Монады в функциональном программировании

Филипп Уодлер, Университет Глазго? Отдел Информационных Технологий, Университет Глазго, G12 8QQ, Шотландия

(wadler@dcs.glasgow.ac.uk)

Резюме. Описывается использование монад для структурирования функциональных программ. Монады обеспечивают удобную рабочую среду для моделирования эффектов, найденных в других языках, таких как глобальное состояние, обработка исключений, продукции, или недетерминизм. Подробно рассмотрены три исследования: как монады упрощают модификацию простого вычислителя; как монады действуют в качестве базиса типа данных массивов, подчиненных оперативному обновлению; и как монады могут использоваться, чтобы строить анализаторы.

1 Введение. Буду ли я чист или нечист?

Функциональное программное сообщество делится на два лагеря. Чистые языки, такие как Miranda0 и Haskell, являются лямбда-исчислением, чистым и простым. Нечистые языки, такие как Scheme и Standard ML (Стандартный Стакан), увеличивают лямбда-исчисление с рядом возможных эффектов, таких как назначения, исключения, или продолжения. Чистые языки проще в относительных рассуждениях и могут извлечь выгоду из ленивой оценки, в то время как нечистые языки предлагают выгоды эффективности и иногда делают возможным более компактный способ выражения.

Недавний прогресс в теоретической вычислительной науке, особенно в областях теории типа и теории категории, предложил новые подходы, которые могут объединить выгоды чистых и нечистых школ. Эти заметки описывают один из них - использование монад, чтобы внести нечистые эффекты в чистые функциональные языки.

Понятие монады, которая является результатом теории категории, было применено Moggi, чтобы структурировать деописательную семантику языков программирования [13, 14]. Та же самая техника может быть применена, чтобы структурировать функциональные программы [21, 23].

Приложения монад будут иллюстрированы тремя исследованиями. Секция 2 вводит монады, показывая, как они могут использоваться для структурирования простого вычислителя с учётом облегчения дальнейших изменений. Секция 3 описывает законы, удовлетворенные

? Появляется в: J. Jeuring и E. Meijer, редакторы, Передовое Функциональное Программирование (Advanced Functional Programming),

Слушания Весенней школы B*astad, в мае 1995, Springer Verlag Лекционные Заметки в Computer Science 925. Предыдущая версия этой заметки появилась в: M. Broy, редактор, Исчисления Проекта Программы (Program Design Calculi), Слушания Летней Школы Marktoberdorf, 30 июля - 8 августа 1992. Некоторые опечатки исправлены в августе 2001; благодарность Дэну Фридману за их обнаружение. 0 Miranda - торговая марка Research Software Limited (Ограниченного Программного обеспечения Исследования).

монадами. Секция 4 показывает, как монады обеспечивают новое решение старой проблемы обеспечения обновляемого состояния в чистых функциональных языках. Секция 5 применяет монады к проблеме построения рекурсивных анализаторов спуска; это представляет интерес в его собственном праве, потому что это обеспечивает парадигму для последовательного выполнения операций и чередования, двух из центральных концепций вычисления.

Сомнительно, что методы структурирования, представленные здесь были бы обнаружены без понимания, предоставленного в соответствии с теорией категории. Но когда-то обнаруженные они легко выражены без какой-либо ссылки на категориальные «вещи». Никакое знание теории категорий не обязательно для чтения этих заметок.

Примеры будут даваться на Haskell, но никакое его знание также не потребуется. То, что требуется, - мимолетные дружественные отношения с основами чистого и нечистого функционального программирования; для общего развития см. [3, 12]. Языки для ссылок - Haskell [4], Miranda [20], Standard ML [11], и Scheme [17].

2 Вычисление монад.

Чистые функциональные языки имеют то преимущество, что весь поток данных сделан явным. И это же неудобство: иногда он крайне явный.

Программа на чистом функциональном языке написана как набор уравнений. Явный поток данных гарантирует, что значение выражения зависит только от его свободных переменных. Следовательно, замена эквивалентного на эквивалентное всегда допустима, делая такие программы особенно простыми для общих рассуждений. Явный поток данных также гарантирует, что порядок вычисления является несущественным, делая такие программы, восприимчивыми к ленивым вычислениям.

Именно в отношении модульности явный поток данных становится и благословением, и проклятием. С одной стороны, это - предел в модульности. Все данные и во всех данных предоставлены декларация и доступность, обеспечивая максимум гибкости. С другой стороны, это - низшая точка модульности. Сущность алгоритма может стать скрытой под арматурой, требующей нести данные от его точки создания к его точке использования.

Скажем, я записываю вычислитель на чистом функциональном языке.

- Чтобы ко всему прочему добавить обработку ошибок, я должен изменить каждое рекурсивное обращение для проверки и оперативных ошибок, соответственно. Если бы я использовал нечистый язык с исключениями, никакого подобного реструктурирования не было бы необходимо.
- Чтобы добавить подсчет выполненных операций, я должен изменить каждое рекурсивное обращение, чтобы распределить такой подсчет соответственно. Если бы я использовал нечистый язык с глобальной переменной, которая может быть увеличена, никакого подобного реструктурирования не было бы необходимо.
- Для добавления трассировки выполнения, я должен изменить каждое рекурсивное обращение, чтобы распределить такие трассеры соответственно. Если бы я использовал нечистый язык, который выполняет продукцию как побочный эффект, никакого подобного реструктурирования не было бы необходимо.

Или я могу использовать монаду.

Эти заметки показывают, как использовать монады для структурирования вычислителя так, чтобы изменения, упомянутые выше, было просто сделать. В каждом случае, все, что требуется, это переопределить монаду и сделать несколько местных изменений.

Этот стиль программирования восстанавливает часть гибкости, обеспеченной различными особенностями нечистых языков. Это также может быть применено, когда нет никакой соответствующей нечистой особенности. Это не устраняет напряженность между гибкостью, предоставленной явными данными и краткостью, предоставленной неявной арматурой; но это действительно до некоторой степени повышает качество.

Методика применима не только к вычислителям, но и на широкий диапазон функциональных программ. В течение множества лет, Глазго был вовлечен в построение компилятора для функционального языка Haskell. Компилятор самостоятельно написан в Haskell, и использует методику структурирования, описанную здесь. Хотя эта бумага описывает использование монад в программах длинной с десяток строк, мы имеем опыт использования их в программе на три порядка большей величины.

Мы начинаем с основного вычислителя для простых термов, затем рассматриваем изменения, которые имитируют исключения, состояние, и вывод. Мы анализируем их на общность, и обобщаем из них концепцию монады. Затем мы показываем, как каждое из изменений вписывается в монадическую рабочую область.

2.1 Разновидность ноль: основной вычислитель.

Вычислитель действует на термы, которые для целей иллюстрации были взяты чрезмерно простыми.

data Term = Con Int j Div Term Term
Терм является или постоянной Con a, где a - целое число, или частное, Div t u, где t и u - термы.

Основной вычислитель - непосредственная простота.

eval :: Term ! Int eval (Con a) = a eval (Div t u) = eval t \Xi eval u

Функциональная оценка приводит терм к целому числу. Если терм - константа, константа возвращена. Если терм - частное, его подтермы оценены, и частное вычислено. Мы используем '\Xi', чтобы обозначить целочисленное деление.

А вот это обеспечит выполняющиеся примеры.

answer ; error :: Term answer = (Div (Div (Con 1972) (Con 2)) (Con 23)) error = (Div (Con 1) (Con 0))

Вычислительный ответ оценки приводит к значению ((1972 \Xi 2) \Xi 23), который есть 42. Основной вычислитель не включает обработку ошибок, так что результат ошибки оценки не определен.

2.2 Разновидность один: Исключения.
Говорят, что желательно добавить проверку ошибок так, чтобы второй пример выше возвратил разумное сообщение об ошибках. На нечистом языке, это легко достигнуто с использованием исключений.

На чистом языке, обработка особых ситуаций может быть эмулирована вводом типа, чтобы представить вычисления, которые могут возбудить исключение.

data M a = Raise Exception j Return a type Exception = String

Значение типа M или имеет форму Raise e, где e - исключение, или Return a, где a - значение типа a. В соответствии с соглашением, a будет использоваться и как переменная типа, как M a, и как переменная, располагающаяся по значениям того типа, который в Return a.

(Слово о различии между объявлениями 'data' и 'type'. Объявление 'data' вводит новый тип данных, в данном случае M, и новые конструкторы для значений этого типа, в данном случае Raise и Return. Объявление 'type' вводит новое имя для существующего типа, в данном случае Exception становится другим именем для String.)

Это является прямым, но утомительным - приспосабливать вычислитель к этому представлению.

eval :: Term ! M Int

eval (Con a) = Return a eval (Div t u) = case eval t of

Raise e ! Raise e Return a !

case eval u of

Raise e ! Raise e Return b !

if b == 0

then Raise "divide by zero" else Return (a \Xi b)

В каждом запросе вычислителя форма результата должна быть проверена: если исключение было возбуждено, оно перевозбуждено, и если значение было возвращено, оно обработано. Применение нового вычислителя, чтобы ответить на результаты (Return 42), при применении его к кодам ошибки (Raise "divide by zero").

2.3 Разновидность два: Состояние.
Забывать ошибки за мгновение - говорят, что это желательно для подсчета числа делений, выполненных во время оценки. На нечистом языке это легко достигнуто при помощи состояния. Первоначально установите данную переменную в обнулённое состояние, и увеличивайте её каждый раз, когда происходит деление.

На чистом языке, состояние может быть эмулировано вводом типа для представления вычислений, воздействующих на состояние.

type M a = State ! (a; State) type State = Int

Теперь значение типа M a является функцией, которая принимает начальное состояние и возвращает вычисленное значение, соединенное с заключительным состоянием.

И снова это является прямым, но утомительным - приспосабливать вычислитель к этому представлению.

eval :: Term ! M Int

eval (Con a) x = (a; x) eval (Div t u) x = let (a; y) = eval t x in

let (b; z) = eval u y in (a \Xi b; z + 1)

В каждом запросе вычислителя, в старом состояние должно быть передано и новое состояние, извлеченное из результата и перешедшее соответственно. Ответ вычисления оценки 0 кодов (42; 2), такой: с начальным состоянием 0 ответ - 42, и с заключительным состоянием - 2, указывая, что два раздела были выполнены.

2.4 Разновидность три: Вывод.
Наконец, говорят желательно отображать трассер (след) выполнения. На нечистом языке, это легко сделано, вставкой команды вывода в соответствующие точки.

На чистом языке, вывод может быть эмулирован вводом типа, чтобы представить вычисления, которые генерируют вывод.

type M a = (Output; a) type Output = String

Теперь значение типа M a состоит из вывода, сгенерированного и соединенного с вычисленным значением.

Все же снова это является прямым, но утомительным - приспосабливать вычислитель к этому представлению.

eval :: Term ! M Int eval (Con a) = (line (Con a) a; a) eval (Div t u) = let (x ; a) = eval t in

let (y; b) = eval u in (x ++ y ++ line (Div t u) (a \Xi b); a \Xi b)

line :: Term ! Int ! Output line t a = "eval (" ++ showterm t ++ ") (" ++ showint a ++ "-"

В каждом запросе вычислителя выводы должны быть собраны и ассемблированы, чтобы сформировать вывод запроса включения. Функциональная строка генерирует одну строку вывода. Здесь showterm и showint конвертируют термы и целые числа к строкам, ++ связывает строки, и "-" представляет строку, состоящую из newline.

Ответ вычисления оценки возвращает пару (x; 42), где x - строка

eval (Con 1972) (1972 eval (Con 2) (2 eval (Div (Con 1972) (Con 2)) (986 eval (Con 23) (23 eval (Div (Div (Con 1972) (Con 2)) (Con 23))) (42

которая представляет трассер вычисления.

В дальнейшем стоит обсудить, может казаться, что программы на нечистых языках проще для изменений, чем те же на чистых языках. Но иногда истинно обратное. Говорят, что было бы желательно изменить предыдущую программу, чтобы отобразить трассер выполнения в обратном порядке:

eval (Div (Div (Con 1972) (Con 2)) (Con 23))) (42 eval (Con 23) (23 eval (Div (Con 1972) (Con 2)) (986 eval (Con 2) (2 eval (Con 1972) (1972

Это непосредственно просто достижимо с чистой программой: только замените терм

x ++ y ++ line (Div t u) (a \Xi b) with the term

line (Div t u) (a \Xi b) ++ y ++ x :

Нечистую же программу не столь просто изменить, чтобы достигнуть такого эффекта. Проблема состоит в том, что вывод происходит как побочный эффект вычисления, но теперь желаем отобразить результат вычисления терма перед отображением вывода, сгенерированного этим вычислением. Это может быть достигнуто разнообразными путями, но все они требуют существенной модификации в нечистой программе.

2.5 Монада-вычислитель.
Каждая из разновидностей интерпретатора имеет подобную структуру, которая может реферироваться, чтобы привести к понятию монады.

В каждой разновидности мы вводили тип вычислений. Соответственно, M представлял вычисления, которые могли возбудить исключения, действовать на состояние, и генерировать вывод. К настоящему времени читатель предположит, что M замещает монаду.

Первоначальный вычислитель имеет тип Term ! Int, в то время как в каждой разновидности его тип принял форму Term ! M Int. Вообще, функция типа a ! b заменена функцией типа a ! M b. Это может читаться как функция, которая принимает параметр типа и возвращает результат типа b с возможным дополнительным эффектом, зафиксированным M. Этот эффект может быть действием на состояние, генерацией вывода, возбуждение исключения, или что Вы пожелаете (what have you).

Какие операции требуются на типе M? Экспертиза примеров показывает две. Сначала, мы нуждаемся в способе превратить значение в вычисление, которое возвращает это значение и не делает ничего иного.

unit :: a ! M a

Второе, мы нуждаемся в способе применить функцию типа a ! M b к вычислению типа M a. Удобно записать их в порядке, при котором параметр прибывает перед функцией.

(?) :: M a ! (a ! M b) ! M b

Монада это тройка (M ; unit; ?), состоящая из конструктора типа M и двух операций данных полиморфных типов. Эти операции должны удовлетворить трём законам, данным в Разделе 3.

Мы будем часто записывать выражения в форме

m ? *a: n, где m и n - выражения, и a - переменная. Форма *a: n - это лямбда-выражение с контекстом связанной переменной существа выражения n. Вышеупомянутое может читаться следующим образом: исполните вычисление m, свяжите со значением результата, и затем исполните вычисление n. Типы обеспечивают полезное руководство. Из типа (?) мы можем видеть, что выражение m имеет тип M a, переменная имеет тип a, выражение n имеет тип M b, лямбда-выражение *a: n имеет тип a ! M b, и всё выражение имеет тип M b.

Вышеупомянутое подходит для выражения

let a = m in n:

На нечистом языке, это читается также: исполните вычисление m, свяжите со значением результата, затем исполните вычисление n, и возвратите его значение. Здесь типы не говорят ничто, чтобы отличить значения от вычислений: выражение m имеет тип a, переменная a имеет тип a, выражение n имеет тип b, и всё выражение имеет тип b. Аналогия с 'let' объясняет выбор порядка параметров к ?. Это удобно для параметра m, чтобы появиться перед функцией *a: n, начиная с вычисления m, выполненного перед вычислением n.

Вычислитель легко перезаписан в терминах этих абстракций.

```
eval :: Term ! M Int eval (Con a) = unit a eval (Div t u) = eval t ? *a: eval u ? *b: unit (a \Xi  b)
```

Слово о старшинстве: лямбда-абстракция связывает наименее сильно, а приложение связывает наиболее сильно, так что последнее уравнение эквивалентно следующему.

```
eval (Div t u) = ((eval t) ? (*a: ((eval u) ? (*b: (unit (a \Xi  b))))))
```

Тип Term ! M Int указывает, что вычислитель берет терм и исполняет вычисление, приводящее к целому числу. Вычисление (Con a), возвращает только a. Чтобы вычислять (Div t u), сначала вычислите t, свяжите с результатом, затем вычислите u, свяжите b с результатом, и затем возвратите \Xi b.

Новый вычислитель немного более сложен, чем первоначальный основной вычислитель, но он намного более гибок. Каждая из разновидностей, данных выше, может быть достигнута, просто изменением определения M, модуля, и ?, и произведение одной или двух местных модификаций. Больше нет необходимости переписывания полного вычислителя для достижения этих простых изменений.

2.6 Разновидность ноль, пересмотренная: основной вычислитель.
В простейшей монаде, вычисление не имеет никакого отличия от значения.

```
type M a = a unit :: a ! I a unit a = a
```

```
(?) :: M a ! (a ! M b) ! M b a ? k = k a
```

Это называют монадой тождества: M - функция тождества на типах, unit - функция тождества, и ? является только приложением.

Берём M, unit и ? как изложено выше в монаде-вычислителе Раздела 2.5 и упрощаем коды основного вычислителя Раздела 2.1

2.7 Разновидность один, пересмотренная: Исключения.
В монаде-исключении, вычисление может или возбудить исключение или возвратить значение.

```
data M a = Raise Exception j Return a
```

```
type Exception = String
```

```
unit :: a ! M a unit a = Return a
```

```
(?) :: M a ! (a ! M b) ! M b m ? k = case m of
```

```
Raise e ! Raise e Return a ! k a
```

```
raise :: Exception ! M a raise e = Raise e
```

Модуль запроса просто возвращает значение a. Запрос m ? k исследует результат вычисления m: если это - исключение, оно перевозбуждается, иначе функция k применяется к возвращенному значению. Также как ? в монаде тождества функциональное приложение ? в монаде исключения можно рассматривать как форму строгого функционального приложения. Наконец, запрос raise e, возбуждает исключение e.

Чтобы добавить обработку ошибок к монаде-вычислителю, возьмите монаду как изложено выше. Затем только замените модуль (\Xi b) на

```
if b == 0

then raise "divide by zero" else unit (a \Xi  b)
```

Это соразмерно с изменением, требуемом на нечистом языке.

Как и можно было ожидать, этот вычислитель эквивалентен вычислителю с исключениями Раздела 2.2. В частности, разворачивая определения unit и ? в этом разделе и упрощая коды вычислителя этого раздела.

2.8 Разновидность два, пересмотренная: Состояние в монаде состояния.
Вычисление принимает начальное состояние и возвращает соединение, соединенное с финальным состоянием.

```
type M a = State ! (a; State) type State = Int

unit :: a ! M a unit a = *x : (a; x )

(?) :: M a ! (a ! M b) ! M b m ? k = *x : let (a; y) = m x in

let (b; z ) = k a y in (b; z )

tick :: M () tick = *x : ((); x + 1 )
```

Модуль вызова возвращает вычисление, которое принимает начальное состояние x и возвращает значение a и конечное состояние x; то есть, оно возвращает a и оставляет состояние неизменным. Вызов m ? k выполняет вычисление m в начальном состоянии x, приводя к значению a и промежуточному состоянию y; затем исполняет вычисление k в состоянии y, приводя к значению b и конечному состоянию z. Импульс вызова увеличивает состояние, и возвращает пустое значение (), чей тип также описан как ().

На нечистом языке, операция подобная импульсу была бы представлена функцией типа () ! (). Поддельный параметр () обязан задерживать эффект, пока функция не применена, и так как тип вывода - (), можно предположить, что целью функции является побочный эффект. Напротив, здесь импульс имеет тип M (): никакой поддельный параметр не является необходимым, и вид M явно указывает, какой эффект может произойти.

Чтобы добавить подсчет выполнений в монаду-вычислитель, возьмите монаду как вышеизложено. Затем только замените модуль (a \Xi b) на

```
tick ? *(): unit (a \Xi  b)
```

(Здесь * : e эквивалентно *x : e, где x :: () - некоторая новая переменная, которая не появляется в e; это указывает, что значение, связанное лямбда-выражением должно быть ().) И снова, это соразмерно с изменением, требуемом на нечистом языке. Упрощение кодов вычислителя с состоянием Раздела 2.3.

2.9 Разновидность три, пересмотренная: Вывод.
В монаде вывода вычисление состоит из сгенерированного вывода, соединенного с возвращенным значением.

```
type M a = (Output; a) type Output = String

unit :: a ! M a unit a = (" "; a)

(?) :: M a ! (a ! M b) ! M b m ? k = let (x ; a) = m in

let (y; b) = k a in (x ++ y; b)

out :: Output ! M () out x = (x ; ())
```

Модуль вызова не возвращает никакого вывода соединённого с a. Вызов m ? k извлекает вывод x и значение из вычисления m, затем извлекает вывод y и значение b из вычисления k a, и возвращает вывод, сформированный конкатенацией x и y, соединенный со значением b. Вызов x возвращает вычисление с выводом x и пустым значением ().

Чтобы добавить трассеры выполнения к монаде-вычислителю, возьмите монаду как вышеизложено. Тогда в предложении для Con a замените unit a на
out (line (Con a) a) ? *(): unit a
a в предложении для Div t u замените unit (a \Xi b) на
out (line (Div t u) (a \Xi b)) ? *(): unit (a \Xi b)
И опять же, это соразмерно с изменением, требуемой на нечистом языке. Упрощение кодов вычислителя с выводом Раздела 2.4

Чтобы получить вывод в обратном порядке, все, что требуется, - это изменить определение ?, заменой x ++ y на y ++ x. Это соразмерно с изменением, требуемом в чистой программе, и скорей всего более просто, чем изменение, требуемое в нечистой программе.

Вы могли бы подумать, что единственное различие между чистыми и нечистыми версиями - то, что нечистые версии отображают вывод, как он вычисляется, в то время как чистая версия не отобразит ничего, пока полное вычисление не завершится. Фактически, если чистый язык ленив, тогда вывод будет отображен возрастающим способом, как происходит вычисление. Кроме того, это также случится, если порядок вывода обратный, который намного более трудно организовать на нечистом языке. Действительно, самый простой способ организовать его может состоять в том, чтобы смоделировать ленивую оценку.

3. Законы монад.
Операции монад удовлетворяют трём законам.

- Левый модуль. Вычислите значение a, свяжите b с результатом, и вычислите n. Результат тот же самый как n со значением, замененным на переменную b.

unit a ? *b: n = n[a=b]:
- Правый модуль. Вычислите m, свяжите результат с a, и возвратите a. Результат такой же, как m.

m ? *a: unit a = m:

- Ассоциативность. Вычислите m, свяжите результат с a, вычислите n, свяжите результат с b, вычислите o. Порядок круглых скобок в таком вычислении является несоответствующим.

m ? (*a: n ? *b: o) = (m ? *a: n) ? *b: o: Контекст переменной включает o слева, но исключает o справа, так что этот закон допустим только, когда пустота (free) не появляется в o.

Двойную операцию с левым и правым модулем, который является ассоциативным, называют моноидом. Монада отличается от моноида тем, что правый операнд вовлекает операцию связывания.

Чтобы продемонстрировать использование этих законов, мы доказываем, что добавление является ассоциативным. Просмотрите вариант вычислителя, основанного на добавлении, а не делении.

data Term = Con Int j Add Term Term eval :: Term ! M Int eval (Con a) = unit a eval (Add t u) = eval t ? *a: eval u ? *b: unit (a \Xi b)

Рассмотрим это вычисление
Add t (Add u v) and Add (Add t u) v ; оба вычисляют одинаковый результат. Упростите левый терм:

eval (Add t (Add u v)) = f def'n eval g
eval t ? *a: eval (Add u v) ? *x : unit (a + x) = f def'n eval g

```
eval t ? *a: (eval u ? *b: eval v ? *c: unit (b + c)) ? *x : unit (a + x ) = f
associative g
eval t ? *a: eval u ? *b: eval v ? *c: unit (b + c) ? *x : unit (a + x ) = f left
unit g
eval t ? *a: eval u ? *b: eval v ? *c: unit (a + (b + c))
```

Упростите правый терм наподобие нижеследующего:

```
eval (Add (Add t u) v ) = f as before g
eval t ? *a: eval u ? *b: eval v ? *c: unit ((a + b) + c)
```

Из результата следует ассоциативность добавления. Это доказательство тривиально; без законов монады, это было бы невозможно.

Доказательство работает в любой монаде: исключение, состояние, вывод. Предполагается, что код - как приведен выше: если это не так, тогда доказательство также должно измениться. Раздел 2.3 изменил программу, добавив запросы за импульс сигнала времени (to tick). В этом случае, ассоциативность все еще держится, и может быть продемонстрирована, используя закон
tick ? *(): m = m ? *(): tick, который держится столь долго во время импульса сигнала времени - единственное действие при состоянии в пределах m. Раздел 2.4 изменил программу, добавив запросы в линию (to line). В этом случае, добавление больше не ассоциативно, в том смысле, что изменение круглых скобок изменит трассер (trace), хотя вычисления будут все еще приводить к тому же самому значению.

А вот другой пример, показывающий, что для каждой монады мы можем определить следующие операции.

```
map :: (a ! b) ! (M a ! M b) map f m = m ? *a: unit (f a)
join :: M (M a) ! M a join z = z ? *m: m
```

Операция map просто применяет функцию к результату, к которому приводит вычисление. Чтобы вычислять map f m, сначала вычислите m, свяжите с результатом, и затем возвратите f a. Операция объединения сложнее. Позвольте z быть вычислением, которое непосредственно приводит к вычислению. Чтобы вычислять объединение z, сначала вычислите z, свяжите m с результатом, и затем поведение, как и при вычислении m. Таким образом, объединение сглаживает ошеломляющий (mind-boggling) двойной уровень вычисления до "выполненного из завода" (run-of-the-mill) единственного уровня вычисления. Как мы увидим в Разделе 5.1, списки формируют монаду, и для этого монада map применяет функцию к каждому элементу списка, и объединение связывает список списков.

Используя id для функции тождества (id x = x), и (\Delta) для композиции функций ((f \Delta g) x = f (r x)), можно сформулировать множество законов.

```
map id = id map (f \Delta  g) = map f \Delta  map g
map f \Delta  unit = unit \Delta  f map f \Delta  join = join \Delta  map (map f )
join \Delta  unit = id join \Delta  map unit = id join \Delta  map join = join \Delta
join
m ? k = join (map k m)
```

Доказательство для каждого - простое последствие определений map и join и трех законов монад.

Часто монады определены не в терминах unit и ?, а скорее в терминах unit, join, и map [10, 13]. Три закона монад заменены в соответствии с первыми семью из этих восьми законов выше. Если определяете ? в соответствии с восьмым законом, тогда три закона монады являются следствием. Следовательно, эти два определения эквивалентны.

4 Состояние. Массивы играют центральную роль в компьютерах, потому что они близко соответствуют текущей архитектуре. Программы замусорены поисками в массивах типа x [i] и обновлениями массивов наподобие x [i]: = v. Эти операции популярны, потому что поиск в массиве осуществляется единственной индексированной командой выборки, и обновление массива единственно индексированной памятью.

Это просто - добавить массивы к функциональному языку, и просто обеспечить эффективный поиск в массивах. С другой стороны, как обеспечивать эффективное обновление массива - это длинная история! Монады обеспечивают новый ответ на этот старый вопрос.

Другой вопрос с длинной историей - желательно ли базировать программы на обновлении массива. С тех пор как такое большое усилие вошло в развивающиеся алгоритмы и архитектуру, основанную на массивах, мы обойдем эти дебаты и просто предположим, что ответ - да.

Есть важное различие между способом, которым монады используются в предыдущем разделе и способе, которым монады используются здесь. Предыдущий раздел показал, что монады помогают использовать существующие особенности языка более эффективно; этот раздел показывает, как монады могут помочь определять новые особенности языка. Никакое изменение в языке программирования не требуется, но применение должно обеспечить новый абстрактный тип данный, возможно как часть стандартной вводной части.

Здесь монады используются, чтобы управлять состоянием внутри программы, но те же самые методы могут быть использованы, чтобы управлять внешним состоянием: выполнять ввод-вывод, или связываться с другими языками программирования. Реализация Haskell'a из Глазго использует дизайн, основанный на монадах, чтобы обеспечить ввод-вывод и язык межнационального общения, работающий с обязательным языком программирования C [15]. Этот дизайн был принят для версии 1.3 стандарта Haskell'a.

4.1 Массивы. Позвольте Arr быть типом массивов, берущих индексы типа Ix и приводящих к значениям типа Val. Ключевые операции на этом типе

newarray :: Val ! Arr; index :: Ix ! Arr ! Val; update :: Ix ! Val ! Arr ! Arr :

Запрос newarray v возвращает массив со всем набором входов в v; запрос index i x возвращает значение по индексу i в массиве x; и запрос update i v x возвращает массив, где индекс, i имеет значение v и остаток, идентичный x. Поведение этих операций определено в соответствии с законами

index i (newarray v) = v ; index i (update i v x) = v ; index i (update j v x) = index i x ; if i 6= j .

Практически, эти операции были бы более сложны; например, если Вы нуждаетесь в способе определить индексные границы. Но вышеупомянутое позволяет объяснить основные пункты.

Эффективный способ осуществлять операцию обновления состоит в том, чтобы записать поверх указанный вход массива, но на чистом функциональном языке это безопасно, только если нет других указателей на массив существующий, когда операция обновления выполнена. Массив, удовлетворяющий этому свойству, называют единственносвязным (введено Шмидтом [Schmidt]) [18].

Рассмотрите формирование интерпретатора для простого императивного языка. Абстрактный синтаксис для этого языка представлен следующими типами данных.

data Term = Var Id j Con Int j Add Term Term data Comm = Asgn Id Term j Seq Comm Comm j If Term Comm Comm data Prog = Prog Comm Term

Здесь Id : baastad:tex ; v1 :1 :1 :11996 =02 =2915 : 17 : 01wadlerExp - неопределенный тип идентификаторов. Терм - это переменная, константа, или сумма двух термов; команда - это назначение, последовательность двух команд, или условного выражения; и программа состоит из команды, сопровождаемой термом.

Текущее состояние выполнения будет оформлено массивом, где индексы - идентификаторы, и соответствующие значения - целые числа.

type State = Arr type Ix = Id type Val = Int

Вот - интерпретатор.

```
eval :: Term ! State ! Int eval (Var i) x = index i x eval (Con a) x = a eval (Add t
u) x = eval t x + eval u x
exec :: Comm ! State ! State exec (Asgn i t) x = update i (eval t x ) x exec (Seq c d
) x = exec d (exec c x ) exec (If t c d ) x = if eval t x == 0 then exec c x else
exec d x
elab :: Prog ! Int elab (Prog c t) = eval t (exec c (newarray 0 ))
```

Это близко напоминает денотационную семантику. Вычислитель для термов берет терм и состояние и возвращает целое число. Вычисление переменной осуществлено, индексированием состояния. Управляющая программа для команд берет команду и начальное состояние и возвращает конечное состояние. Означивание осуществлено, модифицированием состояния. Elaborator для программ берет программу и возвращает целое число. Он выполняет команду в начальном состоянии, где вся карта идентификаторов установлена в 0, затем оценивает данное выражение в конечном состоянии и возвращает его значение.

Состояние в этом интерпретаторе единственносвязное: в любой момент выполнения есть только один указатель на состояние, так что безопасно модифицировать состояние на месте. Для этого, чтобы работать, операция обновления должна оценить новое значение перед размещением его в массиве. Иначе, массив может содержать замкнутое выражение, которое непосредственно содержит указатель на массив, нарушая свойство единственной связности. В семантических термах говорится, что обновление строго во всех трех из их параметров.

Множество исследователей предложило исследования, которые определяют, использует ли данная функциональная программа массив единственносвязным способом, с намерением включения такого анализа в компилятор оптимизации. Большинство этих исследований основано на абстрактной интерпретации [1]. Хотя было немного успеха в этой области, исследования имеют тенденцию быть столь дорогими, что являются очень тяжелыми [2, 7].

Даже если такие исследования были бы реальны, их использование может быть неблагоразумно. Оптимизация обновления может затронуть время программы и использование пространства в соответствии с порядком величины или больше. Программист должен быть уверен, что такая оптимизация произойдет, чтобы знать, что программа будет работать соответственно быстро и в пределах доступного пространства. Это может быть лучше для программиста указать явно, что массив должен быть единственносвязный, а не оставлять это на откуп капризам компилятора оптимизации.

И снова, множество исследователей предложило методы для того, чтобы указать, что массив единственносвязный. Большинство этих методов основано на системах типа [6, 19, 22]. Эта область кажется многообещающей, хотя сложности этих систем типа остаются огромными.

Следующий раздел представляет другой способ указать явно намерение, что массив должен быть единственносвязным. Естественно, это основано на монадах. Преимущество этого метода состоит в том, что это работает с существующими системами типа, используя только идею относительно абстрактного типа данных.

4.2 Массивы-трансформаторы. Монада массивов-трансформаторов - просто монада трансформаторов состояний, с состоянием, взятым, чтобы быть массивом. Определения M, unit, ? являются прежними.

```
type M a = State ! (a; State) type State = Arr
unit :: a ! M a unit a = *x : (a; x )
(?) :: M a ! (a ! M b) ! M b m ? k = *x : let (a; y) = m x in
let (b; z ) = k a y in (b; z )
```

Предварительно, нашим состоянием было целое число, и мы имели дополнительный импульс сигнала времени (tick) операции, действующий на состояние. Теперь наше состояние - массив, и мы имеем дополнительные операции, соответствующие созданию, индексации, и обновлению массива.

```
block :: Val ! M a ! a block v m = let (a; x ) = m (newarray v ) in a
```

```
fetch :: Ix ! M Val fetch i = *x : (index i x ; x )
assign :: Ix ! Val ! M () assign i v = *x : ((); update i v x )
```

Вызов block v m создает новый массив со всеми местоположениями, инициализированными к v, применяет монаду m к этому начальному состоянию, чтобы привести к значению a и конечному состоянию x, освобождает массив, и возвращает a. Вызов fetch i возвращает значение по индексу i в текущем состоянии, и оставляет неизменным состояние. Вызов assign i v возвращает пустое значение (), и модифицирует состояние так, чтобы индекс i содержал значение v.

Небольшая мысль показывает, что эти операции действительно единственносвязные. Единственная операция, которая могла дублировать массив - выборка, но это может быть осуществлено следующим образом: сначала выберите вход по данному индексу в массиве, и затем возвратите пару, состоящую из этого значения и указателя на массив. В семантических термах выборка (fetch) строго в массиве и индексе, но не в значении, расположенном по индексу, и означивание (assign), строго в массиве и индексе, но не означенном значении.

(Этот отличается от предыдущего раздела, где для интерпретатора, чтобы быть единственносвязным было необходимо для обновления быть строгим в данном значении. В этом разделе, как мы увидим, эта строгость удалена, но поддельная последовательность введена для оценки термов. В следующем разделе, поддельная последовательность удалена, но строгость будет повторно введена.)

Мы можем теперь создать M в абстрактный тип данных, поддерживающий эти пять операций: unit, ?, block , fetch, и assign. Операция block играет центральную роль, поскольку она единственная, которая не имеет M в его своем типе результата. Без block не было бы никакого способа записать программу, используя M, которая не имела бы M в своем типе вывода.

Создание M в абстрактный тип данных гарантирует, что единственное связывание сохраняется, и, следовательно, безопасно осуществить означивание с оперативным обновлением. Использование абстракции данных является необходимым для этой цели. Иначе, можно было бы записать программы типа

```
*x : (assign i v x ; assign i w x ) that violate the single threading property.
```

Интерпретатор может теперь быть перезаписан следующим образом.

```
eval :: Term ! M Int eval (Var i) = fetch i eval (Con a) = unit a eval (Add t u) =
eval t ? *a: eval u ? *b: unit (a + b)
exec :: Comm ! M () exec (Asgn i t) = eval t ? *a: assign i a exec (Seq c d ) = exec
c ? *(): exec d ? *(): unit () exec (If t c d ) = eval t ? *a:
if a == 0 then exec c else exec d
elab :: Prog ! Int elab (Prog c t) = block 0 (exec c ? *(): eval t ? *a: unit a)
```

Типы показывают, что вычисление терма возвращает целое число и может обратиться или изменить состояние, и что выполнение терма не возвращает ничего и может обратиться или изменить состояние. Фактически, вычисление только обращается к состоянию и никогда не изменяет его - вскоре мы рассмотрим усовершенствованную систему, которая позволит нам показать это.

Абстрактный тип данных для M гарантирует, что исполнить обновления на месте безопасно - никакой специальной методики анализа не требуется. Просто увидеть, как интерпретатор монады может быть получен из оригинала, и (используя определения, данные ранее), доказательство их эквивалентности является прямым.

Перезаписанный интерпретатор является немного более длинным, чем предыдущая версия, но возможно немного более легким для чтения. Например, выполнение (Seq c d) может читаться: вычислите выполнение c, затем вычислите выполнение d, затем не возвращайте ничего. Сравните это с предыдущей версией, которая имеет расстраивающее свойство, когда exec d появляется слева от exec c.

Недостаток{препятствие} этой программы состоит в том, что она вводит слишком много последовательности. Вычисление (Add t u), может читаться: вычислите оценку t, свяжите a с результатом, затем вычислите оценку u, свяжите b с результатом, затем возвратите a + b. Это неудачно, т.к. это налагает поддельное упорядочение на вычисление t и u, которое не присутствовало в первоначальной программе. Порядок не имеет значения, потому что, хотя вычисление зависит от состояния, оно не заменяет его. Для исправления этого мы увеличим монаду трансформаторов состояний M со второй монадой M 0 из читателей состояний.

4.3 Читатели массива.
Повторный вызов, где монада трансформаторов массива берет начальный массив и возвращает значение и конечный массив.

```
type M a = State ! (a; State) type State = Arr
```

Соответствующая монада читателей массива берет массив и возвращает значение. Никакой массив не возвращен, потому что он принят идентичным первоначальному массиву.

```
type M 0 a = State ! a unit0 :: a ! M 0 a unit0 a = *x : a
(?0) :: M 0 a ! (a ! M 0 b) ! M 0 b m ?0 k = *x : let a = m x in k a x
fetch0 :: Ix ! M 0 Val fetch0 i = *x : index i x
```

Запрос unit0 игнорирует данное состояние x и возвращает a. Запрос m ?0 k исполняет вычисление m в данном состоянии x, приводя к значению a, затем исполняет вычисление k в том же самом состоянии x. Таким образом, unit0 отказывается от состояния и ?0 дублирует это. Запрос fetch0 i возвращает значение в данном состоянии x по индексу i.

Ясно, что вычисления, которые только читают состояние - подмножество вычислений, которые могут читать и записывать состояние. Следовательно, должен быть способ принудить вычисление в монаде M 0 внутри одной в монаде M.

```
coerce :: M 0 a ! M a coerce m = *x : let a = m x in (a; x )
```

Запрос coerce m исполняет вычисление m в начальном состоянии x, приводя a, и возвращается соединенный с состоянием x. Функция coerce удовлетворяет множеству математических свойств, которые будут вскоре обсуждены.

И снова, эти операции поддерживают единственносвязность, если соответственно применены. Определения ?0 и coerce должны оба быть строго в промежуточном значении a. Это гарантирует, что, когда coerce m выполнено в состоянии x, вычисление m x уменьшится до формы a, которая не содержит никаких существующих указателей на состояние x перед возвращенной парой (a; x). Следовательно, будет только один существующий указатель на состояние всякий раз, когда оно модифицировано.

Монада коммутативна, если она удовлетворяет закону
m ? *a: n ? *b: o = n ? *b: m ? *a: o: Контекст a включает n справа, a не слева, так что этот закон допустим только, когда a не является пустым в n. Точно так же b не должно оказаться пустым в m. В коммутативной монаде порядок вычисления не имеет значения.

Состояние монады-читателя коммутативна, в то время как состояние монады-трансформера - нет. Так что никакой поддельный порядок не наложен на вычисления в состоянии монады-читателя. В частности запрос m ?0 k может безопасно быть осуществлен так, чтобы m и k были вычислены параллельно. Однако конечный результат должен все еще быть строг в a. Например, с аннотациями, используемыми в процессоре GRIP, ?0 мог быть определен следующим образом.

```
m ?0 k = *x : let a = m x in
let b = k a x in par a (par b (seq a b))
```

Два вызова par являются искрой (spark) параллельных вычислений a и b, и вызов seq ждёт a, чтобы уменьшить до недоосновного значения (non-bottom)неоснования, перед тем как возвратить b.

Эти операции могут быть упакованы в два абстрактных типа данных M и M 0, поддерживая восемь операций unit, ?, unit0, ?0, block, assign, fetch0, и coerce. Абстракция гарантирует единственносвязность, так как assign может быть осуществлено оперативным обновлением.

Интерпретатор может быть снова перезаписан.

```
eval :: Term ! M 0 Int eval (Var i) = fetch0 i eval (Con a) = unit0 a eval (Add t u)
= eval t ?0 *a: eval u ?0 *b: unit0 (a + b)
exec :: Comm ! M () exec (Asgn i t) = coerce (eval t) ? *a: assign i a exec (Seq c d
) = exec c ? *(): exec d ? *(): unit () exec (If t c d ) = coerce (eval t) ? *a:
if a == 0 then exec c else exec d
```

Эта отличается от предыдущей версии по вычислению, описанному в термах M 0, а не M, и вызовы coerce окружают вызовы вычисления в других двух функциях. Новые типы проясняют, что оценка зависит от состояния, но не изменяет его, в то время как exec может и зависеть от состояния и изменять его.

Чрезмерная последовательность предыдущей версии была устранена. В вычислении (Add t u), эти два подвыражения могут быть оценены или по-порядку, или одновременно.

Морфизм монады от монады M 0 к монаде M - это функция h:: M. 0 a ! M a, которая сохраняет структуру монады:

```
h (unit0 a) = unit a; h (m ?0 *a: n) = (h m) ? *a: (h n):
```

Часто случается, что Вы желаете использовать комбинацию монад для достижения определённой цели, и морфизмы монады играют ключевую роль в преобразовании от одной монады к другой [9].

В частности, coerce - это морфизм монады, и это следует непосредственно из того, что две версии интерпретатора эквивалентны.

4.4 Заключение/
Как функциональный язык может обеспечить оперативное обновление массива - это старая проблема. Этот раздел предоставил новое решение, состоящее из двух абстрактных типов данный с восьмью операциями между ними. Никакое изменение в языке программирования не требуется, кроме как обеспечить реализацию этих типов, возможно, как часть стандартной вводной части (prelude).

Открытие такого простого решения является неожиданностью, на фоне рассматриваемого изобилия более сложных решений, которые были предложены.

Иной способ выразить то же самое решение, основанный на продолжении принятого стиля, был впоследствии предложен Хадаком [Hudak] [8]. Но решение Хадака было вдохновлено монадным решением, и монадное решение все еще, кажется, имеет немного маленьких преимуществ [15].

Почему это решение не было обнаружено двадцать лет назад? Одна из возможных причин состоит в том, что типы данных вовлекают функции более высокого порядка необходимым способом. Обычный axiomatisation массивов вовлекает только функции первого уровня, и вполне возможно это не происходило ни с кем - поиск абстрактного типа данных, основанного на функциях более высокого порядка. Те монады вели к открытию решения, которое должно рассчитаться как точка для получения от них пользы.

5 Синтаксические анализаторы.
Синтаксические анализаторы - великая история успеха теории компьютерных вычислений. Формализм НОРМАЛЬНОЙ ФОРМЫ БЕКУСА-НАУРА обеспечивает краткий и точный способ описать синтаксис языка программирования. Математические испытания могут определить, неоднозначна ли грамматика НОРМАЛЬНОЙ ФОРМЫ БЕКУСА-НАУРА или вырождена. Преобразования могут произвести эквивалентную грамматику, которая проще для анализа. Компиляторы компиляторов могут преобразовать спецификацию грамматики высокого уровня в эффективную программу.

Этот раздел показывает, как монады обеспечивают простую структуру для того, чтобы создать рекурсивные синтаксические анализаторы спуска. Это в самом деле представляет интерес, также и потому что основные структуры синтаксического анализа - последовательности и чередования - являются фундаментальными для всей компьютерной теории. Он также обеспечивает демонстрацию того, как монады могут моделировать отслеживание в обратном порядке (backtracking) (или ангельский недетерминизм).

5.1 Списки.
Наше представление синтаксических анализаторов, зависит от списков. Списки являются вездесущими в функциональном программировании, и удивительно, что мы сумели продвинуться пока, только упоминая их. Фактически, они казались скрытыми, поскольку строки - просто списки символов.

Сделаем обзор в качестве некоторого примечания. Мы записываем [a] для типа списка со всеми элементами типа a, и : для `cons'. Таким образом [1; 2; 3] = 1 : 2 : 3 : [], и оба имеют тип [Int]. Строки - списки символов, т.о. String и [Char] эквивалентны, и "monad" - только сокращение для [`м`; `о`; 'n'; 'a'; 'd'].

Это, возможно, не удивительно, эти списки формируют монаду.

```
unit :: a ! [a] unit a = [a]

(?) :: [a] ! (a ! [b]) ! [b] [ ] ? k = [ ] (a : x ) ? k = k a ++ (x ? k)
```

Вызов unit просто формирует список модулей, содержащий a. Запрос m ? k применяет k к каждому элементу списка m, и добавляет в конец вместе заканчивающиеся списки.

Если монады содержат эффекты, и списки формируют монаду, соответствуют ли списки некоторому эффекту? Да, действительно, и эффект, которому они соответствуют - выбор. Можно подумать о вычислении типа [a] как о предложении выбора значений, одном для каждого элемента списка. Монадный эквивалент функции типа a ! b - функция типа a ! [b]. Это предполагает выбор результатов для каждого параметра, и, следовательно, соответствует отношению. Операция unit соответствует отношению тождества, которое связывает каждый параметр только с собой. Если k
:: a! [b] и h:: b! [c], тогда
*a: k a? *b: h b:: a! [c]
соответствует относительной композиции k и h.

Нотация понимания списка обеспечивает удобный способ управлять списками. Необходимое поведение походит на понимание множеств, кроме существенности порядка. Например,

```
[ sqr a j a  [1 ; 2 ; 3 ] ] = [1 ; 4 ; 9 ] [ (a; b) j a  [1 ; 2 ]; b  "list" ] = [(1
; `l'); (1 ; `i'); (1 ; `s'); (1 ; `t');

(2 ; `l'); (2 ; `i'); (2 ; `s'); (2 ; `t')]
```

Нотация понимания списка четко транслируется в операции монады.

```
[ t j x  u ] = u ? *x : unit t [ t j x  u; y  v ] = u ? *x : v ? *y: unit t
```

Здесь t - выражение, x и y - переменные (или более широкие части (patterns)), и u, и v - выражения, которые вычисляют в списках. Связи между пониманиями и монадами были описаны подробно в другом месте [21].

5.2 Представляющие синтаксические анализаторы.
Синтаксические анализаторы представлены таким же образом, как и состояния трансформаторов.

```
type M a = State ! [(a; State)] type State = String
```

То есть синтаксический анализатор для типа берет состояние, представляющее строку, которая анализируется, и возвращает список из того, что содержит значение типа анализируемой строки, и состояния, представляющего остающуюся строку, чтобы все же анализироваться. Список предоставляет все возможные альтернативные синтаксические

анализы входного состояния: будет пусто, если нет никакого синтаксического анализа, имеет один элемент, если есть один синтаксический анализ, имеет два элемента, если есть два различных возможных синтаксических анализа, и так далее.

Рассмотрим простой синтаксический анализатор для арифметических выражений, который возвращает дерево типа уже предварительно разобранное.

```
Cdata Term = Con Int j Div Term Term
```

Скажем, мы имеем синтаксический анализатор для таких термов.

term :: M Term. Вот некоторые примеры его использования.

```
term "23 " = [(Con 23 ; " ")] term "23 and more" = [(Con 23 ; " and more")] term "not
a term" = [ ] term "((1972 \Xi  2 ) \Xi  23 )" = [(Div (Div (Con 1972 ) (Con 2 ))
(Con 23 )); " ")]
```

Синтаксический анализатор m является однозначным, если для каждого ввода x список возможных синтаксических анализов m x является или пустым или имеет в точности один элемент. Например, терм однозначен. Неоднозначный синтаксический анализатор может возвратить список с двумя или больше альтернативными синтаксическими анализами.

5.3 Синтаксический анализ элемента.
Основной синтаксический анализатор возвращает первый элемент ввода, и терпит неудачу, если ввод истощен.

```
item :: M Char
item [ ] = [ ] item (a : x ) = [(a; x )]
```

Вот два примера.

```
item " " = [ ] item "monad" = [(`m'; "onad")]
```

Ясно, что элемент однозначен.

5.4 Последовательность
Чтобы сформировать синтаксические анализаторы в монаду, нам требуется операции unit и ?.

```
unit :: a ! M a unit a x = [(a; x )]

(?) :: M a ! (a ! M b) ! M b (m ? k ) x = [(b; z ) j (a; y)  m x ; (b; z )  k a y]
```

Синтаксический анализатор unit a принимает ввод x, и приводит к одному синтаксическому анализу со значением a, соединенным с остающимся вводом x. Синтаксический анализатор m ? k берет ввод x; синтаксический анализатор, для которого m применен, чтобы ввести x, приводящее к тому, что каждый анализирует значение соединенное с остающимся вводом y; тогда синтаксический анализатор k a применен к вводу y, приводящему к тому, что для каждого синтаксического анализа a значение b соединено с конечным сохранением вывода z.

Таким образом, unit соответствует пустому синтаксическому анализатору, который не использует никакого ввода, и ? соответствует последовательности синтаксических анализаторов.

Два элемента могут анализироваться следующим образом.

```
twoItems :: M (Char; Char) twoItems = item ? *a: item ? *b: unit (a; b)
```

Вот два примера.

```
twoItems "m" = [ ] twoItems "monad" = [((`m'; `o'); "nad")]
```

Синтаксический анализ успешен, только если есть не менее двух элементов в списке.

Три закона монад выражают, что пустой синтаксический анализатор идентичен последовательности, и что последовательность является ассоциативной.

```
unit a ? *b: n = n[a=b] m ? *a: unit a = m m ? (*a: n ? *b: o) = (m ? *a: n) ? *b: o
```

Если m однозначен, и k однозначен для каждого a, то m ? k также однозначен.

5.5 Чередование
Синтаксические анализаторы также могут быть объединены чередованием.

```
zero :: M a zero x = [ ]

(\Phi ) :: M a ! M a ! M a (m \Phi  n) x = m x ++ n x
```

Синтаксический анализатор zero берет ввод x и всегда терпит неудачу. Синтаксический анализатор m \Phi n берет ввод x и приводит к тому, что все синтаксические анализы m применяются к вводу x и все синтаксические анализы n, применяются к тому же самому вводу x.

Вот синтаксический анализатор, который анализирует один или два элемента ввода.

```
oneOrTwoItems :: M String oneOrTwoItems = (item ? *a: unit [a])

\Phi  (item ? *a: item ? *b: unit [a; b])
```

Вот три примера.

```
oneOrTwoItems " " = [ ] oneOrTwoItems "m" = [("m"; " ")] oneOrTwoItems "monad" =
[("m"; "onad"); ("mo"; "nad")]
```

Последние коды двух альтернативных синтаксических анализаторов показывают, что чередование может привести к неоднозначным синтаксическим анализаторам.

Синтаксический анализатор, который всегда терпит неудачу, идентичен чередованию, и чередование ассоциативно.

```
zero \Phi  n = n

m \Phi  zero = m m \Phi  (n \Phi  o) = (m \Phi  n) \Phi  o
```

Кроме того, zero - действительно нуль ?, и ? распределяется через \Phi.

```
zero ? k = zero m ? *a: zero = zero

(m \Phi  n) ? k = (m ? k) \Phi  (n ? k)
```

? распределяется справа через \Phi не только потому, что мы представляем альтернативные синтаксические анализы упорядоченным списком; если бы мы использовали неупорядоченное мультимножество, то m ? *a: (k \Phi h a) = (m ? k) \Phi (m ? h) также оставалось бы верным. Однозначный синтаксический анализатор в большинстве случаев приводит к списку длины один, так что порядок - несоответствующий, и, следовательно, этот закон, также верен всякий раз, когда любая сторона однозначна (что подразумевает также обе стороны).

5.6 Фильтрация.
Синтаксический анализатор может быть фильтрован путем объединения его с предикатом.

```
(\Lambda ) :: M a ! (a ! Bool) ! M a m \Lambda  p = m ? *a: if p a then unit a else
zero
```

Дан синтаксический анализатор m и предикат на значениях p, синтаксический анализатор m \Lambda p применяет синтаксический анализатор m, чтобы прийти к значению a; если p a держится, он успешен со значением a, иначе он терпит неудачу. Обратите внимание,

что фильтрация написана в терминах предварительно определенных операторов, и не нужно обращаться непосредственно к состоянию.

Позвольте isLetter и isDigit быть очевидными предикатами. Вот два синтаксических анализатора.

```
letter :: M Char letter = item \Lambda  isLetter

digit :: M Int digit = (item \Lambda  isDigit) ? *a: unit (ord a \Gamma  ord `0')
```

Первый успешен, только если следующий входной элемент - символ, а второй успешен, только если это - цифра. Второй также преобразует цифру к ее соответствующему значению, используя ord :: Char ! Int, чтобы преобразовать символ к его коду ASCII. Предположение, что \Lambda имеет более высокое старшинство чем ? позволило бы некоторым круглым скобкам быть пониженными из второго определения.

Синтаксический анализатор для a буквального распознает единственно выделенный символ.

```
lit :: Char ! M Char lit c = item \Lambda  (*a: a == c)
```

Литерал синтаксического анализатора c преуспевает, если ввод начинается с символа c, и терпит неудачу иначе.

```
lit `m' "monad" = [(`m'; "onad")] lit `m' "parse" = [ ]
```

Из предыдущих законов следует, что фильтрация сохраняет нуль (zero) и распределяется через чередование.

```
zero \Lambda  p = zero (m \Phi  n) \Lambda  p = (m \Lambda  p) \Phi  (n \Lambda  p)
```

Если m - однозначный синтаксический анализатор, то m \Lambda p.

5.7 Итерация.
Отдельный синтаксический анализатор может выполниться с помощью итераций, приводя к списку анализируемых значений.

```
iterate :: M a ! M [a] iterate m = (m ? *a: iterate m ? *x : unit (a : x ))

\Phi  unit [ ]
```

Дан синтаксический анализатор m, синтаксический анализатор выполняет итерации m, применяя синтаксический анализатор m последовательно нуль или большее количество раз, возвращая список всех анализируемых значений. В списке альтернативных синтаксических анализов, самый длинный синтаксический анализ возвращается сначала.

Вот пример.

```
iterate digit "23 and more" = [([2 ; 3 ]; " and more");

([2 ]; "3 and more"); ([ ]; "23 and more")]
```

Вот один способ проанализировать номер.

```
number :: M Int number = digit ? *a: iterate digit ? *x : unit (asNumber (a : x ))
```

Здесь asNumber берет список из одной или более цифр и возвращает соответствующий номер. Вот - пример.

```
number "23 and more" = [(23 ; " and more");

(2 ; "3 and more")]
```

Это дает два возможных синтаксических анализа: тот, который анализирует обе цифры, и тот, который анализирует только отдельную цифру. Номер по-определению содержит не менее одной цифры, так что нет никакого синтаксического анализа с нулевыми цифрами.

Как показывают последние примеры, часто более естественно проектировать итератор (iterator) только с целью привести к самому длинному возможному синтаксическому анализу. Следующий раздел описывает способ достигнуть этого.

5.8 Смещенный выбор.
Чередование, описанное как m \Phi n, приводит ко всем синтаксическим анализам, к которым приводит m, следующее за всеми синтаксическими анализами, к которым приводит n. Для некоторых целей, более разумно выбрать один или другой: все синтаксические анализы m, если есть любая возможность, и все синтаксические анализы n иначе. Это называют смещенным выбором.

```
(ff) :: M a ! M a ! M a (m ff n) x = if m x 6 == [ ] then m x else n x
```

Смещенный выбор, описанный как m ff n, приводит к тем же самым синтаксическим анализам, что и m, если m неудачен при получении любых синтаксических анализов, когда это приводит к тем же самым синтаксическим анализам, как и n.

Вот итерация, перезаписанная со смещенным выбором.

```
reiterate :: M a ! M [a] reiterate m = (m ? *a: reiterate m ? *x : unit (a : x ))

ff unit [ ]
```

Единственное различие - замена \Phi на ff. Вместо того чтобы приводить к списку всех возможных синтаксических анализов с наиболее длинным первым, теперь это приводит только к самому длинному возможному синтаксическому анализу.

Вот предыдущий пересмотренный пример.

```
reiterate digit "23 and more" = [([2 ; 3 ]; " and more")] Из чего следует, что номер взят, чтобы быть перезаписанным повторением (reiterate).

number :: M Int number = digit ? *a: reiterate digit ? *x : unit (asNumber (a : x ))
```

Вот пример, который показывает немного из того, как неоднозначные синтаксические анализаторы могут использоваться, чтобы искать пространство возможностей. Мы используем повторение, чтобы найти все способы взять один или два элемента из строки, нуль или большее количество раз.

```
reiterate oneOrTwoItems "many" = [(["m"; "a"; "n"; "y"]; " ");

(["m"; "a"; "ny"]; " "); (["m"; "an"; "y"]; " "); (["ma"; "n"; "y"]; " "); (["ma"; "ny"]; " ")]
```

Это объединяет чередование (в oneOrTwoItems) со смещенным выбором (в reiterate). Есть несколько возможных синтаксических анализов, но для каждого синтаксического анализа oneOrTwoItems был применен, пока полный ввод не был использован. Хотя этот пример несколько причудлив, подобная методика могла бы использоваться, чтобы найти все способы размена доллара на nickels, гривенники и четвертаки.

Если m и n однозначны, то m ff n и reiterate m также однозначны. Для однозначных синтаксических анализаторов последовательность распределяется направо через смещенный выбор:

```
(m ? k) ff (m ? h) = m ? *a: k a ff h a
```

всякий раз, когда m однозначен. В отличие от случая с чередованием, последовательность не распределяется влево через смещенный выбор даже для однозначных синтаксических анализаторов.

5.9 Синтаксический анализатор для термов

Теперь можно написать синтаксический анализатор для термов, ссылаясь на первоначальное. Вот грамматика для полностью заключенных в скобки термов, выраженных в НОРМАЛЬНОЙ ФОРМЕ БЕКУСА-НАУРА.

```
term ::= number j `(' term `\Xi ' term `)'
```

Это транслируется непосредственно в нашу нотацию следующим образом. Обратите внимание, что наша нотация, в отличие от НОРМАЛЬНОЙ ФОРМЫ БЕКУСА-НАУРА, определяет точно, как создать возвращенное значение.

```
term :: M Term term = (number ? *a:

unit (Con a)) \Phi  (lit `(' ? * :

term ? *t: lit `\Xi ' ? * : term ? *u: lit `)' ? * : unit (Div t u))
```

(Здесь * : e эквивалентно *x: e, где x - некоторая новая переменная, которая не появляется в e; это указывает на то, что значение, связанное лямбда-выражением не представляет интерес.) Примеры использования этого синтаксического анализатора показывались ранее.

Вышеупомянутый синтаксический анализатор написан с чередованием, но поскольку оно однозначно, оно, возможно, точно также могло быть написано со смещенным выбором. То же самое истинно для всех синтаксических анализаторов в следующем разделе.

5.10 Левая рекурсия

Вышеупомянутый синтаксический анализатор работает только для полностью заключенных в скобки термов. Если мы позволяем незаключенные в скобки термы, то оператор \Xi должен связаться слева. Обычный способ выразить такую грамматику в НОРМАЛЬНОЙ ФОРМЕ БЕКУСА-НАУРА следующий.

```
term ::= term `\Xi ' factor j factor factor ::= number j `(' term `)'
```

Это транслируется в нашу нотацию следующим образом.

```
term :: M Term term = (term ? *t:

lit `\Xi ' ? * : factor ? *u: unit (Div t u)) \Phi  factor

factor :: M Term factor = (number ? *a:

unit (Con a)) \Phi  (lit `(' ? * :

term ? *t: lit `)' ? * : unit t)
```

Нет никакой проблемы с фактором, но любая попытка применить терм приводит к бесконечному циклу. Проблема состоит в том, что первый шаг терма должен применить терм и продвинуться
к бесконечному регрессу. Это называют проблемой левой рекурсии. Это трудность для всех рекурсивных синтаксических анализаторов спуска, функциональных или иных.

Решение состоит в том, чтобы перезаписать грамматику для терма в следующей эквивалентной форме.

```
term ::= factor term0

term0 ::= `\Xi ' factor term0 j unit
```

где, как обычно, unit обозначает пустой синтаксический анализатор. Тогда он транслируется непосредственно в нашу нотацию.

```
term :: M Term
```

```
term = factor ? *t: term0 t

term0 :: Term ! M Term term0 t = (lit `\Xi ' ? * :

factor ? *u: term0 (Div t u)) \Phi  unit t
```

Здесь term0 анализирует остаток от терма; требуется параметр, соответствующий анализируемому терму и т.д.

Это приносит желаемый эффект.

```
term "1972 \Xi  2 \Xi  23 " = [((Div (Div (Con 1972 ) (Con 2 )) (Con 23 )); " ")]
term "1972 \Xi  (2 \Xi  23 )" = [((Div (Con 1972 ) (Div (Con 2 ) (Con 23 ))); " ")]
```

Вообще, леворекурсивное определение

```
m = (m ? k ) \Phi  n
```

может быть перезаписано как

```
m = n ? (closure k)
```

где

```
closure :: (a ! M a) ! (a ! M a)

closure k a = (k a ? closure k ) \Phi  unit a
```

Here m :: M a, n :: M a, and k :: a ! M a.

5.11 Ленивые улучшения.

Как правило, программа могла бы быть представлена как функция от списка символов - ввода - к другому списку символов - выводу. При ленивом вычислении, обычно только часть ввода должна быть прочитана прежде, чем первая часть списка вывода произведена. Такое 'на линии' (on-line) поведение является необходимым для некоторых целей.

Вообще, неблагоразумно ожидать такого поведения от синтаксического анализатора, так как вообще не может быть известно, что ввод успешно анализируется, пока все из него не прочтено. Однако в некоторых специальных случаях можно надеяться добиться большего успеха.

Предполагается применение повторения m к строке, начинающейся с образца m. В этом случае, синтаксический анализ не может терпеть неудачу: независимо от остатка строки, можно было бы ожидать синтаксический анализ, который приводит к списку, начинающемуся с анализируемого значения. Под ленивой оценкой можно было бы ожидать наличия способности генерировать вывод, соответствующий первой цифре, прежде чем остающийся ввод будет прочтен.

Но это - не то, что есть на самом деле: синтаксический анализатор читает ввод полностью прежде, чем любой вывод будет сгенерирован. То, что является необходимым - это некоторый способ кодировать то, что синтаксический анализатор reiterate m всегда успешен. (Даже если начало ввода не соответствует m, это будет восприниматься как значение пустого списка.) Это обеспечено функцией guarantee.

```
guarantee :: M a ! M a guarantee m x = let u = m x in (fst (head u); snd (head u)) :
tail u
```

Здесь fst (a; b) = a; snd (a; b) = b; head (a : x) = a, и tail (a : x) = x .

А теперь reiterate с добавлением guarantee.

```
reiterate :: M a ! M [a] reiterate m = guarantee ( (m ? *a: reiterate m ? *x : unit
(a : x ))

ff unit [ ])
```

Это гарантирует, что reiterate m и все его рекурсивные обращения возвращают список с не менее чем одним ответом. В результате, поведение при ленивой оценке существенно улучшено.

Предыдущее объяснение является высоко эксплуатируемым и стоит отметить, что денотационная семантика обеспечивает полезный альтернативный подход. Позвольте ? обозначить программу, которая не заканчивается. Можно проверить это со старым определением
reiterate digit (`1' : ?) = ? в то время как с новым определением
reiterate digit (`1' : ?) = ((`1' : ?); ?) : ? Таким образом, при условии, что ввод начинается с символа '1', но остаток от ввода является неизвестным, со старым определением ничего не известно о выводе, в то время как с новым определением известно, что вывод приводит не менее чем к одному синтаксическому анализу, значение которого - список, который начинается с символа '1'.

Другие синтаксические анализаторы могут также извлечь выгоду из разумного использования guarantee, и в особенности iterate может быть изменено наподобие reiterate.

5.12 Заключение.
Мы видели, что монады обеспечивают полезную структуру для того, чтобы структурировать рекурсивные синтаксические анализаторы спуска. Пустой синтаксический анализатор (empty parser) и последовательность (sequencing) соответствуют непосредственно unit и ?, и законы монад отражают, что последовательность ассоциативна и имеет пустой синтаксический анализатор как модуль. Синтаксический анализатор неудачи (failing parser) и чередование (alternation) соответствуют zero и \Phi, которые удовлетворяют законам, отражающим, что чередование является ассоциативным и имеет синтаксический анализатор неудачи как модуль, и что последовательность распределяется через чередование.

Последовательность и чередование фундаментальны не только для синтаксических анализаторов, но и для большей части компьютерных наук. Если монады фиксируют последовательность, то разумно спросить: что объединяет вместе последовательность и чередование? Может быть unit, ?, zero, и \Phi, вместе с вышеназванными законами обеспечивают такую структуру. Необходимы дальнейшие эксперименты. Есть одна обнадёживающая вещь - то, что небольшое изменение монады синтаксического анализатора приводит к вероятной модели охраняемого командного языка Диджкстры (Dijkstra's guarded command language).

Ссылки (если нужно, я их допереведу)

1. S. Abramsky и C. Hankin, Абстрактная Интерпретация Декларативных Языков, Эллис Хорвуда, 1987.
2. A. Bloss, анализ Обновления и эффективное выполнение функциональных совокупностей. В 4'th Симпозиум по Функциональным Языкам программирования и Компьютерной Архитектуре, ACM, Лондон, сентябрь 1989.
3. R. Bird и P. Wadler, Введение в Функциональное Программирование. Прентис Хол, 1987.
4. P. Hudak, S. Пеитон Джоунс и P. Wadler, редакторы, Сообщение относительно Языка программирования Haskell: Версия 1.1. Техническое сообщение, Йельский Университет Университета и Глазго, август 1991.
5. J.-Y. Girard, Линейная логика. Теоретическая Информатика, 50:1-102, 1987.
6. J. Guzm'an и P. Hudak, Единственное-переплетенное полиморфное исчисление лямбды. На Симпозиуме IEEE по Логике в Информатике, Филадельфии, июне 1990.
7. P. Hudak, семантическая модель подсчета ссылки и его абстракции (детальное резюме). На Конференции ACM по Шепелявят и Функциональное Программирование, стр 351-363, Кембридж, Штат Массачусетс, август 1986.
8. P. Hudak, изменчивые абстрактные типы данных на основе продолжения, или как иметь ваше состояние и munge это также. Техническое сообщение YALEU/DCS/RR-914, Отдел Информатики, Йельского Университета, июля 1992.
9. D. Король и P. Wadler, Комбинируя монады. В Симпозиуме Глазго по Функциональному Программированию, Эре, июль 1992. Симпозиумы в Вычислении Ряда, Springer Verlag, для представления.
10. S. Mac Lane (Переулок Макинтоша), Категории для Рабочего Математика, Спринджер-Верлага, 1971.
11. R. Milner, M. Tofte, и R. Harper, определение Standard ML (Стандартного СТАКАНА). Пресс MIT, 1990.
12. L. C. Paulson, ML для Рабочего Программиста. Пресс Университета Кембриджа, 1991.
13. E. Moggi, Вычислительное лямбда-исчисление и монады. На Симпозиуме по Логике в Информатике, Asilomar, Калифорния; IEEE, июнь 1989. (Более длинная версия доступна как техническое сообщение от Университета Эдинбурга.)
14. E. Moggi, абстрактное представление языков программирования. Примечания курса, Университет Эдинбурга.
15. S. L. Пеитон Джоунс и P. Wadler, Обязательное функциональное программирование. В 20'th Симпозиум по Принципам Языков программирования, Чарлстона, Южная Каролина; ACM, январь 1993.
16. Г. Plotkin, Вызов по имени, вызов по значению, и *-исчисление. Теоретическая Информатика, 1:125-159, 1975.
17. J. Rees и W. Clinger (редакторы)., revised3 сообщают относительно алгоритмического языка Scheme. ACM SIGPLAN Уведомления, 21 (12):37-79, 1986.
18. D. Шмидт, Обнаруживая глобальные переменные в деописательных спецификациях. Сделка ACM на Языках программирования и Системах, 7:299-310, 1985.
19. V. Swarup, США. Reddy, и E. Ireland, Назначения для применимых языков. На Конференции по Функциональным Языкам программирования и Компьютерной Архитектуре, Кембриджу, Штат Массачусетс; LNCS 523, Springer Verlag, август 1991.
20. D. A. Turner, краткий обзор Miranda. В D. A. Turner, редактор, Темы Исследования в Функциональном Программировании. Аддизон Уэсли, 1990.
21. P. Wadler, Постигая монады. На Конференции по Lisp (Шепелявят) и Функциональному Программированию, Nice, Франция; ACM, июнь 1990.
22. Есть использование для линейной логики? Конференция по Частичной Оценке и Манипуляции Программы SemanticsBased (PEPM), Нью-Хейвен, Штат Коннектикут; ACM, июнь 1991.
23. P. Wadler, сущность функционального программирования (приглашенный разговор). В 19'th Симпозиум по Принципам Языков программирования, Альбукерке, Нью-Мексико; ACM, январь 1992.