаю, что этот доклад (и интерес к нему со стороны других клиентов) был одним из факторов, ускоривших появление этих нововведений и в значительной мере определивших их функциональность.

Прототипирование с помощью функциональных языков

Применение функциональных языков для моделирования цифровых электронных схем

Сергей Зефиров, Владислав Балин
thesz@fprog.ru, gaperton@fprog.ru

Аннотация

В статье рассказывается о применении функциональных языков в моделировании аппаратуры. Приведены критерии, по которым были выбраны функциональные языки вместо распространенных в индустрии инструментов, и представлены результаты работы.

The article discusses the use of functional programming languages in hardware modelling. The criteria are provided for choosing the functional programming languages over the more mainstream tools, and the results of the authors' work are described.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/2260.html.

3.1. Введение

При разработке ПО часто применяют прототипирование — быструю разработку ключевой части алгоритма или фрагмента решения с целью изучения его свойств. Чем раньше мы обнаруживаем ошибки, тем дешевле нам обходится их исправление, и в особенности это касается ошибок в выборе подхода к решаемой проблеме. Цель прототипирования — проверить правильность выбранного подхода на раннем этапе разработки или получить новое знание об интересной нам предметной области, проведя эксперимент. По результатам эксперимента можно скорректировать подход к проблеме. Прототипирование с успехом применяется в разных областях инженерной деятельности, начиная с разработки автомобилей и самолетов и заканчивая разработкой новых моделей одежды. Возможность быстро и с низкими затратами изготавливать прототипы приносит наибольшую отдачу в тех областях, в которых цена ошибки в выборе подхода и технических решений крайне высока. Раннее прототипирование в том или ином виде совершенно необходимо в следующих ситуациях:

- Длительный цикл разработки. Если природа задачи такова, что разработка состоит из большого количества этапов, у вас просто не хватит времени исправить ошибку в подходе. Скажем, разработка программно-аппаратных комплексов именно такова. В особенности, если у вас…

- …высокая стоимость получения результата. Постановка на производство автомобиля, как и изготовление опытных образцов, является дорогой операцией. Не хватит не только времени, но и денег. На данных этапах надо действовать наверняка. Что сложно, если налицо…

- …отсутствие должного опыта у инженерной группы. В случае инновационной разработки, когда вы делаете продукт на уровне лучших аналогов (или превосходящий их), это всегда так, ибо вы делаете то, что до вас почти никто не делал.

Разработка микроэлектроники, как отличный пример проектов такого типа, имеет длительный (типичная длительность проекта — 1,5 года) и многоступенчатый цикл разработки, в котором задействовано много разных специальностей. В микроэлектронике крайне велика стоимость получения результата и, следовательно, высока цена ошибки. Набор фотошаблонов для современных тех-процессов (тоньше чем 90 нм), без которого не получить образцы микросхем, стоит миллионы долларов и изготавливается фабрикой в срок от 2 месяцев, а бюджет относительно простых проектов составляет от единиц до десятков миллионов долларов. В таких условиях группа разработки продукта должна действовать наверняка — ошибка в требованиях или в выборе подхода к проблеме неприемлема и закончится для проекта фатально. Стоит отметить, что подобное происходит не только в разработке микроэлектроники — некоторые чисто программные проекты также во многих аспектах обладают подобными характеристиками. В связи с этим многие компании-разработчики микроэлектроники выполняют моделирование на раннем этапе для проверки своих решений. Опре-

деленного стандарта в данный момент не существует, применяемые инструменты варьируются от одной компании к другой. Ниже мы рассмотрим существующие средства прототипирования микроэлектроники с их сильными и слабыми сторонами и расскажем об опыте применения функциональных языков программирования в данной задаче.

3.2. Инструменты прототипирования компонентов

Описание рассматриваемого микроэлектронного устройства, такого как микропроцессор, представляет собой цифровую логическую схему, которая может быть сведена к комбинации элементов памяти (триггера) и логического элемента 2–и–не, соединенных проводами. Схема может быть описана в терминах этих (и более сложных) элементов, набор которых называется «библиотекой» и предоставляется фабрикой. Аналогом такого описания в программировании является язык ассемблера, специфичный для каждой их процессорных архитектур.

Сейчас разработка устройства в терминах библиотечных элементов является скорее исключением, и для описания аппаратуры применяются языки высокого уровня, такие как Verilog и VHDL, наиболее популярен из них первый.

Verilog крайне прост в изучении — каждый блок устройства описывается функцией, аргументами которой являются отдельные «провода» и «массивы проводов» (то есть, биты и битовые массивы). Это язык чрезвычайно низкого уровня по меркам современных языков программирования — в нем полностью отсутствуют типы. Язык содержит ограниченное «синтезируемое» подмножество, которое может быть оттранслировано в цифровую схему при помощи САПР.

Несинтезируемые конструкции похожи на конструкции языков разработки ПО и применяются для разработки моделей и тестов для целей верификации синтезируемых схем. В целом, программирование на полном Verilog менее затратно, чем на синтезируемом подмножестве, и может быть использовано для прототипирования устройства перед его реализацией. Однако, оно продолжает оставаться крайне затратным, а модель устройства — слишком детальной для целей архитектурного прототипирования.

Разработчиками САПР поддерживается два языка для решения данной проблемы. Первый из них — SystemC — на деле языком не является, это некоторый фреймворк для C++. По сути, данная библиотека позволяет писать на C++ в стиле Verilog, в том числе описывая и синтезируемые конструкции. Авторы SystemC надеялись, что развитые языковые средства C++ позволят программистам и инженерам описывать более сложные модели.

Вторым языком является SystemVerilog. Это последняя редакция языка Verilog, расширенная современными конструкциями вроде классов и типов. Упрощая структурирование крупной системы для разработчиков аппаратуры, данный язык в целом сохраняет подход Verilog и не адресует проблем архитектурного прототипирования.

Основные требования к инструментам прототипирования микроэлектроники перечислены ниже, в порядке убывания важности:

- Низкие затраты на разработку. Чем раньше будет создан прототип, тем лучше.
- Низкая стоимость внесения изменений. Прототип не является чем-то статичным, в него очень часто вносятся изменения. Чем короче цикл внесения изменений, тем больше экспериментов удастся провести на этапе проектирования. В идеальном случае цикл внесения изменений не должен превышать нескольких дней.
- Скорость работы модели (скорость моделирования). Чем выше скорость, тем большие по объёму тесты можно будет подать на вход модели. Разница в поведении на малом и большом тестах может быть значительной[1] и существенно повлиять на выбор подхода к решению.
- Масштабируемость инструмента реализации. Возможно ли задействовать большее число людей для повышения скорости реализации.
- Встроенность в цикл разработки: возможность постепенного уточнения описания компонента с целью перехода к описанию уровня синтезируемой модели.

SystemVerilog и SystemC в основном нацелены на решение последних двух пунктов требований, и практически не адресуют первые три.

Отдельно стоит упомянуть машины состояний. Электронная аппаратура практически вся построена на них, и сложность варьируется от простых машин с парой состояний (есть данные — нет данных) до сложных составных. Большая часть машин состояний не может быть представлена в виде упрощаемого цикла[2] и выглядит в простейшем случае примерно так:

```
loop1:
   action 1;
loop2:
   action 2;
   if condition then goto loop1;
   action 3;
   goto loop2;
```

Хорошее средство прототипирования позволяет описывать машины состояний просто и безопасно. Чем проще такое описание, тем быстрее можно получить точный прототип компонента, тем выше скорость реализации — первый пункт в перечне выше.

[1] Например, кодирование видео. Разница между требуемой пропускной способностью подсистемы памяти для видео-потоков телевидения высокой и стандартной чёткости составляет > 13 раз, тогда как разница в общем объёме данных не более 5 раз ($1920 * 1080/(720 * 576) = 5$).

[2] Неупрощаемый цикл определяется как цикл с несколькими точками входа.

В общем случае в микропроцессоре присутствуют несколько зон с разной тактовой частотой, и при их соединении в общую систему могут возникать ошибки. Однако таких соединений очень мало, ошибки хорошо известны и их ловят и исправляют очень инженерными способами, например, осциллографом и паяльником[3]. А вот сами зоны с одной тактовой частотой весьма обширны, и большая часть ошибок кроется именно в них. Поэтому инструмент для прототипирования может не иметь возможности моделировать системы с разной тактовой частотой и всё равно быть очень полезным.

Если попытаться посмотреть на функциональные языки с точки зрения инструмента для прототипирования аппаратуры, то выводы будут достаточно интересны:

- Функциональные языки предлагают простой и безопасный способ описания сложных машин состояний на алгебраических типах. Компиляторы функциональных языков автоматически проверяют некоторые инварианты машин состояний. Это позволяет ускорить реализацию и оборот идей.

- Функциональные языки имеют отличные компиляторы и дают возможность дёшево добиться параллельного выполнения программы. Значит, скорость моделирования будет высокой.

- Функциональные языки позволяют упростить создание отдельных компонентов. Несмотря на то, что компоненты имеют состояние, переход между состояниями определяется чистой функцией, которую проще оттестировать и корректность которой даже можно доказать.

- Функциональные языки никак не встроены в процесс получения конечного результата[4]. Мы не можем взять описание компонента на функциональном языке и получить описание на логических вентилях путём уточнения описания. Все тесты придётся переносить на языки описания аппаратуры отдельным этапом, весьма вероятно, вручную.

Принимая во внимание цель прототипирования, «синтезируемость» от прототипа не требуется, требуется только потактовая аккуратность — совпадение временных диаграмм с точностью до такта. Это позволяет существенно сократить уровень детальности модели, подняв при этом скорость разработки и моделирования.

Изменяя подход к описанию цифровой схемы, мы теряем возможность постепенного уточнения этой схемы, и код прототипа будет выкинут при переходе от проектирования к разработке. Это будет вполне оправдано в случае, если изменение подхода даст существенный выигрыш по первым двум пунктам требований по сравнению с существующими инструментами.

[3] Современными их аналогами, конечно же.

[4] Lava [10], Hydra [9] и ForSyDe [4] позволяют надеяться на то, что ситуация изменится в ближайшем будущем.

3.3. Моделирование аппаратуры с помощью функциональных языков

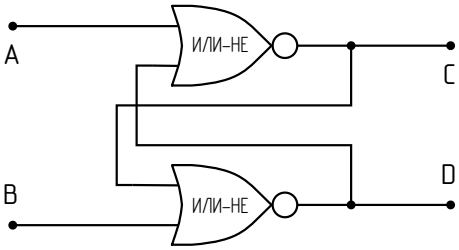
3.3.1. Общий подход

Общий подход прост: на основе текущего состояния и текущих входных данных надо рассчитать текущие выходные данные и состояние для следующего такта (k — номер компонента; состояние, входные и выходные сигналы представляют собой кортежи):

$$(\mathbf{O}_{ki}, \mathbf{I}_{ki+1}) = F_k(\mathbf{I}_{ki}, \mathbf{S}_{ki})$$

Получается рекурсивная функция.

Между компонентами практически всегда существует кольцевая зависимость. Её наличие приводит к необходимости упорядочения вычислений выходов на основе входов. Ниже приведён RS-триггер на ИЛИ–НЕ элементах:



RS-триггер — это простое устройство с памятью на элементах логики. При подаче 1 на провод A (вход S, Set) и 0 на провод B (вход R, Reset) мы должны получить 1 на проводе C (выход Q, прямой выход), который останется при сбросе A в 0. И наоборот, при подаче пары 01 на AB мы должны получить на прямом выходе 0, который останется на входе после сброса провода B в 0.

Выход одного элемента зависит от входа другого и существует вариант входных данных, когда эта схема находится в самовозбуждённом состоянии (если с самого начала подать на оба входа 1).

По идее, компоненты не должны знать о порядке подачи данных на входы, и состояние компонентов не должно быть видно снаружи. В случае SystemC это реализовано при помощи библиотеки, которая перезапускает код вычисления по приходу очередного сообщения на один из входов, состояние инкапсулировано внутри класса. В VHDL эквивалентная схема вычислений обеспечивается семантикой самого языка: процессы, обрабатывающие реакцию элементов на воздействия, запускаются по каждому изменению сигналов, и изменённые в результате вычислений сигналы распространяются далее, запуская другие процессы.

Одним из вариантов абстрагирования от порядка вычислений является использование ленивых вычислений. Развитие событий во времени можно представить списком событий. Если скрестить эти два приёма, получатся ленивые списки событий [5].

Этот подход не является новым и широко известен как декомпозиция системы на «потоках» (streams). Подход хорошо описан в курсе SICP [11, 1, гл. 3.5] и является старейшим из известных подходов к моделированию состояния в «чистых» функциональных языках.

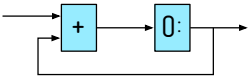
Ленивые списки могут применяться для обеспечения ввода-вывода.[5] Однако их применение в реальных программах затруднено,[6] поскольку события из внешнего мира могут приходить в произвольном порядке. В случае моделирования аппаратуры порядок фиксирован, и ленивые списки являются самым простым вариантом при использовании языков со ссылками (Lisp, семейство ML, Haskell) или с ленивым порядком вычислений (Clean, Haskell).

3.3.2. Вариант на языке Haskell

Начнём сразу с примеров. Построим накапливающий сумматор — небольшой элемент с состоянием, которое равно сумме всех принятых входных значений. На выход будет поступать результат суммирования. Итак:

```
runningSum :: [Int] → [Int]
runningSum inputs = sum
    where
        sum = 0 : (zipWith (+) sum inputs)
```

Вот его схема:



Функция **zipWith** применит первый аргумент — двуместную функцию — к одинаковым по порядку элементам второго и третьего аргументов. (+) — это синоним безымянной функции $\lambda x\, y.\, x+y$. Оператор (**:**) — это оператор конструирования списков. Слева находится голова списка, справа хвост.

Результат работы приведён ниже:

```
Prelude> runningSum [0,0,1,0,0,−1,10,0,0,−1]
[0,0,0,1,1,1,0,10,10,10,9]
Prelude> runningSum [1,0,1,0,0,−1,10,0,0,−1]
[0,1,1,2,2,2,1,11,11,11,10]
```

[5] В библиотеке языка Haskell этот атавизм можно наблюдать до сих пор [8].
[6] Интересное обсуждение связанных с этим проблем содержится в [3].

Сумма поступает на выход с задержкой в один такт. Самый первый элемент всегда 0 — мы сформировали задержку путём добавления 0 в голову списка сумм. Второй элемент — функция от 0 (**sum**`[0]`) и `inputs[0]`. Третий элемент — функция от **sum**`[1]` (`0+inputs[0]`) и `inputs[1]`, и так далее.

Второй пример будет чуть более сложным — RS-триггер на элементах ИЛИ–НЕ. У него два входа и два выхода: вход R (RESET, сброс), вход S (SET, установка) и выходы Q (прямой) и Q' (инверсный).

```
nor :: [Bool] → [Bool] → [Bool]
nor xs ys = False : zipWith nor' xs ys
    where
        nor' a b = not (a || b)

rsTrigger :: [Bool] → [Bool] → ([Bool], [Bool])
rsTrigger r s = (q, q')
    where
        q  = nor s q'
        q' = nor r q
```

Схема создания `nor` похожа на `runningSum`: вводим задержку с помощью (`False : `*`вычисление`*), само вычисление сводится к применению чистой функции к входам поэлементно.

rsTrigger уже отличается от предыдущих функций: на входе каждого nor есть выход другого nor. Такое зацикливание без всякого дополнительного программного текста возможно из-за ленивого порядка вычислений: компилятор формирует вычисления, а библиотека времени выполнения сама выстроит их по порядку.

Вот результат работы rsTrigger:

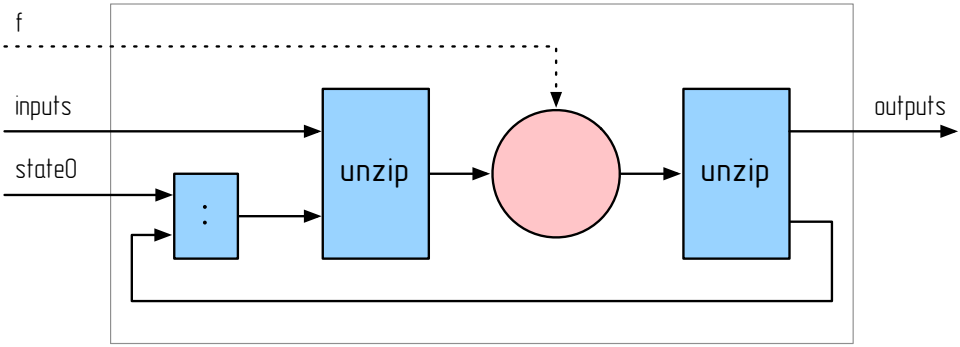
```
*Main> rsTrigger (replicate 5 False) (replicate 5 False)
([False,True,False,True,False,True]
,[False,True,False,True,False,True])
*Main> rsTrigger (True : replicate 4 False) (replicate 5 False)
([False,True,True,True,True,True]
,[False,False,False,False,False,False])
*Main> rsTrigger (replicate  5 False) (True : replicate 4 False)
([False,False,False,False,False,False]
,[False,True,True,True,True,True])
```

Самый первый запуск приводит к самовозбуждению схемы. Второй (`r` активен) приводит к установке `q` в 0, сбросу. Третий пример показывает работу входа s.

В RS-триггере единицей времени моделирования является задержка на элементе ИЛИ–НЕ. В накапливающем сумматоре единица времени не указана, это такт работы какой-то схемы. В принципе, при моделировании сложной аппаратуры пользуются именно вторым вариантом, в качестве единицы модельного времени берётся один такт всей схемы.

Накапливающий сумматор интересен потому, что в нём текущее состояние и входы определяют значение текущих выходов и следующего состояния. Такая схема весь-

ма распространена, она называется автоматом Мили (Mealy machine). Легко оформив её в отдельный примитив, можно писать только чистые функции преобразования данных.



Вот приблизительный код этой функции ядра с примером применения — реализацией накапливающего сумматора:

```
mealy :: (state → input → (state,output))
         → state
         → [input]
         → [output]
mealy f startState inputs = outputs
    where
        (nextStates,outputs) = unzip $ zipWith f states inputs
    states = startState : nextStates

runningSumMealy :: [Int] → [Int]
runningSumMealy = mealy (λsum i → (sum+i,sum)) 0
```

Результат работы `runningSumMealy`:

```
*Main> runningSumMealy [0,0,1,0,0,−1,10,0,0,−1]
[0,0,0,1,1,1,0,10,10,10]
*Main> runningSumMealy [0,0,1,0,0,−1,10,0,0,−1,0,0]
[0,0,0,1,1,1,0,10,10,10,9,9]
```

Отличие есть в конце вычислений, так как входной список конечен.[7] К счастью, мы используем бесконечные списки, и это отличие просто не обнаруживает себя.

3.3.3. Вариант на языке Erlang

Язык программирования Erlang является строгим, и языковые конструкции, соответствующие «потокам», в нем отсутствуют. Описываемый подход, тем не менее,

[7] Количество выходных элементов равно количеству входных из-за применения **`zipWith`**.

возможно реализовать на базе очередей сообщений и процессов.

В данном случае каждый блок будет также представлен бесконечно-рекурсивной функцией, работающей в своём процессе. Однако, вместо того, чтобы работать со списками, функция должна принимать и отправлять сообщения. Логику посылки и отправки сообщений возможно выделить в отдельный модуль (как это сделано для модуля gen_server стандартной библиотеки).

Получающийся подход к моделированию схемы полностью эквивалентен подходу с «потоками», за исключением того, что очередь сообщений между входами и выходами элементов не является первоклассной конструкцией.[8] В связи с этим, для соединения компонентов требуется поддержка ядра библиотеки.

Библиотека должна отслеживать факт получения всех входных данных, в противном случае возможно спутать входные данные разных тактов и получить неверные результаты.

Мы решили использовать Erlang для моделирования аппаратуры из-за его существенно более простого синтаксиса. Опрошенные инженеры сказали, что модели на Erlang воспринимаются ими проще — аргументы функций в скобках более привычны, последовательность операций более прозрачна (к примеру, **`where`** в Haskell «параллелен» в том смысле, что можно переставлять определения без изменения семантики).

3.3.4. Алгебраические типы и полиморфизм

Алгебраические типы при моделировании аппаратуры используются для описания машин состояний и структурирования передаваемой информации. Последнее, вместе с параметрическим полиморфизмом, существенно помогает в работе.[9]

В качестве примера можно привести описание команд микропроцессора.

После этапа декодирования команды содержат индексы регистров, из которых будет производиться чтение.

```
data Cmd = Nop
         | Load DestReg Reg
         | Add DestReg Reg Reg
```

После этапа чтения операндов структура команд не изменится, поменяется только содержимое тех полей, что имели тип `Reg`.

```
data ReadCmd = Nop
             | Load DestReg Word32
             | Add DestReg Word32 Word32
```

Разумно будет параметризовать команды:

```
data Cmd reg = Nop
             | Load DestReg reg
```

[8] First class value — сущность, для которой доступны любые операции языка.

[9] Это справедливо и для динамически типизированных языков, таких как Erlang, тем более, что полиморфизм типов данных в Erlang практически ничем не ограничен.

```
        | Add DestReg reg reg
```

Ниже приведены два примера: один из plasma [7] (свободно распространяемая реализация ядра MIPS [6] на VHDL), другой из тестовой реализации похожего ядра MIPS на Haskell.

Вот код декодера команд plasma:

```
entity control is
  port(opcode   : in std_logic_vector(31 downto 0);
       intr_signal : in std_logic;
       rs_index  : out std_logic_vector(5 downto 0);
       rt_index  : out std_logic_vector(5 downto 0);
       rd_index  : out std_logic_vector(5 downto 0);
       imm_out   : out std_logic_vector(15 downto 0);
       alu_func  : out alu_function_type;
       shift_func : out shift_function_type;
       mult_func : out mult_function_type;
       branch_func : out branch_function_type;
       a_source_out : out a_source_type;
       b_source_out : out b_source_type;
       c_source_out : out c_source_type;
       pc_source_out: out pc_source_type;
       mem_source_out:out mem_source_type;
       exception_out: out std_logic);
end; --entity control

architecture logic of control is
begin

control_proc: process(opcode, intr_signal)
  variable op, func : std_logic_vector(5 downto 0);
  variable rs, rt, rd : std_logic_vector(5 downto 0);
  variable rtx      : std_logic_vector(4 downto 0);
  variable imm      : std_logic_vector(15 downto 0);
  variable alu_function : alu_function_type;
begin
  alu_function := ALU_NOTHING;
  shift_function := SHIFT_NOTHING;
  mult_function := MULT_NOTHING;
  a_source := A_FROM_REG_SOURCE;
  b_source := B_FROM_REG_TARGET;
  c_source := C_FROM_NULL;
  pc_source := FROM_INC4;
  branch_function := BRANCH_EQ;
  mem_source := MEM_FETCH;
  op := opcode(31 downto 26);
  rs := '0' & opcode(25 downto 21);
```

```
rt := '0' & opcode(20 downto 16);
rtx := opcode(20 downto 16);
rd := '0' & opcode(15 downto 11);
func := opcode(5 downto 0);
imm := opcode(15 downto 0);
is_syscall := '0';

case op is
when "000000" => --SPECIAL
  case func is
  when "000000" => --SLL r[rd]=r[rt]<<re;
    a_source := A_FROM_IMM10_6;
    c_source := C_FROM_SHIFT;
    shift_function := SHIFT_LEFT_UNSIGNED;

  when "000010" => --SRL r[rd]=u[rt]»re;
    a_source := A_FROM_IMM10_6;
    c_source := C_FROM_shift;
    shift_function := SHIFT_RIGHT_UNSIGNED;
  when "011011" => --DIVU s→ lo=r[rs]/r[rt]; s→ hi=r[rs]%r[rt];
    mult_function := MULT_DIVIDE;

  when "100000" => --ADD r[rd]=r[rs]+r[rt];
    c_source := C_FROM_ALU;
    alu_function := ALU_ADD;


when "000001" => --REGIMM
  rt := "000000";
  rd := "011111";
...
```

Обратите внимание, что команды и их данные разнесены. Поэтому проверить соответствие команд данным можно только пересмотром кода или тестированием.

Вот код декодера команд на Haskell:

```
type RI = IntSz FIVE -- register index.

data Dest = Dest RI
    deriving (Eq,Show)

data Imm5 = Imm5 (IntSz FIVE)
    deriving (Eq,Show)
data Signed = Signed | Unsigned
    deriving (Eq,Show)
data MIPSCmd reg =
        Nop
```

```
    |   Trap
    |   Add reg reg Dest
    |   And reg reg Dest
    ...

---- Decoder.

mipsDecode :: Word32 → MIPSCmd (IntSz FIVE)
mipsDecode word = cmd
 where
  high6 :: IntSz SIX
  low26 :: IntSz SIZE26
  (high6,low26) = castWires word
  base_rs :: IntSz FIVE
  rt :: IntSz FIVE
  offset :: IntSz SIZE16
  (base_rs,rt,offset) = castWires low26
  cmd = case (high6,base_rs,rt) of
    (0x00,_,_) → mipsDecodeSpecial low26
    (0x02,_,_) → J (Imm26 low26)
    (0x20,_,_) → LB Signed base_rs (RI rt) (Imm16 offset)
    (0x01,_,_) → mipsDecodeRegImm (base_rs,rt,offset)
    (0x07,_,_) → BGT base_rs rt (Imm16 offset)
    (0x06,_,_) → BLE base_rs rt (Imm16 offset)
    (0x05,_,_) → BNE base_rs rt (Imm16 offset)
    ...

mipsDecodeRegImm ::
    (IntSz FIVE, IntSz FIVE, IntSz SIZE16)
    → MIPSCmd (IntSz FIVE)
mipsDecodeRegImm (base_rs,rt,offset) = cmd
  where
    cmd = case (base_rs,rt) of
            (0x00,0x11) → BAL (Imm16 offset)
            ...

mipsDecodeSpecial :: IntSz SIZE26 → MIPSCmd (IntSz FIVE)
mipsDecodeSpecial low26 = cmd
  where
    rs, rt,rd, bits5 :: IntSz FIVE
    specialOp :: IntSz SIX
    (rs,rt,rd,bits5,specialOp) = castWires low26
    cmd = case (rs,rt,rd,bits5,specialOp) of
      (0,0,0,0,0) → Nop
      (rs,rt,rd,0x00,0x21) → AddU rs rt (RI rd)
      ...
```

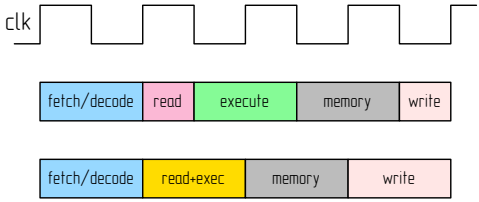
Все заметные отличия сводятся к возможности более эффективного синтеза описаний компонентов. Именно поэтому данные декодера plasma идут по своим отдельным шинам. Зато при реализации выполнения команды на Haskell мы не перепутаем местами индекс регистра приёмника и одного из операндов. Нам не надо заводить типы ALU_NOTHING, MULT_NOTHING и им подобные, у нас есть тип **`Maybe`**. В общем и целом преобразования получаются проще и надёжнее.

При построении конечных автоматов алгебраические типы ограничивают множество данных, доступных на каждом шаге. Одно это упрощает жизнь, поскольку, с одной стороны, не приходится следить за использованием неопределённых данных, а, с другой стороны, приходится продумывать пути вычисления данных, необходимых на каждом этапе, дисциплинируя разработчика.

Практически все конечные автоматы реализуются через примитив `mealy`, показанный выше. Функция преобразования состояния — первый параметр `mealy` — чистая и может быть проверена на все краевые случаи отдельно от системы. При тестировании не надо строить код, который приведёт внутреннее состояние в надлежащее, достаточно подать его на вход нашей функции преобразования.

3.3.5. Потактовая точность

О потактовой точности следует рассказать чуть более подробно. Рассмотрим пример работы конвейера типичного RISC процессора.



Вверху показано изменение сигналов тактовой частоты, затем идёт конвейер команд для настоящего процессора [7], а внизу находится конвейер такой же длины в тактах, все операции которого начинаются на одном и том же изменении тактовой частоты.

Видимые изменения будут производится на шагах memory и write, где будут выполняться запросы к памяти и записи в регистровый файл. Завершение этапа write у второго варианта точно совпадает с завершением его же в первом варианте, завершение этапа memory сдвинуто на половину такта, но при этом темп выполнения остался тот же.

Темп выполнения и длительность конвейеров в целом весьма важны, и, в принципе, достаточно следить только за сохранением этих двух параметров, подстраивая остальные под свои нужды.

3.4. Результаты применения подхода в жизни

Мы использовали язык Haskell для создания модели современного микропроцессора с системой команд MIPS32 и внеочередным выполнением команд (out-of-order execution), параллельно были созданы модель контроллера памяти и модель динамической памяти. Модели контроллера памяти и самой памяти служили гарантией, что модель нашего микропроцессорного ядра будет работать в условиях, максимально приближенных к реальным — нам было известно, что многие идеи, хорошие на бумаге, показывали плохие результаты при соединении с обычной памятью со всеми её задержками.

Над проектом работала команда из двух программистов и двух инженеров; по паре «инженер с программистом» на модель ядра и на модель инфраструктуры памяти. Инженеры выступали в роли консультантов и контролёров, их время использовалось только частично.

Уже через четыре месяца после запуска работ проект смог показать работу системы на разных задачах с учётом всех интересовавших нас параметров. Известные нам примеры менее сложных проектов (без модели памяти, только ядро с моделью кэша) потребовали нескольких человеко-лет для достижения такого же уровня модели.

Модель системы получилась высоко параметризованной: 14 параметров ядра процессора (размер кэшей и связанных буферов, размер окна предпросмотра, параметры предпросмотра и т. д.) и 9 параметров контроллера памяти (параметры алгоритма и размеры буферов) и самой памяти (пропускная способность).

Общий объём кода модели — 3400 полезных строк кода. Модель ядра микропроцессора — 2100 строк кода.

Модель опровергла некоторые наши первоначальные предположения, основным из которых являлся тезис о возможности (с использованием доступных нашей команде ресурсов) построения микропроцессорного ядра, способного декодировать видео высокого разрешения. После провала прототипа микропроцессора мы решили использовать специализированные аппаратные модули для декодирования видео и сосредоточиться на проблемах, критичных для нашей системы-на-кристалле: видеоконтроллер для телевидения высокой чёткости, декодирование и демультиплексирование потоков цифрового телевидения, DRM,[10] интеграционные задачи разного уровня.

Возвращаясь к вопросу о скорости, стоит отметить проведённое нами неформальное сравнение моделей двух, примерно одинаковых по функциональности, устройств. Одна из моделей была написана на SystemC, другая — на Haskell. Модель на языке Haskell показала заметное увеличение скорости моделирования по сравнению с моделью на SystemC, от 2 раз (с включёнными отчётами о работе устройств) до 100 (без отчётов). Причины столь гигантского разрыва кроются во «встроенной» в ленивые языки «оптимизации» — отчёты просто не вычислялись, а расходы на протягивание отложенных вычислений по иерархии компонентов минимальны. Второй причиной является то, что к ленивым спискам были применены методы оптимизации, давно и

[10] Digital Rights Management, защита видео- и аудиоинформации.

хорошо известные функциональному сообществу [2].

3.5. Заключение

Наш опыт говорит о том, что функциональные языки вполне применимы для моделирования цифровых электронных компонентов.

Помимо простоты написания описаний моделей, функциональные языки дают ещё и высокую скорость моделирования.

Строгая система типов с выводом типов и алгебраическими типами данных позволяет описывать типичные микроэлектронные решения просто и безопасно.

3.6. Краткий обзор библиотек моделирования аппаратуры

Lava

Страничка: http://raintown.org/lava/

Самый известный язык описания аппаратуры, встроенный в Haskell. Схема описывается комбинаторами.

Алгебраические типы данных не поддерживаются.

Hydra

Страничка: http://www.dcs.gla.ac.uk/~jtod/Hydra/

Язык описания аппаратуры. Позволяет описывать комбинаторную логику, получать нетлисты[11] (результат синтеза).

В реализации используется Template Haskell, что позволяет из одного и того же текста программы на Haskell получить и модель на бесконечных списках, и результат преобразований в библиотечные элементы.

Алгебраические типы данных не поддерживаются.

Hierarchical Sorting Dataflow Machine

Страничка: http://thesz.mskhug.ru/svn/hiersort/ (SVN)

Пример модели аппаратуры с использованием бесконечных списков. Ядро модели находится в подкаталоге core, содержит порядка 25 строк кода и может быть использовано для написания других моделей аппаратуры.

Получение синтезируемого описания модели не предусмотрено.

Алгебраические типы данных поддерживаются.

[11] Netlist — очень подробный уровень описания аппаратуры, представляет из себя просто список компонентов с соединяющими их проводами.

Другие языки

Для OCaml было создано два языка описания аппаратуры: HDCaml и Confluence. Оба языка больше не поддерживаются, оставшиеся исходные коды можно найти на `http://funhdl.org/`. Ни в одном из них не было сделано попытки реализовать поддержку ни алгебраических типов данных, ни полиморфизма.

Литература

[1] *Abelson H., Sussman G. J.* Structure and Interpretation of Computer Programs, 2nd Edition. — The MIT Press, 1996. `http://mitpress.mit.edu/sicp/`.

[2] *Andrew Gill J. L., Jones S. L. P.* A short cut deforestation. — 1993. — Преобразования вычислений над ленивыми списками. В частности, устранение промежуточных списков и ненужного выделения памяти.

[3] *Carlsson M., Hallgren T.* Fudgets — purely functional processes with applications to graphical user interfaces. — 1998. — Варианты ввода-вывода для чистых функциональных языков, отличные от монадического подхода и подхода с уникальными типами.

[4] ForSyDe: Formal System Design. — Язык описания аппаратуры, встроенный в Хаскель через Template Haskell. Уровнем чуть выше, чем Hydra.

[5] *John Matthews B. C., Launchbury J.* Microprocessor specification in hawk. — Язык спецификации цифровых электронных схем, встроенный в Хаскель.

[6] Mips, microprocessor without interlocked pipeline stages. — URL: `http://en.wikipedia.org/wiki/MIPS_architecture` (дата обращения: 28 сентября 2009 г.). — Один из самых простых RISC процессоров (и, пожалуй, самый элегантный).

[7] Plasma - most mips i(tm) opcodes. — URL: `http://www.opencores.org/project,plasma` (дата обращения: 28 сентября 2009 г.). — Минималистическая реализация ядра MIPS.

[8] Prelude.interact :: (String -> String) -> IO (). — Функция диалога с клиентом, URL: `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:interact` (дата обращения: 28 сентября 2009 г.). — Один из первых вариантов ввода-вывода для ленивых функциональных языков.

[9] The Hydra Computer Hardware Description Language. — URL: `http://www.dcs.gla.ac.uk/~jtod/Hydra/` (дата обращения: 28 сентября 2009 г.). — Высокоуровневый язык описания аппаратуры с использованием Template Haskell.

[10] The Lava hardware description language. — URL: http://raintown.org/lava/ (дата обращения: 28 сентября 2009 г.). — Один из первых языков описания аппаратуры, встроенный в Haskell.

[11] *Харольд Абельсон, Джеральд Джей Сассман*. Структура и интерпретация компьютерных программ. — М.: Добросвет, 2006.

Использование Scheme в разработке семейства продуктов «Дозор-Джет»

Алексей Отт
alexott@fprog.ru

Аннотация

Данная статья представляет собой краткий обзор использования функционального программирования в разработке семейства продуктов «Дозор-Джет» — программного обеспечения для контентной фильтрации трафика, применяемого для предотвращения утечек производственной информации, и соблюдения законодательства в части сохранения информации [2], [1].

This article provides a brief overview of the use of the functional programming in the development of the «Dozor-Jet» family of products. The «Dozor-Jet» software employs traffic content filtering to provide enterprise information leak prevention and ensure legal compliance.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/2368.html.

*Я хочу поблагодарить своих коллег по «Инфосистемам Джет» за помощь в написании этой статьи.

4.1. Что такое «Дозор-Джет»?

Семейство продуктов «Дозор-Джет» состоит из нескольких продуктов, реализующих функциональность по фильтрации почтового и веб-трафика. История данного семейства продуктов началась в 1999 году с разработки системы мониторинга и архивирования почтовых сообщений (СМАП) для одного крупного заказчика. Первая версия продукта[1] была поставлена заказчику в начале 2000 года [4]. Постепенно систему начали внедрять и у других заказчиков, и она стала обретать черты «коробочного» продукта. К 2006 году система была внедрена уже у более, чем 200 клиентов, среди которых были крупные государственные и коммерческие организации, объем продаж достиг нескольких миллионов долларов в год.[2]

В 2005 году был выпущен еще один продукт семейства «Дозор-Джет» — система контроля веб-трафика (СКВТ) [3], на основе которой были затем разработаны и другие продукты этого семейства.

Отличительной чертой линейки продуктов по сравнению с конкурирующими продуктами являлась возможность построения сложных условий обработки трафика, поддержка всех используемых в России и СНГ кодировок,[3] автоматическое определение типов файлов и кодировок документов, большое количество поддерживаемых форматов документов и архивов. Все это позволяло создавать очень гибкие политики безопасности и предотвращать утечки важной для организации информации.

С точки зрения программиста эта линейка продуктов интересна тем, что при её разработке активно использовался язык программирования Scheme (а именно, PLT Scheme). СМАП «Дозор-Джет» практически полностью написан на этом языке (за исключением небольших платформенно-зависимых частей, написанных на языке C), а в СКВТ Scheme используется для реализации серверной части веб-интерфейса.

4.2. Архитектура систем

В данном разделе приводится краткое описание архитектуры продуктов семейства «Дозор-Джет».[4]

4.2.1. Архитектура СМАП «Дозор-Джет»

СМАП «Дозор-Джет» может функционировать в двух режимах: режиме фильтрации почты, когда полученные сообщения обрабатываются в соответствии с опреде-

[1] Первая версия продукта была разработана силами трех разработчиков, и после начала продаж группа постепенно была увеличена до восьми человек, по мере увеличения функциональности продукта.

[2] Данные по продажам взяты из пресс-релиза компании, опубликованного в журнале Jet Info, №10 за 2006-й год, стр. 22.

[3] Это было одной из проблем при использовании продуктов иностранных компаний.

[4] Описания архитектуры взяты из официальной документации соответствующих продуктов, которую вы можете найти на сайте компании: `http://www.jetsoft.ru/`. Документация также содержит подробное описание возможностей продуктов данного семейства.

ленной политикой безопасности, после чего система принимает решение о дальнейшей отправке или задержании сообщения, и режиме архивации, когда почтовые сообщения после обработки могут быть сохранены в долговременном архиве.

СМАП «Дозор-Джет» состоит из нескольких взаимодействующих между собой подсистем (см. рис. 4.1). К основным подсистемам относятся:

- Подсистема приема почтовых сообщений, обеспечивающая прием почты от внешних клиентов и серверов. На этом этапе производится первоначальная фильтрация почтовых сообщений для предотвращения рассылки спама и несанкционированной отправки почты через почтовый сервер. Эта подсистема была реализована на базе SMTP-прокси из другого продукта компании — Z–2.

- Подсистема фильтрации, которая обеспечивает обработку почтовых сообщений в соответствии с политикой безопасности и принимает решение о дальнейшей судьбе почтового сообщения – должно ли оно быть отправлено получателю, помещено в архив, требуется ли уведомить администратора безопасности или совершить сразу несколько действий.

- Подсистема выполнения действий над письмами, которая выполняет конкретные действия, определенные в процессе фильтрации почтовых сообщений. Эта подсистема также используется для выполнения периодических и отложенных действий.

- Подсистема управления, предоставляющая веб-интерфейс, через который администратор может управлять как политикой безопасности, так и системными настройками продукта. Данная подсистема состоит из серверной части, написанной на Scheme, и клиентской части, написанной на JavaScript. Политики безопасности вместе с другой информацией хранятся в базе данных. Также через веб-интерфейс производится работа с архивом почтовых сообщений, хранящимся в базе данных.

- Подсистема архивации, реализующая интерфейс к базе данных (Oracle или PostgreSQL) и обеспечивающая работу с почтовыми сообщениями, хранимыми в базе данных.

- Монитор ресурсов, который отслеживает наличие всех необходимых для работы процессов, следит за свободным местом на диске и в базе данных, и в случае неполадок, оповещает системного администратора.

Практически все подсистемы СМАП, за исключением подсистемы приема почтовых сообщений и клиентской части подсистемы управления, написаны на языке Scheme.

Модульная архитектура СМАП позволяет разнести разные подсистемы по нескольким серверам (когда это было необходимо), обеспечивая балансировку нагрузки между серверами и надежность работы комплекса. Эксплуатация продукта у

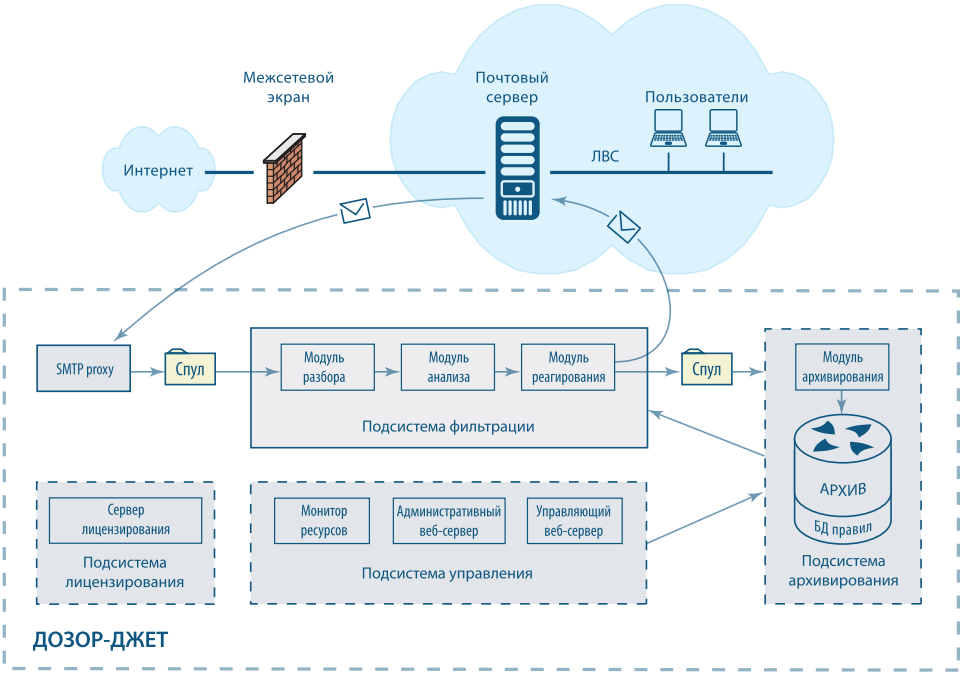


Рис. 4.1. Архитектура СМАП «Дозор-Джет»

больших клиентов показала, что система может обрабатывать десятки гигабайт почтовых сообщений в день даже при использовании одного сервера фильтрации (при выделенном сервере базы данных).

4.2.2. Архитектура СКВТ «Дозор-Джет»

В отличие от СМАП, СКВТ «Дозор-Джет» работает только в одном режиме — режиме фильтрации трафика. Система состоит из следующих подсистем, стандартных для систем фильтрации веб-трафика:

- Подсистема фильтрации трафика, выполняющая проверку передаваемых данных на соответствие политике безопасности, а также архивирование передаваемой информации (веб-почта и т. п.) в СМАП «Дозор-Джет»;
- Подсистема управления, предоставляющая веб-интерфейс администратора, предназначенный для работы с политиками безопасности, предоставления отчетов о работе системы и выполнения прочих административных задач;
- Подсистема кэширования данных на базе кэш-сервера Squid.

В данном продукте на языке Scheme написана только серверная часть подсистемы управления, при этом используются те же самые библиотеки, что и в подсистеме управления СМАП «Дозор-Джет». В связи с этим реализация веб-интерфейса СМАП «Дозор-Джет» практически полностью соответствует реализации интерфейса СКВТ «Дозор-Джет».

4.3. Почему Scheme?

Язык Scheme[5] был выбран по ряду причин, подробнее описанных ниже:

- *Простота и выразительность языка.* Scheme — достаточно простой язык, с легковесной средой выполнения. Существовавший на то время стандарт языка, R5RS, занимал около 50 страниц, в отличие от значительно более «подробного» Common Lisp. Простота языка позволяла быстро вводить новых разработчиков в курс дела, даже если до прихода в компанию они не имели опыта разработки на языке Scheme.

  В процессе разработки использовались макросы Scheme, с помощью которых сильно сокращался объем кода, который необходимо было поддерживать, и программа писалась в терминах целевой предметной области. Стоит отметить, что размер кода системы СМАП (вместе с различными модулями расширений и дополнительными утилитами) составляет порядка 35 тысяч строк на Scheme и несколько тысяч строк кода на языках C и C++. Для сравнения, размер кода одного из конкурирующих продуктов (с меньшей функциональностью), написанного на языке C++, составляет более 200 тысяч строк.

  Сокращение размера кода позволило уменьшить общее количество ошибок в продукте, а достаточно небольшой процент кода на C и C++ позволил избежать типичных для этих языков ошибок, таких как утечки памяти и ошибки доступа к памяти.

- *Возможность интерактивной разработки.* Используя REPL, разработчик может писать и тестировать код без пересборки всего продукта, что существенно ускоряет разработку частей продукта. За счет более короткого цикла разработки время вывода продуктов на рынок было существенно сокращено.

- *Возможность выполнения сгенерированного кода во время выполнения программы* — политика безопасности, созданная администратором безопасности, преобразовывается в исходный код Scheme, проводится оптимизация этого кода и выполняется как часть программы, обновляясь на лету, без перезапуска процессов.

[5]Для первых версий СМАП «Дозор-Джет» использовалась PLT Scheme v103, но позднее, в 2004 году, был выполнен переход на версию PLT Scheme v30x, в которой реализовано много полезных вещей, таких как FFI, система модулей и т. п.

- *Переносимость*. Кроссплатформенность Scheme позволила использовать СМАП «Дозор-Джет» на разных Unix-совместимых операционных системах — различные дистрибутивы Linux, Sun Solaris на процессорах Sparc и x86/AMD, HP-UX на процессорах PA-RISC. Платформенно-зависимая часть продукта составлял несколько сотен строк на языке C и в основном использовалась для реализации привязки к различным библиотекам, в том числе и для баз данных.

Еще одной причиной выбора этого языка, было наличие наработок в части создания веб-приложений на языке Scheme — наличие готовых компонентов, что также позволило уменьшить время вывода продукта на рынок.

4.4. Использование DSL

В продуктах семейства также используются и Domain-Specific Languages — для библиотеки определения типов, созданной для замены библиотеки `libmagic` и утилиты `file`, был спроектирован отдельный язык с Lisp-образным синтаксисом. Данный язык позволяет разработчику или администратору безопасности описывать процедуру точного определения типа файла, используя логические операторы, и функции доступа к содержимому определяемого файла ([2], с. 35). Пример процедуры определения типов, написанной на этом языке приведен ниже:

```
; RAR archives
(and (= @0 "Rar!\x1a")
     (byte @44)
     (<= (byte @35) 4))
  => (format "RAR archive data, v~x, " (byte @44))
  => (case (byte @35)
       ((0) "os: MS-DOS")
       ((1) "os: OS/2")
       ((2) "os: Win32")
       ((3) "os: Unix"))

(= @0 "MZ")
> (and (lelong @#x3c)
       (= @(lelong @#x3c) "PE\x0\x0"))
    => "MS Windows PE"
> (and (lelong @#x3c)
       (= @(lelong @#x3c) "NE"))
    => "MS Windows 3.x NE"
> #t
    => "MS-DOS executable"
    => (and (= @24 "@") "\b, OS/2 or MS Windows")
```

```
      => (and (search "Rar!" @#xc000 10000)
"\b, Rar self-extracting archive")
      => (and (= @11696 "PK\003\004")
"\b, PKZIP SFX archive v1.1")
```

В приведенном примере показаны процедуры определения типов для архивов в формате RAR, а также для исполняемых файлов MS DOS и MS Windows. Как видно из примера, язык позволяет описывать достаточно сложные проверки, недоступные в существующих библиотеках, включая объявление и использование локальных (для правил) переменных.

Политики безопасности СМАП также можно рассматривать как пример программы на DSL. Обладая соответствующими знаниями, можно модифицировать политику, не прибегая к использованию веб-интерфейса. Сформированная администратором политика безопасности, компилируется в программу на Scheme, загружается в среду выполнения, и автоматически начинает применяться ко всем новым почтовым сообщениям. Пример скомпилированной политики безопасности вы можете увидеть ниже:

```
(define |filter-file:Confidential data| (make-filter-file "filter-file.100"))
(define |message:Confidential data|
  '(("from" "admin@localhost")
     ("to" "security_admin@localhost")
     ("subject" "Confidential data found!"))
    "Mail with confidential data was found"
    "Confidential data"))
(define (|cond:All mail| message)
  (log-condition "cond:All mail" (lambda (message) #t) message))
(define (|cond:Executable files| message)
  (log-condition "cond:Executable files"
   (lambda (message)
     (iter-mime-part1
      (rfc822:message-info-body (filtering-info-message message))
      (lambda (x)
        (string-contains-ci? (mime-part-content-type x) "executable"))))
   message))
(define (|cond:Confidential data| message)
  (log-condition "cond:Confidential data"
   (lambda (message)
     (iter-mime-part1
      (rfc822:message-info-body (filtering-info-message message))
      (lambda (x)
        (text-match-regexp-list?
         (mime-part-text-file x)
         (filter-file-escaped-regexp-list
          |filter-file:Confidential data|)))))
   message))
(define (|set:Template rule set| message)
  (call/ec
   (lambda (return)
     (let ((result (|cond:All mail| message)))
       (when result (action:relay-message message) (return #t)))
     #f)))
(define (|set:Main Rule| message)
  (call/ec
   (lambda (return)
     (let ((result (|cond:Confidential data| message)))
       (when result
         (action:archive-message message)
         (action:send-notification message #t |message:Confidential data|)
         (return #t)))
     (let ((result (|cond:Executable files| message)))
       (when result (action:void) (return #t)))
     (let ((result (|cond:All mail| message)))
       (when result (action:relay-message message) (return #t)))
     #f)))
|set:Main Rule|
```

В данной политике определяется несколько условий («All mail», «Confidential data» и «Executable files») и действий, из которых формируется политика безопасности под названием «Main Rule». Данная политика выполняет следующие действия с проходящей почтой:

- проверяет текст писем (включая вложения) на наличие конфиденциальной информации, и в случае обнаружения, информирует администратора безопасности о таком письме, вместе с его задержанием в архиве;
- удаляет письмо в том случае, если в письме передается исполняемый файл;
- доставляет адресатам всю остальную почту.

Каждое условие определенное администратором превращается в набор итераторов по соответствующим частям письма — заголовков письма, вложений, текстовых частей и т. п. К этим частям применяются базовые условия, определенные администратором, в случае срабатывания которых, условие возвращает истинное значение, и выполняются действия, указанные администратором. Базовые условия могут объединяться друг с другом, используя логические операции, что позволяет формировать сложные условия проверки частей письма.

4.5. Реализация СМАП

4.5.1. Подсистема фильтрации

Подсистема фильтрации является самой важной подсистемой СМАП «ДозорДжет» — она обрабатывает всю почту, попадающую в систему, и, в зависимости от политики безопасности, определенной администратором безопасности, принимает решение о дальнейшей её судьбе.

Для определения политики безопасности администратор безопасности может использовать различные комбинации из условий и действий. При этом список условий и действий не является фиксированным, они могут добавляться путем подключения новых модулей, расширяющих функциональность продукта.

В качестве условий могут использоваться различные параметры обрабатываемых писем[6]: содержимое заголовков письма, кем оно отправлено и кому оно предназначено, количество, размер и формат вложений и многое другое. При этом отдельные условия могут комбинироваться в более сложные посредством логических операторов.

В зависимости от срабатывающих условий администратор может определять различные действия и их комбинации: дальнейшую отправку письма, помещение письма

[6]В СМАП практически для всех параметров используются собственные процедуры определения корректных значений формата файлов, кодировки текста и т. д. Это позволяло избежать проблем с неправильным указанием параметров письма в MUA, а также правильно обрабатывать письма с целенаправленно измененным содержимым: переименованными файлами и т. п.

в архив, отправку уведомления о письме, выдачу запроса отправителю письма на явное подтверждение отправки письма и т. д.

В текущей версии продукта все условия по умолчанию являются «ленивыми» — вычисление соответствующих параметров (разбор заголовков письма, извлечение вложений из письма, извлечение текста из документов) откладывается до первого вызова соответствующего условия. Это позволяет снизить нагрузку на систему, особенно в тех случаях, когда политика безопасности достаточно простая и не требует разбора всех вложений.

Обработка вложений обычно производится с помощью внешних утилит, которые извлекают вложенные объекты (для архивов) и текст (для документов). Затем извлеченные части передаются на обработку в соответствии с политикой безопасности, и процесс повторялся заново до тех пор, пока не будет извлечен последний вложенный элемент, или размер распакованных данных не достигнет заданного предельного размера.

4.5.2. Веб-интерфейс

Веб-интерфейс администратора безопасности состоит из серверной и клиентской частей. Клиентская часть написана на JavaScript и использует AJAX для получения данных от сервера и представления их в браузере.

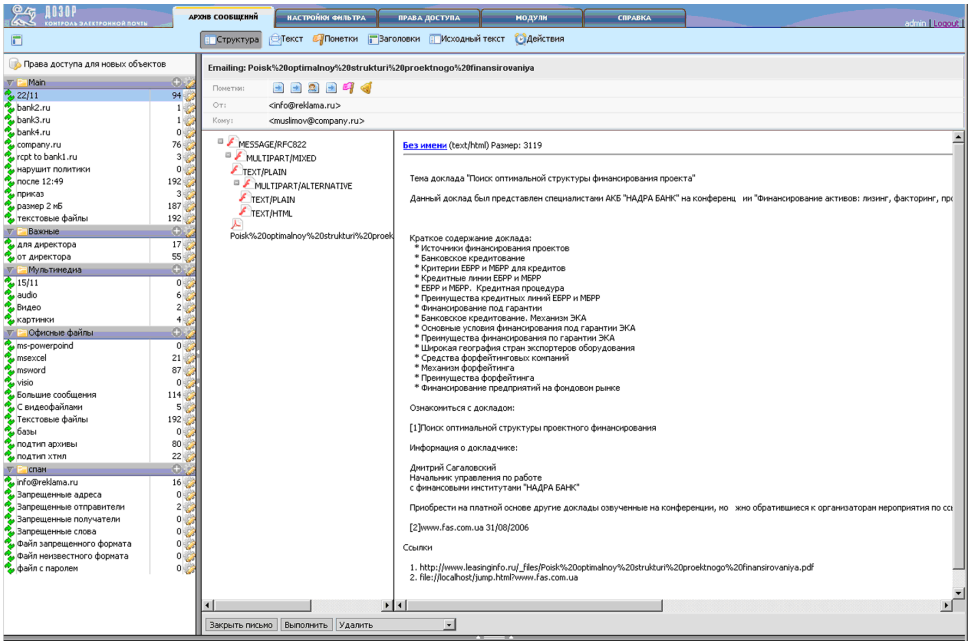


Рис. 4.2. Пример интерфейса СМАП «Дозор-Джет»

Серверная часть реализована на Scheme и использует самостоятельно написанный mod_mzscheme для выполнения кода внутри веб-сервера Apache.[7] Веб-сервер производит отдачу статического и динамического контента. Динамический контент генерируется из файлов, содержащих шаблоны, которые заполняются данными из базы данных или конфигурационных файлов. Возможность выполнения кода на Scheme внутри веб-сервера позволяет существенно ускорить работу веб-интерфейса за счет однократной загрузки кода и сохранению открытых соединений с базой данных между запросами к серверу.

Пример интерфейса администратора безопасности вы можете видеть на рис. 4.2.

4.6. Основные результаты

Как показывает десятилетний опыт разработки и эксплуатации продуктов линейки «Дозор-Джет», функциональные языки могут применяться для разработки коммерческих продуктов, поставляемых конечным пользователям. За счет оперирования высокоуровневыми понятиями и применения REPL, цикл разработки продуктов может быть ускорен, а количество разработчиков может сравнительно малым, что позволяет выводить продукты на рынок в более короткие сроки и с меньшими затратами.

Литература

[1] *Алексей Отт*. О контентной фильтрации. Продолжение темы // *Jet Info*. — 2006. — № 10. — С. 2–21.

[2] *Олег Слепов*. Контентная фильтрация // *Jet Info*. — 2005. — № 10.

[3] *Олег Слепов, Алексей Отт*. Контроль использования Интернет-ресурсов // *Jet Info*. — 2005. — № 2.

[4] *Александр Таранов, Владимир Цишевский*. Система мониторинга и архивирования почтовых сообщений // *Jet Info*. — 2001. — № 9. — С. 10–28.

[7] В последних версиях СКВТ вместо связки Apache + mod_mzscheme используется веб-сервер, поставляемый вместе с PLT Scheme.

Как украсть миллиард
или Давайте сделаем это по-быстрому

Александр Самойлович
samoylovich@fprog.ru

Аннотация

В статье описывается создание программы типа crawler, которая обходит интернет-сайт, находит и сохраняет на диске данные нужных типов. Программа будет написана на языке Erlang. Разработка программы будет вестись «сверху вниз». Цель статьи — показать, как свойства Erlang позволяют значительно ускорить разработку и выполнение программы по сравнению с использованием других языков, не требуя при этом от программиста практически никаких дополнительных усилий.

The article describes the creation of a spider software which crawls the web site in search of files of certain types, and saves them to disk. The program is written in Erlang using the «top-down» approach. This article aims to show how Erlang allows for significant acceleration of development process and minimization of the run time of the program compared to other programming languages with no extra effort on the part of the programmer.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/2735.html.

5.1. Введение

Эта программа появилась в результате решения прикладной задачи. Несколько лет назад мне понадобилось выкачать с некоторого сайта хранящиеся на нем картинки. Картинок было много, примерно несколько тысяч, может быть, десятки тысяч. Прихотливой рукой автора они были щедро разбросаны по всему сайту. Придумать простой алгоритм описания того, где они лежат, мне не удалось. Поэтому в конце концов была написана программа, которая обходила весь сайт, находила картинки и сохраняла их. Задача оказалась достаточно удобной для использования в целях самообразования. С тех пор, начиная изучать новый язык программирования, я пишу на нем очередную реализацию этой программы. Предпоследняя из них была составлена на Scheme. Программа работала и выдавала разумные результаты, но дождаться окончания её работы мне не удалось. Насколько я понимаю, единственным ограничением производительности была однопоточность. Программа была переписана на Erlang. Этот процесс и воспроизводится в статье.

Документация

Документации, статей и книг про Erlang сейчас существует намного меньше, чем, например, про Haskell. На английском языке выпущено две книги: Programming Erlang (автор Joe Armstrong) и Erlang Programming (авторы Francesco Cesarini и Simon Thompson). Описание языка, примеры и иные материалы можно найти на официальном сайте Erlang: www.erlang.org. Например, по адресу `http://www.erlang.org/download/getting_started-5.4.pdf` можно найти введение в Erlang. Собрание немногочисленных «рецептов» (cookbook) лежит здесь: `http://schemecookbook.org/Erlang/WebHome`.

Алгоритм работы

Алгоритм работы нашей программы прост:

1) получаем текст страницы;
2) находим в ней все интересующие нас ссылки;
3) в зависимости от типа ссылки либо сохраняем ссылку (не содержимое), либо переходим к пункту 1, либо игнорируем её;
4) когда все интересующие нас ссылки найдены, сохраняем содержимое этих ссылок на диске.

Сохранение картинки разделено на два шага (1—3 и 4) специально. Сначала мы сохраняем только ссылку на картинку, а потом получаем ее содержимое. Так было сделано потому, что не все картинки были мне интересны, и хотелось иметь возможность редактировать список ссылок для сохранения вручную.

Архитектура

Одна из главных особенностей Erlang — легковесные процессы. В связи с этим проектирование программ на Erlang — это разбиение задач на независимые процессы и определение способов взаимодействия между ними. Процессы в Erlang создаются легко и стоят дешево, поэтому наличие тысяч или десятков тысяч процессов в одной задаче — вполне обычное дело. Процессы в Erlang можно рассматривать как аналоги классов в C++ или Java. Для каждой задачи (независимой активности) создаётся отдельный процесс. Процессы могут обмениваться между собой сообщениями. Посылка сообщений асинхронна. Послав сообщение, процесс продолжает свою работу. Сообщения, передаваемые между процессами, обслуживающимися одним и тем же экземпляром виртуальной машины Erlang, не теряются. Посланное сообщение попадает в очередь сообщений процесса независимо от того, исполняется он на той же машине или где-то в сети. Данные между процессами не разделяются, а копируются. Процессы не имеют доступа к внутренностям друг друга.

В реальных проектах часто создаётся иерархическая система рабочих и наблюдающих процессов, благодаря чему достигается высокая надежность приложения. Но это тема для отдельной большой статьи. А наше приложение — игрушечное, и нам потребуется лишь очень упрощенная система контроля за исполняющимися процессами.

В нашей задаче процессы будут использоваться для получения содержимого вебстраниц из интернета. Для каждой обрабатываемой единицы текста, будь то страница или строка, заведем отдельный процесс. После того, как страница будет проанализирована, и вся необходимая информация извлечена, найденные ссылки надо будет сохранить в файле. Файл один, а процессов много. Если все они станут писать одновременно — неприятностей не избежать, нужно будет решать задачу синхронизации. Это возможно, но есть более простой подход, решающий проблему автоматически: мы заведем специальный процесс, который будет писать в файл, а все остальные будут посылать ему сообщения о том, что они хотели бы записать.

Дочерние процессы могут посылать два вида сообщений: сообщения записывающему процессу о том, что нужно записать данные в файл, и сообщения родительскому процессу о завершении своей работы.

Процесс записи будет понимать два вида сообщений: сообщения от рабочих процессов с данными для записи и сообщение от родительского процесса о том, что данных для записи больше не будет, и можно приступать с сохранению их на диск.

Замечание о долгоживущих процессах

Как уже говорилось, типичная программа на Erlang состоит из большого количества процессов. В разных задачах эти процессы имеют схожее поведение. Они создаются, контролируют поведение друг друга, обмениваются сообщениями и т. д. Такие общие черты поведения называются шаблонами поведения. В реализацию Erlang входит библиотека OTP, в которой, помимо прочего, реализованы шаблоны поведения процессов, чтобы разработчику не было необходимости каждый раз заново это по-

ведение программировать. Но так как мы хотим не только уметь пользоваться библиотечными абстракциями, но и знать, как они устроены, мы организуем долгоживущий сервисный процесс вручную, используя рекурсию, не прибегая к помощи модуля `gen_server` из библиотеки OTP.

Организация кода, модули

Файлы с кодом программ на Erlang называются модулями, расширение файлов с исходным текстом должно быть .erl. Изнутри модуля его функции могут быть вызваны просто по имени, а снаружи модуля — по длинному имени, состоящему из названия модуля и названия функции. Наш модуль будет называться `download.erl`, а функция, которая обходит сайт и сохраняет адреса файлов на диске — `start()`. Снаружи, например, из командной строки Erlang, эта функция будет вызываться как `download:start()`.

В начале каждого модуля обязательно идёт заголовок, содержащий имя этого модуля, которое должно совпадать с именем файла без расширения. После заголовка мы описываем, какие функции можно вызывать извне этого модуля. По умолчанию, все функции модуля доступны только изнутри, снаружи модуля они не видны. Для того, чтобы функцию можно было вызывать извне, она должна быть описана командой `-export(...)`. К ней мы вернемся в конце статьи. А сейчас, для упрощения отладки, сделаем все функции модуля публичными:

```
-module(download).
-compile(export_all).
```

Замечания об отладке

В жизни программа создавалась «сверху вниз», практически в той же последовательности, как это представлено в статье. Работоспособность каждой вновь написанной функции проверялась путем ее вызова в командной строке Erlang с соответствующими параметрами. Для того, чтобы скомпилировать функцию более высокого уровня без реализации всех функций более низкого уровня, нам необходимо создать функции-заглушки. Например, пусть нужно написать такую заглушку для функции f(P1, P2). Если возвращаемое значение нас не интересует, то заглушка может выглядеть как:

```
f(_P1, _P2) -> ok.
```

Символы подчеркивания перед аргументами — это сообщение компилятору о том, что в дальнейшем аргументы не используются, и их значения не важны. Если этого не сделать, компилятор выдаст предупреждение о неиспользованных переменных.

В Erlang есть настоящие отладчики, например, debugger, но для нашего приложения вполне достаточно диагностических сообщений в консоли.

Комментарии

Комментарии в Erlang однострочные. Комментарием считается все, начиная с символа % и до конца строки. Комментарии разных уровней принято выделять разным количеством %. Например, комментарии, относящиеся ко всему модулю — %%%, комментарии, описывающие функцию — %%, а комментарии к строке кода — %.

Константы

Во время работы нам потребуется несколько параметров, например, URL сайта, который мы будем обходить, и имя файла, куда будут записываться промежуточные результаты. Их можно задать, определив их константами, воспользовавшись механизмом макросов или передать аргументами в функцию. Для простоты, заведем несколько функций, которые будут возвращать нужные нам строки:

```
get_start_base() -> "http://airwar.ru/".
get_start_url() -> get_start_base() ++ "image/".
out_file_name() -> "pictures.txt".
```

5.2. Разработка

Главная функция обхода

Функция `start()` будет вызываться из командной строки. В начале работы она произведет инициализацию библиотеки `inets`, которая будет читать для нас данные из интернета. Затем мы создадим процесс Writer, записывающий на диск интересующие нас ссылки. Каждый процесс, который хочет что-то записать, должен будет послать процессу Writer сообщение об этом. Функция обработки веб-страниц `process_page()` сделает всю работу по разбору текстов страниц и определению типов найденных ссылок. И, наконец, мы информируем записывающий процесс, что все данные обработаны, и он может завершать свою работу.

```
start() ->
    inets:start(),
    Writer = start_write(),
    process_page(Writer, get_start_url()),
    stop_write(Writer).
```

Процессы создаются стандартной функцией `spawn()`. При запуске процесса ей передаётся функция процесса. Процесс завершится вместе с завершением этой функции. Сам же `spawn()` возвращает родителю идентификатор запущенного процесса-потомка, «pid». Полученный идентификатор можно использовать в качестве адреса при посылке процессу сообщений: `Pid ! AnyMessage`.

Мы хотим, чтобы наш процесс жил достаточно долго для того, чтобы обработать неизвестное заранее число сообщений, поэтому мы будем использовать рекурсию, которая позволит нам принять и обработать потенциально неограниченное количество сообщений одно за другим. Один вызов функции служит для обработки одного сообщения. Накапливать данные мы будем, передавая их аргументом рекурсивно вызываемой функции. Так как вначале работы список сохраненных адресов картинок пуст, аргументом для первого вызова будет пустой список `[]`.

У функции `spawn()` есть несколько вариантов вызова. В этой статье мы будем пользоваться только одним их них:

```
start_write() -> spawn(fun write_proc/0).
```

Здесь мы передали в `spawn()` безаргументную функцию `write_proc()`, которую определим ниже. Также начальная функция процесса, передаваемая в `spawn()`, может создаваться и при помощи конструкции вида:

```
fun(A) -> A + 42 end
```

Так порождается анонимная функция, аналог $\lambda\alpha \rightarrow \alpha$ `+ 42` в Haskell. Мы ещё воспользуемся этим способом.

Управлять процессом Writer мы будем при помощи двух сообщений — `write` и `stop`. Для их посылки используем две функции:

```
stop_write(W) ->
    W ! stop.

write(W, String) ->
    W ! {write, String}.
```

Функция `stop_write()` посылает сообщение, состоящее из одного атома `stop`. Функция `write()` для посылки сообщения использует пару `{write, String}`, состоящую из атома `write` и произвольных данных **`String`**. Цель введения таких однострочных функций для посылки сообщения — изоляция логики программы от деталей реализации.

Функция обработки цикла сообщений процесса Writer

Функция обработки цикла сообщений процесса Writer должна знать о существовании сообщений `stop` и `{write, String}`, описанных в предыдущем разделе. Для получения сообщений в Erlang в общем случае используется конструкция:

```
receive
    Pattern1 when Guard1 -> expr_1_1, ..., expr_1_N;
    Pattern2 when Guard2 -> expr_2_1, ..., expr_2_N;
    ...
    PatternM when GuardM -> expr_M_1, ..., expr_M_N
after
    Timeout -> expr_1, ..., expr_N
```

```
end
```

Ветвь будет выполнена, только если будет найдено соответствие PatternN, и при этом GuardN будет истинным. Guard — это выражение, которое принимает значение `true` или `false`. Часть **`when`** `Guard` не является обязательной. В этой статье она нам не понадобится, поэтому выражение `receive` будет выглядеть так:

```
receive
    stop -> expr_1_1, ..., expr_1_N;
    {write, String} -> expr_2_1, ..., expr_2_N;
after
    Timeout -> expr_1, ..., expr_N
end
```

Дойдя до блока `receive`, процесс остановится и будет ждать, пока в очереди сообщений что-то не появится. Если очередь не пуста, программа попытается найти соответствие между сообщением и образцами, описанными в разных ветвях `receive`. Сопоставление будет удачным только в том случае, если удалось найти соответствие для всех величин, входящих в данные. Сейчас у нас таких образцов два — `stop` и **`{write, String}`**. Если соответствие найдено, программа выполнит правую часть выражения, если нет — программа будет ожидать в `receive` бесконечно. Бесконечно? Но это же не то, что нам надо. Что будет, если по каким-то причинам сообщение вообще не дойдет? Тогда функция никогда не закончится, и процесс не завершится. Для обработки этого случая и используется ветвь `after Timeout`. Если в течение `Timeout` миллисекунд ни одного соответствия между пришедшим сообщением и ветвью `receive` не будет найдено, то выполнится ветвь `after`:

```
write_proc() -> write_loop([], 0).

write_loop(Data, DataLen) ->
    receive
        stop ->
            io:format("Saving ~b entries~n", [DataLen]),
            {ok, F} = file:open(out_file_name(), write),
            [io:format(F, "~s~n", [S])   ||  S <- Data],
            file:close(F),
            io:format("Done~n");
        {write, String} ->
            %io:format("Adding: ~s~n", [String]),
            case DataLen rem 1000 of
                0 -> io:format("Downloaded: ~p~n", [DataLen]);
                _ -> ok
            end,
            write_loop([String|Data], 1 + DataLen)
    after 10000 ->
            io:format("Stop on timeout~n"),
            stop_write(self()),
```

```
            write_loop(Data, DataLen)
    end.
```

В случае получения сообщения `stop` Writer откроет файл и для каждого элемента списка `Data` вызовет функцию записи в файл, после чего файл будет закрыт, и цикл обработки сообщений завершится.

Если придёт сообщение `{write, String}`, то `write_loop/2` для каждого тысячного сообщения напечатает диагностику на консоль и вновь вызовет себя, уже с новыми аргументами. Рекурсия, а именно, концевая рекурсия — это обычный способ организации циклов в Erlang. Так как переменные в Erlang не изменяются, для сохранения данных между вызовами функций используются их аргументы. В нашем случае мы добавляем величину String, полученную из сообщения, к списку данных `Data`, который был у нас на момент вызова функции `write_loop/2`. После этого рекурсивного вызова процесс `Writer` окажется там же, где и до прихода сообщения — в ожидании сообщения в инструкции `receive`.

В нашем случае ветвь `after` служит упрощённым обработчиком ошибок. Его логика такова: если в течение некоторого времени мы не получили ни одного сообщения о том, что нужно сохранить какие-то данные, это значит, что все, что может быть найдено, уже записано. Если какие-либо процессы не смогли завершиться нормально к этому моменту, то скорее всего это обозначает, что они уже никогда и завершатся в связи с неизвестными нам ошибками. Величина 10 секунд выбрана произвольно. Она должна быть достаточно велика, чтобы превышать возможные задержки, возникающие при нормальной работе сайта.

Чтобы избежать дублирования кода, в ветви `after` процесс посылает сообщение `stop` себе же, а затем вызывает функцию `write_loop/2`, чтобы его обработать. Просто выйти из функции мы не хотим, так как реальная запись данных в файл происходит именно при обработке сообщения `stop`. Посылка `stop` из `Timeout` позволит нам сохранить на диске те данные, которые уже накопились к этому моменту.

Одна из первых версий программы не содержала списка обработанных данных `Data`. Она немедленно записывала их на диск при получении сообщения `{write, String}`. Обработчик сообщения `stop` только закрывал файл. К сожалению, получилось не очень хорошо. Файловые операции в Erlang, предоставляемые библиотечным модулем `io`, довольно медленны, поэтому процесс записи на диск работал недостаточно быстро. Собрать как можно больше данных в памяти, а затем записать их в один прием — это один из примеров оптимизации скорости работы, рекомендованный Джо Армстронгом, одним из авторов языка.

Обработка одной страницы

Веб-страницу мы будем обрабатывать построчно, чтобы упростить нашу учебную задачу. Сначала прочитаем исходный текст страницы, вызвав функцию `get_url_contents()`, которую мы вскоре опишем. Если чтение закончилось с ошибкой, то есть возвращенное значение отличается от `{ok, Data}`, немедленно за-

кончим функцию. Если чтение было успешным, разобьем полученный текст на строки, и для каждой строки запустим процесс обработки строки. Функция обработки страницы не должна завершаться до тех пор, пока не закончатся все порожденные ею процессы обработки строк.

```
process_page(W, Url) ->
    MyPid = self(),
    case get_url_contents(Url) of
        {ok, Data} ->
            Strings = string:tokens(Data, "\n"),
            Pids = [spawn(fun() ->
                          process_string(W, MyPid, Url, Str)
                          end)  || Str <- Strings],
            collect(length(Pids));
        _ -> ok
    end.
```

В этом коде наибольший интерес представляет функция `collect()`. Её назначение — ждать, пока все процессы обработки строк не завершатся:

```
collect(0) -> ok;
collect(N) ->
    %io:format("To collect: ~p~n", [N]),
    receive
        done -> collect(N - 1)
    end.
```

Реализация этой функции состоит из двух ветвей. При вызове функции Erlang попытается найти соответствие между формальными и фактическими параметрами. Если `collect()` вызвана с нулевым аргументом, она вернет атом `ok`, и на этом всё закончится. Если же она вызвана с положительным аргументом, она будет ждать в `receive` до тех пор, пока не получит сообщение `done`, после чего она вызовет себя же с уменьшенным аргументом и снова будет ждать сообщения. Это значит, что, будучи вызвана, функция `collect(N)` не завершится, пока не получит N сообщений`done`. Так как мы вызвали её с аргументом, равным количеству запущенных процессов-обработчиков строк, она будет ждать до тех пор, пока количество полученных сообщений `done` не сравняется с количеством строк в странице.

Получение содержимого URL адреса

Для чтения данных, расположенных по известному URL, мы будем пользоваться библиотечной функцией `http:request()`. В случае успеха чтения (коды возврата 200 и 201) мы вернем прочитанное содержимое. Если ошибка произошла по вине сервера (коды возврата 5XX), то мы подождем и вновь повторим попытку. В случае любой другой ошибки вернем атом `failed`, что будет означать, что чтение не удалось. Если чтение не удалось по вине библиотеки http (ветвь `{error, Why}`), мы тоже повторим попытку чтения после паузы.

```
get_url_contents(Url) -> get_url_contents(Url, 5).
get_url_contents(Url, 0) -> failed;
get_url_contents(Url, MaxFailures) ->
  case http:request(Url) of
    {ok, {{_, RetCode, _}, _, Result}} -> if
      RetCode == 200;RetCode == 201 ->
        {ok, Result};
      RetCode >= 500 ->
        % server error, retry
        %io:format("HTTP code ~p~n", [RetCode]),
        timer:sleep(1000),
        get_url_contents(Url, MaxFailures-1);
      true ->
        % all other errors
        failed
      end;
    {error, _Why} ->
      %io:format("failed request: ~s : ~w~n", [Url, Why]),
      timer:sleep(1000),
      get_url_contents(Url, MaxFailures-1)
  end.
```

Функция обработки одной строки

При обработке строки мы попытаемся найти в этой строке URL. Если он будет найден, мы его обработаем. При выходе из функции сообщим родительскому процессу, что функция обработки строки завершилась. Это приведет к уменьшению счетчика ожидания `collect()` на 1.

```
process_string(W, Parent, Dir, Str) ->
    case extract_link(Str) of
        {ok, Url} -> process_link(W, Dir, Url);
        failed    -> ok
    end,
    done(Parent).

done(Parent) ->
    Parent ! done.
```

Функция `done()` посылает родительскому процессу сообщение, состоящее из одного атома `done`.

Извлечение URL из строки

Извлечение URL из строки HTML произвольного вида — задача непростая. Мы будем решать её очень приблизительно, при помощи регулярного выражения. Ис-

пользуемый метод имеет множество ограничений: он не найдет URL, разбитый на несколько строк, или второй URL в строке. Но для наших учебных целей он вполне подойдет. В случае успеха мы вернём пару, состоящую из атома и найденной величины `{match, Value}`, а в случае неуспеха — только атом `failed`:

```
extract_link(S) ->
    case re:run(S, "href *= *([^>]*)>", [{capture, all_but_first, list}]) of
        {match, [Link]} -> {ok, string:strip(Link, both, $")};
        _               -> failed
    end.
```

Обработка URL

Для определения типа входного URL вновь воспользуемся регулярными выражениями. Типов URL, которые мы умеем распознавать, три:
image — ссылка на картинку,
page — ссылка на другую страницу и
strange — URL, не являющийся ни одним из двух предыдущих типов.
Найдя картинку, мы пошлем процессу `Writer` сообщение о том, что этот адрес нужно сохранить. Если адрес соответствует типу `page`, для него необходимо вызвать ту же функцию `process_link()`, с которой начались наши вычисления в функции `start()`. Для типа `strange` мы просто напечатаем диагностику.

```
process_link(W, Dir, Url) ->
    case get_link_type(Url) of
        image -> process_image(W, Dir ++ Url);
        page  -> process_page(W, Dir ++ Url);
        _     -> process_other(W, Dir ++ Url)
    end.


%% Site-specific heuristics.
get_link_type(Url) ->
    {ok, ReImg} = re:compile("\\.(gif|jpg|jpeg|png)",
                             [extended, caseless]),
    {ok, RePage} = re:compile("^[^/]+/$"),
    case re:run(Url, ReImg) of
        {match, _} -> image;
        _          -> case re:run(Url, RePage) of
                          {match, _} -> page;
                          _          -> strange
                      end
    end.

process_image(W, Url) ->
    write(W, Url).
```

```
process_other(_W, _Url) ->
    %io:format("Unexpected URL: ~p~n", [Url])
    ok.
```

Результат работы функции start()

Мы закончили часть кода, которая обходит сайт и сохраняет в файле адреса картинок для последующей обработки. Никакая дополнительная сборка программы не требуется. В моем случае уже через несколько секунд работы в файле pictures.txt оказалось записано более 55 тысяч адресов.

Главная функция сохранения

Отредактировав файл с адресами картинок pictures.txt, приступим с получению самих картинок из сети и сохранению их на диске.

```
save(Path) ->
    inets:start(),
    {ok, Data} = file:read_file(Path),
    L = string:tokens(binary_to_list(Data), "\n"),
    save_loop(0, L).
```

Вызов функции `download:save()` происходит независимо от вызова функции `download:start()`. Между двумя этими вызовами сессия Erlang не обязана сохраняться. После того, как `start()` отработает, мы вполне можем закрыть сессию, отредактировать файл со ссылками на картинки, а сохранять их через пару дней. Поэтому мы вновь проинициализируем модуль `inets`, читающий для нас данные из интернета. Сначала прочитаем файл с адресами картинок. При этом функция `file:read_file()` вернёт данные в бинарном формате. Превратим бинарные данные в список (а строки в Erlang — это списки) и разобьем одну длинную строку на части, считая перевод строки разделителем. В конце вызовем функцию, осуществляющую цикл сохранения.

Ограничение количества процессов

Одна из первых версий программы запускала независимый процесс сохранения для каждого адреса. К сожалению, это не заработало. Я не смог получить ни одной картинки. Я не знаю истинной причины. Возможно, модуль http не захотел работать со слишком большим количеством одновременных соединений. Может быть, сайт оказался не готов к такой нагрузке. Мы воспользуемся решением, которое успешно обходит все возможные трудности одновременно. Ограничим количество одновременно читающих/пишущих процессов, например, двумя сотнями. Число 200 выбрано экспериментальным путем. Разницы в скорости сохранения при 20 и 200 процессах я не

заметил, но совершенно точно, что 200 процессов работают быстрее, чем 2. С другой стороны, число процессов не должно быть очень большим, чтобы не натолкнуться на ограничение операционной системы на количество одновременно открытых файлов.

Логика ограничения количества процессов реализована в функции `save_loop()`. Она включает в себя четыре ветви.

```
save_loop(0, []) ->
    io:format("saving done~n", []);
save_loop(Running, []) ->
    receive
        done ->
            io:format("to save: ~p~n", [Running]),
            save_loop(Running - 1, [])
    end;
save_loop(Running, [U|Us]) when Running < 200 ->
    S = self(),
    spawn(fun() -> save_url(S, U) end),
    save_loop(Running + 1, Us);
save_loop(Running, Us) ->
    receive
        done ->
            io:format("to save: ~p~n", [Running + length(Us)]),
            save_loop(Running - 1, Us)
    end.
```

Первая ветвь выполнится в том случае, если запущенных процессов уже не осталось, s и список входных URL для сохранения пуст. Эта ветвь прервет цикл сохранения.

Вторая ветвь выполнится в том случае, если список входных URL пуст, а количество запущенных процессов — любое. Цикл сохранения будет ждать в `receive` до тех пор, пока какой-нибудь процесс не сообщит о своём завершении. Тогда цикл снова вызовет себя же, уменьшив счётчик исполняемых процессов на 1.

Третья ветвь вызовется при непустом входном списке и количестве исполняющихся процессов меньшем двухсот. Цикл запустит новый процесс сохранения и вновь вызовет себя же, увеличив счетчик исполняющихся процессов на 1.

Четвертая ветвь выполнится в том случае, если не найдено ни одного из предыдущих соответствий. Она отвечает случаю, когда количество запущенных процессов превышает максимально разрешённое. Эта ветвь будет ждать в `receive` до тех пор, пока один из процессов не закончится, после чего снова вызовет `save_loop()`, уменьшив счётчик процессов на 1.

Сохранение одного URL

Сохраняя содержимое URL, мы должны по URL сформировать путь, по которому мы будем сохранять картинку, создать при необходимости все промежуточные дирек-

тории, прочитать файл из интернета, сохранить его на диске и сообщить родительскому процессу о том, что функция сохранения файла завершилась.

```
save_url(Parent, Url) ->
    Path = url_to_path(Url),
    ensure_dir(".", filename:split(Path)),
    case get_url_contents(Url) of
        {ok, Data} ->
            %io:format("saving ~p~n", [Url]),
            file:write_file(Path, Data);
        _ -> ok
    end,
    done(Parent).
```

Функция `url_to_path()` жульнически проста. Она откусывает от абсолютного URL протокол и имя сервера:

```
url_to_path(Url) ->
    string:substr(Url, length(get_start_base())+1).
```

Функция `sure_path()` в качестве входных аргументов использует имя текущей директории и список, состоящий из всех промежуточных директорий и имени файла. Этот список получается в результате вызова `filename:split(Path)`:

```
ensure_dir(_Dir, [_FileName]) -> ok;
ensure_dir(Dir, [NextDir|RestDirs]) ->
    DirName = filename:join(Dir, NextDir),
    file:make_dir(DirName),
    ensure_dir(DirName, RestDirs).
```

Если входной список директорий состоит только из имени файла, функция ничего не делает, в противном случае она вычисляет имя очередной директории, создает её при помощи `file:make_dir(DirName)` и вновь вызывает себя с только что вычисленной текущей директорией и укороченным списком, лишенным головы.

Export

Наша программа закончена. Осталось только привести её в соответствие с правилами хорошего тона программирования на Erlang. Сейчас все функции, описанные в модуле download.erl, видны снаружи. Для запуска программы мы пользуемся только двумя из них — `start` и `save`. Сделаем видимыми только их. Для этого заменим:

```
-compile(export_all).
```

на:

```
-export([start/0, save/1]).
```

Величины /0 и /1 называются арностью и обозначают количество аргументов функции. Функции с одинаковыми именами, но разной арностью являются разными функциями.

5.3. Заключение

Как уже упоминалось, программа которая находит на сайте картинки, выкачивает их и сохраняет на диске, была сначала написана на Scheme, а потом на Erlang. При практически одинаковом размере кода и одинаковых временных затратах на разработку, программа на Erlang работает в сотни раз быстрее. Это обусловлено природой задачи, которая легко поддаётся распараллеливанию. Поддержка параллельности на Erlang не требует от программиста практически никаких дополнительных усилий. Трудности синхронизации процессов на Erlang не идут ни в какое сравнение с теми, с которыми приходится сталкиваться, программируя многопоточное приложение в C или C++. Модель процессов, принятая в Erlang, позволяет программисту самостоятельно определять удобные ему механизмы синхронизации.

Алгебраические типы данных и их использование в программировании

Роман Душкин
darkus@fprog.ru

Аннотация

Статья рассматривает важную идиому программирования — алгебраический тип данных (АТД). Приводится теоретическая база, которая лежит в основе практического применения АТД в различных языках программирования. Прикладные аспекты рассматриваются на языке функционального программирования Haskell, а также кратко на некоторых других языках программирования.

Algebraic Data Types (ADT) is an important programming idiom. A theoretical background is given that forms a foundation for practical application of ADT in various programming languages. Practical aspects of ADTs are discussed using Haskell and are also briefly outlined for certain other programming languages.

Обсуждение статьи ведётся по адресу
http://community.livejournal.com/fprog/2921.html.

Введение

В 1903 году английский математик Бертран Рассел предложил антиномию в рамках языка классической («наивной») теории множеств Георга Кантора, которая показала несовершенство введённого им определения множества: «*множество есть многое, мыслимое как единое*»[1] [7, 12]:

> *Пусть K — множество всех множеств, которые не содержат сами себя в качестве своего подмножества. Ответ на вопрос «содержит ли K само себя в качестве подмножества?» не может быть дан в принципе. Если ответом является «да», то, по определению, такое множество не должно быть элементом K. Если же «нет», то, опять же по определению, оно должно быть элементом самого себя.*

В общем, куда ни кинь — всюду клин. Ситуация парадоксальна.

Данная антиномия (более известная под названием «парадокс Рассела») поколебала основы математики и формальной логики, что вынудило ведущих математиков того времени начать поиск методов её разрешения. Было предложено несколько направлений, начиная от банального отказа от теоретико-множественного подхода в математике и ограничения в использовании кванторов (интуиционизм, основоположником которого был голландский математик Лёйтзен Брауэр), до попыток аксиоматической формализации теории множеств (аксиоматика Цермело — Френкеля, аксиоматика Неймана — Бернайса — Гёделя и некоторые другие). На сегодняшний день аксиоматические теории множеств, дополненные *аксиомой выбора* или другими аналогичными аксиомами, как раз и служат одним из возможных оснований современной математики.

Позже австрийский философ Курт Гёдель показал, что для достаточно сложных формальных систем всегда найдутся формулы, которые невозможно вывести (доказать) в рамках данной формальной системы — первая теорема Гёделя о неполноте [14]. Данная теорема позволила ограничить поиски формальных систем, дав математикам и философам понимание того, что в сложных системах всегда будут появляться антиномии, подобные той, что предложил Б. Рассел.

В конечном итоге парадокс Рассела и запущенные им направления исследований в рамках формальных систем привели к появлению теории типов, которая, наряду с упомянутыми аксиоматическими теориями множеств и интуиционизмом, является одним из способов разрешения противоречий наивной теории множеств. Сегодня

[1] Впрочем, Г. Кантор дал достаточно чёткое математическое определение множества [2]:

Unter einer «Menge» verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objekten m unserer Anschauung oder unseres Denkens (welche die «Elemente» von M genannt werden) zu einem Ganzen.

«Под «множеством» мы понимаем произвольную коллекцию M в целом, состоящую из отдельных объектов m (которые называются «элементами» M»), которые существуют в нашем представлении или мыслях». Данное определение показывает, что Г. Кантор заложил основы перехода математики от туманных размышлений к точным символическим формулировкам.

под теорией типов понимается некоторая формальная система, дополняющая наивную теорию множеств [13]. Теория типов описывает области определений и области значений функций — такие множества элементов, которые могут быть значениями входных параметров и возвращаемыми результатами функций. Общее понимание теории типов в рамках информатики заключается в том, что обрабатываемые данные имеют тот или иной тип, то есть принадлежат определённому множеству возможных значений.[2]

В частности, решение приведённой в начале статьи антиномии было предложено самим Б. Расселом как раз в рамках теории типов. Решение основано на том, что множество (класс) и его элементы относятся к различным логическим типам, тип множества выше типа его элементов. Однако многие математики того времени не приняли это решение, считая, что оно накладывает слишком жёсткие ограничения на математические утверждения.

В рамках общей теории типов разработано определённое количество теорий, абстракций и идиом, описывающих различные способы представления множеств значений и множеств определений функций. Одна из них — *алгебраический тип данных* (другими важнейшими теориями в рамках дискретной математики являются комбина́торная логика, λ-исчисление и теория рекурсивных функций) [10, 6, 11]. Именно эта идиома и является предметом рассмотрения настоящей статьи, поскольку она имеет серьёзное прикладное значение в информатике в целом и в функциональном программировании в частности. К примеру, в языке Haskell любой непримитивный *тип данных* является алгебраическим.

Вместе с тем надо отметить, что в «чистой» математике типы рассматриваются как некие объекты манипуляции. Например, в типизированном λ-исчислении, в котором явно вводится понятие типа, типы изучаются исключительно как *синтаксические сущности*. Типы в типизированном λ-исчислении — это не «множества значений», а просто «бессмысленные» наборы символов и правила манипуляции ими.

К примеру, если говорить о простом типизированном λ-исчислении, то в нём даже нет констант типов вроде **`Int`**, **`Bool`** и т. д. Типы λ-исчисления — это выражения специального вида, составленные из значков ($\to$), ($*$) и круглых скобок «(» и «)» по простым правилам:

1) $*$ — тип;

2) если α и β — типы, то $(\alpha \to \beta)$ — тип.

Другими словами, типы — это строки вида $* \to (* \to *) \to *$. Именно строки, а не множества значений. Иногда, конечно, вводят и константы типов, но принци-

[2] Вместе с тем, типы в информатике появились из-за необходимости сопоставлять идентификатору внутреннее представление идентифицируемого объекта. Типы в информатике и типы в математике — это немного разные понятия. Понятие типов в информатике основано на математическом понятии, но не совсем тождественно ему (имеются расширения, необходимые для практических применений). Робин Милнер, автор и главарь разработчиков функционального языка ML, был одним из первых, кто попытался применить математические типы для выбора внутреннего представления программных сущностей [4], что породило определённую путаницу.

пиальной разницы это не вносит.

В связи с этим в дальнейшем изложении под понятием «тип» даже в математическом смысле будет иметься в виду конкретная интерпретация для прикладного применения. Именно прикладная интерпретация математического понятия имеет значение при переходе к информатике и программированию.

6.1. Мотивация

Перед рассмотрением теоретических основ АТД имеет смысл сравнить реализацию этой идиомы на языке программирования, в котором в явном виде это понятие отсутствует (например, язык семейства C) с описанием того же АТД на языке, где это понятие является естественным (наиболее «продвинутым» в отношении АТД является язык Haskell).

Например, пусть имеется задача реализовать тип данных, представляющий двоичное дерево,[3] при этом такой тип должен предоставлять разработчику возможности построить определённые дополнительные механизмы контроля внутренней структуры данных и доступа ним, к примеру — осуществлять проверки корректности значений, присваиваемых отдельным внутренним полям. Далее эти дополнительные требования к типам будут рассмотрены во всех подробностях.

Пусть имеется обычное определение структуры, при помощи которой выражается двоичное дерево (в таком дереве данные, по сути, хранятся только в узловых вершинах; «листовой» считается вершина, у которой оба поддерева пусты):

```
struct Tree {
    int value;
    struct Tree *l;
    struct Tree *r;
};
```

Этот вариант определения не совсем подходит для описанных целей, потому что для доступа к элементам этой структуры, а также для разнообразных проверок целостности и непротиворечивости, потребуется писать дополнительные функции.[4] Как бы сделать так, чтобы транслятор автоматически делал за разработчика «чёрную работу» по созданию вспомогательных программных конструкций?

Структуру `Tree` можно переопределить примерно следующим образом (впрочем, это — не совсем корректное переопределение, поскольку в этом случае данные будут храниться только в листовых вершинах, а не в любых вершинах дерева; тем не менее, такое переопределение вполне достаточно для целей статьи):

[3] Здесь специально в познавательных целях опускается тот момент, что подобные типы данных давно уже реализованы в стандартных библиотеках большинства развитых языков программирования.

[4] Например, для данного конкретного определения необходимо написать служебную функцию проверки того, что указатели `l` и `r` ненулевые, а если и нулевые, то эта ситуация корректна (нулевые указатели на дочерние поддеревья могут быть только у листовых вершин).

```
struct Tree {
    union {
        int value;
        struct {
            struct Tree *left;
            struct Tree *right;
        } branches;
    } data;

    enum {
        LEAF,
        NODE
    } selector;
};
```

В этом определении имеются два взаимозависимых элемента: объединение **data** и перечисление selector. Первый элемент содержит данные об узле дерева, а второй идентифицирует тип первого элемента. Если в качестве значения в объединении **data** содержится поле value, то значением элемента selector должно быть LEAF. Соответственно, если в первом элементе находится структура branches, скрывающая в себе два указателя на такие же двоичные деревья (левое и правое поддерево), то во втором элементе должно быть значение NODE. Опять налицо необходимость иметь внешние по отношению к этому определению инструменты, которые следят за семантической непротиворечивостью значений типа.[5]

Разработчику придётся в явном виде писать примерно такие функции для доступа на запись к полям определённой выше структуры:

```
void setValue (Tree *t, int v) {
    t->data.value = v;
    t->selector   = LEAF;
}

void setBranches (Tree *t, Tree *l, Tree *r) {
    t->data.branches.left  = l;
    t->data.branches.right = r;
    t->selector            = NODE;
}
```

В функциях доступа к значениям структуры на чтение придётся вводить дополнительные проверки того, что тип получаемого значения соответствует запрошенному. В итоге необходимо будет явно проводить проверки как в функциях доступа, так и во всех функциях, которым этот доступ необходим. В представленном выше типе имеется только две альтернативы, а в некоторых случаях таких альтернатив может

[5]Конечно, эту задачу можно реализовать через класс, но ничего иного, как помещение структуры данных и функций для её обработки под одним именем программной сущности, это не даст — фактически всё будет то же самое.

быть и десять, и больше, так что объём функций доступа будет возрастать пропорционально количеству полей в структуре.

Итак видно, что определение типа `Tree` в языке типа C получилось достаточно громоздким, но при этом была решена важная задача — здесь явно определена возможность выбора между двумя альтернативами: (`value`, `LEAF`) и (`branches`, `NODE`). Такой тип позволяет, по сути, хранить совершенно разнородные данные в зависимости от своего назначения в каждом конкретном случае — для листовых элементов двоичного дерева хранятся числовые значения, для узловых — указатели на левое и правое поддеревья соответственно. Для понимания того, какой именно тип используется в каждом таком конкретном случае, имеются «метки» из перечисления `selector`. Но цена этого — дополнительные «метки» и множество явно описываемых проверок в функциях доступа.

А вот определение того же типа данных на языке программирования Haskell:

```
data Tree = Leaf Int
          | Node Tree Tree
```

Это определение описывает тип, который может быть представлен двумя видами значений (все такие виды разделены символом вертикальной черты (`|`)). Первый вид значений, помеченный «меткой» `Leaf`, представляет собой целое число, хранимое в листовой вершине двоичного дерева. Второй вид значений — узловая вершина дерева, хранящая ссылки на левое и правое поддеревья (соответственно, используется метка `Node`).

А вот тот же тип, но уже годный для хранения значений произвольного типа, а не только целых чисел:

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

Здесь переменная типов α является «заменителем» для любого другого типа (можно даже придумать ситуацию, когда в листовых вершинах двоичного дерева хранятся двоичные деревья и т. д. до бесконечности[6]). В языках типа C (C++, C#, Java и др.) для этих же целей можно воспользоваться шаблонами, и заинтересованный читатель может самостоятельно реализовать такой шаблон, а также функцию для контроля целостности значений к нему (после чего можно будет сравнить определения, хотя бы по количеству использованных скобок).

В языке Haskell и многих других функциональных языках программирования в идиому для представления алгебраических типов данных (а при помощи ключевого слова **`data`** определяются именно они) уже включаются механизмы контроля внутренней целостности и непротиворечивости. Кроме того, такое формальное описание типов позволяет автоматически генерировать шаблоны функций обработки значений. Для включения своей семантики в программу разработчику необходимо лишь заполнить их (для типовых задач вообще возможно генерировать такие же типовые функции полностью автоматически).

[6] В информатике такие деревья называются «деревьями высшего порядка» (англ. *high-order trees*).

Можно подвести итоги о преимуществам АТД, выделив явные положительные моменты в использовании этой идиомы функционального программирования:

1) АТД позволяют разработчику не тратить время на написание служебных функций и методов для проверки целостности и непротиворечивости типов данных, а зачастую и на написание методов доступа на запись и на чтение к полям таких типов.

2) АТД — это программная сущность для определения гарантированно безопасных размеченных объединений.

3) Наконец, АТД предоставляют возможность описания взаимно рекурсивных типов данных, то есть являются компактной статически безопасной реализацией связных структур.

Для понимания отличительных особенностей и преимуществ АТД в программировании далее будут представлены теоретические аспекты этого понятия (с применением нескольких математических формул уровня первого курса технического вуза), после чего в двух последних разделах будут приведены способы реализации теоретического понятия в прикладных языках программирования. Основное повествование ведётся на языке Haskell, даётся краткое описание той части синтаксиса языка, которая связана с АТД. Для других языков программирования, где явно реализованы АТД, просто приводятся примеры определений.

6.2. Теоретические основы

В теории есть два способа описания АТД. Первый использует теоретико-множественный подход и соответствующую нотацию. Ознакомление с этим аспектом позволит уяснить, как именно развивалось это понятие, и как оно попало в информатику. Второй использует специально разработанную нотацию для так называемого синтаксически-ориентированного конструирования типов и функций для их обработки (нотация Ч. Хоара). Данная нотация позволяет уже более или менее читабельно описывать на математическом языке типы данных в рамках теории типов. Интересно видеть, как данная нотация была преобразована при реализации языков программирования.

6.2.1. Определение АТД

Алгебраический тип данных неформально можно определить как множество значений, представляющих собой некоторые контейнеры, внутри которых могут находиться значения каких-либо иных типов (в том числе и значения того же самого типа — в этом случае имеет место рекурсивный АТД). Множество таких контейнеров и составляет сам тип данных, множество его значений.

Алгебраический тип данных — размеченное объединение декартовых произведений множеств или, другими словами, размеченная сумма прямых произведений множеств.

С теоретической точки зрения алгебраическим типом данных является размеченное объединение множеств (иначе называемое «дизъюнктным объединением»)[7], под которым понимается видоизменённая классическая операция объединения — такая операция приписывает каждому элементу нового множества метку (или индекс), по которой можно понять, из какого конкретно множества элемент попал в объединение. Соответственно, каждый из элементов размеченного объединения в свою очередь является декартовым произведением некоторых иных множеств.

Пусть есть набор множеств $A_i, i \in I$, из которых создаётся их размеченное объединение. В этом случае под размеченным объединением понимается объединение пар:

$$\coprod_{i \in I} A_i = \bigcup_{i \in I} \{(x, i) \mid x \in A_i\}.$$

Здесь (x, i) — упорядоченная пара, в которой элементу x приписан индекс множества, из которого элемент попал в размеченное объединение. В свою очередь каждое из множеств A_i *канонически* вложено в размеченное объединение, то есть пересечения канонических вложений A_i^* всегда пусты, даже в случаях, когда пересечения исходных множеств содержат какие-либо элементы. Другими словами, каноническое вложение имеет вид:

$$A_i^* = \{(x, i) \mid x \in A_i\},$$

а потому

$$\forall i, j \in I, i \neq j : A_i^* \cap A_j^* = \varnothing.$$

Итак, АТД — это размеченное объединение, то есть элементы такого типа с математической точки зрения представляют собой пары (x, i), где i — индекс множества (метка типа), откуда взят элемент x. Но чем являются сами элементы x? Теория говорит о том, что эти элементы являются декартовыми произведениями множеств, которые содержатся внутри контейнеров A_i. То есть, каждое множество A_i, из которых собирается размеченное объединение, является декартовым произведением некоторого (возможно, нулевого) числа множеств. Именно эти множества и считаются «вложенными» в контейнер АТД, вложение обеспечивает операция декартова произведения.

Другими словами, каждое множество A_i представляет собой декартово произведение:

$$A_i = A_{i1} \times A_{i2} \times \ldots \times A_{in},$$

[7] Для заинтересованных читателей имеет смысл дать англоязычное наименование термина — *tagged union* или *disjoint union*.

где множества $A_{ik}, k = \overline{1,n}$ являются произвольными (в том числе нет ограничений на рекурсивную вложенность). Поскольку элементами декартова произведения множеств являются кортежи вида

$$x = (x_1, x_2, \ldots x_n),$$

в целом АТД можно записать как

$$\coprod_{i \in I} A_i = \bigcup_{i \in I} \{((x_1, x_2, \ldots x_{n_i}), i)\}. \tag{6.1}$$

В общем случае декартово произведение вообще может быть представлено пустым кортежем. Тогда считается, что соответствующий контейнер A_i не содержит никаких значений внутри себя, а в размеченное объединение канонически вкладывается единственный элемент этого множества — $((), i)$. Данная ситуация возможна тогда, когда в АТД включается метка i ради самой себя, то есть в АТД содержится нуль-арное каноническое множество A_i^*, имеющее единственный элемент. Обычно это требуется для определения перечислений (эта ситуация и её реализация будут продемонстрированы далее в статье).

Для лучшего понимания того, что представляет собой АТД, можно представить общую формулу произвольного АТД в виде диаграммы. Сообразуясь с формулой 6.1, произвольный АТД можно изобразить так, как показано на рис. 6.1.

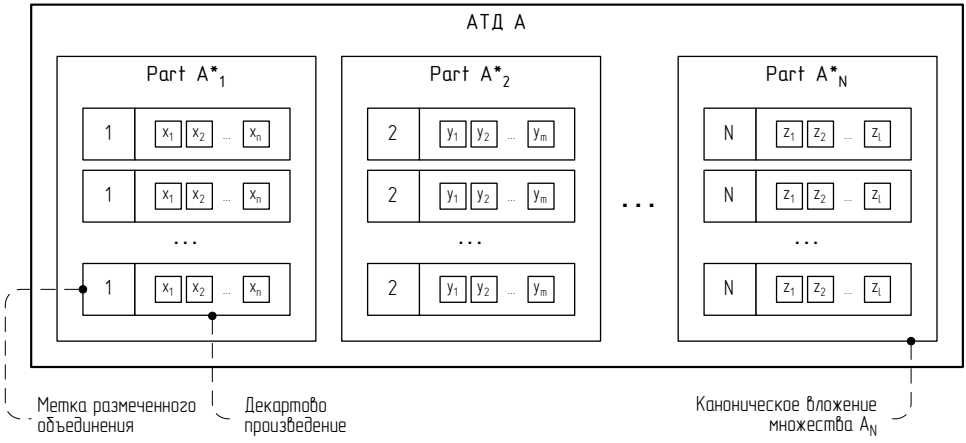


Рис. 6.1. Схема произвольного АТД

В качестве примера АТД в математической нотации можно рассмотреть тип `Tree`, введённый ранее:

```
data Tree α = Leaf α
```

```
  | Node (Tree α) (Tree α)
```

Данный тип есть размеченное объединение двух множеств `Leaf` и `Node`, так что $A_1 \equiv Leaf$, $A_2 \equiv Node$. Множество A_1 есть декартово произведение одного произвольного множества a. Значения этого множества «упаковываются» в контейнер A_1. Соответственно, множество A_2 есть декартово произведение двух одинаковых множеств $Tree(a)$, то есть налицо рекурсивное определение АТД.

6.2.2. Синтаксически-ориентированное конструирование

Как было показано выше, при создании АТД используются две операции: декартово произведение и размеченное объединение. Эти операции вместе с понятиями теории типов были взяты за основу так называемого синтаксически-ориентированного подхода к конструированию типов, предложенного Чарльзом Энтони Хоаром [1]. Дополнительно о типах в языках программирования можно прочесть в [5].

Данный подход предлагает более удобную нотацию для представления типов, нежели формальные математические записи в теоретико-множественной нотации. В данной нотации типы именуются словами английского языка, начинающимися с заглавной буквы, причём конкретные типы имеют вполне конкретные названия: $List$, $Tree$ и т. д., а переменные типов (то есть такие обозначения, вместо которых можно подставлять произвольный тип) — просто буквы с начала алфавита, возможно с индексами: A, B, C_1, C_n и т. п.

Также в качестве ключевых слов в нотации Ч. Хоара используются слова constructors, selectors, parts и predicates (а также эти же слова в форме единственного числа). Данные ключевые слова используются для ввода наименований отдельных «элементов» АТД. Под «элементами» АТД понимаются различные сущности в составе типа — отдельные размеченные множества и типы, упакованные в контейнеры.

- Конструкторы (constructors) — это наименования функций, создающих декартовы произведения из состава АТД.
- Селекторы (selectors) — это специальные утилитарные функции, обеспечивающие получение отдельных значений из декартовых произведений.
- Части (parts) — наименования отдельных канонических множеств размеченного объединения.
- Предикаты (predicates) — функции, позволяющие идентифицировать принадлежность заданного значения конкретному множеству из состава размеченного объединения.

Далее в настоящем разделе каждый из этих элементов описывается более подробно.

Наконец, два символа, (+) и (×), используются для записи определений типов. Знак (+) обозначает размеченное объединение, а знак (×) используется для обозначения декартова произведения.

В качестве примера определения АТД в данной нотации можно привести классическое определение АТД «список элементов типа A»:

$$List(A) = NIL + (A \times List(A))$$

$nil, prefix$ = constructors $List(A)$
$head, tail$ = selectors $List(A)$
$NIL, nonNIL$ = parts $List(A)$
$null, nonNull$ = predicates $List(A)$

Здесь A — произвольный тип данных, так называемая переменная типов, вместо которой в конкретных случаях можно подставлять любой необходимый тип. Например, если необходимо иметь список целых чисел, то достаточно подставить Int вместо всех вхождений символа A.

На представленном примере можно пояснить основные понятия нотации синтаксически-ориентированного конструирования. Первая строка определения описывает сам тип $List(A)$. Список — это размеченное объединение пустого списка NIL и декартова произведения элемента типа A со списком таких же элементов. Значением типа $List(A)$ может быть либо пустой список, либо непустой, который есть пара (декартово произведение двух множеств), первым элементом которой является значение обёртываемого типа, а вторым — список (в том числе и пустой). Это значит, что «в конце» каждого конечного списка должен находиться пустой список как базис рекурсии.

Следующие четыре строки определяют «элементы» АТД «список». Первая из них определяет два конструктора, каждый из которых соответствует одной из частей размеченного объединения. Конструктор nil создаёт пустой список NIL, конструктор $prefix$ создаёт непустой список соответственно. Этот конструктор принимает на вход значение заданного типа и список, а возвращает пару, первым элементом которой является указанное значение, вторым — список. Следовательно, типы конструкторов можно определить так[8]:

$$\#nil = List(A)$$
$$\#prefix = A \to (List(A) \to List(A))$$

Таким образом, тип конструктора типа A_i, являющегося декартовым произведением типов $A_{i1}, A_{i2}, \ldots \times A_{in}$, определяется формулой:

$$\#\,\text{constructor}\, A_i = A_{i1} \to (A_{i2} \to \ldots (A_{in} \to A) \ldots),$$

[8]Запись $\#x$ означает «тип значения x» (в теории типов функции также имеют типы, поэтому в качестве значения x может выступать и функция).

то есть конструктор одного декартова произведения принимает на вход значения «оборачиваемых» типов (тех, которые помещаются в контейнер), а возвращает значение целевого, своего АТД.[9] Все части АТД, каждая из которых есть декартово произведение, имеют по одному конструктору. Сам АТД является в таком случае размеченным объединением *частей*, а части обозначаются ключевым словом parts.

Для всех конструкторов, которые создают «контейнеры», имеются так называемые *селекторы*. Селектор — это функция, которая возвращает одно определённое значение из контейнера. В случае типа $List(A)$ селекторы есть только у части $nonNIL$ (непустой список). Первый селектор $head$ возвращает голову списка, то есть значение типа A, а второй $tail$ — хвост списка, то есть второй элемент пары в декартовом произведении. Типы селекторов можно понять по их назначению:

$$\#head = List(A) \to A$$
$$\#tail = List(A) \to List(A)$$

Другими словами, каждый селектор имеет тип вида $A \to A_{ik}$, и такой селектор принимает на вход значение типа АТД, а возвращает заданное значение из контейнера. Селекторы имеют место только для декартовых произведений, а для каждой части типа имеется столько селекторов, сколько типов упаковывается в соответствующий контейнер. Для каждой части типа верно следующее равенство, называемое «аксиомой тектоничности»[10]:

$$\forall x \in A_i : \text{constructor}\, A_i(s_{i1}x)(s_{i2}x)\ldots(s_{in_i}x) = x,$$

где $s_{i1}, s_{i2}, \ldots s_{in_i}$ — селекторы соответствующих компонентов декартова произведения.

Уже упомянутые части типа — это множества, включённые в АТД посредством размеченного объединения. Для АТД определяются предикаты, при помощи которых можно выявить, к какому конкретно множеству в рамках размеченного объединения относится значение. Наличие таких предикатов — одно из свойств размеченности объединения. Соответственно, сколько в АТД частей, столько и предикатов. Части задаются ключевым словом parts, предикаты — predicates. Для предикатов верна следующая аксиома:

$$(x \in A_i) \Rightarrow (P_i x = 1)\&(\forall j \neq i : P_j x = 0).$$

Наличие такой аксиомы необходимо для того, чтобы для произвольного значения АТД можно было выявить ту конкретную часть, к которой это значение принадлежит.

[9] Здесь необходимо отметить, что в теории и функциональном программировании имеется понятие «обобщённого алгебраического типа данных» (ОАТД), конструкторы которого могут в общем случае возвращать значения не своего типа.

[10] Под *тектоничностью* понимается внутренняя согласованность структуры. Наличие этой аксиомы гарантирует, что значение типа можно «пересобрать» из его отдельных компонентов, что, в частности, позволяет применять такие методики разработки программных средств, как интроспекция данных.

Далее можно будет применять селекторы конкретной части (применение селекторов к значению из другой части приведёт к ошибке согласования типов — необходимо помнить о типе селекторов). Соответственно, при реализации в языках программирования такие предикаты позволяют использовать механизм *сопоставления с образцом* (подробно рассказывается ниже в разделе про язык Haskell).

Нотация Ч. Э. Хоара также позволяет представлять АТД в виде диаграмм, на которых представлено древовидное описание структуры АТД. Такие деревья состоят из двух или трёх уровней. На первом уровне изображается вершина АТД с его наименованием. На втором уровне перечисляются части типа. Если часть представляет собой декартово произведение, то для данной части на третьем уровне перечисляются типы компонентов части. Рёбра дерева, ведущие с первого на второй уровень, помечаются наименованиями предикатов. Соответственно, рёбра, ведущие со второго на третий уровень, помечаются наименованиями селекторов.

Произвольный АТД выглядит так, как показано на рис. 6.2.

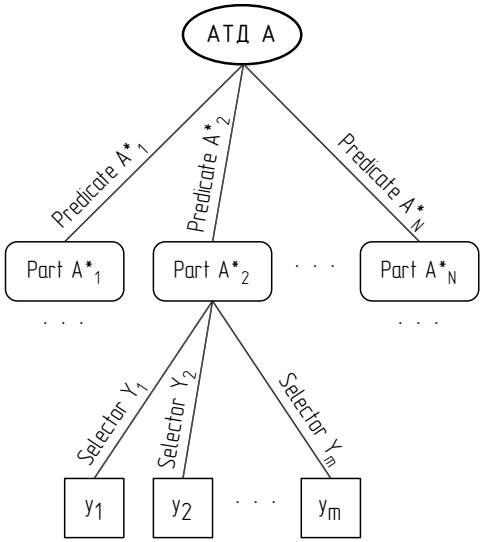


Рис. 6.2. Древовидное представление АТД

В качестве примера представления АТД в виде дерева можно привести диаграмму для типа $List(A)$, показанную на рис. 6.3. На приведённой диаграмме пунктирной линией показано рекурсивное включение типа $List(A)$ в качестве одного из своих компонентов.

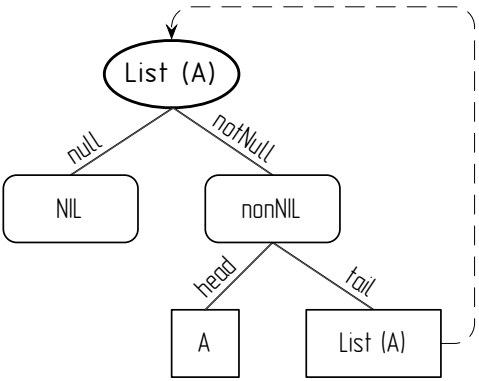


Рис. 6.3. Древовидное представление типа $List(A)$

6.2.3. Примеры описания АТД

В заключение теоретического введения можно привести ряд примеров определений различных АТД. Так, следующее определение используется для типа с поэтическим названием «розовый куст»[11]:

$Rose(A) = NIL + A \times List(Rose(A))$

$nil, rose$ = constructors $Rose(A)$
$node, branch$ = selectors $Rose(A)$
$NIL, Rose$ = parts $Rose(A)$
$null, isRose$ = predicates $Rose(A)$

Вот определение типа с названием «верёвка» (данный тип представляет собой список элементов с чередующимися типами A и B):

$Rope(A, B) = NIL + B \times (Rope(B, A))$

$nil, rope$ = constructors $Rope(A, B)$
$element, twisted$ = selectors $Rope(A, B)$
$NIL, Twist$ = parts $Rope(A, B)$
$null, isTwisted$ = predicates $Rope(A, B)$

А вот и определение двоичного дерева, в вершинах которого находятся элементы

[11]Необходимо отметить, что тип $Rose(A)$ взят из Standard Template Library. Этот тип являются достаточно широко используемым контейнерным типом. Впрочем, в STL имеются определения огромного количества контейнерных типов, заинтересованному читателю рекомендуется использовать STL для оттачивания своих навыков в определении АТД. Это поможет дополнительно осознать преимущества и выгоды АТД.

типа A[12]:

$$Tree(A) = Empty + A \times Tree(A) \times Tree(A)$$

$empty, node$ = constructors $Tree(A)$
$element, left, right$ = selectors $Tree(A)$
$Empty, Node$ = parts $Tree(A)$
$isEmpty, isNode$ = predicates $Tree(A)$

Ну а определение двоичного дерева, приведенного в разделе 6.1, выглядит так:

$$Tree(A) = A + Tree(A) \times Tree(A)$$

$leaf, node$ = constructors $Tree(A)$
$element, left, right$ = selectors $Tree(A)$
$Leaf, Node$ = parts $Tree(A)$
$isLeaf, isNode$ = predicates $Tree(A)$

Как видно из примеров, ничего особенно сложного в нотации синтаксически-ориентированного конструирования АТД нет. Стоит отметить, что абстрактная математическая нотация сегодня используется крайне редко, поскольку можно воспользоваться одним из языков программирования, в котором поддерживается понятие АТД.

6.3. АТД в языке программирования Haskell

Одним из языков программирования, где наиболее полно и близко к теории синтаксически-ориентированного конструирования представлены алгебраические типы данных, является функциональный язык Haskell. В связи с этим для изучения применения АТД в прикладном программировании имеет смысл обратить пристальное внимание на синтаксис этого языка. Далее в этом разделе будет кратко рассмотрен общий вид определения АТД с некоторыми примерами, пояснён механизм *сопоставления с образцами*, а также приведена классификация АТД.

6.3.1. Общий вид определения АТД в языке Haskell

Как уже упоминалось, в языке Haskell любой непримитивный *тип данных* является алгебраическим.[13] АТД вводятся в программу при помощи ключевого слова **`data`**,

[12]Это определение отличается от того определения двоичного дерева, которое рассмотрено в начале статьи — здесь в каждом узле дерева находится определённое значение, а листовые вершины отличаются от узловых тем, что оба поддерева пусты.

[13]Впрочем, с одной стороны, примитивные типы также могут быть представлены в виде конечных или бесконечных перечислений, что делает возможным их определение посредством АТД. С другой стороны, в языке Haskell имеется понятие «*функциональный тип*», то есть тип функции как программной сущно-

использование которого определяется следующим образом [3]:

data [*context* =>] *simpletype* = *constrs* [*deriving*],

где:

- *context* — контекст применения переменных типов (необязательная часть определения);
- *simpletype* — наименование типа с перечислением всех переменных типов, использующихся во всех конструкторах АТД (если переменные типов не используются, то ничего не указывается);
- *constrs* — перечень конструкторов АТД, разделённых символом (|) (данный символ соответствует операции размеченного объединения);
- *deriving* — перечень классов, для которых необходимо автоматически построить экземпляры определяемого типа (необязательная часть определения).

Контекст и экземпляры классов являются понятиями из системы типизации с параметрическим полиморфизмом и ограниченной квантификацией типов, которая используется в языке Haskell. Эти понятия выходят за рамки статьи, поэтому в дальнейших примерах эти части, и без того являющиеся необязательными, будут пропущены. Тем не менее, в дальнейшем (в последующих статьях на темы теории типов и систем типизации функциональных языков программирования) этот вопрос будет подробно рассмотрен, поэтому заинтересованному читателю рекомендуется уже сейчас «держать в уме» все эти нюансы. Кроме того, в компоненте *constrs* могут быть специальные отметки строгости конструкторов, сами конструкторы могут быть инфиксными (об этом чуть позже), а также содержать именованные поля (об этом тоже написано ниже).

Наименование АТД и все его конструкторы по обязательным соглашениям о наименовании, принятым в языке Haskell, должны начинаться с заглавной буквы. Наименование типа и наименования его конструкторов находятся в разных пространствах имён, поэтому в качестве наименования одного из конструкторов можно вполне использовать слово, являющееся наименованием всего типа (эта ситуация часто используется в случаях, когда у определяемого АТД один конструктор). После наименования типа, как уже было сказано, должны быть перечислены все переменные типов, использующиеся в конструкторах, причём переменные типов опять же по соглашениям должны начинаться со строчной буквы (обычно в практике используются начальные буквы латинского алфавита — `a`, `b` и т. д., в некоторых специальных нотациях языка используются строчные греческие буквы α, β и т. д.).

В некоторых случаях конструктор АТД может иметь специальное наименование, составленное из небуквенных символов. Для определения такого конструктора его

сти (не тип значения, возвращаемого функцией, а именно тип функции). Примеры функциональных типов приведены в теоретическом разделе (см., например, стр. 106); в языке программирования Haskell используются именно такие типы функций.

наименование должно начинаться с символа двоеточия (**:**). Способ использования подобных конструкторов ограничен — они должны быть бинарными и использоваться в инфиксной форме (то есть располагаться между своими аргументами). Впрочем, любой бинарный конструктор, как и произвольная бинарная функция, может быть записан в инфиксной форме посредством заключения его наименования в обратные апострофы (`). Возвращаясь к конструкторам с небуквенными наименованиями, можно отметить, что сам символ (**:**) является одним из конструкторов типа «список». Его формальное определение таково:

```
data [α] = []
         | α:[α]
```

Это определение не является правильным с точки зрения синтаксиса языка Haskell, но оно вшито в таком виде в ядро языка, чтобы вид списков в языке был более или менее удобным для восприятия (таким образом, переопределить конструктор (**:**) нельзя). Если определять список самостоятельно, то определение этого типа будет выглядеть примерно так:

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

Как можно заметить, каждый конструктор типа начинается с заглавной буквы (`Nil` и **`List`**), причём второй конструктор совпадает по наименованию с наименованием всего типа (как уже сказано, наименования типов и их конструкторов лежат в разных пространствах имён и употребляются в различных контекстах, поэтому неоднозначностей в интерпретации наименований не возникает, чем удобно воспользоваться). Первый конструктор пустой, зато второй определяет декартово произведение двух типов: некоторого типа α (здесь символ α — переменная типов) и типа **`List`** α, причём в данном случае в качестве аргумента конструктора используется наименование АТД. Это важно чётко понимать для уяснения сути определения. Определение могло бы быть переписано таким образом (второй конструктор называется в традиции языка Lisp):

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

А вот как, к примеру, определяется специальный тип для представления обычных дробей (данный тип определён в стандартном модуле **`Prelude`**)[14]:

```
data Ratio α = α :% α
```

Здесь, как видно, применён бинарный инфиксный конструктор. Этот конструктор принимает на вход два значения, а возвращает их связанную упорядоченную пару. Данная пара является представлением дроби. Соответственно, для значений этого типа определены такие селекторы (наименования приведенных функций переводятся с английского языка как «числитель» и «знаменатель» соответственно):

[14]Определение, конечно, несколько иное, но, как было заявлено ранее, особенности системы типизации языка Haskell в настоящей статье рассматриваться не будут.

```
numerator :: Ratio α → α
numerator (x :% y) = x

denominator :: Ratio α → α
denominator (x :% y) = y
```

Из этих определений видно, что функции-селекторы как бы раскладывают АТД на элементы, возвращая тот из них, который требуется по сути функции. Другими словами, селекторы в языке Haskell аналогичны функциям-аксессорам или операторам доступа к полям данных в других языках программирования. Например, запись «**denominator** n» в языке Haskell аналогична записи «`n.denominator()`» в языке Java.

6.3.2. Сопоставление с образцом

Представленные выше функции **numerator** и **denominator** показывают один важнейший механизм, реализованный в языке Haskell — сопоставление с образцами. Этот механизм используется в языке в нескольких аспектах, одним из главных является сопоставление с образцами при определении функций. Этот аспект необходимо рассмотреть более внимательно.

«Образцом» называется выражение, в котором некоторые или все поля декартова произведения (если декартово произведение не пусто, то есть конструктор не представляет собой простую метку размеченного объединения) заменены *свободными переменными*[15] для подстановки конкретных значений таким образом, что при сопоставлении происходит *означивание параметров образца* — однозначное приписывание переменным образца конкретных значений. Для АТД образцом является указание одного из конструкторов с перечислением переменных или конкретных значений в качестве его полей.

Сопоставление с образцом проходит следующим образом. Из всех клозов функции[16] выбирается первый по порядку, сопоставление со всеми образцами которого произошло успешно. Успешность заключается в корректном сопоставлении конкретных входных аргументов функции с соответствующими образцами. Сопоставление, как уже сказано, должно происходить однозначно и непротиворечиво. Константа сопоставляется только с такой же константой, свободной переменной в образце присваивается конкретное значение.

Этот процесс можно пояснить на примере. Пусть есть определение функции:

```
head :: [α] → α
head []     = error "Empty list has no first element."
head (x:xs) = x
```

[15]Свободной называется такая переменная, которая встречается в теле функции, но при этом не является параметром этой функции.

[16]Клозом (от англ. *clause*) называется одна запись в определении, определяющая значение функции для конкретного набора входных параметров.

Здесь первая строка является сигнатурой, описывающей тип функции, вторая и третья — два клоза функции **head** соответственно. Сигнатура функции может не указываться в исходных кодах, так как строго типизированный язык Haskell имеет механизмы для однозначного вычисления типа любого объекта в наиболее общей форме (соответственно, наличие или отсутствие сигнатур функций не влияет на их работоспособность). Впрочем, многие разработчики говорят о том, что наличие сигнатур функций рядом с определениями позволяет более чётко понимать смысл и суть функции.

Первый клоз определяет значение функции в случае, если на вход функции подан пустой список. Как видно, функция предназначена для возвращения первого элемента («головы») списка, а потому для пустого списка её применение ошибочно. Используется системная функция **error**. Второй клоз применим для случаев, когда список не является пустым. Непустой список, как уже говорилось в теоретической части — это пара, первым элементом которой является некоторое значение, а вторым — список оставшихся элементов, «хвост» списка. В языке Haskell эти элементы декартова произведения соединяются при помощи конструктора (`:`), что и представлено в образце. Две свободные переменные — `x` и `xs` — это параметры образца, которые получают конкретные значения в случае корректного применения функции. Например:

```
> head [1, 2, 3]
```

сопоставит с переменной `x` конкретное значение `1`, а с переменной `xs` — значение `[2, 3]`. Соответственно, результатом выполнения функции будет значение переменной `x`, то есть `1`.

Приведённый пример можно понимать и по-другому. Раз список есть пара, созданная при помощи конструктора (`:`), то список `[1, 2, 3]` может быть представлен как `(1:[2, 3])`, а ещё вернее как `(1:(2:(3:[])))`. В данном случае вызов **head** `[1, 2, 3]` приведёт к следующей последовательности вычислений:

```
> let (x:xs) = (1:[2, 3])
  in  x
```

или, что то же самое:

```
> let x  = 1
      xs = [2, 3]
  in  x
```

В конечном итоге, поскольку свободная переменная `xs` не участвует в вычислении результата, оптимизирующий транслятор языка, основанного на ленивых вычислениях, проведёт такое преобразование[17]:

```
> let x = 1
  in  x
```

[17] О ленивой стратегии вычислений можно прочесть в статье [9] и дополнительных источниках, в том числе перечисленных в указанной статье.

Итогом вычислений будет значение переменной `x`, то есть `1`.

Теперь можно рассмотреть более сложный пример. Пусть определён АТД для представления бинарного дерева (такой же, как в теоретической части):

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

Вот функция, которая вычисляет максимальную глубину заданного дерева. Уже по виду определения АТД можно сказать, что у неё должно быть не менее двух клозов, по одному на каждый конструктор АТД:

```
depth :: Tree α → Int
depth Empty        = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)
```

Первый клоз функции определяет её значение для первого конструктора АТД `Tree`, то есть для пустого дерева. Второй клоз определяет значение функции уже для непустого дерева. У непустого дерева есть хранимый в узле элемент и два поддерева — левое и правое. Функции `depth` хранимый в узле элемент неинтересен, а потому в образце применена «маска подстановки» для неиспользуемых элементов АТД — (_). Этот символ при сопоставлении с образцами означает, что на его место может быть подставлено всё, что угодно. Кроме того, это единственный символ, который можно использовать несколько раз в одном образце. Два других компонента конструктора `Node`, `l` и `r`, сопоставляются с левым и правым поддеревьями соответственно.

Процесс сопоставления с образцом устроен таким образом, что не требует от значений АТД быть быть величинами, над которыми определена операция сравнения[18] — возможность выбора конкретной части АТД обеспечивается наличием предикатов (необходимо вспомнить аксиому для предикатов АТД). Вместе с тем это накладывает дополнительные ограничения — нельзя написать определение функции, подобное следующему:

```
isElement x []     = False
isElement x (x:xs) = True
isElement x (y:xs) = isElement x xs
```

Здесь во втором клозе в образцах переменная `x` используется два раза, что недопустимо в образцах.

Таким образом, при сопоставлении с образцом происходит сравнение конструктора поданного на вход функции значения с конструктором в образце. Это значит, что технология сопоставления с образцом является очень мощной и гибкой при определении функций — она не требует явного определения функций сравнения величин.

6.3.3. Классификация АТД

Остаётся кратко упомянуть о дополнительной классификации АТД в языке Haskell. При помощи АТД определяются любые типы данных, включая те, для которых

[18] В терминах Haskell — типы не обязаны быть экземплярами класса **`Ord`**.

в других языках программирования имеются отдельные ключевые слова. Например, простое перечисление на языке Haskell выражается как АТД, все конструкторы которого пусты:

```
data Weekday = Monday
             | Tuesday
             | Wednesday
             | Thursday
             | Friday
             | Saturday
             | Sunday
```

Декартовым типом называется такой АТД, который имеет только один конструктор декартова произведения (иногда декартов тип называют также *типом-произведением*). Приведённый ранее пример типа **`Ratio`** представляет собой декартов тип. Обычно декартовы типы используются для определения записей с именованными полями. Для этих целей в языке Haskell имеется специальная синтаксическая конструкция:

```
data Timestamp = Timestamp
               {
                 year    :: Int,
                 month   :: Month,
                 day     :: Int,
                 hour    :: Int,
                 minute  :: Int,
                 second  :: Int,
                 weekday :: Weekday
               }
```

В приведённом случае транслятором языка будут автоматически сгенерированы функции доступа к перечисленным полям декартова типа, имеющие такие же наименования, как и поля. Так, если есть переменная `ts` типа `Timestamp`, то вызов выражения `year ts` позволит получить из этой переменной значение первого поля в декартовом произведении. Использование выражения `ts{year = 1984}` позволяет установить значение первого компонента декартова произведения. Символ равенства (`=`) здесь означает не присваивание, а копирование объекта с установкой в определённых полях новых значений (впрочем, оптимизирующие трансляторы могут действительно делать замену в соответствующих ячейках памяти в случаях, если известно, что старый объект больше не будет использован).

Ещё одним специфическим АТД является *тип-сумма*. Такой тип состоит из набора конструкторов, каждый из которых «обёртывает» только одно значение. Ближайшим аналогом такого типа в языках типа C является объединение (ключевое слово **`union`**). Например:

```
data SourceCode = ISBN String -- Код книги.
                | ISSN String -- Код периодического издания.
```

Наконец, осталось упомянуть, что приведённое деление АТД на типы в языке Haskell достаточно условно. Никто не запрещает сделать АТД, в котором тринадцать конструкторов будут пустыми, а четырнадцатый представлять собой структуру с именованными полями. Таким образом видно, что сама по себе концепция АТД позволяет достаточно гибко представлять типы данных в языках программирования.

Более подробно с АТД и методиками программирования на языке Haskell с их применением можно ознакомиться в книге [8].

6.4. АТД в других языках программирования

Помимо рассмотренного в предыдущем разделе языка Haskell концепция АТД явно реализована в следующих языках программирования (перечень дан по алфавиту)[19]:

- F#;
- Hope;
- Nemerle;
- OCaml и большинство языков семейства ML;
- Scala;
- Visual Prolog.

В этом разделе кратко рассмотрены особенности использования АТД в перечисленных языках программирования.

В языке программирования F# АТД реализованы ограниченно исключительно в виде безопасных с точки зрения типизации размеченных объединений (**`union`**), то есть все АТД в языке F# являются типами-суммами. Например:

```
type SomeType =
              | Constructor1 of int
              | Constructor2 of string
```

Язык Hope стал первым языком программирования, в котором концепция АТД и механизм сопоставления с образцами были реализованы в полной мере. Этот язык вдохновлял разработчиков последующих языков программирования — Miranda и Haskell. Синтаксис языка Hope достаточно необычен, но само понятие АТД отражено в нём в полном объёме. Для размеченного объединения используется символ (`++`), для декартова произведения — символ (`#`). Типы в декартовых произведениях заключаются в скобки. Например, для бинарного дерева АТД определяется так:

[19] В список не включён язык Miranda как прародитель языка Haskell. В этих языках синтаксис для определения АТД практически совпадает.

```
data tree == empty + node (num # tree # tree);
```

Язык Nemerle является C-подобным языком программирования для платформы .NET, основное достоинство которого заключается в поддержке как объектно-ориентированной, так и функциональной парадигм программирования (впрочем, язык Haskell также позволяет это делать, особенно его объектно-ориентированные потомки Mondrian, O'Haskell и Haskell++). АТД в языке Nemerle называются вариантами и полностью соответствуют теории синтаксически-ориентированного конструирования Ч. Э. Хоара. Синтаксис же несколько необычен для C-подобного языка:

```
variant Colour
{
  | Red
  | Orange
  | Yellow
  | Green
  | Cyan
  | Blue
  | Violet
  | RGB {r : int; g : int; b : int;}
}
```

Как видно, размеченному объединению соответствует символ (|), а декартову произведению — символ (**;**). Наименования полей в декартовых произведениях обязательны.

Язык OCaml является одним из серии языков ML, который использует функциональную, объектно-ориентированную и процедурную парадигмы программирования. Само семейство языков ML имеет достаточно серьёзный вес в мире функционального программирования, а потому без реализации АТД в этих языках не обошлось.[20] В этом языке, как и в языке Haskell, применяется параметрический полиморфизм (использование переменных типов).

Вот как, к примеру, определяется АТД для представления колоды карт:

```
type suit = Spades | Diamonds | Clubs | Hearts;;

type card =
    Joker
  | Ace   of suit
  | King  of suit
  | Queen of suit
  | Jack  of suit
  | Number of suit * int
;;
```

[20]Собственно, некоторые концепции уже упомянутого языка F# также были основаны на языке OCaml, что видно из синтаксиса

Теоретическая концепция АТД реализована в OCaml в полном объёме. Размеченное объединение как обычно представляется символом (|), а декартово произведение — символом (*). В АТД могут быть как пустые декартовы произведения, так и полноценные, а также их произвольная комбинация.

Язык Scala является Java-подобным мультипарадигменным языком программирования (как обычно заявляются объектно-ориентированная, функциональная, процедурная парадигмы). АТД в этом языке реализованы достаточно своеобразно при помощи концепции класса. Тем не менее эта реализация полностью соответствует теории. Для представления АТД и использования технологии сопоставления с образцами используется специальный вид классов:

```
abstract class Expression
case class Sum (l: Tree, r: Tree) extends Expression
case class Var (n: String) extends Expression
case class Const (v: Int) extends Expression
```

Декларации за ключевыми словами **`case class`** являются таким специальным видом классов, каждый из которых представляет конструктор декартова произведения того АТД, также представимого в виде класса, который он расширяет. Синтаксис достаточно необычен, но о позволяет использовать всю силу концепции АТД.

Если **`case class`** Expression объявить как `sealed`, то компилятор сможет проверять полноту разбора случаев при сопоставлении с образцом типа `Expression`, зато в противном случае разработчик сможет расширять множество выражений, добавив, к примеру, тип выражений `Product`. Таким образом, Scala поддерживает модульные декларации **`case class`**, но может и давать определенные гарантии корректности.

Наконец, язык Visual Prolog является наследником логического языка Prolog, в котором реализована объектно-ориентированная парадигма программирования, а также некоторые особенности функциональной парадигмы. Данный язык позволяет реализовывать графические приложения при помощи описания логики их работы в виде предикатов.

В этом языке для определения типов используется понятие «домен», то есть область определения предиката. Можно определять сложные домены так, чтобы предикаты могли принимать значения любого типа, а не не только `true` и `false` («истина» и «ложь»). При определении домена в виде АТД символ (`;`) используется для размеченного объединения, а (`,`) —для декартова произведения, причём элементы последнего должны быть заключены в круглые скобки после наименования конструктора. Вот пара примеров:

```
domains
  category = art; nom; cop; rel.
  tree = case(category, string); world(tree, tree); silence.
```

В данном примере домен `category` является перечислением, составленным из четырёх констант. Домен `tree` представляет собой обычный АТД, состоящий из трёх конструкторов, первые два из которых представляют собой декартовы произведения двух соответствующих компонентов.

Таков краткий обзор реализаций концепции АТД в различных языках программирования. Он демонстрирует, что АТД могут быть успешно использованы в различных подходах к построению программных средств.

Заключение

Алгебраические типы данных являются достаточно мощным и универсальным средством для описания структур данных при программировании. Понимание теоретических основ этой концепции позволит прикладным программистам глубже осознавать процессы, происходящие в разрабатываемых ими программных средствах. Более того, на практике, концепция АТД во многих случаях даёт разработчикам программных средств возможность избавиться от синтаксического мусора в определениях типов и увидеть суть структур данных на ранних этапах разработки программ.

Литература

[1] *Dahl O.-J., Dijkstra E. W., Hoare C. A. R.* Structured Programming. — Academic Press, 1972.

[2] *G. C.* Beitrage zur Begrundung der transfiniten Mengenlehre. // *Math. Ann.* — 1895. — Vol. 46.

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

[4] *Milner R.* A calculus of communicating systems. — Springer (LNCS 92), 1980.

[5] *Pierce B. C.* Types and Programming Languages. — MIT Press, 2002. — (имеется перевод книги на русский язык URL: http://newstar.rinet.ru/~goga/tapl/ (дата обращения: 28 сентября 2009 г.)). http://www.cis.upenn.edu/~bcpierce/tapl.

[6] *Вольфенгаген В. Э.* Методы и средства вычислений с объектами. Аппликативные вычислительные системы. — М.: АО «Центр ЮрИнфоР», 2004. — 789 с.

[7] *Гарднер М.* А ну-ка, догадайся! — М.: Мир, 1984. — 212 с.

[8] *Душкин Р. В.* Справочник по языку Haskell. — М.: ДМК Пресс, 2008. — 544 с.

[9] *Зефиров С. А.* Лень бояться. // *Журнал «Практика функционального программирования»*. — 2009. — № 1. http://fprog.ru/2009/issue01/.

[10] *Клини С. К.* Введение в метаматематику. — М.: ИЛ, 1957.

[11] *Петер Р.* Рекурсивные функции. — М.: ИЛ, 1954. — 264 с.

[12] *Розанова М.С.* Современная философия и литература. Творчество Бертрана Рассела / Под ред. Б. Г. Соколова. — СПб.: Издательский дом Санкт-Петербургского государственного университета, 2004.

[13] *Уайтхед А. Н.* Основания математики. / Под ред. Г. П. Ярового, Ю. Н. Радаева. — Самара: Изд-во «Самарский университет», 1954.

[14] *Успенский В. А.* Теорема Гёделя о неполноте. — М.: Наука, 1982. — 110 с.