Санкт-Петербургский государственный университет
Факультет прикладной математики — процессов управления

Котельников Евгений Борисович

Магистерская диссертация

Синтаксические расширения Scala для поддержки вычислений с эффектами

Направление 010300

Фундаментальные информатика и информационные технологии

Магистерская программа «Технологии баз данных»

Руководитель магистерской программы,
кандидат физ.-мат. наук,
доцент — Сергеев С.Л.

Научный руководитель,
ст. преподаватель — Севрюков С.Ю.

Рецензент,
аспирант, LAMP,
École Polytechnique Fédérale de Lausanne — Бурмако Е.Н.

Санкт-Петербург
2013

Оглавление

Аннотация

В работе рассматриваются способы формализации вычислений с эффектами некоторыми классами алгебраических структур и ряд синтаксических расширений функциональных языков программирования для работы с ними. В качестве теоретической базы для вывода формальных свойств систем эффектов используется теория категорий. В работе показано, как ограничения вычислительного контекста влияют на выразительность синтаксиса. Результатом является разработанное расширение языка Scala, предоставляющее более модульный синтаксис вычислений с эффектами, чем было представлено ранее.

Введение

Движущей идеей функционального программирования является приближение программирования к математике [58]. Многие идеи, возникающие в контексте функциональных языков (и часто произрастающие из попыток формализовать различные их аспекты), впоследствии оказываются заимствованы в промышленных языках и занимают место в числе лучших практик.

Так, отказ от использования изменяемого состояния, значительно упрощающий денотационную семантику языка, общепризнанно стал самым действенным способом борьбы со сложностью параллельных приложений.

Исследования в области теории типов привели к появлению языков, позволяющих закодировать множество инвариантов на уровне типов, что позволяет превратить в ошибки времени компиляции многие классы ошибок времени выполнения (вплоть до полного устранения последних в тотальных языках вроде Agda [41]).

Следующим заимствованием может стать отделение побочных эффектов (а в широком смысле, любых вычислительных эффектов) от чистых вычислений. Возникшая изначально в качестве инструмента для поддержки *функциональной чистоты* [52] языков, эта идея оказалась полезным принципом при проектирование программ.

Суть разделения сводится к введению специальных типов данных, инкапсулирующих значение в некотором вычислительном контексте. В отличии от чистого значения, эффективное значение допускает конечный набор операций над собой, зависящий от использованной формализации, такой как аппликативный функтор или монада.

Необходимость организовывать работу с эффективными значениями с помощью специальных операторов привела к появлению ряда расширений функциональных языков программирования, упрощающих синтаксис вычислений с эффектами. Все они основаны на введении специальных форм, раскрываемых в вызовы методы некоторого фиксированного интерфейса, задающего эффект.

Данное исследование представляет собой попытку создания модульного расширения функционального языка программирования, приближающего синтаксис вычислений с эффектами к синтаксису чистых вычислений.

Постановка задачи

Для формализации эффектов в работе используются понятия функтора, аппликативного функтора и монады. Известно, что они образуют иерархию (т.е. каждый аппликативный функтор является и просто функтором, а каждая монада является аппликативным функтором). Эта иерархия затрагивает и выразительную силу вычисления, которое можно описать в виде операций над эффективными значениями. При этом ни в одном языке не реализовано синтаксического расширения, поддерживающего всю иерархию целиком.

Также ни в одном языке не была сделана попытка приблизить синтаксис вычислений с эффектами к синтаксису чистых вычислений. Расширение обычно представляет собой введение специальных синтаксических форм, в общем случае плохо компонуемых друг с другом и с чистыми вычислениями.

Целью исследования является разработка синтаксического расширения функционального языка программирования, устраняющего указанные недостатки. Реализация расширения должна:

1. Упрощать описание вычислений с эффектами, формализованными в виде функторов, аппликативных функторов и монад.

2. Предоставлять синтаксис, максимально приближенный к синтаксису чистых вычислений.

3. Единообразно поддерживать вычисления со всей иерархией описанных выше структур.

4. В зависимости от вычислительной силы использованной формализации эффекта предоставлять адекватные синтаксические средства для описания допустимых в ней вычислений.

Для реализации был выбран язык программирования Scala и доступные в нём средства метапрограммирования.

Глава 1. Вычислительные эффекты

1.1. Понятие вычислительного эффекта

Построение языка программирования связано с описанием способа интерпретации допустимых в нём выражений. Одним из таких способов интерпретации является заданная *денотационная семантика* (известная также как семантика Скотта-Стрейчи [55]). Её суть заключается в конструировании математических объектов (*денотаций*, англ. "denotation"), соответствующих сущностям языка. К примеру, в языке программирования с целыми числами, заданных типом $Integer$, объекты этого типа соответствуют элементам множества $\mathbb{Z}$, а выражение вида $2 + 3$ соответствует применению математической операции сложения целых чисел.

Одной из ключевых особенностей *чисто функциональных* языков программирования (таких как Haskell [22]) является то, что интерпретация выражений математическими объектами распространяется и на функции. Так, функция с типом $Integer \to Integer$ является математической функцией над целыми числами.

Легко видеть, какие при такой интерпретации могут возникнуть трудности, к примеру, при попытке описать семантику императивных языков программирования. Вызов функции в общем случае изменяет состояние конечного автомата и притом может вообще не произвести результата. Ясно, что для такой функции тяжело найти аналог в математике[1].

Более того, на практике промышленного программирования встречается масса примеров вычислений, не формализуемых в виде обыкновенных функций. Рассмотрим некоторые из них.

1. **Побочные эффекты.** Иными словами, действия, производимые функцией в дополнении к возврату значения. Сюда входит изменение состояния, наблюдаемое взаимодействие с окружающим миром (такое как ввод-вывод, взаимодействие по сети или системные вызовы), обработка исключений и т.п. Функции, производящие побочный эффект не обладают свойством *ссылочной прозрачности* [53] (англ. "referential transparency"), а потому трудны для изучения.

[1] Тем не менее, это возможно, хотя, учитывая операционную природу императивных вычислений, для их формализации чаще прибегают к *операционной семантике*.

2. **Незавершающиеся вычисления**, реализующиеся бесконечными циклами либо неограниченной рекурсией.

3. **Вычисления с отсутствующим результатом.** Сюда относятся и функции, производящие побочный эффект, и функции, заданные не на всей области определения (т.е. производящие ошибку времени исполнения при попытке вычислить себя на некоторых аргументах), такой как поиск элемента в (возможно пустом) списке.

4. **Недетерминированные вычисления.** Функция возвращает не одно значение, а суперпозицию нескольких значений. Композиция недетерминированных функций возвращает результат, составленный из всех комбинаций недетерминированных значений.

5. **Продолжения** (англ. "continuations") являются абстрактным представлением состояния потока исполнения программы и часто используются для кодирования других управляющих структур в языках программирования.

Подобные примеры, требующие особых формализмов при описании семантики вычислений называются *вычислительными эффектами* (англ. "computational effects"). Производящую эффект функцию будем называть *эффективной функцией* (англ. "effectful function"), а возвращаемое ею значение — *эффективным значением* (англ. "effectful value").

1.2. Способы формализации

Вычисления с эффектами исследуются главным образом в статически-типизированных функциональных языках программирования (таких как упомянутый выше Haskell или Scala [42]). Важным вопросом является построение формальной модели, на основе которой можно построить семантику вычислений с эффектами. Чаще всего это делается кодированием эффекта в типе функции, либо введением системы эффектов, ортогональной системе типов. В первом случае разделяется тип чистой функции $\alpha \to \beta$ и тип эффективной функции $\alpha \to f\ \beta$, где f — это, в некотором смысле, контейнер, содержащий чистое значение. Удобно рассматривать f как алгебраическую структуру [54] с заданным набором операций.

1.2.1. Системы типов и эффектов

Системы типов и эффектов (англ. “type and effect system”) стали одним из первых подходов к формализации вычислительных эффектов. В нём в язык по отдельности вводится система типов и система эффектов.

Жувлот и Гиффорд [28] представили алгоритм *алгебраической реконструкции* (англ. “algebraic reconstruction”) типов и эффектов для полиморфного языка с поддержкой процедур, как объектов первого рода.

Талпин и Жувлот [3] разработали *дисциплину типов и эффектов* (англ. “type and effect discipline”) — формальный фреймворк, позволяющий реконструировать главный тип (англ. “principal type”) и минимальный эффект выражения в неявно типизированном полиморфном функциональном языке программирования с поддержкой императивных языковых конструкций.

Тофте и Талпин [56, 57] разработали *исчисление регионов* (англ. “region calculus”). Предложенный ими подход помещает все доступные во время исполнения значения (включая замыкания) в *регионы*, образующими стек. Специальный анализ выводит места, где регионы могут быть аллоцированы и деаллоцированы. Основное практическое применение этого формализма — построение безопасной модели памяти для языков программирования. Так, язык Cyclone [25] представляет собой клон C с аннотациями эффектов, повышающими надежность при работе с выделением и освобождением памяти.

Бауэр и Претнар построили язык Eff [7], реализующий *обработчики эффектов* (англ. “effect handlers”) для обработки состояния, I/O, недетерминизма, бэктрекинга и др.

Разновидностью легковесной системы эффектов можно назвать *проверяемые исключения* (англ. “checked exceptions”) в Java [20] и других языках. Функция аннотируется типами выбрасываемых исключений, а специальная проверка следит за тем, чтобы реализация не выходила за пределы этих аннотаций — т.е. что никакое выражение внутри функции не выбрасывает исключение, не аннотированное в её сигнатуре. Этот анализ несколько усложняется наличием отношения подтипизации у классов исключений.

Существуют попытки добавить легковесную систему эффектов в Scala [51], расширив тип функции аннотацией производимой ей эффекта.

1.2.2. Функторы, идиомы, монады

Наиболее популярным подходом к формализации эффектов является использование перечисленных алгебраических структур. Все они будут в деталях разобраны далее. Пока достаточно сказать, что функтор обобщает понятие контекста, к содержимому которого можно применить чистую функцию и получить изменённый контекст. Аппликативный функтор (также называемый *идиома*, англ. “idiom”) расширяет функтор на случай нескольких контекстов одного типа и чистой функции от нескольких аргументов, которую можно поаргументно применить к содержимому контекстов. Монада обобщает аппликативный функтор на случай зависимости значения одного содержимого контекста от другого.

Вадлер и Тиманн [66] показали, что монады можно свести к системе эффектов.

1.2.3. Обобщения и специализации монад

Появление монад в Haskell, как средства для описания вычислительных эффектов, катализировало исследования моноидальных алгебраических структур, пригодных для этой же цели. В частности, были представлены следующие обобщения.

Стрелка (англ. “arrow”) — обобщение монады, предложенное Джоном Хьюзом [23]. В частности, стрелки допускают нотацию вычислений, который могут быть частично статическими (независимыми от входа) или принимать несколько входов. Линдли и др. разработали *исчисление стрелок* [32], метаязык для работы со стрелками наподобие метаязыка для работы с монадами. Ими же были рассмотрены взаимосвязи между всеми этими объектами [33]. Для Haskell существует расширение, добавляющее стрелочную нотацию в язык [44].

Относительная монада (англ. “relative monad”) определяет, следуя теоретико-категорному определению, моноид в категории функторов между строго различными категориями (в противовес эндофункторам в определении монады). Альтеркирх и др. [1,2] предложили примеры вычислений, не формализуемых с помощью монад, но допустимых в относительных монадах.

Помимо полезных обобщений, есть и полезные специализации, вносящие дополнительные свойства ценой уменьшения выразительной силы.

Аддитивная монада (англ. “additive monad”) получается добавлением к монаде ассоциативной операции и единицы (фактически — моноида). В Haskell ей соответствует класс типов *MonadPlus* [4]. Аддитивная монада формализует наличие или отсутствие результатов вычисления внутри монадического контекста и допускает свёртку на моноиде из эффективных значений.

Свободная монада (англ. “free monad”) — минимальная структура, удовлетворяющая всем законам монады. Их особенность в том, что они, в отличии от монад, коммутируют.

1.2.4. Дуальные алгебраические структуры

Монады и функторы могут рассматриваться в дуальных категориях, что порождает, соответственно комонады [61] и кофункторы. В отличии от монад, комонады предоставляют операции для композиции функций со структурированным входом (а не выходом). Применительно к вычислениям с эффектами они интересны с точки зрения исследования зависимости вычислений от среды, в которой они были запущены [46]. Комонады используются для описания потоков данных (англ. “dataflow programming”) [59], синтеза атрибутов в атрибутивных грамматиках [60], векторных вычислениях и др. Для вычислениях в комонадах в Haskell предложена специальная **codo**-нотация [43].

1.2.5. Расширения систем типов

Некоторые из формализаций эффектов задействуют специальные свойства систем типов языка программирования. Следует выделить следующее.

Зависимые типы. Эдвин Брейди предложил [10] подход с использованием обработчиков эффектов для зависимо-типизированного языка Idris. Мотивацией для этого подхода является упрощение добавления эффектов к вычислению (в качестве примера показывается наращивание возможностей абстрактный интерпретатора), поскольку подход с использованием монад предполагает написание для этого трансформеров монад.

Линейные типы. Также известные как *уникальные типы* [5]. Уникальные типы позволяют управлять наблюдаемостью побочных эффектов. Например, зная, что на некоторое значение нет посторонних ссылок, можно его изменить, сохранив функциональную чистоту. Используется в языке Clean [12].

1.3. Синтаксические расширения

Для каждой пары чистых функций $\alpha \to \beta$ и $\beta \to \gamma$ существует композиция типа $\alpha \to \gamma$. Эффективные же функции, в общем случае, не композируются — нельзя построить $\alpha \to f\ \gamma$ из $\alpha \to f\ \beta$ и $\beta \to f\ \gamma$.

В зависимости от структуры f, над эффективным значением допустими некоторые операции, которые можно использовать для построения сложных вычислений. Так, если контейнер $f\ \alpha$ поддерживает операцию *map* типа $(\alpha \to \beta) \to f\ \alpha \to f\ \beta$, можно построить композицию чистой функции g и эффективной функции h как $map\ g\ (h\ a)$.

Необходимость использовать комбинаторы над эффективными значениями приводит к заметному усложнению кода, что даёт мотивацию для создания синтаксических расширений, упрощающих синтаксис вычислений с эффектами.

Некоторые функциональные языки программирования предоставляют специальный синтаксис, обычно для монад и аппликативных функторов. Так, Haskell вводит синтаксис *выделения монад* (англ. “monad comprehension”), обобщающий *выделение списков* (англ. “list comprehension”) и **do**-нотацию. Её аналог также реализован в Idris [9], Scala (известный как **for**-нотация) и OCaml (**perform**-нотация). F♯ поддерживает *вычислительные выражения*. Синтаксис для аппликативных вычислений, известный как идиоматические скобки, реализован в Idris и Strathclyde Haskell Enhancement. Все они будут подробно рассмотрены далее.

Глава 2. Теоретические основания

В работе рассматривается формализация вычислений с эффектами аппликативными функторами и монадами. В качестве теоретический основы используется аппарат общей алгебры и теории категорий.

2.1. Общая алгебра

Введём несколько понятий из общей алгебры с примерами из различных областей математики и языков программирования. Определения и большая часть примеров взята из [24].

Определение. *Полугруппой* называется алгебраическая структура $(\mathfrak{S}, \circ)$, состоящая из множества $\mathfrak{S}$ и замкнутой на нём ассоциативной бинарной операцией $\circ$.

Исходя из определения, свойствами полугруппы являются:

1. *Свойство замыкания.* $\forall a, b \in \mathfrak{S}.\ a \circ b \in \mathfrak{S}$
2. *Свойство ассоциативности.* $\forall a, b, c \in \mathfrak{S}.\ (a \circ b) \circ c = a \circ (b \circ c)$

Множества чисел $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ и $\mathbb{C}$ образуют полугруппу по сложению и по умножению. Множество матриц $n \times m$ образуют полугруппу по сложению и вычитанию, а множество квадратных неотрицательных матриц $n \times n$ — по умножению. Множество конечных строк фиксированного алфавита Σ образуют полугруппу относительно конкатенации строк (в этом случае его называют *свободной полугруппой* (англ. “free semigroup”)).

Можно найти примеры и в языках программирования, как очевидные, вроде чисел, строк и списков, так и более изощрённые, как лог-файлы, допускающие добавление записи в конец. Легко показать, что такая операция ассоциативна и замкнута относительно типа лог-файла.

Ассоциативность связанной с полугруппой операции может быть основой для оптимизации алгоритмов. Легко показать, как свёртка списка из n элементов полугруппы может быть реализована за $O(\log n)$. На этом факте основан, к примеру, алгоритм бинарного возведения в степень.

Определение. *Моноидом* называется алгебраическая структура $(\mathfrak{S}, \circ, e)$, такая, что $(\mathfrak{S}, \circ)$ образуют полугруппу, а $e \in \mathfrak{S}$ является нейтральным элементом операции $\circ$.

Свойствами моноида являются:

1. *Свойство замыкания.* $\forall a, b \in \mathfrak{S}.\ a \circ b \in \mathfrak{S}$
2. *Свойство нейтральности слева.* $\forall a \in \mathfrak{S}.\ a \circ e = a$
3. *Свойство нейтральности справа.* $\forall a \in \mathfrak{S}.\ e \circ a = a$
4. *Свойство ассоциативности.* $\forall a, b, c \in \mathfrak{S}.\ (a \circ b) \circ c = a \circ (b \circ c)$

Как следует из определения, моноид является частным случаем полугруппы. Пример полугруппы, не являющейся моноидом — $(\mathbb{Z}, \min)$. Очевидно, что $\min$ ассоциативна, при этом на множестве $\mathbb{Z}$ нельзя выбрать наибольший элемент. Тем не менее, большая часть полугрупп может быть расширена до моноида. Так, упомянутые ранее множества чисел $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ и $\mathbb{C}$ являются моноидами относительно сложения и нуля и умножения и единицы. Наличие единичной матрицы позволяет считать множество квадратных матриц моноидом по умножению, а наличие нулевой матрицы — моноидом по сложению. Среди других примеров — $(\mathbb{B}, \wedge, 0)$, $(\mathbb{B}, \vee, 1)$, $(\mathbb{Z}, \mathrm{lcm}, 1)$ и $(\mathbb{Z}, \gcd, 0)$.

Стоит отметить, что операция $\circ$ в общем случае не коммутативна. Обладающие этим свойством моноиды называются *коммутативными*. Примером некоммутативного моноида является множество квадратных матриц с операцией умножения.

Многие абстрактные типы данных в языках программирования обладают моноидальной структурой, например списки (пустой список соответствует нейтральному элементу; непустой список при этом образует полугруппу) и деревья. Вычисления, задействующие свёртку моноидов активно используются в функциональном программировании [50]. Так, сумму списка можно выразить с помощью свёртки моноида по сложению, а нахождение максимального элемента — свёрткой по операции $\max$.

2.2. Теория категорий

Теория категорий традиционно используется для построения рассуждений о семантике языков программирования. Определения и свойства основных понятий взяты из [37] и [6]. Приложения в языках программирования рассматриваются, в частности, в [49] и [39].

2.2.1. Категории, объекты и морфизмы

Определение. *Категорией* $\mathcal{C}$ называется алгебраическая конструкция, состоящая из

1. Набора *объектов*, обозначаемого $\mathrm{Ob}(\mathcal{C})$.

2. Набора *стрелок* (или *морфизмов*), обозначаемого $\mathrm{Hom}(\mathcal{C})$.

3. Операций, назначающих каждой стрелке f объект $\mathrm{Dom}(f)$, её область определения (*домен*, англ. "domain") и объект $\mathrm{Cod}(f)f$, её область значений (*кодомен*, англ. "codomain"). Будем записывать $f : A \to B$, чтобы показать, что $\mathrm{Dom}(f) = A$, а $\mathrm{Cod}(f) = B$. Набор стрелок с доменом A и кодоменом B будем записывать как $\mathcal{C}(A, B)$.

4. Оператора композиции, назначающего каждой паре стрелок f и g таких, что $\mathrm{Cod}(f) = \mathrm{Dom}(g)$, стрелку *композиции* $g \circ f : \mathrm{Dom}(f) \to \mathrm{Cod}(g)$, удовлетворяющую *ассоциативному закону*: для любых $f : A \to B$, $g : B \to C$ и $h : C \to D$ (где A, B, C и D не обязаны быть различными), $h \circ (g \circ f) = (h \circ g) \circ f$.

5. *Тождественной* стрелки $\mathrm{id}_A : A \to A$ для каждого объекта A, удовлетворяющей *закону тождества*: для любой $f : A \to B$, $\mathrm{id}_B \circ f = f$ and $f \circ \mathrm{id}_A = f$.

Рассмотрим несколько примеров категорий:

1. Категория $\mathcal{S}et$, объектами которой являются множества, а морфизмами — тотальные функции между множествами. Композицией морфизмов является теоретико-множественная функция композиции, а тождественным морфизмом является тождественная функция.

2. Любой ориентированный граф задаёт категорию, где объекты — это вершины, а стрелки — это связанные пути в графе. Композицией морфизмов является соединение вершин графа, а тождественным морфизмом является путь, в котором нет ни одного ребра.

3. Любое множество с заданным предпорядком образует категорию. Объектами являются элементы множества, стрелками — пары элементов, для которых определено отношение предпорядка. Композиция морфизмов задаётся как $(a, b) \circ (b, c) = (a, c)$, а тождественным морфизмом является пара, состоящая из одинаковых объектов.

4. Категория $\mathcal{M}on$, объектами которой являются моноиды, а морфизмами — моноидные гомоморфизмы (т.е. отображения, сохраняющие нейтральный элемент и замкнутые относительно операции). Композицией морфизмов является композиция гомоморфизмов, а тождественным морфизмом является моноидальный эндоморфизм.

Для всех этих примеров можно показать выполнимость ассоциативного закона и закона тождества.

В определении сознательно используется термин *набор* (англ. "collection"), а не *множество*. Категория $\mathcal{C}$, чьи $\mathrm{Ob}(\mathcal{C})$ и $\mathrm{Hom}(\mathcal{C})$ являются множествами называется *малой*. Категория $\mathcal{C}$, чей $\mathcal{C}(A, B)$ для любых A и B является множеством, называется *локально малой*. Так, категория $\mathcal{S}et$ не является малой, но является локально малой.

Убрав из определения малой категории существование тождественного морфизма для каждого объекта, можно получить структуру, известную как *полугруппоид* (англ. "semigroupoid").

Для описания категорий часто прибегают к *коммутативным диаграммам*, представляющих собой визуальную нотацию для объектов и стрелок. Объекты на диаграмме показаны как вершины ориентированного графа, стрелки — как его рёбра. Коммутативность диаграммы означает то, что для любой пары вершин X и Y, все пути из X в Y задают одинаковый морфизм. На рисунке 2.1 представлена коммутативная диаграмма категории, состоящей из объектов A, B и C, и морфизмов $f : A \to B$ и $g : B \to C$.

Определение. Категория $\mathcal{C}^{\mathrm{op}}$ называется *дуальной* к категории $\mathcal{C}$, если все свойства $\mathcal{C}^{\mathrm{op}}$ соответствуют свойствам $\mathcal{C}$ с точностью до перестановки

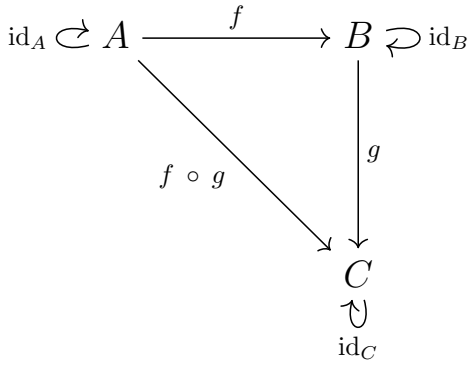

Рис. 2.1. Коммутативная диаграмма композиции морфизмов.

домена и кодомена и порядка композиции морфизмов. На коммутативной диаграмме $\mathcal{C}^{\mathrm{op}}$ выглядит, как $\mathcal{C}$ с перевёрнутыми стрелками.

Определение. *Начальным объектом* (англ. "initial object") I категории $\mathcal{C}$ называется объект, такой, что для любого объекта A существует ровно один морфизм из I в A.

Определение. *Конечным объектом* (англ. "terminal object") T категории $\mathcal{C}$ называется объект, такой, что для любого объекта A существует ровно один морфизм из A в I.

Например, в категории $\mathcal{S}et$ единственным начальным объектом является пустое множество, а любое одноэлементное множество является конечным объектом.

Легко показать, что начальные объекты категории $\mathcal{C}$ являются конечными объектами категории $\mathcal{C}^{\mathrm{op}}$ и наоборот.

Определение. Категория $\mathcal{S}$ называется *подкатегорией* $\mathcal{C}$, если

1. Каждый объект $\mathcal{S}$ является объектом $\mathcal{C}$.
2. Для любой пары $\mathcal{S}$-объектов A и B, $\mathcal{S}(A, B)$ является поднабором $\mathcal{C}(A, B)$.
3. Тождественные морфизмы и морфизмы композиции категорий совпадают.

Интуитивно, подкатегория представляет собой категорию с некоторыми удалёнными объектами и морфизмами.

2.2.2. Функторы и естественные преобразования

Определение. Пусть $\mathcal{C}$ и $\mathcal{D}$ — категории. *Функтором*

$$\mathbf{F} : \mathcal{C} \to \mathcal{D}$$

называется отображение каждого $\mathcal{C}$-объекта A в $\mathcal{D}$-объект $\mathbf{F}(A)$ и $\mathcal{C}$-морфизма $f : A \to B$ в $\mathcal{D}$-морфизм $\mathbf{F}(f) : \mathbf{F}(A) \to \mathbf{F}(B)$, такое, что для всех $\mathcal{C}$-объектов A и допускающих композицию $\mathcal{C}$-морфизмов f и g выполняется:

1. $\mathbf{F}(\mathrm{id}_A) = \mathrm{id}_{\mathbf{F}(A)}$
2. $\mathbf{F}(f \circ g) = \mathbf{F}(f) \circ \mathbf{F}(g)$

Интуитивно, функтор является *гомоморфизмом* между категориями, т.е. отображением, сохраняющим тождества и композицию. После отображения морфизма $f : A \underset{\mathcal{C}}{\to} B$, получаем морфизм[1] $\mathbf{F}(f) : \mathbf{F}(A) \underset{\mathbf{F}(\mathcal{C})}{\longrightarrow} \mathbf{F}(B)$, а композиция $f \circ g$ переходит в $\mathbf{F}(f \circ g)$, как показано на рисунке 2.2.

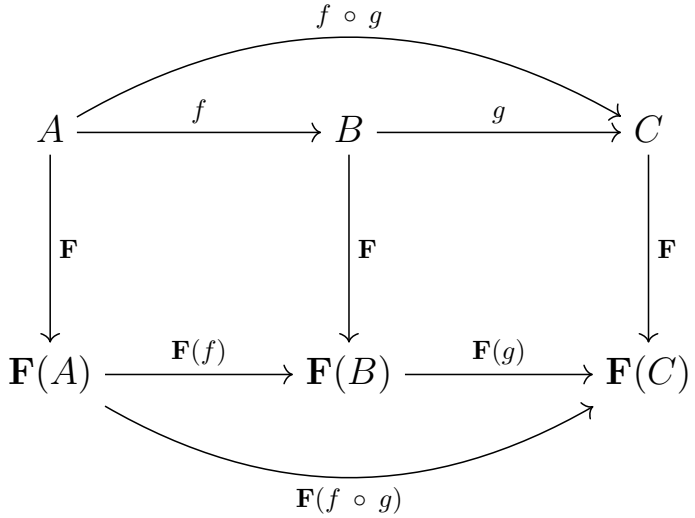

Рис. 2.2. Коммутативная диаграмма функтора $\mathbf{F}$.

Примером функтора является $\mathbf{F} : \mathcal{M}on \to \mathcal{S}et$, отображающий каждый моноид во множество, на котором он задан, а каждый моноидальный гомоморфизм в функцию между множествами, на которых заданы домен и кодомен. Видно, что при таком отображении часть структуры категории теряется. Чтобы отразить это свойство, такой функтор называют *забывающим* (англ. “forgetful”).

[1]Подпись под стрелкой указывает, в какой категории объявлен морфизм.

Определение. Функтор $\mathbf{F} : \mathcal{C} \to \mathcal{C}$ называется *эндофунктором.*

Определение. Функтор $\mathbf{F} : \mathcal{C} \times \mathcal{D}$ от двух аргументов называется *бифунктором.*

Определение. Пусть $\mathcal{C}$ и $\mathcal{D}$ — категории, а $\mathbf{F}$ и $\mathbf{G}$ — функторы из $\mathcal{C}$ в $\mathcal{D}$. *Естественным преобразованием* (англ. "natural transformation")

$$\eta : \mathbf{F} \dot{\to} \mathbf{G}$$

называется отображение, каждому $\mathcal{C}$-объекту A назначающая $\mathcal{D}$-морфизм $\eta_A : \mathbf{F}(A) \to \mathbf{G}(A)$ такой, что для каждого $\mathcal{C}$-морфизма $f : A \to B$ диаграмма на рисунке 2.3 коммутирует в категории $\mathcal{D}$.

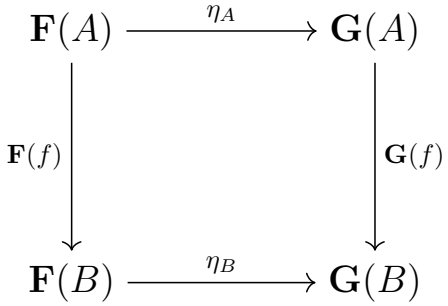

Рис. 2.3. Коммутативная диаграмма естественного преобразования η.

Определение. Если соответствующий каждому объекту A категории $\mathcal{C}$ морфизм η_A является изоморфизмом в категории $\mathcal{D}$, η называется *естественным изоморфизмом* (англ. "natural isomorphism").

Примечательным свойством является то, что для любых категорий $\mathcal{C}$ и $\mathcal{D}$ функторы между ними образуют категорию с морфизмами, заданными естественными преобразованиям.

2.2.3. Моноидальные категории

Определение. Категория $\mathcal{C}$ называется *моноидальной*, если для неё определены

1. Бифунктор $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ (называемым *тензорным* или *моноидальным произведением*).

2. Объект I, называемый *единичным объектом* (англ. “unit object”).

3. Три естественных изоморфизма α, λ и ρ, удовлетворяющих коммутативным диаграммам на рисунках 2.4 и 2.5, которые выражают, что

   3.1. Тензорное произведение ассоциативно: $(A \otimes B) \otimes C = A \otimes (B \otimes C)$.

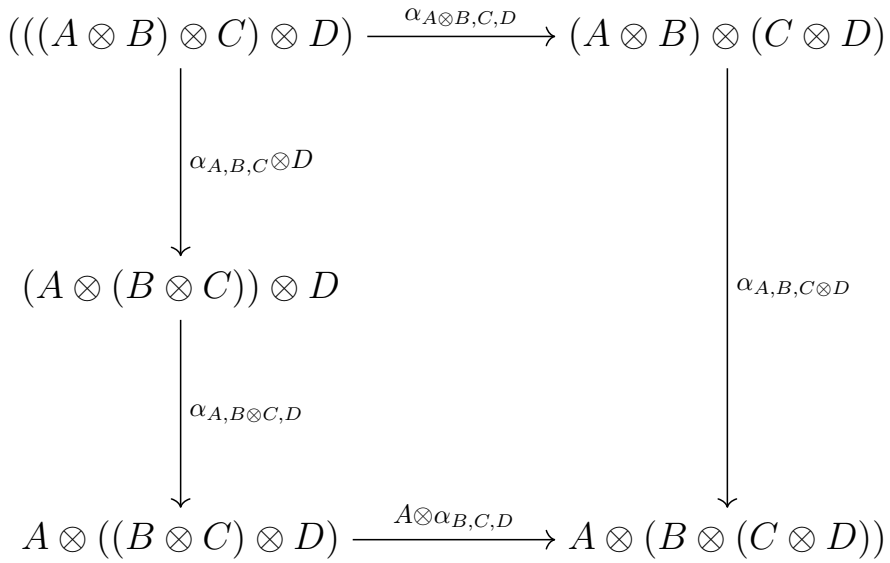

Рис. 2.4. Коммутативная диаграмма естественных изоморфизмов α.

   3.2. I является нейтральным элементом слева и справа относительно тензорного произведения: $A \otimes I = I \otimes A = A$.

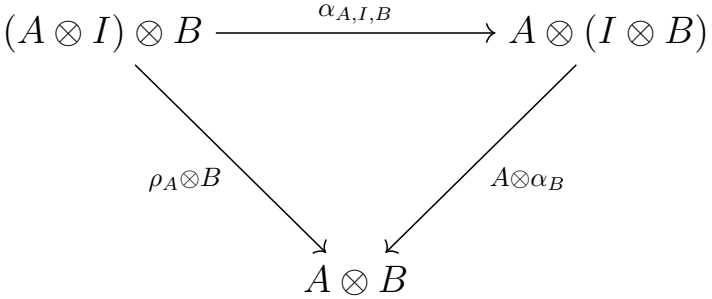

Рис. 2.5. Коммутативная диаграмма естественных изоморфизмов λ и ρ.

Исключив из определения моноидальной категория существование единичного объекта можно получить определение *полумоноидальной категории*.

2.2.4. Приложения в теории языков программирования

Рассмотрим $\mathcal{H}ask$, категорию типов и функций языка Haskell. В ней

1. Объектами являются типы (*String*, [*Int*], *String* $\rightarrow$ *Int*).
2. Морфизмами являются функции (*head*, *tail*, *show*).
3. Композицией морфизмов является композиция функций ($\circ$).
4. Тождественным морфизмом является тождественная функция *id*.
5. Подкатегориями являются, например, все типы, образованные применением какого-либо типового конструктора (такого как *Maybe*, [] или *Either String*).
6. Каждый типовый конструктор f, конструктор данных $F : \alpha \rightarrow f\ \alpha$ и функция *map* с типом $f\ (\alpha \rightarrow \beta) \rightarrow f\ \alpha \rightarrow f\ \beta$ задают функтор из $\mathcal{H}ask$ в подкатегорию $\mathcal{H}ask$, задаваемую f.
7. Единственным эндофунктором является тождественный эндофунктор, задаваемый функцией *id*.
8. Функция *listToMaybe* :: $[a] \rightarrow$ *Maybe a* является примером естественного преобразования $\eta : \mathbf{F} \dot{\rightarrow} \mathbf{G}$ между функторами $\mathbf{F} : \mathcal{H}ask \rightarrow \mathcal{L}st$ и $\mathbf{G} : \mathcal{H}ask \rightarrow \mathcal{M}aybe$.
9. $\mathcal{H}ask$ является моноидальной категорией с бифунктором, задаваемым типом кортежа из двух элементов и единичным объектом ().

Аналогичным образом можно построить $\mathcal{S}cal$ — категорию типов и функций языка Scala. Строго говоря, в обоих случаях рассматривается лишь некоторое пригодное для формализации подмножество языков [17].

Глава 3. Средства реализации

3.1. Язык программирования Scala

Scala [42] представляет собой промышленный объектно-ориентированный и функциональный язык программирования со строгой статической типизацией и возможностью вывода типов. Scala компилируется в байт-код JVM и обращение к библиотечным функциям Java.

Scala был выбран в качестве основного языка разработки в этой работе. В данной главе рассматриваются основные затронутые возможности языка.

По сравнению с Java, Scala содержит множество элементов, относящихся к функциональному программированию. Среди них

1. Строгое разделение изменяемых и неизменяемых переменных. Переменные, отмеченные как **var** доступны как для чтения, так и модификации, в то время как **val**-переменные нельзя изменить после инициализации.

2. Поддержка λ-функций и функций высшего порядка.

3. Поддержка алгебраических типов данных (англ. “algebraic data types”) и сопоставления с образцом (англ. “pattern matching”).

Scala является объектно-ориентированным языком, основными конструкциями которого являются классы, абстрактные классы и трейты. Последние являются расширением интерфейсов в Java, допуская частичную реализацию и основанное на миксинах (англ. “mixin”) наследование, что позволяет избежать проблем с множественным наследованием, присущим другим языкам.

Поддерживается т.н. неявную (англ. “impicit”) область видимости. Все значения, объявленные в ней (с помощью ключевого слова **implicit**) могут быть автоматически подставлены в аргументы вызова функции, также помеченных как **implicit**. Эта возможность используется, например, для упрощения API, когда значение, используемое в вызовах нескольких функций (например, кодировка) может быть объявлено один раз и автоматически подставлено в всех необходимых местах.

Scala обладает развитой системой типов. В частности, доступен вывод типов и типы высшего порядка (англ. “higher-kinded types”). Implicit-функции используются для реализации *неявного приведения типов* (англ. “implicit conversion”).

Scala обладает развитым C-подобным синтаксисом со множеством синтаксического сахара. К примеру, специальный символ _ используется для упрощения синтаксиса λ-функций, вместо выражения $x \Rightarrow x + 1$ допустимо писать _ + 1. Любой метод объекта может быть использован в инфиксной форме. В некоторых случаях допустимо опускать операторные скобки у блочного аргумента функции и писать *foo* { *bar*(); *baz*() } вместо *foo*({ *bar*(); *baz*() }), что активно используется при построении библиотек и встраиваемых языков.

3.2. Средства метапрограммирования

В 2011 году в Scala появились средства для метапрограммирования. Первый технический отчёт [14] представлял реализацию макро-системы под названием *def*-макросы.

Наличие в Scala статической системы типов, системы аннотаций и различных сортов синтаксического сахара даёт пространство для прочих модификаций макро-системы. В 2012 году эксперименты продолжились и в отдельной ветке компилятора появились macro flavors [13] — макро-системы для различных частей языка. В частности, доступны:

1. **Динамические макросы.** Начиная с версии 2.9, Scala позволяет переписывать операции над несуществующими полями и вызовы несуществующих методов у объектов, наследующих трейт *Dynamic* в вызовы методов *selectDynamic*, *updateDynamic* и *applyDynamic*. Динамические макросы позволяют переопределить эти макросы в виде макро-определений и генерировать код при необходимости.

2. **Implicit-макросы** являются обыкновенными *def*-макросами с implicit-аннотацией, что позволяет использовать их как implicit-методы. В данный момент основным сценарием использования implicit-макросов является возможности автоматической материализации реализаций классов типов.

3. **Типовые макросы** являются в некотором смысле макросистемой на уровне типов. Встретив использование макро-типа, компилятор раскрывает его в некоторый другой тип, возможно сгенерировав при этом определение необходимого класса.

4. **Макро-аннотации** концептуально похожи на типовые макросы и предназначены для увеличения модульности компилятора, в частности, с помощью перевода реализации ленивых значений и **case**-класс в стандартную библиотеку в виде макросов.

5. **Нетипизированные макросы.** В отличии от *def*-макросов, принимают на вход нетипизированное синтаксическое дерево (т.е. не содержащее меток типов у узлов дерева) и могут проверить тип всего выражения или некоторого подвыражения позже. Следствием этого является то, что нетипизированный макрос может принимать на вход некорректный Scala-код (синтаксически корректный, но не проходящий проверку типов), но модифицирующий его так, что он становится корректным.

Для реализации синтаксического расширения, представляемого в этой работе, используются нетипизированные макросы. Для разработки синтаксического расширения они предоставляют готовый синтаксис языка (уже разобранное синтаксическое дерево), систему типов и доступ к тайп-чекеру и рефлексии, т.е. фактически неограниченный доступ к манипулициям с интерпретацией выражений языка.

Глава 4. Функторы и идиомы

4.1. Обзор

Понятие *функтора*, применительно к программированию, было впервые введено, вероятно, М. Джонсом [26]. В Haskell оно задаётся как класс типов с единственным методом *map*, применяющий чистую функцию к содержимому контейнера.

class *Functor* f **where**
  $fmap :: (\alpha \to \beta) \to f\ \alpha \to f\ \beta$

Аналогичное определение в Scala состоит из трейта с одним методом.

```
trait Functor[F[_]] {
  def fmap[A, B](f : A ⇒ B) : F[A] ⇒ F[B]
}
```

Аппликативный функтор (или *идиома*, англ. “idiom”) был предложен МакБрайдом и Паттерсоном [36] для описания эффективных вычислений в аппликативном стиле.

class *Functor* $f \Rightarrow$ *Applicative* f **where**
  $pure :: \alpha \to f\ \alpha$
  $(\circledast) :: f\ (\alpha \to \beta) \to f\ \alpha \to f\ \beta$

Его аналог в Scala:

```
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a : A) : F[A]
  def app[A, B](f : F[A ⇒ B]) : F[A] ⇒ F[B]
}
```

Аппликативный функтор является обобщением функтора для функции от нескольких аргументов. *fmap* можно выразить с помощью *pure* и $\circledast$ в виде $fmap\ f\ a = pure\ f\ \circledast\ a$.

4.2. Приложения в языках программирования

Многие встроенные конструкторы типов функциональных языков программирования поддерживают реализацию функтора и идиомы.

Так, реализация типа, представляющего наличие или отсутствие результата вычисления выглядит в Haskell следующим образом.

```
instance Applicative Maybe where
  pure = Just
  (Just f) ⊛ (Just x) = Just (f x)
  _        ⊛ _        = Nothing
```

Его аналог в Scala выглядит как

```
val option = new Applicative[Option] {
  def pure[A](a : A) = Some(a)
  def app[A, B](f : Option[A ⇒ B]) =
    x ⇒ (f, a) match {
      case (Some(g), Some(a)) ⇒ Some(f(a))
      case _ ⇒ None
    }
}
```

Реализация функтора и идиомы позволяет безопасно работать с результатом вычисления, которое могло завершиться неудачей. В отличии языков, использующих *null*-значения для описания отсутствия результата (и производящих ошибку времени исполнения при попытке работы с ними), языки с закодированными в типе эффектами гарантируют, что некорректное значение будет обработано. Так, имея функцию add, суммирующую три своих целочисленных аргумента

```
add :: Int → Int → Int
add x y z = x + y + z
```

можно применять её к значениям, содержащим, или не содержащим целое число:

```
pure add ⊛ Just 3 ⊛ Just 2 ⊛ Just 5
```

Конструктор типа может иметь и несколько реализаций эффекта. В частности, для списка определяют и реализацию идиомы, методом ⊛ строящую список из всех комбинаций элементов списка и реализацию, известную как *zipList*, строящую список из поэлементно взятых пар значений списка.

В данной работе понятие функтора и идиомы используется, в первую очередь, в контексте вычислений с эффектами, хотя построенные реализации полезны и сами по себе. Рассмотрим реализацию идиомы для списка. С одной стороны, *fmap* — полезная функция, позволяющая применить операцию к каждому элементу списка, а ⊛ — для комбинирования всех элементов нескольких списков. С другой — список традиционно используется для представления результатов недетерминированных вычислений, и в этом смысле методы функторов можно рассматривать в роли комбинаторов, связывающих чистые и эффективные вычисления. *fmap* соответствует композиции недетерминированного и чистого вычисления, чистая функция применяется к каждому из исходов недетерминированного вычисления, в результате чего получается новое недетерминированное значение. ⊛ соответствует композиции нескольких недетерминированных вычислений, чей результат строится из возможных исходов всех недетерминированных аргументов.

Иногда рассматривают аппликативный функтор без оператора *pure*. Он определён, к примеру, как класс типов *Apply* в пакете `semigoupoids` Эдварда Кхметта и библиотеке `scalaz`. Далее на него будем ссылаться как на «полуидиому».

Поскольку идиома является частным случаем функтора, существуют функторы, не являющиеся идиомами. Пример такого функтора — пара с зафиксированным первым аргументом. Можно определить операцию *fmap*, применяющий функцию ко второму элементу и не затрагивающую первый, но для реализации *pure* и ⊛ необходима нотация для пустого значения и композиции значений типа первого элемента. Построить объект идиомы для пары можно, наложив дополнительное ограничение на соответствие интерфейсу моноида для первого элемента. Также можно построить объект полуидиомы, наложив ограничение на соответствие интерфейсу полугруппы.

4.3. Теоретико-категорные основания

Как и подсказывает название, *Functor f* формализуется функтором из категории $\mathcal{H}ask$ ($\mathcal{S}cal$) в подкатегорию, задаваемую типовым конструктором f. Так, к примеру, реализация функтора для списка является функтором из категории $\mathcal{H}ask$ ($\mathcal{S}cal$) в категорию $\mathcal{L}st$, объектами которой являются списочные типы вида $[\alpha]$, а морфизмами являются функции над списочными типамим вида $[\alpha] \to [\beta]$. Строго говоря, функция *fmap* определяет отображением между морфизмами категорий, а в роли отображения между объектами выступает конструктор данных $F : f\ \alpha$, соответствующий конструктору типа. Обычно считается, что он всегда определён.

Законы следуя формальному определению, все реализации функторов должны удовлетворять следующим законам:

1. Закон тождества.
   $fmap\ id \equiv id$

2. Закон композиции.
   $fmap\ (p \circ q) \equiv (fmap\ p) \circ (fmap\ q)$

И в Haskell, и в Scala реализация функтора должна сама обеспечивать выполнимость этих законов, никакой языковой поддержки для верификации не предоставляется. При этом примечательно, что выполнимость закона композиции следует напрямую из закона тождества, согласно сигнатуре типа функции *fmap* и $(\circ)$, что гарантируется Вадлеровскими «бесплатными теоремами» [63].

Аппликативный функтор *Applicative f* задаётся как слабый моноидальный эндофунктор (англ. “lax monoidal endofunctor”) в категории $\mathcal{H}ask$ ($\mathcal{S}cal$). Рассмотрим это определение.

Определение. *Слабый моноидальный функтор* между моноидальными категориями $(\mathcal{C}, \otimes, I, \lambda, \rho, \alpha)$ и $(\mathcal{C}', \otimes', I', \lambda', \rho', \alpha')$ состоит из функтора $\mathbf{F} : \mathcal{C} \to \mathcal{C}'$ и двух естественных преобразований:

1. $u : I' \to \mathbf{F}(I)$

2. $\circledast : \mathbf{F}(A) \otimes' \mathbf{F}(A) \to \mathbf{F}(A \otimes B)$

таких, что диаграммы на рисунках 4.1, 4.2 (определяющих, что u является нейтральным элементом слева и справа относительно операции $\circledast$) и 4.3 (определяющей ассоциативность $\circledast$) коммутируют.

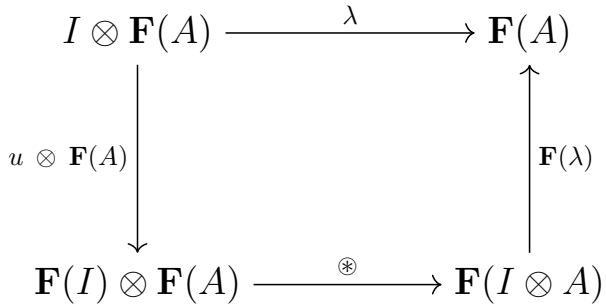

Рис. 4.1. Коммутативная диаграмма естественного преобразования λ.

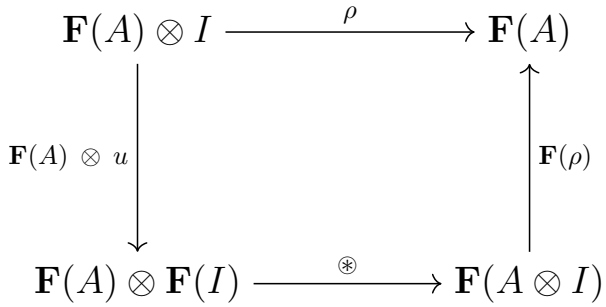

Рис. 4.2. Коммутативная диаграмма естественного преобразования ρ.

Аналогично экземплярам функторов, экземпляры аппликативных функторов обязаны подчиняться следующим законам:

1. Закон тождества.
   $pure\ id\ \circledast\ v\ \equiv\ v$

2. Закон композиции.
   $pure\ (\circ)\ \circledast\ u\ \circledast\ v\ \circledast\ w\ \equiv\ u\ \circledast\ (v\ \circledast\ w)$

3. Закон гомоморфизма.
   $pure\ f\ \circledast\ pure\ x\ \equiv\ pure\ (f\ x)$

4. Закон перестановки.
   $u\ \circledast\ pure\ y\ \equiv\ pure\ (\$\ y)\ \circledast\ u$

Полуидиоме соответствует слабый *полу*-моноидальный эндофунктор(т.е. определённый в полумоноидальной категории).

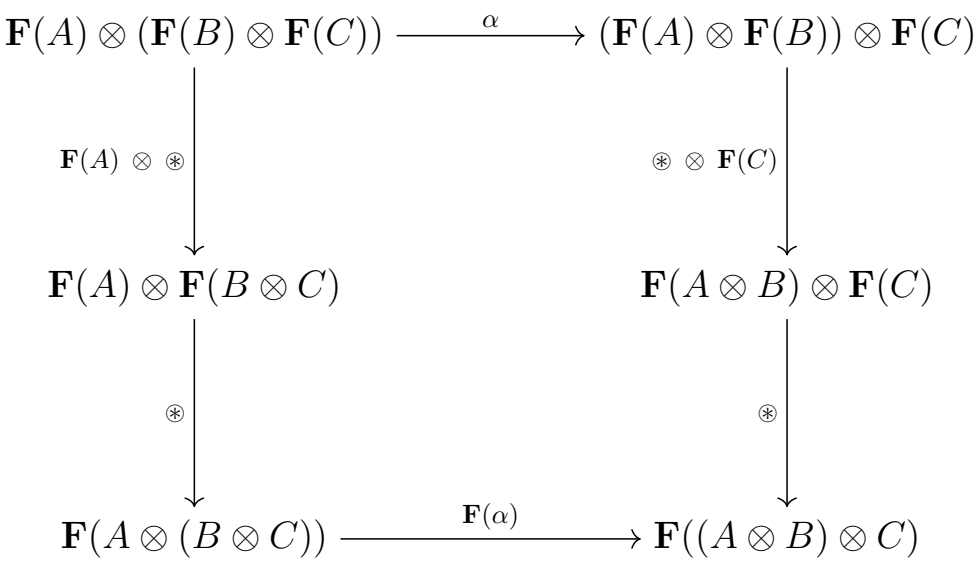

Рис. 4.3. Коммутативная диаграмма естественного преобразования α.

4.4. Синтаксические расширения

В оригинальной статье МакБрайда и Паттерсона [36], посвящённой аппликативным функторам, приводится нотация *идиоматических скобок* (англ. “idiom brackets”), упрощающая синтаксис применения чистой функции к эффективным аргументам. Так, вместо канонической формы записи в виде

$$pure\, f \;\circledast\; u_1 \;\circledast\; \ldots \;\circledast\; u_n$$

запись с идиоматическими скобками имеет вид

$$[\![\, f \;\; u_1 \;\; \ldots \;\; u_n \,]\!].$$

Существует несколько реализаций этой нотации в различных языках. В Haskell она может быть частично реализована с помощью специального класса типов *Idiomatic* и вспомогательных алгебраических типов.

Препроцессор Strathclyde Haskell Enhancement [35] для Glasgow Haskell Compiler [27] предоставляет нотацию вида $(|\, f \;\; a_1 \;\; \ldots \;\; a_n \,|)$. В качестве функции допустимо использовать инфиксный оператор. Кроме того, для идиом, реализующих класс типов *Alternative* [68] (интерфейс аппликативного функтора, расширенный интерфейсом моноида) допустим синтаксис вида $(|\, idiom_1 \mid idiom_2 \mid \ldots \mid idiom_n \,|)$, выбирающий первое встреченное непустое значение среди $idiom_k$.

В Idris реализован свой вариант идиоматических скобок [8], состоящий из конструкций **idiom** и [| *expr* |]. Аргументами в **idiom** передаются функции, соответствующие методам *pure* и ⊛, а также блок кода, в котором конструкции [| *expr* |] будут заменены на вызовы переданных функций.

Специального синтаксиса для простых функторов представлено не было. Конечно, имея реализацию идиомы можно в некоторых случаях использовать идиоматические скобки для функции от одного аргумента (что соответствует функтору). Правда, сгенерирована по прежнему будет пара вызовов *pure* и ⊛ (вместо достаточного вызова *map*).

Аппликативные функторы и идиоматические скобки формализуют понятие *поднятия* (англ. "lifting") в математике [21]. Такая нотация широко используется, например, для композиции функционалов в анализе:

$$(f + g)(x) = f(x) + g(x).$$

Следует отметить ряд ограничений нотации идиоматических скобок.

1. Поддерживается только применения одной чистой функции, поддержки вложенных функций нет. Так, запись $[\![\ \mathit{Just}\ 2\ +\ \mathit{Just}\ 3\ ]\!]$ корректна, а $[\![\ \mathit{Just}\ 2\ +\ \mathit{Just}\ 3\ +\ \mathit{Just}\ 5\ ]\!]$ уже нет, т.к. это вложенный вызов функции сложения.

2. В вызове чистой функции ожидаются только эффективные аргументы, нельзя смешивать чистые и эффективные вычисления. Strathclyde Haskell Enhancement допускает специальный синтаксис, аргумент с префиксом ∼ оборачивается в вызов *pure*, но так или иначе вызов *map* не генерируется ни в каком случае.

3. Идиоматические скобки рассчитаны на вычисления в идиомах, но в действительности они они могут работать и с полуидиомами, более слабом контексте. Вызов функции от одного аргумента может транслироваться в вызов *map*, а от двух и более — в последовательность из *map* и ⊛, вызов *pure* не требуется нигде. Таким образом, предлагаемая в работе МакБрайда и Паттерсона нотация затрагивает больший интерфейс, чем требуется.

4.5. Расширение `scala-workflow`

Результатом работы является разработанное расширение Scala, устраняющее рассмотренные выше ограничения специального синтаксиса вычислений с аппликативными функторами (а также специального синтаксиса для работы с монадами, рассматриваемыми далее). Исходный код расширения, названного `scala-workflow`, открыт и доступен для загрузки по адресу `https://github.com/aztek/scala-workflow`.

4.5.1. Иерархия вычислительных контекстов

Взамен формализации эффекта какой-либо заданной структурой, предлагается более модульный подход. Сила вычислительного контекста наращивается инкрементально добавлением новых методов к пустому базовому трейту *Workflow*, в качестве аргумента принимающего конструктор типа.

```
trait Workflow[F[_]]
```

Методы добавляются с помощью миксинов соответствующих трейтов, отнаследованных от *Workflow*. Так, каждому из рассмотренных ранее методов соответствует свой трейт.

```
trait Pointing[F[_]] extends Workflow[F] {
  def point[A](a : A) : F[A]
}


trait Mapping[F[_]] extends Workflow[F] {
  def map[A, B](f : A ⇒ B) : F[A] ⇒ F[B]
}


trait Applying[F[_]] extends Workflow[F] with Mapping[F] {
  def app[A, B](f : F[A ⇒ B]) : F[A] ⇒ F[B]
}
```

Объявить реализацию вычислительного контекста можно создав объект *Workflow* с подмешанными трейтами и переопределёнными методами,

либо воспользовавшись шорткатом, представляющим одну из рассмотренных структур (важно, что это не самостоятельные определения, а просто синонимы для подмешиваемого набора трейтов).

trait *Functor*[*F*[_]] **extends** *Mapping*[*F*]

trait *SemiIdiom*[*F*[_]] **extends** *Functor*[*F*] **with** *Applying*[*F*]

trait *Idiom*[*F*[_]] **extends** *SemiIdiom*[*F*] **with** *Pointing*[*F*] {
  def *map*[*A*, *B*](*f* : *A* ⇒ *B*) = *app*(*point*(*f*))
}

4.5.2. Алгоритм раскрытия выражений

Одно из важных отличий `scala-workflow` от похожих синтаксических расширений заключается в том, что оно всегда стремится использовать наименее мощный интерфейс реализации вычислительного контекста для сгенерированного кода. Это означает, в частности, что появляется возможность пользоваться идиоматическими скобками для функторов (например, для *Map*[*A*, *B*]).

Реализация `scala-workflow` основана на нетипизированных макросах. Макрос $ принимает в качестве аргумента синтаксическое дерево выражения на Scala. Это выражение обязано быть синтаксически корректным, но не обязано проходить проверку типов.

Алгоритм раскрытия выражений для контекста *Workflow*[*F*] начинает проверять типы всех подвыражений, начиная с наиболее вложенных. В зависимости от результата проверки:

1. Если проверка типов завершилась успешно и результирующий тип равен $F[T]$ для какого-то T, выражение заменяется на синтетический параметр x_n типа T и связывание x_n с реальным значением запоминается в отдельную таблицу.

2. Если проверка типов завершилась успешно и результирующий тип соответствует чистому значению (в заданном контексте), выражение оставляется как есть.

3. В противном случае, алгоритм разбивает выражение на составные части и пытается переписать их.

Алгоритм останавливается, когда переписанное выражение проходит проверку типов. На выходе остаётся таблица связываний синтетических параметров. Далее, все появившиеся в ней синтетические имена переменных генерируются в определения аргументов анонимных функций, а эффективные значения — в аргументы вызовов методов контекста.

Рассмотрим следующий пример.

$$
\begin{array}{l}
context(option)\,\{ \\
\quad \$(2 \times 3 + Some(10) \times Some(5)) \\
\}
\end{array}
$$

Все числа проходят проверку типов и результирующий тип равен Int, поэтому они оставляются как есть. 2×3 так же проходит проверку типов и оставляется как есть. $Some(10)$ и $Some(5)$ оба проходят проверку типов с результирующим типом $Option[Int]$ (вообще говоря, в данном случае они имеют тип $Some[Int]$, что тоже допустимо, поскольку проверка происходит не на равенство типов а на отношение подтипизации), поэтому в таблицу связываний синтетических параметров добавляется две записи.

Сгенерированный код выглядит следующим образом.

$$
\begin{array}{l}
option.app(\\
\quad option.map(\\
\quad\quad (x_1 : Int) \Rightarrow (x_2 : Int) \Rightarrow \\
\quad\quad\quad 2 \times 3 + x_1 \times x_2 \\
\quad)(Some(10)) \\
)(Some(5))
\end{array}
$$

Стоит отметить, что метод *point* генерируется только когда в скобки передано корректное чистое значение. Во всех остальных случаях будут сгенерированы вызовы *map* и *app*. В таблице 4.1 приводятся примеры $-выражений и результат их раскрытия.

$-выражение	**Раскрытое выражение**
$(42)	$option.point(42)$
$(Some(42) + 1)$	$option.map(\ (x_1 : Int) \Rightarrow\ x_1 + 1\ )(Some(42))$
$(Some(2) \times Some(3))$	$option.app(\ option.map(\ (x_1 : Int) \Rightarrow (x_2 : Int) \Rightarrow\ x_1 \times x_2\ )(Some(2))\ )(Some(3))$
$(Some(2) \times Some(3) + Some(5))$	$option.app(\ option.app(\ option.map(\ (x_1 : Int) \Rightarrow (x_2 : Int) \Rightarrow (x_3 : Int) \Rightarrow\ x_1 \times x_2 + x_3\ )(Some(2))\ )(Some(3))\ )(Some(5))$

Таблица 4.1. Примеры раскрытия $-выражений

4.5.3. Определение контекста

Вычислительный контекст задаётся с помощью макроса *context*, который принимает в качестве аргумента либо реализацию контекста, либо конструктор типа $F[\_]$, такой, что в области видимости implicit-значений содержится реализация такого контекста.

Так, следующие два примера эквивалентны. Значение *list* определено в качестве implicit-значения внутри `scala-workflow`.

$$
\begin{array}{l}
context[List]\ \{ \\
\quad \$(List(2,\ 5)\ \times\ List(3,\ 7)) \\
\}
\end{array}
$$

$$
\begin{array}{l}
context(list)\ \{ \\
\quad \$(List(2,\ 5)\ \times\ List(3,\ 7)) \\
\}
\end{array}
$$

Макрос $ использует контекст из наиболее близко определённого блока *context*. В качестве альтернативы можно указать конструктор типа, соответствующая реализация вычислительного контекста которого будет взята из области видимости implicit-значений.

$$\$[List](List(2,\ 5)\ \times\ List(3,\ 7))$$

При таком вызове, $ игнорирует все ограждающие блоки *context* и работает внутри контекста $Workflow[List]$.

Вложенные применения *context* и $ могут быть заменены специальным макросом *workflow*, принимающим в качестве аргумента либо реализацию контекста, либо конструктор типа, и переписывающий блок кода по тем же правилам, что и $.

$$\begin{aligned}&workflow(list)\ \{\\ &\quad List(2,\ 5)\ \times\ List(3,\ 7)\\ &\}\end{aligned}$$

Макрос *workflow* рассчитана на работу со сложными фрагментами кода, в то время как $ рассчитана на простые выражения.

Вместе с расширением поставляется набор встроенных реализаций контекста, определённых внутри трейтов *FunctorInstances*, *IdiomInstances* и *SemiIdiomInstances*. Все они подмешаны к объекту пакета *scala.workflow* и потому доступны при импорте *workflow._*. В качестве альтернативы можно импортировать только сами определения макросов

$$workflow.\{context,\ workflow,\ \$\}$$

и обращаться к реализациям через объекты *Functor*, *SemiIdiom* и *Idiom*.

4.5.4. Композиция эффектов

Известно, что из произвольной пары функторов, полу-идиом и идиом можно построить композицию. Это следует из коммутативности операций над соответствующими алгебраическими структурами. `scala-workflow` определяет набор классов композиции (*FunctorCompose*, *SemiIdiomCompose*

и *IdiomCompose* соответственно) позволяющих, например, имея реализации $Idiom[F]$ и $Idiom[G]$, получить реализацию $Idiom[F[G]]$. Создать объект класса *IdiomCompose* можно либо явно с помощью конструктора класса, либо вызовом метода \$ класса *Idiom* (а по аналогии, и для других классов, поддерживающих композицию).

$context(list \$ option)$ {
  $\$(List(Some(2),\ Some(3),\ None)\ \times\ 10)\ ==$
    $List(Some(20),\ Some(30),\ None)$
}

С помощью этого же синтаксиса возможно комбинировать контексты различных классов, при этом результирующий контекст будет реализовывать более слабый интерфейс из переданных двух. Например, объект $map[String]$ \$ *option* реализует *Functor*, поскольку из двух переданных контекстов, *Functor* у $map[String]$ и *Idiom* у *option* первый слабее.

Композиция контекстов некоммутативна, *list* \$ *option* и *option* \$ *list* задают два различных контекста и значения типа $List[Option[T]]$ будет считаться чистым во втором случае.

$context(option \$ list)$ {
  $\$(Some(List(1,\ 2,\ 3))\ \times\ 10)\ ==\ Some(List(10,\ 20,\ 30))$
}

4.6. Приложения

4.6.1. Интерпретатор с обработкой ошибок

Следуя примеру из [36] рассмотрим, как `scala-workflow` упрощает построение интерпретатора с обработкой ошибок. Определение абстрактного синтаксиса языка с целыми числами, связываниями переменных и операцией сложения выглядит следующим образом.

sealed trait *Expr*
case class $Var(id : String)$ **extends** *Expr*
case class $Val(value : Int)$ **extends** *Expr*

```
case class Add(lhs : Expr, rhs : Expr) extends Expr
```

Значения переменных хранятся в окружении, заданном типом *Env*, фактически являющимся хеш-таблицей. Обращение к окружению происходит с помощью функции *fetch*, возвращающей либо значение переменной, либо *None*.

```
type Env = Map[String, Int]
def fetch(x : String)(env : Env) : Option[Int] = env.get(x)
```

Интерпретатор представляет собой функцию, принимающую выражение, объект окружения и возвращающую либо результат вычисления, либо его отсутствие, в случае, когда одна из использованных внутри выражения переменных не задана в окружении. Ниже приводится её реализация с помощью **for**-нотации.

```
def eval(expr : Expr)(env : Env) : Option[Int] =
  expr match {
    case Var(x) ⇒ fetch(x)(env)
    case Val(value) ⇒ Some(value)
    case Add(x, y) ⇒ for {
      lhs ← eval(x)(env)
      rhs ← eval(y)(env)
    } yield lhs + rhs
  }
```

Следует отметить, что при таком подходе приходится явно передавать объект окружения в рекурсивный вызов и явно именовать проточные вычисления при интерпретации сложения. Реализация может быть значительно упрощена, будучи помещённой в вычислительный контекст вида *Workflow*[*Env* ⇒ *Option*[_]], который может быть построен либо вручную, либо композицией *Workflow*[*Env* ⇒ _] и *Workflow*[*Option*].

```
def eval : Expr ⇒ Env ⇒ Option[Int] =
  context(function[Env] $ option) {
    case Var(x) ⇒ fetch(x)
    case Val(value) ⇒ $(value)
```

$$\textbf{case}\ Add(x,\ y)\ \Rightarrow\ \$(eval(x)\ +\ eval(y))$$
$$\}$$

4.6.2. Недетерминированные вычисления

Недетерминированные значения традиционно представляются с помощью списков. Функцию, возвращающую список, можно интерпретировать как функцию, возвращающую суперпозицию из нескольких результатов вычисления.

$$\textbf{val}\ xs\ =\ List(1,\ 2,\ 3)$$
$$\textbf{val}\ ys\ =\ List(4,\ 5)$$
$$\textbf{val}\ zs\ =\ List(6,\ 7)$$

Поместив блок кода в контекст $Workflow[List]$ можно прозрачно работать с недетерминированными значениями с помощью $-нотации.

Применение чистой функции к недетерминированному значению должно вызывать её для каждого возможного результата вычисления и собирать полученные значения обратно в список.

$$\$(xs\ \times\ 2)\ ==\ List(2,\ 4,\ 6)$$

Вычисления, задействующие два и более недетерминированных значения должны брать комбинации из всех возможных значений и помещать в список все полученные результаты.

$$\$(xs\ \times\ ys)\ ==\ List(4,\ 5,\ 8,\ 10,\ 12,\ 15)$$
$$\$(xs\ \times\ ys\ +\ zs)\ ==\ List(10,\ 11,\ 11,\ 12,\ 14,\ 15,\ 16,\ 17,\ 18,\ 19,\ 21,\ 22)$$

4.6.3. Асинхронное программирование

Многие языки программирования предоставляют интерфейс *Future*, предназначенный для организации асинхронных вычислений. Так, запуск вычисления объекта *Future* не блокирует поток исполнения, а начинает выполнять код в соседнем потоке, позволяя позже заблокироваться для получения результата. В Scala класс *Future* имеет аппликативный интерфейс

(а как будет показано далее, и монадический), а потому его можно использовать для асинхронного программирования в идиоматических скобках.

Рассмотрим функцию *slowly*, которая блокирует исполнение на секунду, после чего вычисляет свой ленивый аргумент.

```
def slowly[A](proc :⇒ A) = {
  Thread.sleep(1000)
  proc
}
```

Идиоматические скобки контекста *Future* позволяют организовывать вычисления в будущем.

```
context[Future] {
  val x = $(slowly(2 × 3))
  val y = $(slowly(4 − 1))
  $(x × y)
}
```

Переданное в скобки чистое значение (такое как $slowly(2 \times 3)$) начинает исполняться асинхронно, а несколько скомбинированных асинхронных значений порождают новое асинхронное значение, ждущее результаты выполнения всех своих аргументов. Приведённый выше код сгенерирует асинхронное значение, результат которого будет доступен примерно спустя секунду после запуска, значения x и y будут выполняться параллельно.

4.6.4. $\mathbb{SKI}$-исчисление и бесточечная нотация

У реализации экземпляра идиомы для функции ($Idiom[A \Rightarrow \_]$[1]) есть примечательное свойство. Его методы *point* и *app* являются, соответственно, комбинаторами $\mathbb{K}$ и $\mathbb{S}$ в $\mathbb{SKI}$-исчислении [16]. Это означает, что с помощью композиции этих двух методов можно построить любой замкнутый λ-терм (иными словами, любую функцию). Так, реализация $\mathbb{I}$-комбинатора выглядит как

[1]Здесь и далее подобная запись выбрана для наглядности и не является корректным кодом на Scala. Правильная запись использует *типовую λ-функцию* и в данном случае имеет вид $Idiom[(\{\textbf{type } \lambda[\alpha] = A \Rightarrow \alpha\})\#\lambda]$.

$$\textbf{def}\ \ id[T]\ =\ function[T].app(\\ \quad function[T\ \Rightarrow\ T].point\\ )(function[T].point)$$

А 𝔹-комбинатор (соответствующий композиции функций) выражается в виде

$$\textbf{def}\ \ b[A,\ B,\ C]\ =\ function[A\ \Rightarrow\ B].app(\\ \quad function[A\ \Rightarrow\ B].point(function[C].app[A,\ B])\\ )(function[C].point)$$

У этой идиомы есть и более практическое применение. Она может быть использована для построения функций в бесточечном стиле (англ. англ. “point-free notation”), т.е. без указания аргументов функции.

В качестве примера, рассмотрим идиому $function[Char]$ и функцию $isLetterOrDigit$, определяющую, является символ буквой или цифрой. Имея в распоряжении пару вспомогательных функций

$$\textbf{val}\ \ isLetter : Char\ \Rightarrow\ Boolean\ =\ \_.isLetter\\ \textbf{val}\ \ isDigit :\ \ Char\ \Rightarrow\ Boolean\ =\ \_.isDigit$$

в традиционном стиле её реализация выглядит как

$$\textbf{val}\ \ isLetterOrDigit\ =\ (ch : Char)\ \Rightarrow\ isLetter(ch)\ \ ||\ \ isDigit(ch)$$

В бесточечном стиле и с применением идиоматических скобок появляется возможность выразить её как

$$\textbf{val}\ \ isLetterOrDigit\ =\ \$(isLetter\ \ ||\ \ isDigit)$$

Методы $Function.compose$ и $Function.andThen$ могут быть использованы для построения более сложных функций. Так, работая в идиоме $function[Double]$ можно скомбинировать функции

$$\textbf{val}\ \ sqrt : Double\ \Rightarrow\ Double\ =\ x\ \Rightarrow\ math.sqrt(x)\\ \textbf{val}\ \ sqr :\ \ Double\ \Rightarrow\ Double\ =\ x\ \Rightarrow\ x\ \times\ x\\ \textbf{val}\ \ log :\ \ Double\ \Rightarrow\ Double\ =\ x\ \Rightarrow\ math.log(x)$$

Следующие пары функций эквивалентны:

$$\textbf{val}\ f\ =\ (x : Double)\ \Rightarrow\ sqrt((sqr(x)\ -\ 1)\ \div\ (sqr(x)\ +\ 1))$$
$$\textbf{val}\ f\ =\ sqrt\ \ compose\ \ \$((sqr\ -\ 1)\ \div\ (sqr\ +\ 1))$$

$$\textbf{val}\ g\ =\ (x : Double)\ \Rightarrow\ (sqr(log(x))\ -\ 1)\ \div\ (sqr(log(x))\ +\ 1)$$
$$\textbf{val}\ g\ =\ log\ \ andThen\ \ \$((sqr\ -\ 1)\ \div\ (sqr\ +\ 1))$$

4.6.5. Аппликативные EDSL

Помимо синтаксической поддержки вычислений с эффектами, `scala-workflow` также находит приложение и в роли синтаксического фреймворка для *встраиваемых предметно-ориентированных языков* (англ. “embedded domain-specific language”, сокращённо EDSL). Как отмечалось ранее, аппликативные функторы являются не только способом описания эффектов, но и полезным формализмом для прикладного программирования.

В качестве примера рассмотрим построение небольшого встраиваемого языка для *реактивного программирования* [18] (англ. “functional reactive programming”). Парадигма реактивного программирования определяет потоки данных с распространением изменения состояния. Классическим примером реактивных вычислений являются электронные таблицы, где изменение значения одной из ячеек влечёт изменение и всех остальных ячеек, зависящих от ней. В контексте Scala реактивное программирование интересно, в частности, в качестве альтернативы шаблону проектирования Observer [34].

Интерфейс ячейки предоставляет доступ к значение типа T. Хранимое значение при этом может зависеть от значений других ячеек. Над ячейкой допустимы две операции: получение содержимого (возможно, сопровождающее пересчётом содержимого других ячеек) и присвоение значения.

$$\textbf{trait}\ Cell[T]\ \{$$
$$\quad \textbf{def}\ !\ :\ T$$
$$\quad \textbf{def}\ :=\ (value\ :\ T)$$
$$\}$$

Для ячейки можно объявить реализацию идиомы, идиоматические скобки которой будут позволять комбинировать чистые и реактивные вы-

числения. Фактически, метод *point* создаёт ячейку, хранящую значение и допускающее его изменение, а метод *app* создаёт ячейку, обращение к которой провоцирует обращение к значениям всех зависящих ячеек.

```
val frp = new Idiom[Cell] {
  def point[A](a : A) = new Cell[A] {
    private var value = a
    def := (a : A) { value = a }
    def ! = value
  }
  def app[A, B](f : Cell[A ⇒ B]) = a ⇒ new Cell[B] {
    def ! = f!(a!)
    def := (value : T) {
      throw new UnsupportedOperationException
    }
  }
}
```

Теперь при работе внутри контекста *Workflow*[*Cell*] появляется возможность все реактивные вычисления проводить в идиоматических скобках, а также прозрачно смешивать чистые и реактивные вычисления.

```
context(frp) {
  val a = $(10)
  val b = $(5)
  val c = $(a + b × 2)
  println(c!)
  b := 7
  println(c!)
}
```

Выведет 20 и 24.

Глава 5. Монады

5.1. Обзор

Монады представляют собой ещё один способ формализации эффектов. Термин *монада* был взят из теории категорий, а её связь с вычислениями показал Могги [40]. Вадлер описал [65, 67] применимость монад в функциональном программировании, а наибольшую известность они получили после того, как вошли в Haskell, в частности, в виде формализации подсистемы ввода-вывода [48].

Монада представляет собой структуру, состоящую из *монадической единицы* и *монадического связывания*. В Haskell её можно представить в виде класса типов с двумя методами.

class $Monad\ \mu$ **where**
  $return\ ::\ \alpha\ \to\ \mu\,\alpha$
  $(\gg\!=)\ \ ::\ \mu\,\alpha\ \to\ (\alpha\ \to\ \mu\,\beta)\ \to\ \mu\,\beta$

Аналогичное описание на Scala выглядит как

trait $Monad[M[\_]]$ {
  def $point[A](a:\ A) : M[A]$
  def $bind[A,\ B](f : A\ \Rightarrow\ M[B]) : M[A]\ \Rightarrow\ M[B]$
}

Нетрудно показать, что монада является частным случаем аппликативного функтора (а соответственно, и обычного функтора). Монадическая единица является оператором *pure*, а оператор $\circledast$ можно выразить в виде $f\ \circledast\ x\ =\ x\ \gg\!=\ (\lambda\,a\ \to\ f\ \gg\!=\ (\lambda\,g\ \to\ return\,(g\ a)))$.

5.2. Приложения в языках программирования

Монада в Haskell представлена классом типов, концептуально схожим с описанным в предыдущем параграфе. В ней также доступен оператор $\gg$ с типом $\mu\,\alpha\ \to\ \mu\,\beta\ \to\ \mu\,\beta$, представляющий собой аналог $\gg\!=$ для вырожденного случая второго аргумента, не зависящего от результата предыдущего вычисления.

Scala не предоставляет специального интерфейса, описывающего монаду, но содержит ряд соглашений по заданию методов, образующих монадическую структуру. Этими методами являются *map* (соответствующий *fmap* в Haskell), *flatMap* (соответствующий $\gg=$) и *filter*, используемый для фильтрации содержимого монады по условию.

Как было сказано ранее, монады являются частным случаем аппликативного функтора. Примером реализации аппликативного функтора, не являющегося монадой, является упомянутый в параграфе 4.2 *zipList*.

Тем не менее, большая часть стандартных конструкторов типа обладают реализацией монады. Вновь рассмотрим тип, представляющий наличие или отсутствие значения.

```
instance Monad Maybe where
  return = Just
  (Just a) >>= f = f a
  _        >>= _ = Nothing
```

Его аналог в Scala выглядит как

```
val option = new Monad[Option] {
  def point[A](a : A) = Some(a)
  def bind[A, B](f : A => Option[B]) = {
      case Some(a) => f(a)
      case _ => None
    }
}
```

В качестве примера использования рассмотрим функцию безопасного деления действительных чисел.

```
divide :: Double -> Maybe Double
divide _ 0 = Nothing
divide x y = Just (x / y)
```

С помощью монады для *Maybe* можно строить сложные вычисления с эффектами. Так, функция, безопасно вычисляющая значение выражения $1 \div (1 \div x + 1 \div y)$ для аргументов x и y может быть построена как

$$
\begin{aligned}
\mathit{complexDivide}\ x\ y\ = {} & \mathit{divide}(1,\ x) \gg\!= \lambda\, x' \ \rightarrow \\
& \mathit{divide}(1,\ y) \gg\!= \lambda\, y' \ \rightarrow \\
& \mathit{divide}(1,\ x' + y')
\end{aligned}
$$

Все вычисления с использованием аппликативных функторов можно переформулировать для монад. Так, пример со сложением чисел из параграфа 4.2 можно переписать в виде

$$
\mathit{Just}\ 3 \gg\!= \lambda\, x \ \rightarrow\ \mathit{Just}\ 2 \gg\!= \lambda\, y \ \rightarrow\ \mathit{Just}\ 5 \gg\!= \mathit{return}\ \$\ \mathit{add}\ x\ y\ z
$$

Необходимо отметить отличия в выразительной силе монад и идиом.

1. Монады вводят нотацию зависимости одного вычисления с эффектами от другого. В предыдущем примере для результатов промежуточных вычислений пришлось вводить имена x, y, z, то время как в аппликативном контексте явного именования не потребовалось, т.ч. в этом смысле монады оказываются более многословными, чем требует постановка задачи. С другой стороны, нотация зависимости между вычислениями позволяет организовывать вычисления, невыразимые в аппликативном контексте, как в определении функции *complexDivide*.

2. В отличии от функторов и идиом, монады не композируются. Иными словами, имея в общем случае экземпляры монад $\mathit{Monad}\ F$ и $\mathit{Monad}\ G$ для конструкторов F и G, нельзя построить экземпляр $\mathit{Monad}\ (F \circ G)$. Построить композицию [29] для двух конкретных экземпляров монад возможно, объявив *трансформер монад* [31] (англ. “monad transformer”). Также стоит отметить, что монады некоммутативны, т.е. $\mathit{Monad}\ (F \circ G)$ и $\mathit{Monad}\ (G \circ F)$ являются двумя разными монадами.

3. При переходе от чистых вычислений к монадическим требуется явно задавать порядок вычислений. Использование стиля *вызова по значению* (англ. “call-by-value”) или *вызова по имени* (англ. “call-by-name”) может приводить к различиям в семантике программы [45]). В аппликативных вычислениях эти различия отсутствуют.

5.3. Теоретико-категорные основания

Определение. *Монадой* в категории $\mathcal{C}$ называется эндофунктор $\mathbf{T}$ и пара естественных преобразований:

1. $\eta : 1_{\mathcal{C}} \dot{\to} \mathbf{T}$, где $1_{\mathcal{C}}$ — тождественный функтор категории $\mathcal{C}$
2. $\mu : \mathbf{T}^2 \dot{\to} \mathbf{T}$, где $\mathbf{T}^2$ — композиция эндофункторов категории $\mathcal{C}$

таких, что диаграммы на рисунке 5.1 (определяющему, что $\mu_X \circ \mathbf{T}(\mu_X) = \mu_X \circ \mu_{\mathbf{T}(X)}$) и 5.2 (определяющему, что $\mu_X \circ \mathbf{T}(\eta_X) = \mu_X \circ \eta_{\mathbf{T}(X)} = 1_{\mathbf{T}}$) коммутируют.

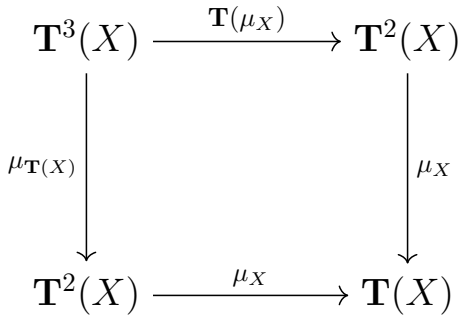

Рис. 5.1. Коммутативная диаграмма естественного преобразования η_X.

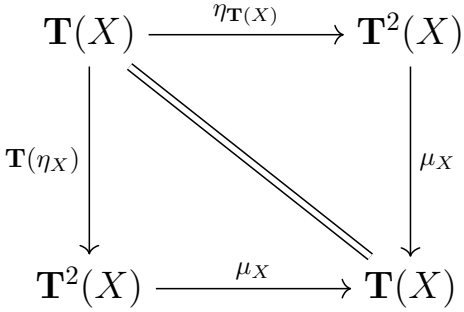

Рис. 5.2. Коммутативная диаграмма естественного преобразования μ_X.

«Категорному» определение монады соответствует следующее определение в Haskell:

```
class Monad μ where
  fmap   :: (α → β) → μ α → μ β
  return :: α → μ α
  join   :: μ (μ α) → μ α
```

Здесь метод *fmap*, фактически, задаёт функтор, *return* задаёт естественное преобразование η, а *join* задаёт естественное преобразование μ. Несложно показать, что «категорное» определение эквивалентно «прикладному». Последнее используется, т.к. более удобно на практике. Действительно, $join\ x\ =\ x\ \gg\!=\ id$, а $x\ \gg\!=\ f\ =\ join\ (fmap\ f\ x)$.

Реализация монады должна удовлетворять нескольким законам.

1. Закон левого тождества.
   $return\ a\ \gg\!=\ f\ \equiv\ f\ a$

2. Закон правого тождества.
   $m\ \gg\!=\ return\ \equiv\ m$

3. Закон ассоциативности.
   $(m\ \gg\!=\ f)\ \gg\!=\ g\ \equiv\ m\ \gg\!=\ (\lambda\, x\ \to\ f\ x\ \gg\!=\ g)$

Более интуитивный набор аксиом можно получить, введя операцию *композиции Клейсли*.

$$(\gg\!\!>)\ ::\ (\alpha\ \to\ \mu\,\beta)\ \to\ (\beta\ \to\ \mu\,\gamma)\ \to\ \alpha\ \to\ \mu\,\gamma$$
$$f\ \gg\!\!>\ g\ =\ \lambda\, x\ \to\ f\ x\ \gg\!=\ g$$

Заменив монадическое связывание на композицию Клейсли получим следующий набор законов.

1. *return* нейтрален слева.
   $return\ \gg\!\!>\ f\ \equiv\ f$

2. *return* нейтрален справа.
   $f\ \gg\!\!>\ return\ \equiv\ f$

3. Композиции Клейсли ассоциативна.
   $(f\ \gg\!\!>\ g)\ \gg\!\!>\ h\ \equiv\ f\ \gg\!\!>\ (g\ \gg\!\!>\ h)$

Теперь явно видна моноидальная структура монады — ассоциативной операцией является композиция Клейсли, а нейтральным элементом — монадическая единица. Фактически, монада является моноидальным объектом в *категории Клейсли*, состоящей из функций вида $\alpha\ \to\ \mu\,\beta$.

5.4. Синтаксические расширения

Заняв заметное место в языках программирования, монады получили и более развитый синтаксис в языках программирования. Рассмотрим некоторые из них.

5.4.1. Haskell

Haskell предоставляет два расширения для монад. Первое, *выделение монад* [64] (англ. “monad comprehension”) обобщает синтаксис *выделения списков* (англ. “list comprehension”). Оно становится доступно при включении опции $\{-\#\ LANGUAGE\ MonadComprehensions\ \#-\}$. С ним, выражения в квадратных скобках, используемые для построения списков

$$[\, x\ +\ y \mid x\ \leftarrow\ [1,\ 2,\ 3],\ y\ \leftarrow\ [4,\ 5,\ 6]\,]$$

могут быть обобщены для произвольной монады, такой как *Maybe a*

$$[\, x\ +\ y \mid x\ \leftarrow\ \mathit{Just}\ 2,\ y\ \leftarrow\ \mathit{Just}\ 3\,]$$

Правила раскрытия выражений подробно описаны в [19]. Помимо связывания эффективных значений поддерживается группировка по условию и «застёгивание» (англ. “zipping”) выражений. Приведённое выше выражение раскрывается в

$$\begin{aligned}&\mathit{Just}\ 2\ \gg\!=\ \lambda\, x\ \rightarrow\\ &\quad \mathit{Just}\ 3\ \gg\!=\ \lambda\, y\ \rightarrow\\ &\qquad \mathit{return}\ (x\ +\ y)\end{aligned}$$

Второе расширение носит название **do**-нотации. Последовательность из связываний в виде стрелки влево раскрывается во вложенный вызов функции $\gg\!=$. Так, выражение вычисление суммы комбинаций из всех элементов двух списков выглядит как

$$\begin{aligned}&\textbf{do}\\ &\quad x\ \leftarrow\ [1,\ 2,\ 3]\\ &\quad y\ \leftarrow\ [4,\ 5,\ 6]\\ &\quad \mathit{return}\ (x\ +\ y)\end{aligned}$$

преобразовывается в

$$[1,\ 2,\ 3] \gg\!= \lambda\, x\ \rightarrow \\ \quad [4,\ 5,\ 6] \gg\!= \lambda\, y\ \rightarrow \\ \quad\quad \mathit{return}\ (x\ +\ y)$$

Допустимо не давать имени связыванию (в таком случае имя переменной и стрелка опускаются), что может быть удобно, если эффективное вычисление не производит результата и используется только для побочного эффекта.

Стоит отметить, что, в отличии от выделения монад, **do**-нотация не производит вызова *return*, в примере выше *return* $(x\ +\ y)$ является значением, возвращаемым из выражения.

5.4.2. Scala

Scala предоставляет расширение, известное как **for**-нотация. Основное отличие от **do**-нотации заключается в том, что она производит вызовы методов *map* и *flatMap* и потому ожидает чистое значение в качестве последнего выражения. Так, выражение вида

$$\textbf{for}\ \{ \\ \quad x\ \leftarrow\ \mathit{List}(1,\ 2,\ 3) \\ \quad y\ \leftarrow\ \mathit{List}(4,\ 5,\ 6) \\ \}\ \textbf{yield}\ x\ +\ y$$

транслируется в

$$\mathit{List}(1,\ 2,\ 3).\mathit{flatMap}(x\ \Rightarrow \\ \quad \mathit{List}(4,\ 5,\ 6).\mathit{map}(y\ \Rightarrow \\ \quad\quad x\ +\ y))$$

Также поддерживается фильтрация по условию, для чего генерируется вызов метода *filter*.

5.4.3. OCaml

Для OCaml [30] была разработана **perfrom**-нотация [15], близкая по синтаксису и реализации к рассмотренным нотациям в Haskell и Scala.

perform
  $x \longleftarrow [1; 2; 3];$
  $y \longleftarrow [4; 5; 6];$
  $return\ (x\ +\ y)$

раскрывается в

$$bind\ [1; 2; 3]\ (\textbf{fun}\ x\ \rightarrow\ bind\ [4; 5; 6]\ (\textbf{fun}\ y\ \rightarrow\ return\ (x\ +\ y)))$$

Стоит отметить, что, по аналогии с идиоматическими скобками, ни для **do**, ни для **for**, ни для **perform** не нужна полноценная монада, достаточно полу-монады, вызов *return* или аналога нигде не генерируется.

5.4.4. F♯

F♯ предлагает более развитое синтаксическое средство для работы с монадами под названием *вычислительные выражения* [38] (англ. "computation expressions"). Суть его заключается в переписывании выражения вида *builderName* { *expression* }, где *builderName* соответствует имени построителя контекста (англ. "builder"), а *expression* — обыкновенный код на F♯, расширенный специальными синтаксическими конструкциями для связывания эффективных значений. Переписывание заключается в генерации вызовов методов построителя контекста.

Построитель контекста представляет собой объект, реализующий методы некоторого интерфейса. Обязательными методами являются *Return* и *Bind*, соответствующие монадической единице и монадического связыванию, а необязательными — ряд методов, соответствующих элементам языка, таким как циклы **for** и **while**, обработка исключений и конструкция **if** без ветви **else**.

Внутри переписываемого выражения различаются связывания чистых значений конструкциями **let**, **do**, **yield** и **use** и эффективных значений конструкциями **let!**, **do!**, **yield!** и **use!**.

В качестве примера рассмотрим работу с построителем контекста для типа $'a\ option$. Его интерфейс выглядит как

```
type OptionBuilder =
    new : unit → OptionBuilder
    member Zero : unit → 'a option
    member Bind : 'a option * ('a → 'b option) → 'b option
    member Return : 'a → 'a option
    member ReturnFrom : 'a option → 'a option
```

А реализация как

```
type OptionBuilder() =
    member b.Zero() = None
    member b.Bind(x, f) =
        match x with
            | Some x → f x
            | None → None
    member b.Return x = Some x
    member b.ReturnFrom x = x : _ option
```

Теперь объявив *opt* объектом построителя, можно пользоваться им как конструкцией языка.

```
let opt = OptionBuilder()
```

Будем рассматривать приложение, запрашивающее у пользователя простое число. В случае, если введённое число простое (что проверяется вхождением в заданный извне список *primes*), будет выведен его порядковый номер в списке, в противном случае будет выведено сообщение об ошибке.

Функция ввода простого числа.

```
let inputInt32() = opt {
    let str = Console.ReadLine()
    let success, value = Int32.TryParse str
    if success then return value
}
```

Функция нахождения номера простого числа.

```
let tryInputPrime() = opt {
    printfn "Enter prime number :"
    let! prime = inputInt32()
    let! index = List.tryFindIndex ((=) prime) primes
    return prime, index + 1
}
```

Наконец, интерактивный диалог с пользователем.

```
match tryInputPrime() with
    | Some(prime, index) →
        printfn "You did enter prime number %d (%d)" prime index
    | None →
        printfn "The number you entered wasn't prime"
```

Также для вычислительных выражений добавляется расширение, названное *джойнады* [47] (англ. “joinads”), вводящее конструкцию **match!**, расширяющее нотацию сопоставления с образцом и используемое для параллельного и асинхронного программирования.

5.5. Поддержка монад в `scala-workflow`

Как было показано ранее, монады является частным случаем аппликативных функторов, добавляющими возможность специфицировать порядок вычислений с эффектами.

В качестве примера рассмотрим безопасную функцию деления вещественных чисел.

```
def divide(x : Double, y : Double) = if (y == 0) None else Some(x / y)
```

Выражение $(*divide*(1, *divide*(2, 3))) нельзя раскрыть с помощью одних только идиом, поскольку для того, чтобы вычислить результат вызова внешнего *divide*, необходимо вычислить результат вызова вложенного *divide*. Для поддержки вложенных вычислений с эффектами необходимо расширить `scala-workflow` поддержкой монадических вычислений.

Эта поддержка монад сводится к добавлению новой операции для *Workflow*, расширению алгоритма раскрытия выражений и, как следствие, расширению поддерживаемого внутри $ подмножества Scala.

5.5.1. Расширенная иерархия вычислительных контекстов

В иерархию вычислительных контекстов добавляется трейт *Binding*, содержащий метод *bind*, соответствующий монадическому связыванию.

```
trait Binding[F[_]] extends Workflow[F] {
  def bind[A, B](f : A ⇒ F[B]) : F[A] ⇒ F[B]
}
```

Также добавляются новые синонимы, соответствующие монаде и полумонаде. По аналогии с *Idiom*, для монады можно реализовать метод *app* с помощью *point* и *bind*.

```
trait SemiMonad[F[_]] extends SemiIdiom[F] with Binding[F]

trait Monad[F[_]] extends Idiom[F] with Binding[F] {
  def app[A, B](f : F[A ⇒ B]) =
    bind(a ⇒ bind((g : A ⇒ B) ⇒ point(g(a)))(f))
}
```

Поскольку монады не композируются, метода *compose*, как у прочих структур, у них нет. Возможность объявлять трансформеры монад и составлять композицию с их помощью, конечно, остаётся.

5.5.2. Расширенный алгоритм раскрытия выражений

Алгоритм раскрытия выражений остаётся, по существу, тем же, но добавляется специальный анализ зависимостей между эффективными значениями. В случае, когда одно подвыражение зависит от другого, будет сгенерирован вызов *bind*, а не *app*. В Таблице 5.5.2 показаны $-выражения, производящие вызов *bind*.

Корректная обработка зависящих друг от друга эффективных вычислений позволяет добавить в поддерживаемое внутри идиоматических скобок подмножество Scala **val**-объявления. Таблица 5.5.2 показывает, как будут переписаны некоторые из таких объявлений.

$-выражение	**Раскрытое выражение**

$ {
  val $x = Some(10)$
  $x + 2$
}
	$option.map($
	  $(x : Int) \Rightarrow$
	  $x + 2$
	$)(Some(10))$

$ {
  val $x = Some(10)$
  val $y = Some(5)$
  $x + y$
}
	$option.bind($
	  $(x : Int) \Rightarrow$
	  $option.map($
	  $(y : Int) \Rightarrow$
	  $x + y$
	  $)(Some(5))$
	$)(Some(10))$

$ {
  val $x = Some(10)$
  val $y = x - 3$
  $x \times y$
}
	$option.map($
	  $(x : Int) \Rightarrow$
	  val $y = x - 3$
	  $x \times y$
	$)(Some(10))$

$ {
  val $x = Some(10)$
  $divide(x, 2)$
}
	$option.bind($
	  $(x : Int) \Rightarrow$
	  $divide(x, 2)$
	$)(Some(10))$

$ {
  val $x = Some(10)$
  $x \times Some(2) + Some(5)$
}
	$option.bind($
	  $(x : Int) \Rightarrow$
	  $option.app($
	  $option.map($
	  $(x_1 : Int) \Rightarrow (x_2 : Int) \Rightarrow$
	  $x \times x_1 + x_2$
	  $)(Some(2))$
	  $)(Some(5))$
	$)(Some(10))$

$(2 + {
  val $x = Some(10)$
  $x \times 2$
})
	$option.map($
	  $(x_1 : Int) \Rightarrow$
	  $2 + x_1$
	$)(option.map($
	  $(x_2 : Int) \Rightarrow$
	  $x_2 \times 2$
	$)(Some(10)))$

Таблица 5.1. Раскрытие $-выражений, содержащих `val`-объявления.

$-выражение	**Раскрытое выражение**
$\$(divide(1, 2))$	$divide(1, 2)$
$\$(divide(Some(1.5), 2))$	$option.bind($ $\quad(x_1 : Double) \Rightarrow$ $\quad\quad divide(x_1, 2)$ $)(Some(1.5))$
$\$(divide(Some(1.5), Some(2)))$	$option.bind($ $\quad(x_1 : Double) \Rightarrow$ $\quad\quad option.bind($ $\quad\quad\quad(x_2 : Int) \Rightarrow$ $\quad\quad\quad\quad divide(x_1, x_2)$ $\quad\quad)(Some(2))$ $)(Some(1.5))$
$\$(divide(Some(1.5), 2) + 1)$	$option.bind($ $\quad(x_1 : Double) \Rightarrow$ $\quad\quad option.map($ $\quad\quad\quad(x_2 : Double) \Rightarrow$ $\quad\quad\quad\quad x_2 + 1$ $\quad\quad)(divide(x_1, 2))$ $)(Some(1.5))$

Таблица 5.2. Раскрытие $-выражений, производящие вызов *bind*.

5.6. Приложения

5.6.1. Вычисления с ошибками

Scala не поддерживает проверяемые исключения, т.е. в сигнатуре функции нельзя указать выбрасываемые ей исключения. Фактически, информация о вычислительном эффекте игнорируется компилятором и доступна только во время выполнения. Начиная с версии 2.10 в Scala доступна специальная монада `scala.util.Try`, которая позволяет обернуть чистое значение в аннотацию возможно выброшенного в процессе работы исключения. Конструктор *Try* позволяет превратить вычисление в эффективное значение.

При этом переход от стиля программирования с исключениями на стиль программирования с *Try* сопровождается серьёзными изменениями в синтаксисе, избежать которых помогает `scala-workflow`.

Для примера рассмотрим просто код, получающий XML-документ

по сети. Его прямая реализация с использованием исключений выглядит следующим образом.

```
def fetchXml(address : String) : Elem = {
  val url = new URL(address)
  val source = io.Source.fromURL(url)
  val contents = source mkString ""
  XML.fromString(contents)
}
```

Система исключений инкапсулирует по-крайней мере три ошибочных ситуации в этом коде. Во-первых, *address* может оказаться невалидным URL, во-вторых попытка сделать запрос по сети может закончиться неудачей, в-третьих строка *contents* может оказаться некорректным XML-документом. При этом в тип функции никак не сообщается, что она на этапе выполнения она может произвести исключение.

Закодировав результат функции в типе *Try*[*Elem*] можно воспользоваться **for**-синтаксисом, чтобы переписать реализацию функции.

```
def fetchXml(address : String) : Try[Elem] = for {
  url ← Try(new URL(address))
  source ← Try(io.Source.fromURL(url))
  contents = source mkString ""
  xml ← Try(XML.fromString(contents))
} yield xml
```

Помимо оборачивания вычислений, выбрасывающих исключения, в конструктор *Try*, пришлось изменить структуру самой функции. Благодаря `scala-worflow`, этот же код можно записать как

```
def fetchXml(address : String) : Try[Elem] = workflow[Try] {
  val url = Try(new URL(address))
  val source = Try(io.Source.fromURL(url))
  val contents = source mkString ""
  Try(XML.fromString(contents))
}
```

Отличие от кода, не кодирующего эффекты в типе минимально — добавилась лишь аннотация вычислительного контекста.

5.6.2. Вычисления с накапливаемым выводом

Рассмотрим реализацию монады *Writer*, используемой для вычислений с накапливаемым побочным эффектом. Результатом таких вычислений становится пара из вычисленного значения и накопленного эффекта. Формализуем его в виде класса *Writer*.

```
case class Writer[R, O](result : R, output : O)
```

Накапливаемый результат должен реализовывать интерфейс моноида. Нейтральный элемент моноида соответствует отсутствующему эффекту вычисления, а ассоциативная операция позволяет слить два эффекта в один. Следующее определение формализует моноид в Scala.

```
trait Monoid[A] {
  def empty : A
  def append : (A, A) ⇒ A
}
```

Теперь можно объявить реализацию монады *Writer*.

```
implicit def writer[O : Monoid] = new Monad[Writer[, O]] {
  private val monoid = implicitly[Monoid[O]]
  def point[A](a : A) = Writer(a, monoid.empty)
  def bind[A, B](f : A ⇒ Writer[B, O]) = {
    case Writer(a, o) ⇒
      val Writer(b, o2) = f(a)
      Writer(b, monoid.append(o, o2))
  }
}
```

В качестве накапливаемого вывода рассмотрим тривиальный пример лог-файла, состоящего из списка строк, представляющих записи в логе.

```
case class Log(entries : List[String])
```

Реализация моноида для лога, фактически, опирается на реализацию моноида для списка.

```
implicit val logMonoid = new Monoid[Log] {
  def empty = new Log(Nil)
  def append = {
    case (Log(oldentries), Log(newentries)) ⇒
      Log(oldentries ++ newentries)
  }
}
```

В качестве эффективной функции объявим *log*, возвращающую экземпляр *Writer* с отсутствующим значением и логом, состоящим из одной записи.

```
def log(message : String) =
  Writer[Unit, Log]({}, new Log(List(message)))
```

Теперь можно строить вычисления внутри контекста *Writer*[_, *Log*], реализация монады разделит чистое значение и эффект.

```
val Writer(result, log) = workflow(writer[Log]) {
  log("Calculating the sum...")
  val sum = 2 + 2

  val product = 2 × 2

  log("Calculating the subtraction...")
  val subtraction = 10 − 7

  log("Calculating the division...")
  30 ÷ 5
}
```

Полученный код представляет собой чисто функциональную версию записи в лог файл. В его традиционном императивном аналоге, функция *log* производила бы неконтролируемый системой типов побочный эффект.

В этой же реализации результат вычисления доступен в переменной *result* и равен 6, а накопленный лог-файл оказывается в переменной *log*. При этом `scala-workflow` позволило не менять реализацию при переходе от императивной версии к чисто функциональной.

5.6.3. Асинхронное программирование

В параграфе 4.6.3 было упомянуто, что класс *Future* реализует монадический интерфейс. Рассмотрим отличия асинхронного программирования в монадическом контексте от аппликативного.

Перепишем функцию *slowly* так, чтобы она сразу возвращала *Future*.

$$
\begin{array}{l}
\textbf{def}\ slowly[A](proc : \Rightarrow\ A)\ =\ Future\ \{ \\
\quad Thread.sleep(1000) \\
\quad proc \\
\}
\end{array}
$$

Рассмотрим выражение $\$(slowly(slowly(2)\ \times\ 2))$, допустимое в монадическом, но не аппликативном контексте. Ранее было отмечено, что одно из отличий между ними заключается в том, что монады поддерживают нотацию зависимости между результатами промежуточных вычислений. Так и в этом случае, внешний вызов *slowly* сможет начать вычислений только когда будет известен результат вызова внутреннего, а значит и вычисление всего выражение займёт две секунды.

Перепишем код из параграфа 4.6.3 в монадическом контексте.

$$
\begin{array}{l}
workflow[Future]\ \{ \\
\quad \textbf{val}\ x\ =\ slowly(2\ \times\ 3) \\
\quad \textbf{val}\ y\ =\ slowly(4\ -\ 1) \\
\quad x\ \times\ y \\
\}
\end{array}
$$

Запустив код, можно убедиться, что и его выполнение теперь занимает не одну секунду, а две, фактически, каждое выражение стало блокирующим! Это связано с тем, что блоки с **val** всегда раскрываются в вызовы методов **bind**, даже в случае, когда зависимостей между связываниями эффективных значений нет.

5.6.4. Монадические EDSL

По аналогии с EDSL, построенными на основе аппликативных функторов, для формализации вычислений можно использовать и монады. Упомянутая ранее возможность монад производить вычисления, зависящие от результата друг друга позволяет описывать произвольный поток управления для встраиваемого языка, что значительно повышает их выразительную силу.

Рассмотрим пример встраиваемого в Scala стекового языка программирования. Его реализация основана на вычислении в монаде *State*, причём поток выполнения зависит от состояния стека и вычисление может завершиться досрочно в случае возникновения ошибки обращения к стеку. `scala-workflow` позволяет построить конкатенативный синтаксис, близкий к синтаксису аналогичных языков (таких как Forth [11] и Joy [62]), чего было бы невозможно добиться с использованием одних только **for**-выражений.

Будем представлять стек в виде обыкновенного неизменяемого списка, а результат выполнения программы как либо модифицированное состояние стека, либо сообщение об ошибке (такое как «stack underflow»).

$$\textbf{type}\ Stack\ =\ List[Int]$$
$$\textbf{type}\ Result\ =\ Either[String,\ Stack]$$

Интерпретация программы задействует монаду *state*, которая игнорирует результат выполнения каждой отдельной команды, но сохраняет состояние стека.

$$\textbf{val}\ stackLang\ =\ state[Result]$$
$$\textbf{def}\ execute(program:\ State[Unit,\ Result])\ =\ program.state(Right(Nil))$$

Операции над стеком являются функциями типа $Stack \Rightarrow Result$. Чтобы иметь возможность пользоваться ими внутри какого-нибудь вычислительного контекста, необходимо поднять их в монаду.

$$\textbf{def}\ command(f:\ Stack\ \Rightarrow\ Result)\ =$$
$$\quad State[Unit,\ Result](st\ \Rightarrow\ (\{\},\ right[String].bind(f)(st)))$$

Теперь, пользуясь функцией *command*, можно объявить несколько команд, работающих со стеком.

```
def put(value : Int) = command {
  case stack ⇒ Right(value :: stack)
}

def dup = command {
  case head :: tail ⇒ Right(head :: head :: tail)
  case _ ⇒ Left("Stack underflow while executing ‘dup‘")
}

def rot = command {
  case a :: b :: stack ⇒ Right(b :: a :: stack)
  case _ ⇒ Left("Stack underflow while executing ‘rot‘")
}

def sub = command {
  case a :: b :: stack ⇒ Right((b − a) :: stack)
  case _ ⇒ Left("Stack underflow while executing ‘sub‘")
}
```

Программой, написанной в конкатенативном стиле является последовательность вызовов операций над стеком внутри идиоматических скобок контекста *stackLang*.

```
context(stackLang) {
  val programA = $ { put(5); dup; put(7); rot; sub }
  execute(programA) == Right(List(2, 5))

  val programB = $ { put(5); dup; sub; rot; dup }
  execute(programB) == Left("Stack underflow while executing ‘rot‘")
}
```

Заключение

Результатом данной работы является расширение языка Scala под названием `scala-workflow`, реализованное с помощью средств метапрограммирования и предназначенное для упрощения синтаксиса вычислений с эффектами, формализованными функторами, аппликативными функторами и монадами. Реализация построена на основе нетипизированных макросов, на момент написания работы доступных в экспериментальной ветке компилятора Scala.

Отметим особенности, отличающие `scala-workflow` от аналогов и составляющих новизну работы:

1. Вычисления с эффектами представляют собой обыкновенный код на Scala, обёрнутый в аннотацию контекста, т.е. максимально приближенный к синтаксису чистых вычислений, в отличии от похожих расширений, вводящих для этой цели специальные синтаксические конструкции.

2. Описывается единообразный синтаксис для вычислений со всеми образующими иерархию формализациями эффектов, чего ранее не было представлено. Идиоматические скобки и **do**/**for**-нотация рассчитаны на работу только с аппликативными функторами и монадами соответственно.

3. Большая модульность. Не ожидается, что инстанс *Workflow* должен реализовывать интерфейс какой-то конкретной структуры, раскрытие выражение всегда происходит в вызовы наиболее слабого необходимого интерфейса, от реализации требуются только те методы, которые нужны в конкретном случае (в отличии, к примеру, от **do**-нотации, которая всегда подразумевает монаду, даже в случаях, когда описываемое ей вычисление может быть выражено и более слабой структурой). До некоторой степени это похоже на вычислительные выражения в F♯, с поправкой на то, что вычислительные выражения всё же ожидают интерфейс монады, расширенный некоторыми дополнительными методами, в то время как контекст `scala-workflow` может реализовывать и всего один самый базовый метод функтора.

Помимо упрощения вычислений с эффектами `scala-workflow` может использоваться и в качестве синтаксического фреймворка для построения аппликативных и монадических EDSL, что следует из двойственной природы рассматриваемых в работе формализаций эффектов.

Необходимо отметить недостатки разработанного решения:

1. Не всегда очевидно, какой код будет в итоге сгенерирован. В отличие от рассмотренных синтаксических расширений, `scala-workflow` не просто применяет правила синтаксического преобразования к выражениям, но рассматривает типы подвыражений, что затрудняет анализ.

2. В частности, не всегда очевидно, какой набор методов контекста необходим для раскрытия заданного участка кода. Это вполне очевидно для простых выражений, но не всегда для сложных.

3. Поскольку макросы, переписывающие выражение, принимают в общем случае некорректное с точки зрения системы типов Scala выражение, становится трудно корректно сообщить об ошибке типов, произошедшей во время раскрытия в случае, если исходное выражение было ошибочно составлено так, что не проходит проверку типов, будучи раскрытым. Текущая реализация не поддерживает полноценной обработки ошибок и этот вопрос, предполагается, будет решён в будущем.

4. Добавление нового нетривиального этапа увеличивает время компиляции (впрочем, ненамного).

5. Implicit-преобразования могут вмешиваться в работу раскрытия выражений, из-за чего подвыражения могут случайно проходить проверку типов. На текущий момент способов решения этой проблемы не найдено.

Дальнейшая работа

`scala-workflow` уже сейчас пригодно для использования и разработка будет продолжена как в теоретическом, так и в прикладном аспектах.

Расширение поддерживаемого множества языка. Scala — сложный язык с очень развитой семантикой. Помимо элементов чисто функциональных языков программирования, здесь присутствует изменяемое состояние, исключения и implicit-преобразования, вычисления с которыми могут потребовать добавления новых методов к *Workflow*. Такой подход очень близок к текущей реализации вычислительных выражений в F♯, где большая часть методов представляет собой поддержку особенностей языка.

Вывод контекста из типа. `scala-worflow` позволяет в большинстве случаев уменьшить количество boilerplate-кода. Уменьшить его ещё больше можно было бы, выводя контекст из типов подвыражений аргумента в простых случаях. Например, считать выражение $List(1,\ 2,\ 3)\ \times\ 2$ помещённым по умолчанию в контекст $Workflow[List]$ и иметь возможность писать $\$(List(1,\ 2,\ 3)\ \times\ 2)$ без указания контекста где-либо. Построение алгоритма вывода контекста связано с рядом проблем:

1. Неясно, как выбрать, по какому типовому аргументу абстрагировать контекст, если используется типовый конструктор от нескольких аргументов. Например, встретив выражение типа $Either[A,\ B]$ неясно, следует ли вывести контекст для $Either[A,\ \_]$ или для $Either[\_,\ B]$.

2. Неясно, какой выбрать подтип, если встреченный в выражении тип участвует в иерархии наследования. Например, сейчас выражение $Some(42)$ имеет тип $Some[Int]$, являющийся подтипом $Option[A]$. Необходимо построить алгоритм, способный вывести в таком случае контекст для *Option*, а не для *Some*.

Вывод контекста из интерфейса. Как упоминалось ранее, Scala не выделяет монаду в отдельный интерфейс, раскрытие **for**-выражения просто ожидает наличие методов *map*, *flatMap* и *filter* у объекта. Сейчас это приводит к тому, что для каждого типа, неявно поддерживающего монадический интерфейс, приходится писать тривиальную реализацию *Workflow*. Хорошей альтернативой стала бы возможность выводить кон-

текст полумонады из наличия методов *map* и *flatMap* в интерфейсе некоторой структуры.

Расширения системы типов. Алгоритм с переписыванием выражений до прохождения проверки типов весьма хрупок и ненадёжен. Хорошей альтернативой стала бы система типов, в которой правило применения типа расширено на случай применения к эффективному аргументу. Исследование подобной системы типов предполагает решение задач проверки типа (англ. “type checking problem”), синтеза типов (англ. “type synthesis problem”) и населённости типа (англ. “type inhabitance problem”), а также вопроса устойчивости (англ. “type soundness”) такой системы типов.

Прочие формализации эффектов. Большой интерес представляют альтернативные способы описания эффектов, рассмотренные в первой части работы. Необходимо понять, существует ли более естественный способ отображения элементов языка в композицию операций над какой-либо алгебраической структурой, формализующей эффективное значение. Это могло бы позволить точнее гранулировать мощь вычислительного контекста и дать возможность для некоторых контейнеров использовать синтаксис, невозможный при текущих правилах отображения.

Список литературы

[1] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, number 6940, pages 297–311. 2010.

[2] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Relative monads formalised. *Journal of Formalized Reasoning*, (6940), 2010.

[3] Jean-Pierre and Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, number section 3, pages 162–173, 1992.

[4] Douglas M. Auclair. MonadPlus: What a Super Monad! *The Monad. Reader Issue 11*, page 39, 2008.

[5] Eric Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 00:1–36, 1996.

[6] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall International Series in Computer Science. Prentice Hall, New York, 1995.

[7] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *arXiv preprint arXiv:1203.1539*, pages 1–25, 2012.

[8] Edwin Brady. Programming in Idris: a tutorial. *http://www.cs.st-andrews.ac.uk/~eb/writings/idris-tutorial.pdf*, 2013.

[9] Edwin C. Brady. Idris, a language with dependent types. *IFL 2008*, 2008.

[10] Edwin C. Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. pages 1–12, 2013.

[11] Leo Brodie. *Thinking Forth*. Punchy Pub, 2004.

[12] T.H. Brus, Marko C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer. Clean — a language for functional graph rewriting. In

Functional Programming Languages and Computer Architecture, pages 364–384, 1987.

[13] Eugene Burmako. Scala Macros: Let Our Powers Combine! *Scala Days*, 2013.

[14] Eugene Burmako and Martin Odersky. Scala Macros, a Technical Report. *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[15] Jacques Carette, Lydia E. van Dijk, and Oleg Kiselyov. Syntax extension for monads in OCaml. *Software package available at http://www.cas.mcmaster.ca/~carette/pa_monad*, 2008.

[16] Haskell Brooks Curry, Robert Feys, William Craig, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972.

[17] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL'06*, pages 206–217, 2006.

[18] Conal M. Elliott. Push-pull functional reactive programming. *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, 2009.

[19] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing back monad comprehensions. *Proceedings of the 4th ACM symposium on Haskell - Haskell '11*, page 13, 2011.

[20] James Gosling. *The Java Language Specification*. Addison-Wesley Professional, 2000.

[21] Ralf Hinze. Lifting Operators and Laws. 2010.

[22] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, and Others. Report on the programming

language Haskell: a non-strict, purely functional language version 1.2. *ACM SIGPLAN Notices*, 27(5):1–164, 1992.

[23] John Hughes. Generalising Monads to Arrows. *Science of computer programming*, 2000.

[24] Nathan Jacobson. *Basic algebra I.* Courier Dover Publications, 2012.

[25] Trevor Jim, Greg Morrisett, and Dan Grossman. Cyclone: A safe dialect of C. *USENIX Annual Technical Conference*, 90, 2002.

[26] Mark Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61, 1993.

[27] Simon Peyton Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93. Citeseer, 1993.

[28] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, (1):303–310, 1991.

[29] David King and Phillip Wadler. Combining monads. *Functional Programming, Glasgow 1992*, pages 134–143, 1993.

[30] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.12. *INRIA*, 2010.

[31] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, 1995.

[32] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. *Journal of Functional Programming*, (October), 2010.

[33] Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 229(5):97–117, 2011.

[34] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. *École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, Technical Report*, 2010.

[35] Conor McBride. The Strathclyde Haskell Enhancement. *http://personal.cis.strath.ac.uk/~conor/pub/she*, 2009.

[36] Conor McBride and Ross Paterson. Functional Pearl: Applicative programming with effects. *Journal of Functional Programming*, 2008.

[37] Saunders McLane. *Categories for the Working Mathematician.* Categories for the Working Mathematician. Springer, 1998.

[38] MSDN Microsoft. Computation Expressions (F#). *http://msdn.microsoft.com/en-us/library/dd233182.aspx.*

[39] John C. Mitchell. *Foundations for Programming Languages*, volume 1. MIT press Cambridge, 1996.

[40] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[41] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[42] Martin Odersky, Philippe Altherr, Vincent Cremet, and Burak Emir. An overview of the Scala programming language. (Section 5), 2004.

[43] Dominic Orchard and Alan Mycroft. A Notation for Comonads. *24th Symposium on Implementation and Application of Functional Languages*, 2012.

[44] Ross Paterson. A New Notation for Arrows. *ACM SIGPLAN Notices*, page 229, 2001.

[45] Tomas Petricek. Evaluation strategies for monadic computations. *Electronic Proceedings in Theoretical Computer Science*, 76(Section 2):68–89, February 2012.

[46] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. *Proceedings of ICALP*, 2013.

[47] Tomas Petricek and Don Syme. Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming. *Practical Aspects of Declarative Languages*, pages 1–15, 2011.

[48] Simon Peyton Jones and Phillip Wadler. Imperative functional programming. *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 71–84, 1993.

[49] Benjamin C. Pierce. *Basic category theory for computer scientists.* Foundations of Computing Series. University Press Group Limited, 1991.

[50] Dan Piponi. Haskell monoids and their uses. *http://blog.sigfpe.com/2009/01/haskell-monoids-and-their-uses.html*, 2009.

[51] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. *ECOOP 2012–Object-Oriented Programming*, 2012.

[52] Amr Sabry. What is a Purely Functional Language? *Journal of Functional Programming*, 1(January), 1998.

[53] Harald Sø ndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990.

[54] Sam Staton. Instances of computational effects: an algebraic perspective. In *Twenty-Eighth Annual ACM/IEEE Symposium on Logic in Computer Science*, 2013.

[55] Joseph E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory.* MIT press, 1977.

[56] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. *Proceedings of the 21st*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94, pages 188–201, 1994.

[57] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[58] David A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

[59] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In *Central European Functional Programming School*, pages 135–167. Springer, 2006.

[60] Tarmo Uustalu and Varmo Vene. Comonadic functional attribute evaluation. *Trends in Functional Programming*, 6:145–162, 2007.

[61] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203:263–284, 2008.

[62] Manfred Von Thun and Others. Joy: Forth's functional cousin. In *Proceedings from the 17th EuroForth Conference*, 2001.

[63] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, number June, pages 347–359, 1989.

[64] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990.

[65] Philip Wadler. Monads for functional programming. *Advanced Functional Programming*, pages 24–52, 1993.

[66] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, January 2003.

[67] Phillip Wadler. The essence of functional programming. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 1992.

[68] Brent Yorgey. The Typeclassopedia. *The Monad Reader*, (13):17, 2009.