Функциональное программирование

1 Введение

Программы на традиционных языках программирования, таких как Си, Паскаль, Java и т.п. состоят их последовательности модификаций значений некоторого набора переменных, который называется *состоянием*. Если не рассматривать операции ввода-вывода, а также не учитывать того факта, что программа может работать непрерывно (т.е. без остановок, как в случае серверных программ), можно сделать следующую абстракцию. До начала выполнения программы состояние имеет некоторое начальное значение σ_0, в котором представлены входные значения программы. После завершения программы состояние имеет новое значение σ', включающее в себя то, что можно рассматривать как «результат» работы программы. Во время исполнения каждая команда изменяет состояние; следовательно, состояние проходит через некоторую конечную последовательность значений:

$$\sigma = \sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n = \sigma'$$

Состояние модифицируется с помощью команд *присваивания*, записываемых в виде `v=E` или `v:=E`, где `v` — переменная, а `E` — некоторое выражение. Эти команды следуют одна за другой; операторы, такие как `if` и `while`, позволяют изменить порядок выполнения этих команд в зависимости от текущего значения состояния. Такой стиль программирования называют *императивным* или *процедурным*.

Функциональное программирование представляет парадигму, в корне отличную от представленной выше модели. Функциональная программа представляет собой некоторое выражение (в математическом смысле); выполнение программы означает вычисление значения этого выражения.[1] С учетом приведенных выше обозначений, считая что результат работы

[1]Употребление термина «вычисление» не означает, что программа может оперировать только с числами; результатом вычисления могут оказаться строки, списки и вообще, любые допустимые в языке структуры данных.

императивной программы полностью и однозначно определен ее входом, можно сказать, что финальное состояние (или любое промежуточное) представляет собой некоторую функцию (в математическом смысле) от начального состояния, т.е. $\sigma' = f(\sigma)$. В функционально программировании используется именно эта точка зрения: программа представляет собой выражение, соответствующее функции f. Функциональные языки программирования поддерживают построение таких выражений, предоставляю широкий выбор соответствующих языковых конструкций.

При сравнении функционального и императивного подхода к программированию можно заметить следующие свойства функциональных программ:

- Функциональные программы не используют переменные в том смысле, в котором это слово употребляется в императивном программировании. В частности в функциональных программах не используется оператор присваивания.

- Как следствие из предыдущего пункта, в функциональных программах нет циклов.

- Выполнение последовательности команд в функциональной программе бессмысленно, поскольку одна команда не может повлиять на выполнение следующей.

- Функциональные программы используют функции гораздо более замысловатыми способами. Функции можно передавать в другие функции в качестве аргументов и возвращать в качестве результата, и даже в общем случае проводить вычисления, результатом которого будет функция.

- Вместо циклов функциональные программы широко используют рекурсивные функции.

На первый взгляд функциональный подход к программированию может показаться странным, непривычным и мало полезным, однако необходимо принять во внимание следующие соображения.

Прежде всего, императивный стиль в программировании не является жестко заданной необходимостью. Многие характеристики императивных языков программирования являются результатом абстрагирования от низкоуровневых деталей реализации компьютера, от машинных кодов к языкам ассемблера, а затем к языкам типа Фортрана и т.д. Однако нет причин полагать, что такие языки отражают наиболее естественный для

человека способ сообщить машине о своих намерениях. Возможно, более правилен подход, при котором языки программирования рождаются как абстрактные системы для записи алгоритмов, а затем происходит их перевод на императивный язык компьютера.

Далее, функциональный подход имеет ряд преимуществ перед императивным. Прежде всего, функциональные программы более непосредственно соответствуют математическим объектам, и следовательно, позволяют проводить строгие рассуждения. Установить значение императивной программы, т.е. той функции, вычисление которой она реализует, может оказаться довольно трудно. Напротив, значение функциональной программы может быть выведено практически непосредственно.

Например, рассмотрим следующую программу на языке Haskell:

```
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

Практически сразу видно, что эта программа соответствует следующей частичной функции:

$$f(n) = \begin{cases} n! & n \geq 0 \\ \bot & n < 0 \end{cases}$$

(Здесь символ $\bot$ означает неопределенность функции, поскольку при отрицательных значениях аргумента программа не завершается.) Однако для программы на языке Си это соответствие не очевидно:

```
int f (int n)
{
    int x = 1;
    while (n > 0)
    {
        x = x * n;
        n = n - 1;
    }
    return x;
}
```

Следует также сделать замечание относительно употребления термина «функция» в таких языках как Си, Java и т.п. В математическом смысле «функции» языка Си не являются функциями, поскольку:

- Их значение может зависеть не только от аргументов;
- Результатом их выполнения могут быть разнообразные *побочные эффекты* (например, изменение значений глобальных переменных)

- Два вызова одной и той же функции с одними и теми же аргументами могут привести к различным результатам.

Вместе с тем функции в функциональных программах действительно являются функциями в том смысле, в котором это понимается в математике. Соответственно, те замечания, которые были сделаны выше, к ним не применимы. Из этого следует, что вычисление любого выражения не может иметь никаких побочных эффектов, и значит, порядок вычисления его подвыражений не оказывает влияния на результат. Таким образом, функциональные программы легко поддаются распараллеливанию, поскольку отдельные компоненты выражений могут вычисляться одновременно.

2 Основы лямбда-исчисления

Подобно тому, как теория машин Тьюринга является основой императивных языков программирования, лямбда-исчисление служит базисом и математическим «фундаментом», на котором основаны все функциональные языки программирования.

Лямбда-исчисление было изобретено в начале 30-х годов логиком А. Черчем, который надеялся использовать его в качестве формализма для обоснования математики. Вскоре были обнаружены проблемы, делающие невозможным его использование в этом качестве (сейчас есть основания полагать, что это не совсем верно) и лямбда-исчисление осталось как один из способов формализации понятия алгоритма.

В настоящее время лямбда-исчисление является основной из таких формализаций, применяемой в исследованиях связанных с языками программирования. Связано это, вероятно, со следующими факторами:

- Это единственная формализация, которая, хотя и с некоторыми неудобствами, действительно может быть непосредственно использована для написания программ.

- Лямбда-исчисление дает простую и естественную модель для таких важных понятий, как рекурсия и вложенные среды.

- Большинство конструкций традиционных языков программирования может быть более или менее непосредственно отображено в конструкции лямбда-исчисления.

- Функциональные языки являются в основном удобной формой синтаксической записи для конструкций различных вариантов лямбда-исчисления. Некоторые современные языки (Haskell, Clean) имеют

100% соответствие своей семантики с семантикой подразумеваемых конструкций лямбда-исчисления.

В математике, когда необходимо говорить о какой-либо функции, принято давать этой функции некоторое имя и впоследствии использовать его, как, например, в следующем утверждении:

Пусть $f\colon \mathbb{R} \to \mathbb{R}$ определяется следующим выражением:

$$f(x) = \begin{cases} 0, & x = 0 \\ x^2 \sin(1/x^2), & x \neq 0 \end{cases}$$

Тогда $f'(x)$ не интегрируема на интервале $[0, 1]$.

Многие языки программирования также допускают определение функций только с присваиванием им некоторых имен. Например, в языке Си функция всегда должна иметь имя. Это кажется естественным, однако поскольку в функциональном программировании функции используются повсеместно, такой подход может привести к серьезным затруднениям. Представьте себе, что мы должны всегда оперировать с арифметическими выражениями в подобном стиле:

Пусть $x = 2$ и $y = 4$. Тогда $xx = y$.

Лямбда-нотация позволяет определять функции с той же легкостью, что и другие математические объекты. *Лямбда-выражением* будем называть конструкцию вида

$$\lambda x.E$$

где E — некоторое выражение, возможно, использующее переменную x.

Пример. $\lambda x.x^2$ представляет собой функцию, возводящую свой аргумент в квадрат.

Использование лямбда-нотации позволяет четко разделить случаи, когда под выражением вида $f(x)$ мы понимаем саму функцию f и ее значение в точке x. Кроме того, лямбда-нотация позволяет формализовать практически все виды математической нотации. Если начать с констант и переменных и строить выражения только с помощью лямбда-выражений и применений функции к аргументам, то можно представить очень сложные математические выражения.

Применение функции f к аргументу x мы будем обозначать как $f\,x$, т.е., в отличие от того, как это принято в математике, не будем использовать скобки[2]. По причинам, которые станут ясны позднее, будем считать, что применение функции к аргументу ассоциативно влево, т.е. $f\,x\,y$

[2]Заметим, что и в математике такие выражения, как $\sin x$ записываются без скобок.

означает $(f(x))(y)$. В качестве сокращения для выражений вида $\lambda x.\lambda y.E$ будем использовать запись $\lambda x\, y.E$ (аналогично для большего числа аргументов). Также будем считать, что «область действия» лямбда-выражения простирается вправо насколько возможно, т.е., например, $\lambda x.x\, y$ означает $\lambda x.(x\, y)$, а не $(\lambda x.x)y$.

На первый взгляд кажется, что нам необходимо ввести специальное обозначение для функций нескольких аргументов. Однако существует операция *каррирования*[3], позволяющая записать такие функции в обычной лямбда-нотации. Идея заключается в том, чтобы использовать выражения вида $\lambda x\, y.x + y$. Такое выражение можно рассматривать как функцию $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, т.е. если его применить к одному аргументу, результатом будет функция, которая затем принимает другой аргумент. Таким образом:

$$(\lambda x\, y.x + y)\, 1\, 2 = (\lambda y.1 + y)\, 2 = 1 + 2.$$

Переменные в лямбда-выражениях могут быть *свободными* и *связанными*. В выражении вида $x^2 + x$ переменная x является свободной; его значение зависит от значения переменной x и в общем случае ее нельзя переименовать. Однако в таких выражениях как $\sum_{i=1}^{n} i$ или $\int_0^x \sin y\, dy$ переменные i и y являются связанными; если вместо i везде использовать обозначение j, значение выражения не изменится.

Следует понимать, что в каком-либо подвыражении переменная может быть свободной (как в выражении под интегралом), однако во всем выражении она связана какой-либо *операцией связывания переменной*, такой как операция суммирования. Та часть выражения, которая находится «внутри» операции связывания, называется *областью видимости* переменной.

В лямбда исчислении выражения $\lambda x.E[x]$ и $\lambda y.E[y]$ считаются эквивалентными (это называется α-эквивалентностью, и процесс преобразования между такими парами называют α-преобразованием). Разумеется, необходимо наложить условие, что y не является свободной переменной в $E[x]$.

[3]от фамилии известного логика Хаскелла Карри, в честь которого назван язык программирования Haskell

3 Лямбда-исчисление как формальная система

Лямбда-исчисление основано на формальной нотации лямбда-терма, составляемого из переменных и некоторого фиксированного набора констант с использованием операции применения функции и лямбда-абстрагирования. Сказанное означает, что все лямбда-выражения можно разделить на четыре категории:

1. **Переменные:** обозначаются произвольными строками, составленными из букв и цифр.

2. **Константы:** также обозначаются строками; отличие от переменных будем определять из контекста.

3. **Комбинации:**, т.е. применения функции S к аргументу T; и S и T могут быть произвольными лямбда-термами. Комбинация записывается как $S\,T$.

4. **Абстракции** произвольного лямбда-терма S по переменной x, обозначаемые как $\lambda x.S$.

Таким образом, лямбда-терм определяется рекурсивно и его грамматику можно определить в виде следующей формы Бэкуса-Наура:

$$\mathit{Exp} = \mathit{Var} \mid \mathit{Const} \mid \mathit{Exp}\ \mathit{Exp} \mid \lambda\ \mathit{Var}\ .\ \mathit{Exp}$$

В соответствие с этой грамматикой лямбда-термы представляются в виде синтаксических деревьев, а не в виде последовательности символов. Отсюда следует, что соглашения об ассоциативности операции применения функции, эквивалентность выражений вида $\lambda x\,y.S$ и $\lambda x.\lambda y.S$, неоднозначность в именах констант и переменных проистекают только из необходимости представления лямбда-термов в удобном человеку виде, и не являются частью формальной системы.

3.1 Свободные и связанные переменные

В данном разделе мы формализуем данное ранее интуитивное представление о свободных и связанных переменных. Множество свободных

переменных $FV(S)$ лямбда-терма S можно определить рекурсивно следующим образом:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(c) &= \emptyset \\ FV(S\,T) &= FV(S) \cup FV(T) \\ FV(\lambda x.S) &= FV(S) \backslash \{x\} \end{aligned}$$

Аналогично множество связанных переменных $BV(S)$ определяется следующими формулами:

$$\begin{aligned} BV(x) &= \emptyset \\ BV(c) &= \emptyset \\ BV(S\,T) &= BV(S) \cup BV(T) \\ BV(\lambda x.S) &= BV(S) \cup \{x\} \end{aligned}$$

Здесь предполагается, что c — некоторая константа.

Пример. Для терма $S = (\lambda x\, y.x)\ (\lambda x.z\ x)$ можно показать, что $FV(S) = \{z\}$ и $BV(S) = \{x, y\}$.

3.2 Подстановки

Интуитивно ясно, что применение терма $\lambda x.S$ как функции к аргументу T дает в результате терм S, в котором все свободные вхождения переменной x заменены на T. Как ни странно, формализовать это интуитивное представление оказывается нелегко.

Будем обозначать операцию подстановки терма S вместо переменной x в другом терме T как $T[x := S]$. Также, как и в определение свободных и связанных переменных, правила подстановки также можно определить рекурсивно. Трудность состоит в том, что необходимо наложить дополнительные ограничения, позволяющие избегать конфликта в именах переменных.

$$\begin{aligned} x[x := T] &= T \\ y[x := T] &= y, \text{ если } x \neq y \\ c[x := T] &= c \\ (S_1\ S_2)[x := T] &= S_1[x := T]\ S_2[x := T] \\ (\lambda x.S)[x := T] &= \lambda x.S \\ (\lambda y.S)[x := T] &= \lambda y.(S[x := T]), \text{ если } x \neq y \text{ и } x \notin FV(S), \text{ либо } y \notin FV(T) \\ (\lambda y.S)[x := T] &= \lambda z.(S[y := z][x := T]) \text{ иначе, где } z \notin FV(S) \cup FV(T) \end{aligned}$$

3.3 Конверсия

Лямбда-исчисление основано на трех операциях конверсии, которые позволяют переходить от одного терма к другому, эквивалентному ему. По сложившейся традиции эти конверсии обозначают греческими буквами α, β и η. Они определяются следующим образом:

- α-конверсия: $\lambda x.S \xrightarrow{\alpha} \lambda y.S[x := y]$ при условии, что $y \notin FV(S)$. Например, $\lambda u.u\ v \xrightarrow{\alpha} \lambda w.w\ u$.
- β-конверсия: $(\lambda x.S)\ T \xrightarrow{\beta} S[x := T]$.
- η-конверсия: $\lambda x.T\ x \xrightarrow{\eta} T$, если $x \notin FV(T)$. Например, $\lambda u.v\ u \xrightarrow{\eta} v$.

Для нас наиболее важна β-конверсия, поскольку она соответствует вычислению значения функции от аргумента. α-конверсия является вспомогательным механизмом для того, чтобы изменять имена связанных переменных, а η-конверсия интересна в основном при рассмотрении лямбда-исчисления с точки зрения логики, а не программирования.

3.4 Равенство лямбда-термов

Используя введенные правила конверсии, можно формально определить понятие равенства лямбда-термов. Два терма равны, если от одного из них можно перейти к другому с помощью конечной последовательности конверсий. Определим понятие равенства следующими выражениями, в которых горизонтальные линии следует понимать как «если утверждение над чертой выполняется, то выполняется и утверждение

под ней»:

$$\frac{S \xrightarrow{\alpha} T \text{ или } S \xrightarrow{\beta} T \text{ или } S \xrightarrow{\eta} T}{S = T}$$

$$\frac{}{T = T}$$

$$\frac{S = T}{T = S}$$

$$\frac{S = T \text{ и } T = U}{S = U}$$

$$\frac{S = T}{S\ U = T\ U}$$

$$\frac{S = T}{U\ S = U\ T}$$

$$\frac{S = T}{\lambda x.S = \lambda x.T}$$

Следует отличать понятие равенства, определяемое этими формулами, от понятия синтаксической эквивалентности, которую мы будем обозначать специальным символом $\equiv$. Например, $\lambda x.x \not\equiv \lambda y.y$, но $\lambda x.x = \lambda y.y$. Часто можно рассматривать синтаксическую эквивалентность термов с точностью до α-конверсий. Такую эквивалентность будем обозначать символом $\equiv_\alpha$. Это отношение определяется так же, как равенство лямбда-термов, за тем исключением, что из всех конверсий допустимы только α-конверсии. Таким образом, $\lambda x.x \equiv_\alpha \lambda y.y$.

3.5 Экстенсиональность

η-конверсия в лямбда-исчислении выражает собой принцип *экстенсиональности*. В общефилософском смысле два свойства называются экстенсионально эквивалентными, если они принадлежат в точности одним и тем же объектам. В математике, например, принят экстенсиональный взгляд на множества, т.е. два множества считаются одинаковыми, если они содержат одни и те же элементы. Аналогично мы говорим, что две функции равны, если они имеют одну и ту же область определения, и для любых значений аргумента из этой области определения вычисляют один и тот же результат.

Вследствие наличия η-конверсии определенное нами выше отношение равенства лямбда-термов экстенсионально. Действительно, если $f\ x$ и $g\ x$ равны для любого x, то в частности $f\ y = g\ y$, где y не является свободной переменной ни в f, ни в g. Следовательно, по последнему правилу в определении равенства лямбда-термов, имеем $\lambda y.f\ y = \lambda y.g\ y$. Теперь, если несколько раз применить η-конверсию, можно получить, что $f = g$. И обратно, экстенсиональность дает то, что каждое применение η-конверсии действительно приводит к равенству, поскольку по правилу β-конверсии $(\lambda x.T\ x)\ y = T\ y$ для любого y, если x не свободна в T.

3.6 Редукция лямбда-термов

Отношение равенства лямбда-термов, разумеется, симметрично. Хотя оно хорошо отражает понятие эквивалентности лямбда-термов, с вычислительной точки зрения более интересно будет рассмотреть асимметричный вариант. Определим отношение редукции (обозначаемое символом $\rightarrow$) следующим образом:

$$\frac{S \xrightarrow{\alpha} T \text{ или } S \xrightarrow{\beta} T \text{ или } S \xrightarrow{\eta} T}{S \rightarrow T}$$

$$\frac{}{T \rightarrow T}$$

$$\frac{S \rightarrow T \text{ и } T \rightarrow U}{S \rightarrow U}$$

$$\frac{S \rightarrow T}{S\ U \rightarrow T\ U}$$

$$\frac{S \rightarrow T}{U\ S \rightarrow U\ T}$$

$$\frac{S \rightarrow T}{\lambda x.S \rightarrow \lambda x.T}$$

Несмотря на то, что термин «редукция» подразумевает уменьшение размера лямбда-терма, в действительности это может быть не так, что

показывает следующий пример:

$$\begin{aligned}(\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x) &\rightarrow (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\\ &\rightarrow (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\\ &\rightarrow \ldots\end{aligned}$$

Тем не менее отношение редукции соответствует систематическим попыткам вычислить терм, последовательно вычисляя комбинации $f(x)$, где f-некоторая лямбда-абстракция. Когда для терма невозможно сделать никакую редукцию, за исключением α-преобразования, будем говорить, что терм находится в *нормальной форме*.

3.7 Редукционные стратегии

Давайте отвлечемся от теоретических рассуждений и вспомним важность рассматриваемых вопросов для функционального программирования. Функциональная программа представляет собой *выражение*, и выполнение ее означает вычисление этого выражения. Употребляя введенные термины, можно сказать, что мы начинаем с соответствующего терма и последовательно применяем редукции до тех пор, пока это возможно. Но какую же именно редукцию применять на каждом конкретном шаге? Отношение редукции не детерминистично, т.е. для некоторого терма t существует несколько различных t_i, таких что $t \rightarrow t_i$. Иногда выбор между ними означает выбор между конечной и бесконечной последовательности редукций, т.е. между завершением и зацикливанием программы. Например, если мы начнем редукцию с самого внутреннего *редекса*[4] в следующем примере, мы получим бесконечную последовательность редукций:

$$\begin{aligned}&(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x))\\ \rightarrow\ &(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x))\\ \rightarrow\ &(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x))\\ \rightarrow\ &\ldots\end{aligned}$$

Однако если мы начнем с самого внешнего редекса, то мы сразу получим:

$$(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)) \rightarrow y,$$

и больше нельзя применить никакую редукцию.

Следующая теорема утверждает, что наблюдение, сделанное в предыдущем примере, верно и в более общем смысле.

[4]От англ. redex (REDucible EXpression)

Теорема 1. *Если $S \to T$, где T находится в нормальной форме, то последовательность редукций, начинающаяся с S, и заключающаяся в том, что для редукции всегда выбирается самый левый и самый внешний редекс, гарантированно завершится и приведет терм в нормальную форму.*

Понятие «самого левого и самого внешнего» редекса можно определить индуктивно: для терма $(\lambda x.S)\ T$ это будет сам терм; для любого другого терма $S\ T$ это будет самый левый и самый внешний редекс в S, и для абстракции $\lambda x.S$ это будет самый левый самый внешний редекс в S. В терминах нашего конкретного синтаксиса мы всегда редуцируем редекс, чей символ λ находится левее всего.

Следующая теорема, известная как теорема Черча–Россера, утверждает, что если мы начнем с терма T и проведем две произвольные конечные последовательности редукций, всегда будут существовать еще две последовательности редукций, которые приведут нас к одному и тому же терму (хотя он может и не находиться в нормальной форме).

Теорема 2. *Если $t \to s_1$ и $t \to s_2$, то существует терм u, такой, что $s_1 \to u$ и $s_2 \to u$.*

Эта теорема имеет несколько важных следствий.

Следствие 1. *Если $t_1 = t_2$, то существует терм u такой, что $t_1 \to u$ и $t_2 \to u$.*

Следствие 2. *Если $t = t_1$ и $t = t_2$, где t_1 и t_2 находятся в нормальной форме, то $t_1 \equiv_\alpha t_2$, т.е. t_1 и t_2 равны с точностью до переименования переменных.*

Следовательно, нормальная форма, если она существует, единственна с точностью до до α-конверсии.

С вычислительной точки зрения это означает следующее. В некотором смысле, стратегия редуцирования самого левого самого внешнего редекса (будем называть ее *нормализованной стратегией*) является наилучшей, поскольку она приведет к результату, если он достижим с помощью какой-либо стратегии. С другой стороны, *любая* завершающаяся последовательность редукций всегда приводит к одному и тому же результату. Более того, никогда не поздно прекратить выполнять редукции по заданной стратегии и вернуться к нормализованной стратегии.

4 Комбинаторы

Теория комбинаторов была разработана еще до создания лямбда-исчисления, однако мы будем рассматривать ее в терминах, введенных нами ранее. *Комбинатором* будем называть лямбда-терм, не содержащий свободных переменных. Такой терм является *замкнутым*; он имеет фиксированный смысл независимо от значения любых переменных.

В теории комбинаторов установлено, что с помощью несколько базовых комбинаторов и переменных можно выразить любой терм без применения операции лямбда-абстракции. В частности, замкнутый терм можно выразить только через эти базовые комбинаторы. Определим эти комбинаторы следующим образом:

$$\begin{aligned} I &= \lambda x.x \\ K &= \lambda x\ y.x \\ S &= \lambda f\ g\ x.f\ x\ (g\ x) \end{aligned}$$

I является функцией идентичности, которая оставляет свой аргумент неизменным. K служит для создания постоянных (константных) функций: применив его к аргументу a, получим функцию $\lambda x.a$, которая возвращает a независимо от переданного ей аргумента. Комбинатор S является «разделяющим»: он берет две функции и аргумент и «разделяет» аргумент между функциями.

Теорема 3. *Для любого лямбда-терма t существует терм t', не содержащий лямбда-абстракций и составленный из комбинаторов S, K, I и переменных, такой что $FV(t') = FV(t)$ и $t' = t$.*

Эту теорему можно усилить, поскольку комбинатор I может быть выражен в терминах S и K. Действительно, для любого A выполняется:

$$\begin{aligned} S\ K\ A\ x &= K\ x\ (A\ x) \\ &= (\lambda y.x)\ (A\ x) \\ &= x \end{aligned}$$

Применяя η-конверсию, получаем, что $I = S\ K\ A$ для любого A. По причинам, которые станут ясны в дальнейшем, мы будем использовать $A = K$. Таким образом, $I = S\ K\ K$, и в дальнейшем символ I можно исключать из выражений, составленных из комбинаторов.

Хотя мы представили комбинаторы как некоторые лямбда-термы, можно разработать теорию, в которой они являются базовым понятием. Также, как и в лямбда-исчислении, можно начать с формального синтаксиса,

вместо лямбда-абстракций содержащего комбинаторы. Вместо правил α, β и η-конверсии, можно ввести правила конверсии для выражений, содержащих комбинаторы, например $K\ x\ y \to x$. Как независимая теория, теория комбинаторов обладает многими аналогиями с лямбда-исчислением, в частности, для нее выполняется теорема Черча–Россера. Однако эта теория менее интуитивно понятно, поскольку выражения с комбинаторами могут быстро становиться сложными и запутанными.

Комбинаторы имеют не только теоретический интерес. Как мы уже говорили, и как станет ясно из последующих глав, лямбда-исчисление может рассматриваться как простой функциональный язык программирования, составляющий ядро настоящих языков, таких как ML или Haskell. Тогда можно сказать, что приведенная выше теорема показывает, что лямбда-исчисление может быть в некотором смысле «скомпилировано» в «машинный код» комбинаторов. Комбинаторы действительно используются как метод реализации функциональных языков не только на уровне программного, но и на уровне аппаратного обеспечения.

5 Лямбда-исчисление как язык программирования

5.1 Введение

5.2 Представление данных в лямбда-исчислении

Программы предназначены для обработки данных, поэтому необходимо начать с того, чтобы зафиксировать лямбда-выражения, представляющие данные. Затем мы определим базовые операции на этих данных. Часто будет возможно показать, как строка s, представляющая некоторое описание в читаемом человеком формате может быть закодирована в виде лямбда-выражения s'. Эта процедура известна как «синтаксический сахар». Будем записывать такую процедуру как

$$s \triangleq s'$$

Эту запись следует читать как «$s = s'$ по определению». Можно трактовать ее как определение некоторой константы s, обозначающей операцию, которую затем можно использовать в стиле, принятом в лямбда-исчислении.

5.2.1 Булевские значения и условия

Для кодирования значений true и false можно использовать любые неравные лямбда-термы, но лучше всего определить их следующим образом:

$$\begin{aligned} \text{true} &\triangleq \lambda x\ y.x \\ \text{false} &\triangleq \lambda x\ y.y \end{aligned}$$

Используя эти определения, можно легко определить условное выражение (напоминающее конструкцию `?:`) языка Си. Заметьте, что это — условное *выражение*, а не условная *команда* (в нашем контексте это понятие не имеет смысла), следовательно, else-часть обязательна:

$$\text{if } E \text{ then } E_1 \text{ else } E_2 \triangleq E\ E_1\ E_2.$$

Действительно:

$$\begin{aligned} \text{if true then } E_1 \text{ else } E_2 &\triangleq \text{true } E_1\ E_2 \\ &\triangleq (\lambda x\ y.x)\ E_1\ E_2 \\ &\triangleq E_1 \end{aligned}$$

и

$$\begin{aligned} \text{if false then } E_1 \text{ else } E_2 &\triangleq \text{true } E_1\ E_2 \\ &\triangleq (\lambda x\ y.y)\ E_1\ E_2 \\ &\triangleq E_2 \end{aligned}$$

Теперь легко определить все обычные логические операторы:

$$\begin{aligned} \text{not } p &\triangleq \text{if } p \text{ then false else true} \\ p \text{ and } q &\triangleq \text{if } p \text{ then } q \text{ else false} \\ p \text{ or } q &\triangleq \text{if } p \text{ then true else } q \end{aligned}$$

5.2.2 Пары и кортежи

Упорядоченную пару можно представить следующим образом:

$$(E_1, E_2) \triangleq \lambda f.f\ E_1\ E_2$$

Скобки здесь необязательны, но мы будем использовать их для того, чтобы вызывать аналогии с соответствующей записью для обозначения

векторов или комплексных чисел в математике. В действительности запятую можно рассматривать просто как некоторый инфиксный оператор, как $+$ или $-$. С учетом данного определения, функции для извлечения компонентов пары можно записать так:

$$\begin{aligned} \mathrm{fst}\, p &\triangleq p\ \mathsf{true} \\ \mathrm{snd}\, p &\triangleq p\ \mathsf{false} \end{aligned}$$

Нетрудно убедиться, что эти определения удовлетворяют требованиям:

$$\begin{aligned} \mathrm{fst}(p,q) &= (p,q)\ \mathsf{true} = \\ &= (\lambda f.f\ p\ q)\ \mathsf{true} = \\ &= \mathsf{true}\ p\ q = \\ &= (\lambda x\ y.x)\ p\ q = \\ &= p \end{aligned}$$

и

$$\begin{aligned} \mathrm{snd}(p,q) &= (p,q)\ \mathsf{false} = \\ &= (\lambda f.f\ p\ q)\ \mathsf{false} = \\ &= \mathsf{false}\ p\ q = \\ &= (\lambda x\ y.y)\ p\ q = \\ &= q \end{aligned}$$

Тройки, четверки и вообще, произвольные n-кортежи можно построить с помощью пар:

$$(E_1, E_2, \ldots, E_n) = (E_1, (E_2, \ldots, E_n)).$$

Следует только ввести соглашения, что оператор «запятая» ассоциативен вправо.

Функции fst и snd можно расширить на случай n-кортежей. Определим функцию селектора, которая получает i-й компонент кортежа p. Будем записывать ее как $(p)_i$. Тогда $(p)_1 = \mathrm{fst}\, p$ и $(p)_i = \mathrm{fst}(\mathrm{snd}^{i-1}\, p)$.

5.2.3 Натуральные числа

Натуральное число n будем представлять в следующем виде:

$$n \triangleq \lambda f\ x.f^n\ x$$

т.е. $0 = \lambda f\ x.x$, $1 = \lambda f\ x.f\ x$, $2 = \lambda f\ x.f\ (f\ x)$ и т.д. Таким образом, числа вводятся индуктивным образом, и употребление символа n в определении сделано просто для удобства. Введенное представление называется «цифрами по Черчу». Это не очень эффективное представление; можно было бы использовать, например, двоичное представление чисел в виде кортежей, состоящих из значений true и false. Однако сейчас нас интересует понятие вычислимости «в принципе», и цифры по Черчу обладают рядом удобных формальных свойств.

Например, легко можно ввести операцию следования (увеличения на 1):

$$\text{SUC} \triangleq \lambda n\ f\ x.n\ f\ (f\ x)$$

Действительно:

$$\begin{aligned}
\text{SUC}\ n &= (\lambda n\ f\ x.n\ f\ (f\ x))\ (\lambda f\ x.f^n\ x) = \\
&= \lambda f\ x.(\lambda f\ x.f^n\ x)\ f\ (f\ x) = \\
&= \lambda f\ x.(\lambda x.f^n\ x)\ (f\ x) = \\
&= \lambda f\ x.f^n\ (f\ x) = \\
&= \lambda f\ x.f^{n+1}\ x = \\
&= n+1
\end{aligned}$$

Также легко ввести функцию проверки числа на 0:

$$\text{ISZERO}\ n \triangleq n\ (\lambda x.\text{false})\ \text{true}$$

поскольку

$$\text{ISZERO}\ 0 = (\lambda f\ x.x)\ (\lambda x.\text{false})\text{true} = \text{true}$$

и

$$\begin{aligned}
\text{ISZERO}\ n+1 &= (\lambda f\ x.f^n\ x)(\lambda x.\text{false})\text{true} = \\
&= (\lambda x.\text{false})^{n+1}\ \text{true} = \\
&= (\lambda x.\text{false})((\lambda x.\text{false})^n\ \text{true}) = \\
&= \text{false}
\end{aligned}$$

Следующие определения вводят сложение и умножение:

$$\begin{aligned}
m+n &\triangleq \lambda f\ x.m\ f\ (n\ f\ x) \\
m*n &\triangleq \lambda f\ x.m\ (n\ f)\ x
\end{aligned}$$

Действительно:

$$\begin{aligned} m+n &= \lambda f\ x.m\ f\ (n\ f\ x) = \\ &= \lambda f\ x.(\lambda f\ x.f^m\ x)\ f\ (n\ f\ x) = \\ &= \lambda f\ x.(\lambda x.f^m\ x)\ (n\ f\ x) = \\ &= \lambda f\ x.f^m\ (n\ f\ x) = \\ &= \lambda f\ x.f^m\ ((\lambda f\ x.f^n\ x)\ f\ x) = \\ &= \lambda f\ x.f^m\ ((\lambda x.f^n\ x)\ x) = \\ &= \lambda f\ x.f^m\ (f^n\ x) = \\ &= \lambda f\ x.f^{m+n}\ x \end{aligned}$$

и

$$\begin{aligned} m*n &= \lambda f\ x.m\ (n\ f)\ x = \\ &= \lambda f\ x.(\lambda f\ x.f^m\ x)\ (n\ f)\ x = \\ &= \lambda f\ x.(\lambda x.(n\ f)^m\ x)\ x = \\ &= \lambda f\ x.(n\ f)^m\ x = \\ &= \lambda f\ x.((\lambda f\ x.f^n\ x)\ f)^m\ x = \\ &= \lambda f\ x.(\lambda x.f^n\ x)^m\ x = \\ &= \lambda f\ x.(f^n)^m\ x = \\ &= \lambda f\ x.f^{mn}\ x \end{aligned}$$

Эти операции на натуральных числах довольно легко представляются в лямбда-исчислении, однако выразить функцию «предшествования» (вычитание 1) оказывается гораздо труднее. Требуется найти лямбда-выражение PRE, такое, что PRE $0 = 0$ и PRE $n + 1 = n$. Эту задачу решил Клини в 1935 году. Идея заключается в том, чтобы отбросить одно применение f в выражении $\lambda f\ x.f^n\ x$. Определим функцию PREFN, удовлетворяющую следующим условиям:

$$\text{PREFN}\ f\ (\text{true}, x) = (\text{false}, x)$$

и

$$\text{PREFN}\ f\ (\text{false}, x) = (\text{false}, f\ x)$$

Тогда можно заметить, что $(\text{PREFN}\ f)^{n+1}\ (\text{true}, x) = (\text{false}, f^n\ x)$ и функцию предшествования можно определить без особых трудностей. Функцию PREFN можно определить, например, так:

$$\text{PREFN} \triangleq \lambda f\ p.(\text{false}, \text{if}\ \ \text{fst}\, p\ \text{then}\ \ \text{snd}\, p\ \text{else}\ f(\text{snd}\, p))$$

Тогда:

$$\text{PRE}\ n \triangleq \lambda f\ x.\,\text{snd}(n\ (\text{PREFN}\ f)\ (\text{true}, x))$$

5.3 Рекурсивные функции

Возможность определять рекурсивные функции — характерная особенность функционального программирования. На первый взгляд, кажется, что в лямбда-исчислении нет способа сделать это. Действительно, важной частью рекурсивного определения является *именование* функции, иначе как мы обратимся к ней в правой части определения? Однако в действительности мы сможем обойтись без этого, хотя, как и в случае с функцией предшествования, этот факт далеко не очевиден.

Ключевым моментом является существование так называемых *комбинаторов неподвижной точки*. Замкнутый лямбда-терм Y называется комбинатором неподвижной точки, если для любого лямбда-терма f выполняется: $f(Y\ f) = Y\ f$. Таким образом, комбинатор неподвижной точки по заданному терму f возвращает неподвижную точку f, т.е. терм x, такой что $f(x) = x$. Первый такой комбинатор, найденный Карри, обычно обозначается как Y. Часто его называют «парадоксальным комбинатором». Он определяется следующим образом:

$$Y \triangleq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Нетрудно убедиться, что данное выражение действительно определяет комбинатор неподвижной точки:

$$\begin{aligned} Y\ f &= (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ f = \\ &= (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) = \\ &= f((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) = \\ &= f(Y\ f) \end{aligned}$$

Хотя с математической точки зрения наши рассуждения верны, с вычислительной стороны такое определение вызывает трудности, поскольку вышеприведенное рассуждение использует лямбда-равенство, а не редукции (в последнем переходе мы использовали *обратную* β-конверсию). По этой причине можно предпочесть следующее определение комбинатора неподвижной точки, принадлежащее Тьюрингу:

$$T \triangleq (\lambda x\ y.y\ (x\ x\ y))\ (\lambda x\ y.y\ (x\ x\ y))$$

(Доказательство того, что $T\ f \rightarrow f(T\ f)$ оставляем в качестве упражнения.)

Несмотря на сделанное выше замечание, будем обозначать комбинатор неподвижной точки по-прежнему как Y. Покажем, как он может помочь в определении рекурсивных функций. Рассмотрим в качестве

примера функцию факториала. Мы хотим определить функцию fact, такую, что:

$$\text{fact}(n) = \text{if ISZERO}\, n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE}\, n)$$

Сперва преобразуем это к следующему эквивалентному виду:

$$\text{fact} = \lambda n.\text{if ISZERO}\, n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE}\, n)$$

Это выражение, в свою очередь, эквивалентно

$$\text{fact} = (\lambda f\ n.\text{if ISZERO}\, n \text{ then } 1 \text{ else } n * f(\text{PRE}\, n))\ \text{fact}$$

Отсюда можно заключить, что fact является неподвижной точкой некоторой функции F следующего вида:

$$F = \lambda f\ n.\text{if ISZERO}\, n \text{ then } 1 \text{ else } n * f(\text{PRE}\, n)$$

Таким образом, $\text{fact} = Y\ F$.

Похожая техника используется для определения взаимно рекурсивных функций, т.е. набора функций, определения которых взаимно зависят друг от друга. Определение вида

$$\begin{aligned} f_1 &= F_1\ f_1 \ldots f_n \\ f_2 &= F_2\ f_1 \ldots f_n \\ \ldots &= \ldots \\ f_n &= F_n\ f_1 \ldots f_n \end{aligned}$$

можно преобразовать, используя кортежи, к одному равенству:

$$(f_1, f_2, \ldots, f_n) = (F_1\ f_1 \ldots f_n, F_2\ f_1 \ldots f_n, \ldots, F_n\ f_1 \ldots f_n)$$

Теперь, если мы запишем $t = (f_1, f_2, \ldots, f_n)$, то каждая из функций f_i в правой части равенства может быть представлена через соответствующий селектор: $f_i = (t)_i$. Таким образом, уравнение можно записать в каноническом виде $t = F\ t$, что дает решение $t = Y\ F$. Отсюда, снова с помощью селекторов, можно получить отдельные компоненты кортежа t.

5.4 Именованные выражения

Ранее мы представляли возможность записывать безымянные функции как одно из преимуществ лямбда-исчисления. Было показано, что рекурсивные функции можно определить, не вводя каких-либо имен. Тем

не менее возможность давать имена выражениям часто бывает полезна, поскольку позволяет избежать долгих и утомительных повторений. Простая форма именования может быть введена как еще одна форма «синтаксического сахара» над чистым лямбда-исчисление:

$$\text{let } x = S \text{ in } T \triangleq (\lambda x.T)\ S$$

Например:

$$(\text{let } z = 2 + 3 \text{ in } z + z) = (\lambda z.z + z)\ (2 + 3) = (2 + 3) + (2 + 3)$$

Можно связать несколько выражений с переменными в последовательном или параллельном стиле. В первом случае мы просто вкладываем let-конструкции друг в друга. Во втором мы записываем связывания, ограничив их символами '{' и '}' и разделяя символом ';':

$$\text{let } \{x_1 = S_1; x_2 = S_2; \ldots; x_n = S_n\} \text{ in } T$$

Это выражение можно рассматривать как «синтаксический сахар» для:

$$(\lambda(x_1, \ldots, x_n).T)\ (S_1, \ldots, S_n)$$

Вместо префиксной формы с let можно ввести постфиксный вариант, который иногда более читаем:

$$T \text{ where } x = S$$

Например, можно записать '$y < y^2$ where $y = 1 + x$'.

Далее, с помощью let-нотации можно определять функции, введя соглашение, что

$$\text{let } f\ x_1\ x_2 \ldots x_n = S \text{ in } T$$

означает

$$\text{let } f = \lambda x_1\ x_2 \ldots x_n.S \text{ in } T$$

Для определения рекурсивных функций можно ввести соглашение по использованию комбинатора неподвижной точки, т.е.

$$\text{let } \text{fact}\, n = \text{if } \text{ISZERO}\, n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE}\, n) \text{ in } S$$

означает let $\text{fact} = Y\ F$ in S.

5.5 На пути к реальному языку программирования

Разработанная нами система «синтаксического сахара» позволяет поддерживать понимаемый человеком синтаксис для чистого лямбда-исчисления, и с ее помощью можно писать программы на языке, очень напоминающем настоящий язык программирования, такой как Haskell или ML.

Программа представляет собой единственное выражение. Однако, имея в распоряжении механизм let для связывания подвыражений с именами, более естественно рассматривать программу как набор *определений* вспомогательных функций, за которыми следует само выражение, например:

$$\begin{array}{l}\text{let } \{\,\text{fact}\, n = \text{if } \text{ ISZERO}\, n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE}\, n);\\ \ldots\\ \}\ \text{in } \ \text{fact}\, 6\end{array}$$

(Заметьте, что это уже напоминает реальную программу на языке Haskell, за тем исключением, что, помимо правил выравнивания, позволяющим обойтись без символов ‘{’, ‘}’ и ‘;’,в Haskell введено соглашение опускать конструкцию let в определениях верхнего уровня.)

Эти определения вспомогательных функций можно интерпретировать как систему уравнений в обычном математическом смысле. Они не задают нам нам явных указаний, каким именно образом вычислять выражения. По этой причине функциональное программирование часто объединяют с логическим в класс *декларативных* методов программирования. Программа определяет ряд желаемых свойств результата и оставляет машине найти способ его вычисления.

Тем не менее программа все же должна быть выполнена каким-либо образом. В действительности выражение-программа вычисляется с помощью «развертывания» всех конструкций до уровня чистого лямбда-исчисления и последующего применения β-конверсий. Таким образом, хотя в самой программе нет никакой информации по ее выполнению, при ее составлении все же подразумевается некоторая стратегия исполнения. В некотором смысле понятие декларативности является отражением только человеческой психологии.

Более того, необходимо ввести некоторое определенное соглашение по используемым редукционным стратегиям, поскольку известно, что выбор различных β-редексов может привести к различному поведению программы в смысле ее завершаемости. Следовательно, нам нужно определить эту стратегию, чтобы получить реальный язык программирования. В последующих разделах мы увидим, как различаются по этому

критерию различные функциональные языки программирования. Однако сперва нам нужно ввести понятие типа.

6 Типы

Типы предоставляют возможность различать отдельные виды данных, такие как булевские значения, натуральные числа и функции. Учет типов может дать то преимущество, что, например, функция не может быть применена к аргументам, имеющим «неправильный» тип. Почему у нас может возникнуть желание ввести типы в лямбда-исчисление и в языки программирования, основанные на нем? Мотивы этого могут возникнуть как со стороны логики, так и со стороны программирования.

Изначально лямбда-исчисление задумывалось как способ формализации всего языка математики. Черч надеялся также включить в лямбда-исчисление теорию множеств. По заданному множеству S можно определить его характеристическую функцию χ_S, такую что:

$$\chi_S(x) = \begin{cases} \text{true}, & x \in S \\ \text{false}, & x \notin S \end{cases}$$

И наоборот, имея унарный предикат P, можно определить множество таких x, что $P(x) = \text{true}$. Кажется, естественно определить предикаты (и, следовательно, множества) в виде произвольных лямбда-выражений. Однако это приводит к противоречиям.

Рассмотрим знаменитый парадокс Рассела. Определим множество R как множество тех множеств, которые не содержат сами себя в качестве элемента:

$$R = \{x | x \notin x\}$$

Тогда мы получаем, что $R \in R \Leftrightarrow R \notin R$, что является, несомненно, противоречием. В терминах лямбда-исчисления, мы полагаем $R = \lambda x.\neg(x\ x)$ и обнаруживаем, что $R\ R = \neg(R\ R)$. Выражение $R\ R$ является неподвижной точкой оператора отрицания.

Парадокс Рассела возникает из-за того, что мы делаем странную даже с точки зрения здравого смысла вещь: мы применяем функцию саму к себе. Конечно, само по себе применение функции к самой себе не обязательно приводит к парадоксу: например, функция идентичности $\lambda x.x$ или константная функция $\lambda x.y$ вполне безобидны. Однако, мы, безусловно, имели бы более ясное представление о функциях, обозначаемых лямбда-термом, если бы мы точно знали области их определения и значений и применяли бы их только к аргументам, принадлежащим

их областям определения. Таковы были причины, по которым Рассел изначально предложил ввести понятие типа.

Типы также возникли, вероятно, вначале без всякой связи с приведенными выше рассуждениями, в языках программирования, и поскольку мы рассматриваем лямбда-исчисление как язык программирования, такая точка зрения не может нас не интересовать. Даже в первых версиях Фортрана различались целые и вещественные числа. Одной из этого была эффективность: зная о типе переменной, можно генерировать более эффективный код и более эффективно использовать память.

Помимо эффективности, по мере развития программирования, типы стали цениться за то, что они предоставляют ограниченную форму статической проверки программ. Многие ошибки программирования, от простых опечаток до более существенных концептуальных ошибок, влекут за собой несоответствие типов, и, следовательно, могут быть обнаружены до запуска программы, на этапе ее компиляции. Более того, типы часто предоставляют полезную информацию для людей, читающих программы. Наконец, типы можно использовать для того, чтобы достичь большей модульности и скрытия данных: они позволяют «искусственно» отделить некоторые данные от их внутреннего представления.

Вместе с тем некоторые программисты недолюбливают типы. Существуют безтиповые языки, как императивные, так и функциональные. Другие языки (к ним относится и Си) являются *слабо типизированными*: компилятор допускает некоторое несоответствие в типах и сам делает необходимые преобразования. Также существуют языки (например, LISP), которые выполняют проверку типов *динамически*, во время выполнения программы, а не во время компиляции.

Найти систему типов, которая не позволяет выполнять полезные статические проверки и в то же время не накладывает на программиста чересчур жестких ограничений — непростая задача, до конца еще не решенная. Система типов, использующаяся в таких языках, как Haskell или ML — важный этап на этом пути, поскольку в ней программисту предоставляется возможность *полиморфизма*: одна и та же функция может использоваться с различными типами. Это оставляет возможность статических проверок, предоставляя в то же время некоторые преимущества слабой или динамической типизации. Более того, программист не обязан указывать *никаких* типов в Haskell или ML: компьютер может вывести наиболее общий тип любого выражения и отвергнуть выражения, не имеющие типа.

6.1 Типизированное лямбда-исчисление

Модифицировать лямбда-исчисление так, чтобы ввести в него понятие типа, довольно нетрудно, однако это приводит к важным последствиям. Основная идея заключается в том, чтобы каждый лямбда-терм имел *тип*, и терм S можно применить к терму T в комбинации $S\,T$, если их типы правильно соотносятся друг с другом, т.е. S имеет тип функции $\sigma \to \tau$ и T имеет тип σ. Результат, $S\,T$, имеет тип τ. Это свойство называется *сильной типизацией*. Терм T должен иметь *в точности* тип σ: приведение типов не допускается, в отличие от языков типа Си, в котором некоторая функция, ожидающая аргумент типа `double` или `float`, может принять аргумент типа `int` и неявно преобразовать его.

Мы будем использовать запись вида $T :: \sigma$ для утверждения «T имеет тип σ». Это напоминает использующуюся в математике стандартную запись вида $f\colon \sigma \to \tau$ для функции f из множества σ в множество τ. Типы можно представлять себе как множества, которым принадлежат соответствующие объекты, т. е. запись $T :: \sigma$ можно интерпретировать как $T \in \sigma$. Однако мы будем рассматривать типизированное лямбда-исчисление как формальную систему, независящую от подобной интерпретации.

6.1.1 Базовые типы

Прежде всего необходимо точно определить, что такое тип. Мы предполагаем, что у нас имеется некоторый набор *базовых* типов, таких как `Bool` или `Integer`. Из них можно конструировать составные типы с помощью *конструкторов типов*, являющихся, по сути, функциями. Дадим следующее индуктивное определение множества Ty_C типов, основывающихся на множестве базовых типов C:

$$\frac{\sigma \in C}{\sigma \in Ty_C}$$

$$\frac{\sigma \in Ty_C, \tau \in Ty_C}{\sigma \to \tau \in Ty_C}$$

Например, возможными типами могут быть *Integer*, *Bool* $\to$ *Bool*, (*Integer* $\to$ *Bool*) $\to$ *Integer* $\to$ *Bool* и т. д. Мы предполагаем, что функциональная стрелка $\to$ ассоциативна вправо, т. е. $\sigma \to \tau \to \nu$ означает $\sigma \to (\tau \to \nu)$.

В дальнейшем мы расширим нашу систему типов двумя способами. Во-первых, мы введем понятие *ти́повых переменных*, являющихся средством для реализации полиморфизма. Во-вторых, мы введем дополнительные конструкторы типов, помимо функциональной стрелки.

Например, введем конструктор $\times$ для типа пары значений. В этом случае к нашему индуктивному определению необходимо добавить:

$$\frac{\sigma \in Ty_C, \tau \in Ty_C}{\sigma \times \tau \in Ty_C}$$

Далее, как и в языке Haskell, можно вводить именованные конструкторы произвольной арности. Мы будем использовать запись $\mathit{Con}\ (\alpha_1, \ldots, \alpha_n)$ для применения n-арного конструктора Con к аргументам α_i.

Важным свойством типов является то, что $\sigma \to \tau \neq \sigma$ (В действительности тип не может совпадать ни с каким своим синтаксически правильным подвыражением.) Это исключает возможность применения терма к самому себе.

6.1.2 Типизации по Черчу и Карри

Существуют два основных подхода к определению типизированного лямбда-исчисления. Первый подход, принадлежащий Черчу — *явная* типизация. Каждому терму сопоставляется единственный тип. Это означает, что в процессе конструирования термов, нетипизированные термы, которые мы использовали ранее, модифицируются с помощью добавления дополнительной характеристики — типа. Для констант этот тип задан заранее, но переменные могут иметь любой тип. Правила корректного формирования термов выглядят тогда следующим образом:

$$\frac{}{v :: \sigma}$$

$$\frac{\text{Константа } c \text{ имеет тип } \sigma}{c :: \sigma}$$

$$\frac{S :: \sigma \to \tau, T :: \sigma}{S\ T :: \tau}$$

$$\frac{v :: \sigma, T :: \tau}{\lambda v.T :: \sigma \to \tau}$$

Однако мы будем использовать для типизации подход Карри, который является *неявным*. Термы могут иметь или не иметь типа, и если терм имеет тип, то он может быть не единственным. Например, функция идентичности $\lambda x.x$ может иметь любой тип вида $\sigma \to \sigma$. Такой подход более предпочтителен, поскольку, во-первых, он соответствует используемому в языках типа Haskell и ML понятию полиморфизма, а во-вторых, позволяет, как и в этих языках, не задавать типы явным образом.

В то же самое время, некоторые формальные детали типизации по Карри несколько сложнее. Мы не определяем понятие типизируемости само по себе, но по отношению к *контексту*, т. е. конечному набору предположений о типах переменных. Мы будем записывать:

$$? \vdash T :: \sigma$$

чтобы обозначить, что «в контексте $?$ терм T может иметь тип σ». Мы будем употреблять запись $\vdash T :: \sigma$ или просто $T :: \sigma$, если суждение о типизации выполняется в пустом контексте. Элементы контекста $?$ имеют вид $v :: \sigma$, т. е. они представляют собой предположения о типах переменных, как правило, тех, которые являются компонентами терма. Будем предполагать, что в $?$ нет противоречивых предположений.

6.1.3 Формальные правила типизации

Правила типизации вводятся совершенно естественно, следует только помнить о том, что $T :: \sigma$ интерпретируется как «T *может* принадлежать типу σ».

$$\frac{v :: \sigma \in ?}{? \vdash v :: \sigma}$$

$$\frac{\text{Константа } c \text{ имеет тип } \sigma}{c :: \sigma}$$

$$\frac{? \vdash S :: \sigma \to \tau, ? \vdash T :: \sigma}{? \vdash S\ T :: \tau}$$

$$\frac{? \cup \{v :: \sigma\} \vdash T :: \tau}{? \vdash \lambda v.T :: \sigma \to \tau}$$

Эти правила являются индуктивным определением отношения типизируемости, и терм имеет некоторый тип, если он может быть выведен по этим правилам. Для примера рассмотрим типизирование функции идентичности $\lambda x.x$. По правилу для переменных имеем:

$$\{x :: \sigma\} \vdash x :: \sigma$$

и отсюда по последнему правилу получаем:

$$\emptyset \vdash \lambda x.x :: \sigma \to \sigma$$

Следуя соглашению о пустых контекстах мы пишем просто $\lambda x.x :: \sigma \to \sigma$.

Можно показать, что все преобразования лямбда-термов сохраняют типы, т. е. если некоторый терм T имел тип σ в контексте $?$, то после преобразования его в T' он будет иметь тот же самый тип.

6.2 Полиморфизм

Система типов по Карри уже дает нам некоторую форму *полиморфизма*, в том смысле, что терм может иметь различные типы. Необходимо различать похожие концепции полиморфизма и *перегрузки*. Оба этих термина означают, что выражение может иметь несколько типов. Однако в случае полиморфизма все типы сохраняют некоторое систематическое сходство друг с другом и допустимы все типы, следующие некоторому образцу. Например, функция идентичности может иметь тип $\sigma \to \sigma$, $\tau \to \tau$ или $(\sigma \to \sigma) \to (\sigma \to \tau)$, но все эти типы имеют похожую структуру. В противоположность этому, при перегрузке функция может иметь различные типы, не связанные друг с другом структурным сходством. Также возможна ситуация когда функция просто определяется для некоторого набора типов. Например, функция сложения может иметь тип int $\to$ int $\to$ int или float $\to$ float $\to$ float, но не bool $\to$ bool $\to$ bool.

6.2.1 let-полиморфизм

К сожалению, наша система типов приводит к некоторым нежелательным ограничениям на полиморфизм. Например, следующее выражение совершенно приемлемо:

$$\text{if } (\lambda x.x) \text{ true then } (\lambda x.x)\ 1 \text{ else } 0$$

Если использовать правила типизации, то можно получить, что это выражение имеет тип int. Два экземпляра функции идентичности имеют типы bool $\to$ bool и int $\to$ int.

Теперь рассмотрим выражение:

$$\text{let } I = \lambda x.x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0$$

Согласно нашему определению, это всего лишь другой способ записи для

$$(\lambda I.\text{if } I \text{ true then } I\ 1 \text{ else } 0)\ (\lambda x.x)$$

Однако это выражение не может быть типизировано по нашим правилам. В нем присутствует только *один* экземпляр функции идентичности, и он должен иметь единственный тип. Это — серьезный недостаток, поскольку мы уже видели, что конструкции с let широко используются в функциональном программировании.

Один из способов преодолеть это ограничение — добавить еще одно правило типизации, в котором let-конструкции рассматривается как первичная:

$$\frac{? \vdash S :: \sigma \text{ и } ? \vdash T[x := S] :: \tau}{? \vdash \text{let } x = S \text{ in } T :: \sigma}$$

Это правило вводит *let-полиморфизм*. В нем выражается тот факт, что при типизации мы рассматриваем связывания по let так, как будто у нас записаны различные экземпляры этих именованных выражений. Дополнительная гипотеза $? \vdash S :: \sigma$ нужна для того, чтобы гарантировать, что терм S типизируем, иначе были бы типизируемы такие выражения как:

$$\text{let } x = \lambda f.f\ f \text{ in } 0$$

6.2.2 Наиболее общие типы

Как уже было сказано, некоторые выражения не имеют типа, например, $\lambda f.f\ f$ или $\lambda f.(f\ \text{true}, f\ 1)$. Типизируемые выражения обычно имеют много типов, хотя некоторые, например, true, имеют только один. Также мы говорили о параметрическом полиморфизме, т. е. все типы выражения могут иметь структурное сходство. В действительности истинно более сильное утверждение: для каждого типизируемого выражения существует *наиболее общий тип* или *основной тип*, и все возможные типы выражения являются экземплярами этого наиболее общего типа. Прежде чем сделать это утверждение строгим, необходимо ввести некоторую терминологию.

Прежде всего, расширим нашу нотацию понятием *ти́повых переменных*. Это означает, что типы могут быть сконструированы с помощью применения конструкторов типа либо к типовым константам, либо к переменным. Будем использовать буквы α и β для типовых переменных, а σ и τ — для произвольных типов. Теперь можно определить понятие замены типовой переменной в некотором типе на другой тип. Это в точности аналогично подстановке на уровне термов, и мы будем использовать ту же нотацию. Например:

$$(\sigma \to \text{bool})[\sigma := (\sigma \to \tau)] = (\sigma \to \tau) \to \text{bool}$$

Формальное определение гораздо проще, чем для термов, поскольку нам нет нужды беспокоится о связываниях переменных. В то же время удобно расширить его параллельными подстановками:

$$\begin{aligned} \alpha_i[\alpha_1 := \tau_1, \ldots, \alpha_k := \tau_k] &= \tau_i \\ \beta[\alpha_1 := \tau_1, \ldots, \alpha_k := \tau_k] &= \beta, \text{ если } \alpha_i \neq \beta \text{ для } 1 \leq i \leq k \\ \text{Con } (\sigma_1, \ldots, \sigma_n)[\theta] &= \text{Con } (\sigma_1[\theta], \ldots, \sigma_n[\theta]) \end{aligned}$$

Для простоты, мы рассматриваем типовые константы как 0-арные конструкторы, т. е. считаем, что int задается как int (). Имея определение подстановки, мы можем определить, что тип σ является *более*

общим, чем тип σ' и записывать этот факт как $\sigma \preceq \sigma'$. Это отношение выполняется тогда и только тогда, когда существует набор подстановок θ такой, что $\sigma' = \sigma\theta$. Например:

$$\begin{array}{rcl} \alpha & \preceq & \alpha \\ \alpha \to \alpha & \preceq & \beta \to \beta \\ \alpha \to \text{bool} & \preceq & (\beta \to \beta) \to \text{bool} \\ \beta \to \alpha & \preceq & \alpha \to \beta \\ \alpha \to \alpha & \not\preceq & (\beta \to \beta) \to \beta \end{array}$$

Теперь можно сформулировать теорему:

Теорема 4. *Каждый типизируемый терм имеет некоторый основной тип, т. е. если $T :: \tau$, то существует некоторый σ, такой что $T :: \sigma$ и для любого σ', если $T :: \sigma'$, то $\sigma \preceq \sigma'$.*

Основной тип является единственным с точностью до переименований типовых переменных. Мы не будем приводить доказательство теоремы: оно не сложно, но довольно длинно. Важно понять, что доказательство конструктивно: оно дает конкретную процедуру для поиска основного типа. Эта процедура известна как *алгоритм Хиндли–Милнера*. Все реализации языков программирования Haskell, ML и других, включают в себя вариант этого алгоритма, так что выражения в них могут быть сопоставлены их основному типу либо отвергнуты как нетипизируемые.

6.3 Сильная нормализация

Вспомним наш пример терма без нормальной формы:

$$\begin{array}{rl} & (\lambda x.y)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)) \\ \to & (\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)) \\ \to & (\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)) \\ \to & \ldots \end{array}$$

В типизированном лямбда-исчислении такого не может случиться, поскольку существует следующая теорема о сильной нормализации:

Теорема 5. *Каждый типизируемый терм имеет нормальную форму и каждая возможная последовательность редукций, начинающаяся с типизируемого терма, завершается.*

На первый взгляд, это хорошо — функциональная программа, соблюдающая дисциплину типизации может быть вычислена любым образом, и она всегда завершится в единственной нормальной форме (единственность следует из теоремы Черча–Россера, которая выполняется и для типизированного лямбда-исчисления). Однако возможность писать незавершающиеся программы существенна для полноты по Тьюрингу, так что мы больше не можем определять все вычислимые функции и даже все полные функции.

Поскольку все определяемые функции полны, мы не можем делать произвольные рекурсивные определения. Действительно, обычный комбинатор неподвижной точки должен быть нетипизируем; $Y \triangleq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ не является правильно типизируемым выражением, поскольку он применяет x сам к себе и x связан. Чтобы вернуть полноту по Тьюрингу, мы просто добавляем способ определения произвольных рекурсивных функций, которые *являются* правильно типизированными. Мы вводим полиморфный оператор рекурсии для всех типов вида:

$$\mathit{Rec} :: ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau$$

Также вводится дополнительное правило редукции: для любого $F :: (\sigma \to \tau) \to (\sigma \to \tau)$ имеем

$$\mathit{Rec}\ F \to F\ (\mathit{Rec}\ F)$$

Теперь будем считать, что рекурсивные определения функций интерпретируются с использованием этих операторов рекурсии.

7 Отложенные вычисления

Мы уже обсуждали, что с теоретической точки зрения, нормальный порядок редукции выражений наиболее предпочтителен, поскольку если любая стратегия завершается, завершится и она. Такая стратегия известна и в традиционных языках программирования (в Алголе 60 ее называют *вызовом по имени*, по таким же правилам «вызываются» параметризованные макроопределения в языке Си.) Однако с практической точки зрения такая стратегия имеет существенные недостатки. Для примера рассмотрим выражение

$$(\lambda x.x + x + x)\ (10 + 5)$$

Нормальная редукция сводит это выражение к следующему:

$$(10 + 5) + (10 + 5) + (10 + 5)$$

Таким образом, необходимо трижды вычислять значение одного и того же выражения. На практике это, разумеется, недопустимо. Существуют два основных решения этой проблемы и они разделяют мир функционального программирования на два лагеря.

В первом подходе отказываются от нормальной стратегии и вычисляют аргументы функций до того, как передать их значения в функцию. Это — обычная практика в таких языках, как Си, Паскаль и т. д. Такой подход называют *передачей по значению*. При этом вычисление некоторых выражений, завершающееся при нормальной стратегии, может привести к бесконечной последовательности редукций. Однако на практике этих случаев можно легко избежать. Обычно такая стратегия позволяет получать довольно эффективный машинный код. Также она предпочтительна в случае гибридных языков, т. е. функциональных языков, содержащих императивные конструкции (присваивания, циклы и т. п.) Языки семейства ML придерживаются такого порядка вычислений.

При другом подходе, использующемся в Haskell и некоторых других языках, нормальная стратегия редукций сохраняется. Однако при этом различные возникающие подвыражения разделяются и никогда не вычисляются более одного раза. Во внутреннем представлении выражения становятся не деревьями, а направленными ациклическими графами. Такая дисциплина вызова называется *ленивой* или *вызовом по необходимости*, поскольку выражения вычисляются только тогда, когда их значения действительно необходимы для работы программы.

Например, в языке Haskell можно сделать следующее определение:

```
bottom = bottom
```

Согласно этому определению, вычисление выражения `bottom` никогда не завершиться. Однако рассмотрим такую функцию:

```
const1 x = 1
```

Тогда значение выражения `const1 bottom` равно 1. Поскольку функция `const1` не нуждается в значении своего аргумента, он не вычисляется. Аналогичная ситуация и со следующим выражением:

```
const1 (1/0)
```

Значение этого выражения также равно 1. Деления на 0 не происходит, поскольку аргумент функции никогда не вычисляется.

Преимуществом такого подхода является то, что он позволяет добиваться большей выразительности при записи функций. Платой за это является сложность реализации и некоторое снижение эффективности.

Действительно, вместо того, чтобы непосредственно вычислить выражение, приходится сохранять информацию о том, как его вычислять. Разумеется, если значение этого выражения не понадобилось в дальнейших вычислениях, мы получаем некоторую экономию, однако в противном случае непосредственное вычисление кажется предпочтительным. Оптимизирующие компиляторы языка Haskell во многих случаях могут сами выявить места программы, в которых можно обойтись без отложенных вычислений.

Конструкторы данных в Haskell также являются функциями (единственное отличие от «настоящих» функций заключается в том, что их можно использовать в шаблонах при сопоставлении с образцом.) В сочетании с «ленивым» вызовом это позволяет определять бесконечные структуры данных. Например, следующее определение задает бесконечный список единиц:

```
ones = 1 : ones
```

Более интересным примером служит бесконечный список целых чисел, начиная с числа `n`:

```
numsFrom n = n : numsFrom (n+1)
```

Термин «бесконечный» здесь — не преувеличение. Определения, подобные приведенным, действительно задают потенциально бесконечные структуры данных. С ними можно работать так же, как и с конечными, например, получить (бесконечный) список квадратов натуральных чисел:

```
squares = map (^2) (numsFrom 1)
```

Разумеется, в реальности мы работаем только с конечной частью нашего списка, однако использование отложенных вычислений позволяет отделить генерацию бесконечной структуры данных от выделения ее конечной части. Для наших примеров конечную часть списка можно выделить, например, с помощью функции `take`, которая по заданному числу `n` и списку возвращает первые `n` элементов этого списка. Так, значение выражения

```
take 5 (numsFrom 1)
```

равно `[1,2,3,4,5]`, а результатом вычисления

```
take 5 squares
```

будет `[1,4,9,16,25]`

Представленный пример дает образец типичной структуры программы, использующейся в Haskell. Программа представляется в виде конструкции `g (f input)`, где `g` и `f` — некоторые функции. Они выполняются вместе строго синхронно. `f` запускается только тогда, когда `g` пытается прочитать некоторый ввод, и выполняется ровно столько, чтобы предоставить данные, который пытается читать `g`. После этого `f` приостанавливается, и выполняется `g`, до тех пор, пока вновь не попытается прочитать следующую группу входных данных. Если `g` заканчивается, не прочитав весь вывод `f`, то `f` прерывается. `f` может даже быть не завершаемой программой, создающей бесконечный вывод, так как она будет остановлена, как только завершится `g`. Это позволяет отделить условия завершения от тела цикла, что является мощным средство модуляризации.

Этот метод называется "ленивыми вычислениями"так как `f` выполняется настолько редко, насколько это возможно. Он позволяет осуществить модуляризацию программы как генератора, который создает большое количество возможных ответов, и селектора, который выбирает подходящие. Некоторые другие системы позволяют программам выполнятся вместе подобным способом, но только функциональные языки использует ленивые вычисления однородно при каждом обращении к функции, позволяя модуляризовать таким образом любую часть программы. Ленивые вычисления, возможно, наиболее мощный инструмент для модуляризации в наборе функционального программиста.

Мы проиллюстрируем мощь ленивых вычислений, программируя некоторые численные алгоритмы. Прежде всего, рассмотрим алгоритм Ньютона–Рафсона для вычисления квадратного корня. Этот алгоритм вычисляет квадратный корень числа z, начиная с начального приближения a_0. Он уточняет это значение на каждом последующем шаге, используя правило:

$$a_{n+1} = (a_n + z/a_n)/2$$

Если приближения сходятся к некоторому пределу a, то $a = (a + z/a)/2$, то есть $a \cdot a = z$ или $a = \sqrt{z}$.

Фактически сведение к пределу проходит быстро. Программа проводит проверку на точность (`eps`) и останавливается, когда два последовательных приближения отличаются меньше чем на `eps`. При императивном подходе алгоритм обычно программируется следующим образом:

```
x = a0;
do{
  y = x;
```

```
    x = (x + z/x) / 2;
  } while (abs(x-y) < eps)
  // теперь x = квадратному корню из z
```

Эта программа неделима на обычных языках. Мы выразим её в более модульной форме, используя ленивые вычисления, и затем покажем некоторые другие применения полученным частям.

Так как алгоритм Ньютона–Рафсона вычисляет последовательность приближений, естественно представить это в программе явно списком приближений. Каждое приближение получено из предыдущего функцией

```
next z x = (x + z/x) / 2
```

То есть `(next z)` — функция, отображающая каждое приближение в следующее. Обозначим эту функцию `f`, тогда последовательность приближений будет: `[a0, f a0, f(f a0), f(f(f a0)),...]` . Мы можем определить функцию, вычисляющую такую последовательность:

```
iterate f x = x : iterate f (f x)
```

Тогда список приближений можно вычислить так:

```
iterate (next z) a0
```

Здесь `iterate` — пример функции с «бесконечным» выводом — но это не проблема, потому что фактически будет вычислено не больше приближений, чем требуется остальным частям программы. Бесконечность - только потенциальная: это означает, что любое число приближений можно вычислить, если потребуется, `iterate` сама по себе не содержит никаких ограничений.

Остаток программы — функция `within`, которая берет допуск и список приближений и, просматривая список, ищет два последовательных приближения, отличающихся не более чем на данный допуск.

```
within eps (a:b:rest) =
    if abs(a-b) <= eps
        then b
        else within eps (b : rest)
```

Собирая части, получаем:

```
sqrt a0 eps z = within eps (iterate (next z) a0)
```

Теперь, когда мы имеем части программы поиска квадратного корня, мы можем попробовать объединить их различными способами. Одна из модификаций, которую мы могли бы пожелать, заключается в использовании относительной погрешности вместо абсолютной. Она больше подходит как для очень малых чисел (когда различие между последовательными приближениями маленькое), так и для очень больших (при округлении ошибки могут быть намного большими, чем допуск). Необходимо определить только замену для `within`:

```
relative eps (a:b:rest) =
    if abs(a-b) <= eps*abs(b)
        then b
        else relative eps (b:rest)
```

Теперь можно определить новую версию sqrt

```
relativesqrt a0 eps z = relative eps (iterate (next z) a0)
```

Нет необходимости переписывать часть, которая генерирует приближения.

Мы повторно использовали `iterate` при генерации последовательности приближений в вычислениях квадратного корня. Конечно же, можно многократно использовать `within` и `relative` в любых численных алгоритмах, которые генерирует последовательность приближений.

Например, запишем алгоритм определения корня функции f методом касательных. Если начальное приближение равно x_0, то следующее приближение вычисляется по формуле $x_1 = x - f(x)/f'(x)$. На языке Haskell этот алгоритм можно записать следующим образом:

```
root f diff x0 eps = within eps (iterate (next f diff) x0)
                          where next f diff x = x - f x / diff x
```

Для работы этого алгоритма необходимо указать две функции: `f`, вычисляющая значение функции f и `diff`, вычисляющую значение f'. Например, чтобы найти положительный корень функции $f(x) = x^2 - 1$ с начальным приближением 2, запишем следующее выражение:

```
root f diff 2 0.001 where f x = x^2 - 1
                          diff x = 2*x
```

Результатом вычисления этого выражения будет число 1.00000004646115. В качестве более сложного примера приведем вычисление корня уравнения $\cos x = x$:

```
root f diff 2 0.001 where f x = cos x - x
                          diff x = -sin x - 1
```

Результатом будет число 0.739085133219815.

8 Классы типов

Классы типов предоставляют возможность контроля над перегрузкой типов. Класс типов представляет собой некоторое множество типов языка, обладающих общими свойствами.

Рассмотрим простой пример: равенство. Существует много типов, для которых определено отношение равенства, однако для других типов оно не определено. Например, определение равенства двух функций невозможно с вычислительной точки зрения, однако было бы неплохо проверять на равенство два списка. Рассмотрим определение функции `elem`, проверяющей, что элемент входит в список:

```
x `elem` [] = False
x `elem` (y:ys) = x == y || (x `elem` ys)
```

Каков тип этой функции. Интуитивно, он должен иметь вид `a -> [a] -> Bool`. Однако это подразумевает, что оператор == имеет тип `a -> a -> Bool`, хотя мы говорили, что он определен не для всех типов. Более того, даже если бы он был определен для всех типов, сравнение двух списков сильно отличается от сравнения двух чисел. В этом смысле, мы ожидаем, что оператор == будет перегружен, чтобы выполнять эти (различные) задачи.

Классы типов решают обе эти проблемы. Они позволяют определять, какие типы являются *экземплярами* каких классов, и предоставлять определения ассоциированных с классом *операций*, перегруженных для каждого типа. Например, определим класс, содержащий оператор равенства:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Здесь `Eq` — имя определяемого класса, а == — единственная операция, определенная в классе. Это определение можно прочесть следующим образом: «тип `a` является экземпляром класса `Eq`, если для него существует перегруженный оператор ==».

Ограничение, гласящее, что тип `a` должен принадлежать классу `Eq` записывается как `Eq a`. Таким образом, `Eq a` выражает ограничение на тип и называется *контекстом*. Контексты располагаются перед описаниями типов:

```
(==) :: (Eq a) => a -> a -> Bool
```

Это означает: «для каждого типа `a`, принадлежащего классу `Eq`, оператор `==` имеет тип `a -> a -> Bool`». Для функции `elem` ограничения запишутся аналогично:

```
elem :: (Eq a) => a -> [a] -> Bool
```

Эта запись выражает тот факт, что функция `elem` определена не для всех типов, а только для тех, для которых определен оператор равенства.

Как указать, какие типы принадлежат классу `Eq` и определить функции сравнения для этих типов? Это делается с помощью *определения принадлежности*:

```
instance Eq Integer where
  x == y = x `integerEq` y
```

Определение оператора `==` называется *методом*. Функция `integerEq` сравнивает на равенство два целых числа. В общем случае в правой части определения допустимо любое выражение, как и при обычном определении функции.

При определении принадлежности также можно использовать контекст. Пусть у нас определен рекурсивный тип дерева:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Тогда можно определить оператор поэлементного сравнения деревьев:

```
instance (Eq a) => Eq (Tree a) where
  Leaf a == Leaf b  =  a == b
  (Branch l1 r1) == (Branch l2 r2) = (l1 == l2) && (r1 == r2)
  _ = _ = False
```

В стандартных библиотеках языка определено много классов типов. В действительности класс `Eq` несколько больше:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

В классе определены две операции, для равенства и для неравенства. Также здесь демонстрируется использование *методов по умолчанию*, в данном случае для операции неравенства. Если метод для какого-либо экземпляра класса пропущен в определении принадлежности, используется метод, определенный в классе, если он существует.

Haskell также поддерживает нотацию *расширения класса*. Например, мы определяем класс `Ord`, который *наследует* все операции класса `Eq`, и кроме того, определяет новые операторы сравнения и функции минимума и максимума:

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

Будем говорить, что `Eq` является *суперклассом* класса `Ord` (соответственно, `Ord` является *подклассом* `Eq`).

9 Монады

Понятие монад является одним из важнейших в языке Haskell. Понимание этой концепции начнем с ряда примеров.

Одной из простейших монад является тип `Maybe`. Напомним его определение:

```
data Maybe a = Nothing | Just a
```

Он используется для возврата результата из функций в случае, если этого результата может и не быть. Например, функция `f` с сигнатурой

```
f :: a -> Maybe b
```

принимает значение типа `a` и возвращает результат типа `b` (обернутый в конструктор `Just`), либо может не вычислить значения и тогда, как сигнал об ошибке, вернет значение `Nothing`.

Типичным примером такого рода функций служат функции для осуществления запроса к базе данных. В случае, если данные, удовлетворяющие критериям запроса, существуют, их следует вернуть; в противном случае возвращается `Nothing`.

Рассмотрим базу данных, содержащую информацию об адресах людей. Она устанавливает соответствие между полным именем человека и его адресом. Для простоты предположим, что имя и адрес задаются строками. Тогда базу данных можно описать следующим образом:

```
type AddressDB = [(String,String)]
```

Таким образом, база данных представляет собой список пар, первым компонентом которых будет имя, а вторым — адрес. Тогда функция `getAddress`, по заданному имени возвращающая адрес, определяется так:

```
getAddress :: AddressDB -> String -> Maybe String
getAddress [] _ = Nothing
getAddress ((name,address):rest) n | n == name = Just address
                                   | otherwise = getAddress rest n
```

Для имен, присутствующих в базе, функция возвращает соответствующий адрес. Если такого имени в базе нет, возвращается `Nothing`.

Пока все выглядит безобидно; проблемы начинаются, когда необходимо осуществлять последовательность запросов. Предположим, у нас есть еще одна база данных, содержащая соответствие между адресами и номерами телефонов:

```
type PhoneDB = [(String,Integer)]
```

Далее, пусть имеется функция `getPhone`, по адресу возвращающая телефон, реализованная также с помощью типа `Maybe`. Реализация этой функции полностью аналогична функции `getAddress`. Как теперь определить функцию, возвращающую телефон по имени человека?

Каков будет тип возвращаемого значения этой функции? Очевидно, что у человека может не оказаться телефона, следовательно, функция должна возвращать значение типа `Maybe Integer`. Далее, значение `Nothing` эта функция может вернуть в следующих случаях:

1. Указанное имя не содержится в базе адресов.
2. Адрес, соответствующий указанному имени, существует, однако он не содержится в базе телефонов.

Исходя из этих соображений, функцию `getPhoneByName` можно определить так:

```
getPhoneByName :: AddressDB -> PhoneDB -> String -> Integer
getPhoneByName a p n = case (getAddress a n) of
                           Nothing -> Nothing
                           Just address -> case (getPhone p address) of
                                               Nothing -> Nothing
                                               Just phone -> Just phone
```

Разумеется, такой стиль программирования не слишком изящен, кроме того, он провоцирует ошибки. В случае, когда уровень вложенности запросов возрастает, растет и объем повторяющегося кода. Как быть? Мудрый программист, всегда стремящийся к повторному использованию кода, определит вспомогательную функцию, которая отражает использующийся в функции `getPhoneByName` шаблон связывания функций, возвращающих значение типа `Maybe`. Назовем эту функцию `thenMB` (от англ. then maybe — тогда, возможно):

```
thenMB :: Maybe a -> (a -> Maybe b) -> Maybe b
thenMB mB f = case mB of
```

```
    Nothing -> Nothing
    Just a  -> f a
```

Рассмотрим эту функцию подробнее. Она принимает два аргумента: значение типа `Maybe a` и функцию, отображающую значение типа `a` в значение типа `Maybe b`. Если первый аргумент содержит значение `Nothing`, второй аргумент игнорируется. Если же первый аргумент содержит реальное значение, обернутое в конструктор `Just`, оно извлекается из него и передается в функцию, являющуюся вторым аргументом. Вспоминая, что в языке Haskell лямбда-абстракция записывается в виде `\x -> expr`, функцию `getPhoneByName` можно записать так:

```
getPhoneByName a p n =
  (getAddress a n `thenMB`
        (\address -> getPhone p address)) `thenMB` (\phone -> Just ph
```

Или, опуская скобки и записывая с более наглядным расположением кода:

```
getPhoneByName a p n = getAddress a n     `thenMB` \address ->
                       getPhone p address `thenMB` \phone   ->
                       Just phone
```

Эту запись следует читать так, будто результат левого аргумента оператора `` `thenMB` `` «присваивается» имени переменной из лямбда-абстракции правого аргумента.

Чего мы добились? Мы определили функцию `thenMB`, комбинирующую вычисления, которые могут либо вернуть результат, либо отказаться его вычислять. Сама функция `thenMB` не зависит от того, какие именно вычисления она комбинирует, лишь бы они удовлетворяли ее сигнатуре. Ее можно использовать не только для нашего примера, но и для любых других аналогичных задач. Она определяет некоторое правило комбинирования вычислений в цепочку, заключающееся в том, что если одно из вычислений не выполнилось, не выполняется и вся цепочка.

Усовершенствуем наш пример. Мы перечисляли случаи, в которых функция `getPhoneByName` может не найти телефон. Однако в любом случае она вернет значение `Nothing`. В реальности нас может интересовать, почему именно она не нашла телефон. Пусть функции `getPhone`, `getAddress` и `getPhoneByName` возвращают значение типа `Value`, который определим следующим образом:

```
data Value a = Value a | Error String
```

Значение типа `Value a` представляет собой либо значение типа `a`, обернутое в конструктор `Value`, либо строковое сообщение об ошибке, содержащееся в конструкторе `Error`. Функцию `getAddress` определим тогда так:

```
getAddress :: AddressDB -> String -> Value String
getAddress [] _ = Error "no address"
getAddress ((name,address):rest) n | n == name = Value address
                                   | otherwise = getAddress rest n
```

В случае ошибки `getAddress` вернет значение `Error "no address"`. Аналогично можно определить и функцию `getPhone`, которая в случае ошибки вернет значение `Error "no phone"`. Тогда функцию `getPhoneByName` можно определить следующим образом:

```
getPhoneByName :: AddressDB -> PhoneDB -> String -> Value Integer
getPhoneByName a p n = case (getAddress a n) of
                         Error s -> Error s
                         Value address -> case (getPhone p address) o
                                            Error s -> Error s
                                            Value phone -> Value phon
```

Здесь мы видим аналогичную проблему, что и с предыдущим определением. Для ее решения воспользуемся тем же примем: определим вспомогательную функцию:

```
thenV :: Value a -> (a -> Value b) -> Value b
thenV mV f = case mV of
               Error s -> Error s
               Value v -> f v
```

С использованием этой функции можно упростить наше определение:

```
getPhoneByName a p n = getAddress a n     `thenV` \address ->
                       getPhone p address `thenV` \phone ->
                       Value phone
```

Нельзя не отметить некоторое сходство в функциях `thenMB` и `thenV`, а также в определениях функции `getPhoneByName`. Забегая вперед, скажем, что тип `Value` также представляет собой пример монады.

Наконец, сделаем другое обобщение нашей исходной задачи. До сих пор мы предполагали, что записи в наших базах данных уникальны, т. е. каждому человеку соответствует только один адрес, а каждому адресу только один телефон. Предположим теперь, что это не так, т.

е. одному человеку может соответствовать несколько адресов, а одному адресу — несколько телефонов. Тогда функции `getPhone`, `getAddress` и `getPhoneByName` должны возвращать списки, и их сигнатуры можно записать следующим образом:

```
getAddress     :: AddressDB -> String -> [String]
getPhone       :: PhoneDB -> String -> [Integer]
getPhoneByName :: AddressDB -> PhoneDB -> String -> [Integer]
```

Предположим, что у нас уже определены функции `getPhone` и `getAddress`. В случае неудачи они возвращают пустые списки, в случае успешного поиска — списки, состоящие из произвольного количества элементов. Как, используя эти функции, определить функцию `getPhoneByName`? Для каждого адреса, возвращенного функцией `getAddress`, она должна вызвать функцию `getPhone`; результаты всех вызовов этой функции необходимо объединить в один список. Определение, учитывающее эти особенности:

```
getPhoneByName a p n =
   case (getAddress a n) of
     [] -> []
     (address:rest) -> getPhone p addresses ++ getPhones p rest
   where getPhones _ [] = []
         getPhones p (x:xs) = getPhone p x ++ getPhones p xs
```

Здесь также можно определить вспомогательную функцию:

```
thenL :: [a] -> (a -> [b]) -> [b]
thenL mL f = case mL of
               [] -> []
               (x:xs) -> f x ++ getRest xs f
             where getRest [] _ = []
                   getRest (x:xs) f = f x ++ getRest xs f
```

С использованием комбинатора определение примет вид:

```
getPhoneByName a p n = getAddress a n     `thenL` \address ->
                       getPhone p address `thenL` \phone ->
                       [phone]
```

Можно догадаться, что список также является монадой.

Итак, что же такое монада? Монада — это некоторый тип данных, подразумевающий определенную стратегию комбинирования вычислений

значения этого типа. Так, монада `Maybe` подразумевает такое комбинирование вычислений, что «не сработавшее» вычисление заставляет «не срабатывать» и всю цепочку вычислений. Монада `Value` подразумевает при этом, что цепочка вычислений должна вернуть сообщение об ошибке от «не сработавшего» вычисления. Монада «список» отображает концепцию вычислений, которые могут вернуть неоднозначный результат, и цепочка вычислений должна учитывать все возможные результаты.

С другой стороны, монаду можно рассматривать как контейнерный тип, т. е. как тип, значения которого содержат в себе некоторое количество элементов другого типа. Особенно ярко это проявляется на примере списков, однако нетрудно заметить, что типы `Maybe` и `Value` также являются контейнерными и могут содержать ноль или одно значение типа, являющегося их параметром.

В стандартной библиотеке языка Haskell определен класс типов, являющихся монадами. Определение этого класса приведено ниже:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a

    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

Данное определение говорит о том, что тип `m` является монадой, если для него определены указанные функции и операторы. При этом необходимо определить только функцию `return` и оператор `>>=`; для остальных функций и операторов имеются определения по умолчанию.

Инфиксный оператор `>>=` является обобщением наших функций `thenMB`, `thenV` и `thenL` (сравните типы этих функций и оператора). Функция `return` предназначена для создания монадического типа из обычного значения: она «вкладывает» значение в монаду, которая в данном случае рассматривается как контейнер.

Тип `Maybe` является монадой; это отражено в следующем коде:

```
instance Monad Maybe where
    (>>=) = thenMB
    return a = Just a
```

Таким образом, функцию `getPhoneByName`, возвращающую значения типа `Maybe Integer`, можно определить следующим образом:

```
getPhoneByName a p n = getAddress a n     >>= \address ->
                       getPhone p address >>= \phone ->
                       return phone
```

Можно дать аналогичные определения для других рассмотренных монад:

```
instance Monad Value where
    (>>=) = thenV
    return a = Value a

instance Monad [ ] where
    (>>=) = thenL
    return a = [a]
```

Оператор >> используется для комбинирования вычислений, которые не зависят друг от друга; это можно видеть из его определения. Функцию `fail` рассмотрим несколько позднее.

В стандартных библиотеках языка Haskell определено много функций для работы с монадами. Например, функция `sequence` принимает в качестве аргумента список монадических вычислений, последовательно выполняет их и возвращает список результатов:

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (c:cs) = c             >>= \x ->
                  sequence cs   >>= \xs ->
                  return (x:xs)
```

Например, выражение

```
sequence [getAddress a name1, getAddress a name2]
```

вернет список из двух элементов, каждый из которых будет содержать результат выполнения соответствующей функции.

Кроме того, поддержка монад встроена непосредственно в язык Haskell. Так называемая do-нотация облегчает запись монадических вычислений. Она позволяет записывать псевдо-императивный код с именованными переменными. Результат монадического вычисления может быть «присвоен» переменной с помощью оператора <-. Выражение справа от этого оператора должно иметь монадический тип `m a`. Выражение слева представляет собой образец, с которым сопоставляется значение *внутри* монады. Например, `(x:xs)` может сопоставиться с `Maybe [1,2,3]`. В

последующих монадических вычислениях можно использовать переменные, связанные в результате этого сопоставления.

Проще всего показать использование do-нотации на примере. Функция `getPhoneByName` запишется таким образом:

```
getPhoneByName a p n = do address <- getAddress a n
                          phone <- getPhone p address
                          return phone
```

Сравните это определение с определением через оператор `>>=`. Ключевое слово `do` вводит правило выравнивания, заключающееся в том, что первый символ, появляющийся после этого слова, задает колонку, в которой должны начинаться последующие строки. Использование символов `{`, `}` и `;` позволяет избежать этого правила:

```
getPhoneByName a p n =
     do { address <- getAddress a n;
          phone <- getPhone p address; return phone }
```

Do-нотация имеет простые правила преобразования в стандартную нотацию:

1. Конструкция вида `do {e1; e2}` преобразуется в `e1 >> e2`
2. Конструкция вида `do {x <- e1; e2}` преобразуется в `e1 >>= \x -> e2`

Таким образом, в этой конструкции нет ничего, что выходило бы за рамки функциональной парадигмы, хотя код, использующий do-нотацию, зачастую напоминает императивный.

Для того, чтобы тип был монадой, ему мало принадлежать к классу `Monad`. Понятие монад пришло из отрасли математики, которая называется теорией категорий. В ней монадами являются объекты, удовлетворяющие набору аксиом; аналогично, для монад языка Haskell должен выполняться ряд законов. Компилятор языка не проверяет их выполнение (это практически невозможно сделать автоматически) и их правильность должна быть гарантирована программистом. Вот эти законы:

1. `return x >>= f` $\equiv$ `f x`
2. `f >>= return` $\equiv$ `f`
3. `f >>= (\x -> g x >>= h)` $\equiv$ `(f >>= g) >>= h`

В принципе, первые два закона утверждают, что `return` является левой и правой единицей для оператора `>>=`, а третий закон утверждает, что оператор `>>=` обладает свойством ассоциативности. Рассмотрим эти законы подробнее.

Если мы рассматриваем монады как вычисления, то `return x` создает тривиальное вычисление, всегда возвращающее значение `x`. Если мы связываем его с другим вычислением `f`, это эквивалентно непосредственному выполнению `f` на `x`. Если этот закон выполняется, следующие две программы должны быть эквивалентны:

```
law1a = do x <- return a
           f x

law1b = do f a
```

Второй закон гласит, что если мы выполнили некоторое вычисление `f` и передали его результат в тривиальное вычисление, которое просто вернет это значение, это будет эквивалентно тому, что мы просто выполнили это вычисление. Следующие две программы должны быть эквивалентны:

```
law2a = do x <- f
           return x

law2b = do f
```

Наконец, третий закон утверждает, что независимо от того, в каком порядке мы группируем действия (слева направо или справа налево), результат не должен меняться. Следующие программы должны быть эквивалентны:

```
law3a = do x <- f
           do y <- g x
              h y

law3b = do y <- do x <- f
                   g x
           h y
```

Можно убедиться, что рассмотренные нами монады удовлетворяют этим законам.

В классе `Monad` определена функция `fail`. Она вызывается в том случае, если при сопоставлении с образцом в do-нотации произошла

ошибка. По умолчанию эта функция печатает сообщение и завершает программу, однако в некоторых монадах ее имеет смысл переопределить. Например, для монады `Maybe` ее определяют следующим образом:

```
fail _ = Nothing
```

Аналогично для списка:

```
fail _ = []
```

Важным видом монад являются монады состояния. В начале наших лекций мы определяли императивные программы как ряд последовательных изменений состояния программы. В императивных языках это состояние присутствует неявно; в функциональных языках его необходимо явно передавать в качестве аргумента функции.

Рассмотрим проблему генерации псевдослучайных последовательностей. В компьютере псевдослучайные последовательности генерируются с помощью рекуррентных соотношений вида $x_k = f(x_{k-1})$, где x_k — k-й элемент последовательности. Примером такого датчика (не очень хорошего), генерирующего вещественные числа в диапазоне от 0 до 1, может служить соотношение

$$x_k = \{11 \cdot x_{k-1} + \pi\},$$

где $\{\}$ — оператор взятия дробной части.

В языке Си можно реализовать функцию без аргументов, при каждом вызове возвращающей новое псевдослучайное число. При этом состояние датчика, представляющее в данное случае предыдущее значение последовательности, можно хранить в глобальной либо в локальной статической переменной. В языке Haskell значение функции полностью определяется ее аргументами и состояние (предыдущее значение последовательности) необходимо передавать явно. Таким образом, функция, вычисляющая площадь куба со случайными сторонами, запишется следующим образом:

```
random x = frac (pi * x + 11)

cube x0 = let x1 = random x0
              x2 = random x1
              x3 = random x2 in
          x1 * x2 * x3
```

Здесь в качестве параметра функции `cube` выступает начальное значение для генератора, а состояние передается из одного вызова функции `random` в другой явно. Здесь легко допустить ошибку, например, как в следующем коде:

```
cube x0 = let x1 = random x0
              x2 = random x1
              x3 = random x1 in
          x1 * x2 * x3
```

В этой программе значения переменных `x2` и `x3` равны! К счастью, монады могут помочь и в этом случае.

Прежде всего, сделаем ряд обобщений. В нашем случае состояние датчика и его возвращаемого значения совпадают. Однако это не обязательно. Во-первых, датчик может учитывать при вычислении очередного элемента последовательности не только предыдущий элемент, но и предшествующий ему и т. д. Во-вторых, вообще полезно разделить состояние, передаваемое от одного вызова к другому и возвращаемое значение, поскольку это позволит рассматривать состояние без привязки к возвращаемому значению. С учетом этих замечаний сделаем следующие определения:

```
data State = ... -- Здесь следует определение типа состояния
type StateT a = State -> (a,State)
random :: StateT Float
```

Тип `StateT` определяет значения, являющиеся *преобразователями состояния*, т. е. функциями, которые преобразуют заданное состояние в другое состояние, возвращая при этом некоторое значение. Таким образом, функция `random` фактически имеет тип `State -> (Float,State)`, т. е. по заданному состоянию она возвращает два значения: очередное псевдослучайное число и новое состояние. Функция `cube` запишется таким образом:

```
cube s0 = let (x1,s1) = random s0
              (x2,s2) = random s1
              (x3,s3) = random s2 in
          x1 * x2 * x3
```

Тип `StateT` можно сделать монадой, если сделать следующие определения:

```
instance Monad StateT where
  st >>= f = \s -> let (x,s1) = st s in
                   f x s1
  return a = \s -> (a,s)
```

Здесь оператор >>= связывает состояния, передавая их из одного вычисления в другое, а функция `return` возвращает тривиальный преобразователь состояния: функцию, которая оставляет состояние неизменным и возвращает указанное значение. Теперь можно применить do-нотацию:

```
cube :: StateT Float
cube = do x1 <- random
          x2 <- random
          x3 <- random
          return (x1 * x2 * x3)
```

Монады состояния важны, поскольку они являются основой для определения операций ввода-вывода в языке Haskell. Ввод-вывод долго был «камнем преткновения» для функциональных языков. Действительно, в них постулируется, что результат функции зависит только от ее аргументов. Однако в этом случае функции не могут производить операций ввода-вывода. Действительно, пусть у нас есть функция `getChar`, которая по заданному дескриптору файла типа `FileHandle` возвращает очередной прочитанный из него символ:

```
getChar :: FileHandle -> Char
```

Эта функция не возвращает одно и то же значения, будучи вызвана с одним и тем же аргументом. Ее результат зависит не только от аргумента, но и от состояния файла, которое меняется каждый раз, как вызывается эта функция.

Одним из решений этой проблемы будет явная передача этого состояния в функции, осуществляющие ввод-вывод. Таким образом, каждая из функций ввода-вывода будет принимать дополнительный параметр типа `World`, описывающим состояние всего внешнего мира, и возвращать, помимо результата, измененное состояние мира. (Разумеется, в реальности нам нет необходимости запоминать все состояние внешнего мира, эта переменная будет фиктивной.) Теперь результат функции зависит только от ее параметров. Однако необходимо быть уверенным, что состояния мира используются только один раз: возможность повторно использовать это состояние означало бы, что программа должна запоминать все предыдущие состояния внешнего окружения, а это совершенно недопустимо, да и не всегда возможно: ведь во внешнее окружение входит в том числе и человек, вводящий символы с клавиатуры!

В некоторых языках (например, Clean) для решения этой проблемы модифицировали систему типов таким образом, что однократность использования переменных типа `World` проверяется компилятором. В языке Haskell эта проблема решена с использованием монад. Определяется тип `IO a`, аналогичный нашему типу `StateT`, только вместо типа `State` используется `World`. Определение оператора связывания `>>=` гарантирует, что это значение не будет использоваться более одного раза. Далее, определяется ряд *базовых операций*, определенных на низком уровне, осуществляющих основные операции ввода-вывода: открыть файл, считать символ из файла и т. п. Важным свойством этих операций является то, что они возвращают значение типа `IO`. Например, функция для считывания символа из файла имеет тип

```
getChar :: FileHandle -> IO Char
```

Наконец, вся программа на языке Haskell представляет собой функцию `main :: IO ()`, содержащую последовательность операций ввода-вывода (их называют *действиями*). Эта функция неявно передает состояние мира в последовательность действий.

Необходимо отметить два важных свойства типа `IO`. Прежде всего, это абстрактный тип данных: пользователю недоступны конструкторы данных этого типа, невозможно извлечь значение из него без использования монадических конструкций и т. д. Далее, не существует способа избавиться от этого типа. Любая функция, использующая значение, помеченное конструктором `IO`, должна возвращать значение, также помеченное этим конструктором. Если функция не содержит в своей сигнатуре типа `IO`, значит, она не выполняет никаких операций ввода-вывода: это гарантируется системой проверки типов. Таким образом, те части программы, которые выполняют ввод-вывод, явно отделены от чисто функционального ядра.

Использование монад позволяет определить небольшой под-язык в рамках языка Haskell и программировать на нем практически в императивном стиле. Предполагается, что опытный программист способен сохранять «императивную» часть программы небольшой и четко отделять ее от функциональной части.