Министерство образования и науки
Российской Федерации
Ярославский государственный университет
им. П. Г. Демидова
Кафедра теоретической информатики

В. А. Башкин

Лямбда-исчисление

Учебно-методическое пособие

Ярославль
ЯрГУ
2018

УДК 510.56(075)
ББК 3973.2-018я73
Б 33

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2018 года

Рецензент
кафедра теоретической информатики

Башкин, Владимир Анатольевич.
Б 33 Лямбда-исчисление: учебно-методическое пособие / В. А. Башкин ; Яросл. гос. ун-т. им. П. Г. Демидова — Ярославль : ЯрГУ, 2018. — 52 с.

В учебно-методическом пособии излагаются основы теории лямбда-исчисления. Рассматриваются базовые элементы формализма, приводятся примеры различных вычислений и структур данных.

Предназначено для студентов факультета информатики и вычислительной техники ЯрГУ, обучающихся по дисциплине «Теория вычислительных процессов и структур».

УДК 510.56(075)
ББК 3973.2-018я73

©ЯрГУ, 2018

Оглавление

Предисловие

Программы на традиционных языках программирования, таких как Си, Паскаль, Java и т. п., состоят из последовательности модификаций значений некоторого набора переменных, который называется состоянием.

Функциональное программирование реализует парадигму, отличную от рассмотренной модели. Функциональная программа представляет собой некоторое выражение (в строгом математическом смысле); выполнение программы означает вычисление значения этого выражения. Считая, что результат работы императивной программы полностью и однозначно определен ее входом, можно сказать, что финальное состояние (или любое промежуточное) являет собой некоторую функцию (в математическом смысле) от начального состояния. В функциональном программировании используется именно такая точка зрения: программа представляет собой выражение, соответствующее функции. Функциональные языки программирования поддерживают построение таких выражений.

Если теория машин Тьюринга является основой императивных языков программирования, то лямбда-исчисление служит базисом и математическим "фундаментом", на котором основаны функциональные языки программирования.

Лямбда-исчисление было изобретено в начале 30-х годов XX века американским логиком Алонзо Черчем, который использовал его в качестве формализма для обоснования математики[1]. Математическим доказательством реализуемости любого алгоритма при помощи одних только функций является фундаментальный

[1] **Краткая историческая справка:** Примерно в 1924 году Моисей Шёнфинкель разработал простую теорию функций. В 1934 году Алонзо Чёрч создал лямбда-исчисление и использовал его для разработки и исследования формальной теории множеств. В 1940-х годах Хаскелл Б. Карри представил комбинаторную логику — теорию функций без переменных.

результат Чёрча об универсальности лямбда-исчисления. Интересно, что лямбда-функции Чёрча появились раньше, чем машины Тьюринга, так что функциональное программирование получило математическое обоснование даже раньше императивного.

В настоящее время лямбда-исчисление является одной из основных формализаций, применяемых в исследованиях, связанных с языками программирования. Это связано в первую очередь со следующими факторами:

- это одна из немногих формализаций, которые могут быть непосредственно использованы 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 отображено в конструкции лямбда-исчисления;
- функциональные языки программирования являются удобной формой синтаксической записи для конструкций различных вариантов лямбда-исчисления; некоторые современные языки (например, Haskell) обеспечивают практически полное соответствие своей семантики с семантикой конструкций лямбда-исчисления.

Лямбда-исчисление лежит в основе большинства существующих функциональных языков программирования и их диалектов: семейства Лиспа (Common Lisp, Scheme, Clojure и др.) и семейства ML (Standard ML, Hope, Miranda, Haskell, F#, Swift и пр.).

1. Лямбда-исчисление

1.1. Лямбда-выражения

Многие императивные языки программирования допускают определение функций только с присваиванием им некоторых имен. Например, в языке Си функция всегда должна иметь конкретное имя. В то же время при работе со значениями (числами, строками, объектами, ...) используются два способа именования — конкретный (константы) и абстрактный (переменные и указатели).

При символьном представлении информации нет принципиальной разницы в природе изображения значений и функций. Следовательно, нет и препятствий для обработки представлений функций теми же средствами, какими обрабатываются значения. Представления функций можно строить из их частей и даже вычислять по мере поступления и обработки информации — именно так работают программы-компиляторы. Однако в мэйнстримовских языках программирования почему-то не принято к такой технике программирования допускать обычных пользователей (программистов).

Используемая в функциональных языках программирования 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. Например, $\lambda x.x^2$ представляет собой функцию, возводящую аргумент в квадрат.

Использование лямбда-нотации позволяет четко разделить случаи, когда под выражением вида $f(x)$ мы понимаем саму функцию f, а когда — её значение в точке x. Кроме того, лямбда-нотация

позволяет формализовать практически все виды математических конструкций. 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 (фактически — сколь угодно сложные).

Применение функции f к аргументу x мы будем обозначать как $f\,x$, то есть, в отличие от того, как это принято в математике, не будем использовать скобки. По причинам, которые станут ясны позднее, будем считать, что применение функции к аргументу ассоциативно влево, то есть $f\,x\,y$ означает $(f(x))(y)$. В качестве сокращения для выражений вида $\lambda x.\lambda y.E$ будем использовать запись $\lambda xy.E$ (аналогично для большего числа аргументов). Также будем считать, что "область действия" лямбда-выражения простирается вправо настолько, насколько это возможно, то есть, например, $\lambda x.xy$ означает $\lambda x.(xy)$, а не $(\lambda x.x)y$.

Формальная нотация (синтаксис).

Существуют три типа λ-выражений:

- *Переменная:* $x, y, z, \ldots$ Используется "математическое", а не "программистское" понимание этого термина. Здесь переменная — это не "именованная область памяти компьютера", а имя, обозначающее произвольную математическую абстракцию[1]. Переменные могут обозначать величины, выражения и даже функции. Более того, в одном и том же выражении одна и та же переменная в разных контекстах может обозначать совершенно разные вещи. Привязка конкретных объектов (выражений) к их абстрактным именам (переменным) производится при помощи *комбинаций* и *абстракций* (см. далее).

  Для обозначения произвольных переменных мы будем использовать символы $V, V_1, V_2, \ldots$.

[1] Ближайшим аналогом такой "переменной" в программировании можно считать аргумент макроподстановки или шаблона функции.

- *Комбинация,* то есть применение функции к аргументу. Если E_1 и E_2 — λ-выражения, то запись

  $$(E_1\ E_2)$$

  обозначает результат применения выражения (функции) E_1 к выражению (аргументу) E_2. E_1 также называется *оператором*, E_2 — *операндом.* Например, вычисление синуса числа π в таком синтаксисе будет выглядеть как $(sin\ \pi)$.

- *Абстракция,* то есть явное указание аргумента λ-выражения. Если V — переменная, а E — λ-выражение, то запись

  $$\lambda V.\ E$$

  обозначает λ-выражение, состоящее из *связанной переменной* V и *тела* E (используется также термин *абстракция выражения* E *по переменной* V).

  Можно представлять себе это выражение как определение функции с телом «E» и заголовком «$\lambda V.$» (сама функция безымянная, V — это её аргумент).

Весь синтаксис лямбда-выражений описывается очень простой формулой БНФ:

```
<выражение> ::= <переменная>
              | (<выражение> <выражение>)
              | (λ <переменная> . <выражение>)
```

Используя буквенные обозначения, ту же самую БНФ-формулу можно переписать ещё более компактно:

$$E ::= V \mid (E_1\ E_2) \mid \lambda V.\ E$$

Как будет показано далее, только что мы описали синтаксис полноценного языка программирования.

Неформальное описание процесса вычислений. Переменные описывают "статическую" структуру лямбда-выражения. Сам процесс вычислений полностью возложен на механизм абстракций и комбинаций, то есть описаний и вызовов функций. Остановимся на нем более подробно.

Лямбда-выражение $\lambda V.\, E$ описывает функцию с аргументом V и телом E, которое содержит какое-то (возможно, нулевое) количество упоминаний переменной V. Имени у этой функции нет, поэтому, чтобы её использовать, нам придётся при каждом вызове полностью записывать её код (то есть само выражение $\lambda V.\, E$). Вызов функции f от аргумента x в лямбда-нотации выглядит как $(f\, x)$ (комбинация f и x), поэтому вызов функции $\lambda V.\, E$ от аргумента G будет выглядеть как

$$((\lambda V.\, E)\ G)$$

Вычисление результата состоит в простой замене всех вхождений переменной V в выражении E на выражение G и отбрасывании заголовка функции $(\lambda V.)$:

$$((\lambda V.\, E)\ G) \to E[V := G]$$

Запись $x[y := z]$ обозначает выражение, полученное из выражения x заменой всех вхождений выражения y на выражение[2] z.

Пример 1.1. $((\lambda x.\, xx)\, z) = xx[x := z] = zz$;
$((\lambda x.\, yz)\, a) = yz[x := a] = yz$;
$((\lambda y.\, yzy)\, (\lambda x.cd)) = yzy[y := (\lambda x.cd)] = (\lambda x.cd)z(\lambda x.cd)$.

Использование абстракций и конверсий — очень сильный механизм, который позволяет закодировать любой алгоритм (любую машину Тьюринга). Рассмотрим несколько примеров простых функций.

[2]Более точное определение подстановки, учитывающее возможность конфликта имён переменных, будет дано в следующем параграфе.

Пример 1.2. Функция идентичности: $(\lambda x.\, x)$.

Действительно, легко убедиться в том, что для любого E выполняется $((\lambda x.\, x)\, E) = E$.

Пример 1.3. Функция переворота аргументов: $(\lambda x.\, (\lambda f.\, (f\, x)))$.

Эта функция, получив на вход последовательно два аргумента (сначала E_1, потом E_2), возвращает их в обратном порядке (сначала E_2, потом E_1):

$$(((\lambda x.\, (\lambda f.\, (f\, x)))\ E_1)\ E_2) = ((\lambda f.\, (f\, x))[x := E_1]\ E_2) =$$
$$((\lambda f.\, (f\, E_1))\ E_2) = (f\, E_1)[f := E_2] = (E_2\ E_1).$$

Упражнение 1.1. Определите, что делают функции:

(i) $(\lambda x.\, (\lambda y.\, y))$;

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

Условные обозначения. Легко заметить, что любое лямбда-выражение содержит немало скобок. Чтобы несколько облегчить синтаксис, будем использовать следующие правила обозначений:

1. В последовательности комбинаций (применений функций) приоритет всегда у самой левой, то есть запись $E_1\, E_2\, \ldots\, E_n$ означает $(\,(\ldots\, (E_1\, E_2)\, \ldots)\, E_n)$. Например:

   $E_1\, E_2$ означает $(E_1\, E_2)$,
   $E_1\, E_2\, E_3$ означает $((E_1\, E_2)\, E_3)$,
   $E_1\, E_2\, E_3\, E_4$ означает $(((E_1\, E_2)\, E_3)\, E_4)$;

2. Область действия заголовка функции («$\lambda V.$») расширяется вправо настолько далеко, насколько это возможно, то есть запись $\lambda V.\, E_1\, E_2\, \ldots\, E_n$ означает $(\lambda V.\, (E_1\, E_2\, \ldots\, E_n))$;

3. В последовательной записи абстракций (объявлений функций) приоритет всегда у самого правого аргумента, то есть запись $\lambda V_1\, \ldots\, V_n.\, E$ означает $(\lambda V_1.\, (\ldots\, .(\lambda V_n.\, E)\, \ldots))$. Например:

   $\lambda x\, y.\, E$ означает $(\lambda x.\, (\lambda y.\, E))$,

$\lambda x\, y\, z.\, E$ означает $(\lambda x.\, (\lambda y.\, (\lambda z.\, E)))$,
$\lambda x\, y\, z\, w.\, E$ означает $(\lambda x.\, (\lambda y.\, (\lambda z.\, (\lambda w.\, E))))$.

Пример 1.4. $\lambda xy.\, (\lambda z.\, zw)yx = (\lambda x.\, (\lambda y.\, (((\lambda z.\, zw)y)x))) = \lambda xy.\, ywx$.

1.2. Конверсии и редукция

Свободные и связанные переменные. Далее нам потребуются понятия свободного и связанного вхождения переменной. Скажем, что вхождение переменной V в выражение E *свободно*, если оно не принадлежит области действия «λV», и *связано* в противном случае. Например, в выражении

$$(\lambda x.\, \underline{y}\, x)(\lambda y.\, \underline{x}\, y)$$

подчеркнуты свободные вхождения переменных x и y (*свободные переменные* данного выражения).

Множество всех свободных переменных выражения E обозначим как $FV(E)$.

Подстановки. Рассмотрим выражения $E_1 = \lambda x.\, xy$ и $E_2 = xy$. Имена переменных в лямбда-исчислении — абстрактные, они не несут никакой смысловой нагрузки. Выражение E_2 мы с тем же успехом можем сформулировать как uw. Однако заметим, что согласно приведенным ранее неформальным правилам "вычисления" лямбда-функций при первом варианте обозначений выражение $(E_1\ E_2)$ вычислится как xyy, а при втором — как uvy. Очевидно, получились разные ответы: в первом случае даже число различных переменных в выражении меньше, чем во втором (две вместо трех). Причина — использование (в первом случае) одного и того же набора свободных переменных.

Таким образом, нам необходимо более строго описать вычисления, чтобы избежать таких конфликтов в именах переменных[3].

[3]Очевидно, что единственной причиной возникновения проблемы являет-

Для этого модифицируем правило подстановки выражения вместо переменной (обозначаемое $E[V := E']$) таким образом, чтобы подстановка не приводила к возникновению проблем, связанных с одинаковыми названиями переменных.

Определим подстановку $E[V := E']$ индуктивно по структуре произвольного лямбда-выражения E:

E	$E[V := E']$
V	E'
V' (где $V \neq V'$)	V'
$E_1\ E_2$	$E_1[V := E']\ E_2[V := E']$
$\lambda V.\, E_1$	$\lambda V.\, E_1$
$\lambda V'.\, E_1$ (где $V \neq V'$ и $V' \notin FV(E')$)	$\lambda V'.\, E_1[V := E']$
$\lambda V'.\, E_1$ (где $V \neq V'$ и $V' \in FV(E')$)	$\lambda V''.\, E_1[V' := V''][V := E']$, где V'' — переменная, такая что $V'' \notin FV(E')$ и $V'' \notin FV(E_1)$

Пример 1.5. Рассмотрим подстановку $(\lambda y.\, y\, x)[x := y]$. Поскольку переменная y свободна в выражении y, здесь следует применить последнее правило подстановки из таблицы. В качестве V'' возь-

ся желание человека использовать в тексте поменьше переменных, то есть везде писать x, y и т. д. Таким образом, самый простой способ избежать конфликтов — договориться о том, что в каждом исходном выражении есть свой собственный набор свободных переменных. Однако, к сожалению, это сделает формулы слишком громоздкими.

мём новую переменную z, которая не встречается ни в выражении x, ни в выражении y:

$$(\lambda y.\, y\, x)[x := y] \equiv \lambda z.\, (y\, x)[y := z][x := y] \equiv \lambda z.\, (z\, x)[x := y] \equiv \lambda z.\, z\, y\,.$$

Упражнение 1.2. Определите результаты подстановок:

(i) $(\lambda y.\, x\, (\lambda x.\, x))[x := (\lambda y.\, y\, x)]$,

(ii) $(y\, (\lambda z.\, x\, z))[x := (\lambda y.\, z\, y)]$.

Конверсии. В основе лямбда-исчисления лежат три операции конверсии, которые позволяют переходить от одного выражения к другому (эквивалентному). Процесс последовательного выполнения конверсий соответствует процессу вычисления функции (выполнения программы), описанной исходным лямбда-выражением.

Конверсии обозначаются греческими буквами α, β и η («альфа», «бета» и «эта») и определяются следующим образом:

- **α-конверсия:** Если $V' \notin FV(E)$, то

$$\lambda V.\, E \;\xrightarrow{\alpha}\; \lambda V'.\, E[V := V']\,,$$

- **β-конверсия:**

$$(\lambda V.\, E)\, E' \;\xrightarrow{\beta}\; E[V := E']\,,$$

- **η-конверсия:** Если $V \notin FV(E)$, то

$$\lambda V.\, (E\, V) \;\xrightarrow{\eta}\; E\,.$$

Рассмотрим конверсии более подробно.

Первый вид — **α-конверсия** — является вспомогательным механизмом для того, чтобы изменять имена связанных переменных. Лямбда-выражение (в данном случае это обязательно абстракция), к которому применяется α-конверсия, называется *α-редексом* (от английского REDucible EXpression). Правило α-конверсии всего лишь говорит о том, что в α-редексе связанные переменные могут быть переименованы, если это не приводит к конфликту имён.

Примеры допустимых α-конверсий:

$$\lambda x.\, x \;\xrightarrow{\alpha}\; \lambda y.\, y\,,$$

$$\lambda x.\, f\, x \;\xrightarrow{\alpha}\; \lambda y.\, f\, y\,.$$

Пример недопустимой α-конверсии:

$$\lambda xy.\, x\, y \;\overset{\alpha}{\not\longrightarrow}\; \lambda yy.\, y\, y\,.$$

Данная конверсия недопустима, поскольку после выполнения подстановки свободная переменная (бывшая x) становится связанной (y).

Второй вид — β**-конверсия** — наиболее важен с точки зрения программирования, поскольку он соответствует вычислению значения функции от аргумента. В качестве β-редекса может выступать только комбинация. Процедура β-конверсии аналогична вызову функции в языке программирования: тело E функции $\lambda V.\, E$ "выполняется" в ситуации, когда "формальному параметру" V присвоено значение "актуального параметра" E'.

Примеры допустимых β-конверсий:

$$(\lambda x.\, f\, x)\, E \;\xrightarrow{\beta}\; f\, E\,,$$

$$(\lambda x.\, x\, x)\, (\lambda y.\, y) \;\xrightarrow{\beta}\; (\lambda y.\, y)\, (\lambda y.\, y) \;\xrightarrow{\beta}\; \lambda y.\, y\,.$$

Третий вид — η**-конверсия** — формализует правило введения и удаления формального параметра. Как и в первом случае, редексом для η-конверсии может быть только абстракция (то есть определение функции). Сама η-конверсия выражает следующее свойство (называемое также свойством *экстенсиональности*): две функции идентичны, если они дают одинаковые результаты при одинаковых значениях входных параметров. Например, η-конверсия показывает, что выражения $\lambda x.\, (\mathbf{sin}\ x)$ и $\mathbf{sin}$ обозначают одну и ту же функцию.

Примеры допустимых η-конверсий:

$$\lambda x.\, f\, x \;\xrightarrow{\eta}\; f\,,$$

$$\lambda y.\, f\, x\, y \;\xrightarrow{\eta}\; f\, x\,.$$

Упражнение 1.3. Покажите, что

$$(\lambda f\, g\, x.\, f\, x\, (g\, x))\, (\lambda x\, y.\, x)\, (\lambda x\, y.\, x) \;=\; \lambda x.\, x\,.$$

Определения $\xrightarrow{\alpha}$, $\xrightarrow{\beta}$ и $\xrightarrow{\eta}$ могут быть обобщены:

- $E_1 \xrightarrow{\alpha} E_2$, если E_2 может быть получено из E_1 посредством α-конверсии какого-то подвыражения;
- $E_1 \xrightarrow{\beta} E_2$, если E_2 может быть получено из E_1 посредством β-конверсии какого-то подвыражения;
- $E_1 \xrightarrow{\eta} E_2$, если E_2 может быть получено из E_1 посредством η-конверсии какого-то подвыражения.

Равенство лямбда-выражений. Используя введенные правила конверсии, можно формально определить понятие *равенства* лямбда-выражений. Два выражения равны (что обозначается как $E_1 = E_2$), если от одного из них можно перейти к другому с помощью конечной последовательности конверсий.

Следует отличать понятие равенства выражений от понятия синтаксической эквивалентности (полного совпадения обозначающих выражения строк), которую мы будем обозначать специальным символом $\equiv$. Например, $\lambda x.x \not\equiv \lambda y.y$, но $\lambda x.x = \lambda y.y$.

Отношение равенства является симметричным: если $E_1 = E_2$, то и $E_2 = E_1$. Обратите внимание на то, что равенство двух выражений совсем не означает, что мы каждое из них можем конвертировать в другое: достаточно наличия конверсий только в одну сторону. Например:

$$(\lambda x.\, x\, x)\, f \;=\; f\, f\,,$$

$$(\lambda x.\, x\, x)\, f \xrightarrow{\beta} f\, f, \text{ но не существует } f\, f \xrightarrow{\alpha,\beta,\eta} (\lambda x.\, x\, x)\, f\,.$$

Редукция лямбда-выражений. С вычислительной точки зрения более интересно будет рассмотреть асимметричный вариант.

Определим отношение редукции лямбда-выражений следующим образом: выражение E_1 *редуцируемо* до выражения E_2 (обозначается $E_1 \longrightarrow E_2$), если от E_1 можно перейти к E_2 с помощью конечной последовательности конверсий. Очевидно, что

$$E_1 \longrightarrow E_2 \ \vee \ E_2 \longrightarrow E_1 \quad \Longrightarrow \quad E_1 = E_2 \,.$$

Несмотря на то что термин "редукция" подразумевает уменьшение размера лямбда-выражения, в действительности это может быть не так, что показывает следующий пример:

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

Тем не менее отношение редукции соответствует систематическим попыткам "вычислить" исходное выражение, последовательно вычисляя комбинации $f(x)$, где f — некоторая лямбда-абстракция. Когда к выражению невозможно применить никакую редукцию, кроме α-конверсии, будем говорить, что это выражение находится в *нормальной форме*.

Упражнение 1.4. Найдите нормальную форму лямбда-выражений (или покажите, что её не существует):

(i) $(\lambda x.\, x\, x)\,(\lambda x.\, x)$,

(ii) $(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$,

(iii) $(\lambda x\, y.\, f\,((\lambda z.\, z\, x)\, g\, y))\, h$.

Редукционные стратегии. Рассмотрим лямбда-выражение

$$(\lambda x.\, y\, x)((\lambda z.\, f\, z)\, g).$$

Легко показать, что оно редуцируемо к выражению $y\,(f\,g)$. Однако получить эту редукцию можно двумя различными способами при различной последовательности выполнения β-конверсий (подчеркнуты редуцируемые β-редексы):

1. $\underline{(\lambda x.\, y\, x)((\lambda z.\, f\, z)\, g)} \xrightarrow{\beta} y\,(\underline{(\lambda z.\, f\, z)\, g}) \xrightarrow{\beta} y\,(f\, g)$,

2. $(\lambda x.\, y\, x)(\underline{(\lambda z.\, f\, z)\, g}) \xrightarrow{\beta} \underline{(\lambda x.\, y\, x)(f\, g)} \xrightarrow{\beta} y\,(f\, g)$.

В рассмотренном случае мы оба раза пришли к результату за два шага. Но иногда выбор между редуцируемыми редексами означает выбор между конечной и бесконечной последовательностями редукций, то есть между завершением и зацикливанием программы. Например, если мы начнем редукцию с самого внутреннего β-редекса в следующем примере, то получим бесконечную последовательность редукций:

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

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

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

Оказывается, существует универсальная стратегия выбора редекса, которая гарантирует отсутствие зацикливаний в случае редуцируемости выражения:

Теорема 1.1. *Если $E \longrightarrow E'$, где E' находится в нормальной форме, то последовательность редукций, начинающаяся с E и заключающаяся в том, что для редукции всегда выбирается самый левый и самый внешний редекс, гарантированно завершится за конечное число шагов и приведет исходное выражение в нормальную форму.*

Понятие “самого левого и самого внешнего” редекса можно определить индуктивно: для выражения $(\lambda V.\, E_1)\, E_2$ это будет само выражение, для любого другого выражения $E_1\, E_2$ это будет

самый левый и самый внешний редекс в E_1, и, наконец, для абстракции $\lambda V.\, E$ это будет самый левый самый внешний редекс в E. Фактически, в условиях нашего конкретного синтаксиса, всегда редуцируется редекс, чей символ λ находится левее всего.

Такую стратегию мы будем называть *нормальной* или *нормализованной*. Она всегда приводит к результату, но не всегда самым эффективным образом. Например, рассмотрим выражение

$$(\lambda x.\, x\, x\, x)\, ((\lambda y.\, f\, y)\, g)\,.$$

Его вычисление по нормальной стратегии требует выполнения четырех β-конверсий:

$$\begin{aligned}(\lambda x.\, x\, x\, x)\, ((\lambda y.\, f\, y)\, g) &\xrightarrow{\beta} ((\lambda y.\, f\, y)\, g)\, ((\lambda y.\, f\, y)\, g)\, ((\lambda y.\, f\, y)\, g) \\ &\xrightarrow{\beta} (f\, g)\, ((\lambda y.\, f\, y)\, g)\, ((\lambda y.\, f\, y)\, g) \\ &\xrightarrow{\beta} (f\, g)\, (f\, g)\, ((\lambda y.\, f\, y)\, g) \\ &\xrightarrow{\beta} (f\, g)\, (f\, g)\, (f\, g)\ .\end{aligned}$$

Но у данного выражения существует и более короткая редукция, состоящая всего из двух конверсий:

$$(\lambda x.\, x\, x\, x)\, ((\lambda y.\, f\, y)\, g) \xrightarrow{\beta} (\lambda x.\, x\, x\, x)\, (f\, g) \xrightarrow{\beta} (f\, g)\, (f\, g)\, (f\, g)\,.$$

Теорема Чёрча-Россера. Следующая фундаментальная теорема утверждает, что, если мы начнем с одного и того же выражения и проведем две произвольные конечные последовательности редукций, всегда будут существовать ещё две последовательности редукций, которые приведут нас к одному и тому же выражению (хотя оно может и не находиться в нормальной форме).

Теорема 1.2. (Чёрч, Россер) *Если* $E \longrightarrow E_1$ *и* $E \longrightarrow E_2$, *то существует* E' *такое, что* $E_1 \longrightarrow E'$ *и* $E_2 \longrightarrow E'$.

Эта теорема имеет несколько важных следствий.

Следствие 1.1. *Если* $E_1 = E_2$, *то существует* E' *такое, что* $E_1 \longrightarrow E'$ *и* $E_2 \longrightarrow E'$.

Следствие 1.2. *Если* $E = E_1$ *и* $E = E_2$, *где* E_1 *и* E_2 *находятся в нормальной форме, то* $E_1 \equiv_\alpha E_2$, *т. е.* E_1 *и* E_2 *синтаксически эквивалентны с точностью до переименования переменных.*

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

С вычислительной точки зрения это означает следующее. Нормальная стратегия является в некотором смысле "универсальной", поскольку она *всегда* приведет к результату, если он достижим с помощью хоть 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 нормальной стратегии.

Отложенные вычисления. С теоретической точки зрения нормальный порядок редукции выражений наиболее предпочтителен, поскольку, если какая либо стратегия завершается, то завершится и она. Такая стратегия известна как *вызов по имени*. С практической точки зрения, как мы видели, такая стратегия имеет существенные недостатки.

Рассмотрим выражение

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

Нормальная редукция приводит его к следующему виду:

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

Таким образом, необходимо трижды вычислять значение одного и того же выражения. На практике это недопустимо. Существуют два основных подхода к решению этой проблемы.

При первом подходе отказываются от нормальной стратегии и вычисляют аргументы функций до того, как передать их значения в функцию. Такой подход называют *передачей по значению*. При этом вычисление некоторых выражений, завершающееся при нормальной стратегии, может привести к бесконечной последовательности редукций. Однако такая стратегия позволяет получать эффективный машинный код.

При другом подходе, использующемся в современных компиляторах языков функционального программирования, нормальная стратегия редукций сохраняется. Однако различные возникающие подвыражения разделяются и никогда не вычисляются более одного раза. Такой способ вызова называется *ленивым* или *вызовом по необходимости*, поскольку выражения вычисляются только тогда, когда их значения действительно необходимы для работы программы. При этом оптимизирующие компиляторы сами выявляют места программы, в которых можно обойтись без отложенных вычислений.

1.3. Комбинаторы

Теория комбинаторов была разработана Шёнфинкелем ещё до создания лямбда-исчисления, однако для удобства мы будем рассматривать ее в терминах лямбда-исчисления.

Комбинатором будем называть лямбда-терм, не содержащий свободных переменных. Такой терм является *замкнутым*, то есть имеющим фиксированный смысл независимо от значения любых используемых переменных.

В теории комбинаторов установлено, что с помощью нескольких 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 их следующим образом:

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

Комбинатор I ("комбинатор тождества") является функцией идентичности, возвращающей свой аргумент.

Комбинатор K (также называемый "канцелятор") 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 (также называемый "коннектор") является "разделяющим": он использует две функции и аргумент и "разделяет" аргумент между функциями.

Помимо перечисленных базовых комбиаторов, в работах создателей комбинаторной логики Моисея Шёнфинкеля и Хаскелла Карри были введены и другие удобные комбинаторы, в частности, "композитор" B, "пермутатор" C и "дубликатор" W. Однако все они выражаются через базовые:

Теорема 1.3. *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. Будем использовать $A = K$. Таким образом, $I = S\ K\ K$ и поэтому I можно исключить изо всех выражений, составленных из комбинаторов.

Мы представили комбинаторы как некоторые лямбда-термы, однако имеется теория, в которой они являются базовым понятием. Так же, как и в лямбда-исчислении, определяется формальный синтаксис, который вместо лямбда-абстракций содержит комбинаторы. Вместо правил α, β и η-конверсии вводятся правила конверсии для выражений, содержащих комбинаторы, например $Kxy \to x$. Как независимая теория, теория комбинаторов обладает многими аналогиями с лямбда-исчислением, в частности для нее выполняется теорема Черча-Россера. Однако эта теория менее интуитивно понятна.

Комбинаторы представляют не только теоретический интерес. Лямбда-исчисление может рассматриваться как простой функциональный язык программирования, составляющий ядро реальных языков программирования, таких как ML или Haskell. Теорема 1.3 показывает, что лямбда-исчисление может быть "скомпилировано" в "машинный код" комбинаторов. Комбинаторы используются как метод реализации функциональных языков на уровне программного, а также аппаратного обеспечения.

2. Язык программирования

Язык лямбда-исчисления на первый взгляд может показаться очень примитивным. Но на самом деле с его помощью можно закодировать практически любые конструкции, необходимые для программирования.

Лямбда-исчисление достаточно выразительно для кодирования булевых значений, упорядоченных пар, натуральных чисел и списков, то есть всех основных структур данных, которые могут возникнуть в программе. Способы кодирования могут показаться не совсем естественными, и они, конечно, не являются вычислительно эффективными. В этом они напоминают машинные коды и программы для машины Тьюринга. Однако, в отличие от программ машины Тьюринга, лямбда-программы сами по себе представляют математический интерес и исследователи снова и снова возвращаются к ним в своих теоретических изысканиях. Многие из рассматриваемых способов кодирования воплощают идею о том, что входные данные программы (функции) могут содержать (частично или полностью) саму её структуру управления.

2.1. Логика

Наша цель — закодировать объекты и структуры таким образом, чтобы они отвечали требуемым свойствам. Например, лямбда-коды логических значений *true* и *false* (обозначаемые **true** и **false** соответственно), а также код логической функции отрицания $\neg$ (обозначаемый **not**) должны удовлетворять следующим двум требованиям:

$$\begin{aligned} \mathbf{not\ true} &= \mathbf{false}, \\ \mathbf{not\ false} &= \mathbf{true}. \end{aligned}$$

Кодирующее конъюнкцию $\wedge$ лямбда-выражение **and** должно удовлетворять требованиям:

$$\begin{aligned}
\textbf{and true true} &= \textbf{true},\\
\textbf{and true false} &= \textbf{false},\\
\textbf{and false true} &= \textbf{false},\\
\textbf{and false false} &= \textbf{false}.
\end{aligned}$$

В свою очередь, кодирующее дизъюнкцию $\vee$ лямбда-выражение **or** должно удовлетворять требованиям:

$$\begin{aligned}
\textbf{or true true} &= \textbf{true},\\
\textbf{or true false} &= \textbf{true},\\
\textbf{or false true} &= \textbf{true},\\
\textbf{or false false} &= \textbf{false}.
\end{aligned}$$

Осталось найти подходящие лямбда-выражения. На самом деле таких выражений существует бесконечно много, но традиционно используются следующие:

$$\begin{aligned}
\texttt{LET}\ \ \textbf{true} &= \lambda xy.\ x\\
\texttt{LET}\ \ \textbf{false} &= \lambda xy.\ y\\
\texttt{LET}\ \ \textbf{not} &= \lambda t.\ t\ \textbf{false}\ \textbf{true}
\end{aligned}$$

Использованный здесь оператор $\texttt{LET}\ A\ E$ обозначает привязку имени A лямбда-выражению E. Далее вместо самого выражения мы можем писать его имя (как уже сделали в случае объявления выражения **not**).

Легко убедиться в том, что введённые нами конструкции удовлетворяют требуемым свойствам. Действительно, свойство отрицания $\textbf{not false} = \textbf{true}$ легко выводится с помощью конверсий:

$$\begin{aligned}
\textbf{not true} &= (\lambda t.\ t\ \textbf{false}\ \textbf{true})\ \textbf{true} && \text{(по определению \textbf{not})}\\
&= \textbf{true}\ \textbf{false}\ \textbf{true} && (\beta\text{-конверсия})\\
&= (\lambda xy.\ x)\ \textbf{false}\ \textbf{true} && \text{(по определению \textbf{true})}\\
&= (\lambda y.\ \textbf{false})\ \textbf{true} && (\beta\text{-конверсия})\\
&= \textbf{false} && (\beta\text{-конверсия})
\end{aligned}$$

Упражнение 2.1. Покажите, что **not false** $=$ **true**.

Похожим образом определяются конъюнкция и дизъюнкция:

$$\texttt{LET}\quad \textbf{and} \;=\; \lambda xy.\; x\; y\; \textbf{false}$$
$$\texttt{LET}\quad \textbf{or} \;=\; \lambda xy.\; x\; \textbf{true}\; y$$

Проверим свойство **and true false** $=$ **false**:

$$\begin{aligned}
\textbf{and true false} &= (\lambda xy.\; x\; y\; \textbf{false})\; \textbf{true false} && (\text{по опр. } \textbf{and})\\
&= (\lambda y.\; \textbf{true}\; y\; \textbf{false})\; \textbf{false} && (\beta\text{-конверсия})\\
&= \textbf{true false false} && (\beta\text{-конверсия})\\
&= (\lambda xy.\; x)\; \textbf{false false} && (\text{по опр. } \textbf{true})\\
&= (\lambda y.\; \textbf{false})\; \textbf{false} && (\beta\text{-конверсия})\\
&= \textbf{false} && (\beta\text{-конверсия})
\end{aligned}$$

Упражнение 2.2. Проверьте остальные свойства операций конъюнкции и дизъюнкции.

При помощи лямбда-исчисления мы можем реализовать даже ветвление. Рассмотрим функцию от трех аргументов **ifthenelse**, обладающую следующими двумя свойствами:

$$\textbf{ifthenelse true}\; E_1\; E_2 \;=\; E_1,$$
$$\textbf{ifthenelse false}\; E_1\; E_2 \;=\; E_2.$$

Таким образом, эта функция — оператор ветвления, который осуществляет переход по ветке E_1 в случае истинности заданного в первом аргументе условия и по ветке E_2 — в случае его ложности. Определенные нами конструкции выражений **true** и **false** (функции от двух аргументов, возвращающие свой первый и второй аргументы соответственно) позволяют потроить очень простое лямбда-выражение для ветвления:

$$\texttt{LET}\quad \textbf{ifthenelse} \;=\; \lambda xyz.\; x\; y\; z$$

Упражнение 2.3. Проверьте свойства оператора ветвления.

2.2. Пары

Мы можем представить упорядоченные пары следующим образом:

$$\texttt{LET} \quad (E_1, E_2) \;=\; \lambda f.f \;\; E_1 \;\; E_2$$

Введём конкретное обозначение:

$$\texttt{LET} \quad \textbf{pair} \;=\; \lambda xyf.f \;\; x \;\; y$$

С точк зрения синтаксиса скобки не являются обязательными, но мы часто используем их для упрощения понимания или для принудительной группировки. Фактически мы просто рассматриваем запятую как инфиксный оператор наподобие $+$. Учитывая приведенное выше определение, соответствующие деструкторы для пар можно определить как:

$$\begin{aligned} \texttt{LET} \quad \textbf{fst}\ p \;&=\; p\ \textbf{true} \\ \texttt{LET} \quad \textbf{snd}\ p \;&=\; p\ \textbf{false} \end{aligned}$$

Легко увидеть, что всё работает как надо:

$$\begin{aligned} \textbf{fst}\ (p,q) \;&=\; (p,q)\ \textbf{true} \\ &=\; (\lambda f.\ f\ p\ q)\ \textbf{true} \\ &=\; \textbf{true}\ p\ q \\ &=\; (\lambda xy.\ x)\ p\ q \\ &=\; p \end{aligned}$$

и, соответственно,

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

Мы можем построить тройки, четверки, пятёрки, действительно произвольные n-кортежи, просто итерируя эту конструкцию "спаривания":

$$(E_1, E_2, \ldots, E_n) = (E_1, (E_2, \ldots, (E_{n-1}, E_n)\ldots))\,.$$

Достаточно просто сказать, что оператор инфиксной запятой является право-ассоциативным, и всё становится понятным без каких-либо дополнительных соглашений. Например:

$$\begin{aligned}\textbf{snd}\ (p,q,r,s) &= (p,(q,(r,s)))\\ &= \lambda f.\ f\ p\ (q,(r,s))\\ &= \lambda f.\ f\ p\ (\lambda f.\ f\ q\ (r,s))\\ &= \lambda f.\ f\ p\ (\lambda f.\ f\ q\ (\lambda f.\ f\ r\ s))\\ &= \lambda f.\ f\ p\ (\lambda g.\ g\ q\ (\lambda h.\ h\ r\ s))\,,\end{aligned}$$

где в последней строке мы ради большей понятности выполнили альфа-конверсию. Хотя кортежи строятся в "плоской" манере, использования повторяющихся объединений позволяет достаточно легко создавать произвольные конечно-ветвящиеся древовидные структуры. Наконец, если кто-то предпочитает обычные функции над декартовыми произведениями нашим "каррированным" функциям, то одно в другое можно легко преобразовать, используя следующие операторы:

$$\begin{aligned}\texttt{LET}\quad CURRY\ f &= \lambda xy.\ f(x,y)\\ \texttt{LET}\quad UNCURRY\ g &= \lambda p.\ g\ (\textbf{fst}\ p)\ (\textbf{snd}\ p)\end{aligned}$$

Эти специальные операции для пар легко обобщаются на произвольные n-кортежи. Например, мы можем определить селекторную функцию для получения i-й компоненты плоского кортежа p. Обозначим эту операцию как $(p)_i$ и определим $(p)_1 = \mathbf{fst}\ p$ и $(p)_i = \mathbf{fst}\ (\mathbf{snd}^{i-1} p)$ для остальных i. Аналогичным образом мы можем обобщить $CURRY$ и $UNCURRY$:

$$\texttt{LET}\quad CURRY_n\ f\ =\ \lambda x_1 \cdots\ x_n.\ f(x_1, \ldots, x_n)$$
$$\texttt{LET}\quad UNCURRY_n\ g\ =\ \lambda p.\ g\ (p)_1 \cdots\ (p)_n$$

Теперь мы можем ввести обозначение $\lambda(x_1, \ldots, x_n).t$ для выражения $UNCURRY_n(\lambda x_1 \cdots x_n.t)$, что дает нам естественный способ записи функций от декартовых произведений.

2.3. Числа

Требуется построить лямбда-выражения, описывающие все целые неотрицательные числа, а также набор примитивных арифметических операций (следования **suc**, предшествования **pre** и сложения **add**). Полученные выражения должны удовлетворять требованиям:

$$\begin{aligned} \mathbf{suc\ n} &= \mathbf{n+1}, \\ \mathbf{pre\ n} &= \mathbf{n-1}, \\ \mathbf{add\ m\ n} &= \mathbf{m+n}. \end{aligned}$$

Существуют различные способы представления натуральных чисел при помощи лямбда-выражений. Следующий считается классическим и принадлежит Алонзо Чёрчу (“числа Чёрча”, “нумералы Чёрча”):

$$\begin{aligned}
\texttt{LET}\quad \mathbf{0} &= \lambda f x.\ x \\
\texttt{LET}\quad \mathbf{1} &= \lambda f x.\ f\ x \\
\texttt{LET}\quad \mathbf{2} &= \lambda f x.\ f\ (f\ x) \\
&\vdots \\
\texttt{LET}\quad \mathbf{n} &= \lambda f x.\ f^n\ x \\
&\vdots
\end{aligned}$$

Арифметические операции **suc** и **add** определяются как:

$$\begin{aligned}
\texttt{LET}\quad \mathbf{suc} &= \lambda n f x.\ n\ f\ (f\ x) \\
\texttt{LET}\quad \mathbf{add} &= \lambda m n f x.\ m\ f\ (n\ f\ x)
\end{aligned}$$

Проще всего понять эти определения, если представлять арифметические действия как операции над унарным представлением натуральных чисел (где в качестве количества единиц берется количество "вызовов" функции справа от точки в соответствующем лямбда-выражении).

Рассмотрим пример вычисления функции следования:

$$\begin{aligned}
\mathbf{suc\ 2} &= (\lambda n f x.\ n\ f\ (f\ x))\ (\lambda g y.\ g\ (g\ y)) \\
&= \lambda f x.\ (\lambda g y.\ g\ (g\ y))\ f\ (f\ x) \\
&= \lambda f x.\ (\lambda y.\ f\ (f\ y))\ (f\ x) \\
&= \lambda f x.\ f\ (f\ (f\ x)) \\
&= \mathbf{3}\,.
\end{aligned}$$

Упражнение 2.4. Покажите, что $\mathbf{add\ 3\ 2} = \mathbf{5}$.

Упражнение 2.5. Покажите, что $\mathbf{add\ m\ n} = \mathbf{m} + \mathbf{n}$.

Упражнение 2.6. Покажите, что $\lambda mn.m$ **suc** n также осуществляет сложение чисел Чёрча.

Выразить функцию предшествования **pre** оказывается гораздо труднее. Эту задачу решил Клини в 1935 году:

$$\texttt{LET}\quad \mathbf{pre} \;=\; \lambda nfx.\; n\;(\lambda gh.\; h\;(g\;f))\;(\lambda u.\; x)\;(\lambda w.\; w)$$

Упражнение 2.7. Покажите, что $\mathbf{pre}\;\mathbf{2} \;=\; \mathbf{1}$.

Операцию умножения можно ввести несколькими различными способами, например:

$$\texttt{LET}\quad \mathbf{mul}_1 \;=\; \lambda mn.\; m\;(\mathbf{add}\;n)\;\mathbf{0}$$
$$\texttt{LET}\quad \mathbf{mul}_2 \;=\; \lambda mnf.\; m\;(n\;f)$$

Первый способ соответствует m-кратному прибавлению числа n к нулю. Второй способ несколько менее очевиден.

Упражнение 2.8. Покажите, что $\mathbf{mul}_1\;\mathbf{2}\;\mathbf{2} \;=\; \mathbf{mul}_2\;\mathbf{2}\;\mathbf{2} \;=\; \mathbf{4}$.

Операция возведения в степень выглядит несложно:

$$\texttt{LET}\quad \mathbf{expt} \;=\; \lambda mnfx.\; n\;m\;f\;x$$

Упражнение 2.9. Проверьте правильность работы **expt** на нескольких примерах.

Определим также предикат **iszero** проверки числа на 0, который соединит построенные нами логику и арифметику:

$$\texttt{LET}\quad \mathbf{iszero} \;=\; \lambda n.\; n\;(\lambda x.\; \mathbf{false})\;\mathbf{true}$$

Упражнение 2.10. Покажите, что:
(i) $\mathbf{iszero}\;\mathbf{0} \;=\; \mathbf{true}$,
(ii) $\mathbf{iszero}\;\mathbf{5} \;=\; \mathbf{false}$.

И, наконец, последним примером в данном разделе будет реализация знаменитой функции Аккермана:

$$\texttt{LET}\quad \mathbf{ack} \;=\; \lambda m.\; m\;(\lambda f n.\; n\;f\;(f\;\mathbf{1}))\;\mathbf{suc}$$

Чтоб убедиться в корректности определения, нам нужно доказать следующие рекурсивные соотношения (классическое определение функции Аккермана):

$$\begin{aligned}\mathbf{ack\ 0\ n} &= \mathbf{n+1},\\ \mathbf{ack\ (m+1)\ 0} &= \mathbf{ack\ m\ 1},\\ \mathbf{ack\ (m+1)\ (n+1)} &= \mathbf{ack\ m\ (ack\ (m+1)\ n)}.\end{aligned}$$

Проверим первое соотношение:

$$\begin{aligned}\mathbf{ack\ 0\ n} &= \mathbf{0}\;(\lambda f n.\; n\;f\;(f\;\mathbf{1}))\;\mathbf{suc\ n}\\ &= \mathbf{suc\ n}\\ &= \mathbf{n+1}\,.\end{aligned}$$

Для оставшихся двух соотношений заметим, что:

$$\begin{aligned}\mathbf{ack\ (m+1)\ n} &= \mathbf{(m+1)}\;(\lambda f n.\; n\;f\;(f\;\mathbf{1}))\;\mathbf{suc\ n}\\ &= (\lambda f n.\; n\;f\;(f\;\mathbf{1}))\;(\mathbf{m}\;(\lambda f n.\; n\;f\;(f\;\mathbf{1}))\;\mathbf{suc})\;\mathbf{n}\\ &= (\lambda f n.\; n\;f\;(f\;\mathbf{1}))\;(\mathbf{ack\ m})\;\mathbf{n}\\ &= \mathbf{n}\;(\mathbf{ack\ m})\;(\mathbf{ack\ m\ 1})\,.\end{aligned}$$

Теперь проверим

$$\begin{aligned}\mathbf{ack\ (m+1)\ 0} &= \mathbf{0}\;(\mathbf{ack\ m})\;(\mathbf{ack\ m\ 1})\\ &= \mathbf{ack\ m\ 1}\end{aligned}$$

и

$$\begin{aligned}\mathbf{ack\ (m+1)\ (n+1)} &= \mathbf{(n+1)}\;(\mathbf{ack\ m})\;(\mathbf{ack\ m\ 1})\\ &= \mathbf{ack\ m}\;(\mathbf{n}\;(\mathbf{ack\ m})\;(\mathbf{ack\ m\ 1}))\\ &= \mathbf{ack\ m}\;(\mathbf{ack\ (m+1)\ n})\,.\end{aligned}$$

Ключом к приведённым выше вычислениям является итерация функции **ack m**.

Таким образом, лямбда-исчисление позволяет достаточно удобно кодировать даже арифметические функции, не являющиеся примитивно-рекурсивными.

Упражнение 2.11. Определите, какую арифметическую операцию над числами Чёрча производит функция $\lambda nfx.\ f\ (n\ f\ x)$.

Упражнение 2.12. Определите, какую арифметическую операцию над числами Чёрча производит функция $\lambda mn.\ n\ m$.

2.4. Списки

Числа Чёрча могут быть обобщены для представления списков. Список $[x_1, x_2, \ldots, x_n]$ в таком случае можно было бы представить как функцию от f и y с телом $fx_1(fx_2 \ldots (fx_n y) \ldots)$. Такие списки содержали бы собственную структуру управления.

В качестве альтернативы представим списки через операцию образования пар. Этот способ кодирования легче понять, потому что он ближе к прикладным реализациям. Список $[x_1, x_2, \ldots, x_n]$ будет представлен как $x_1 :: x_2 :: \ldots :: x_n :: nil$. Чтобы операции были максимально простыми, мы будем использовать два уровня пар. Каждая "ячейка списка" $x :: y$ будет кодироваться как $(\mathbf{false}, (x, y))$, где **false** является отличительным полем (тегом). По правилам, nil тоже следовало бы представить парой, в которой первый компонент был бы равен **true**, но работает и более простое определение.

Итак, вот наше кодирование списков:

$$\begin{aligned}
\texttt{LET}\quad & \mathbf{nil} = \lambda z.\, z \\
\texttt{LET}\quad & \mathbf{cons} = \lambda xy.\, \mathbf{pair}\ \mathbf{false}\ (\mathbf{pair}\ xy) \\
\texttt{LET}\quad & \mathbf{null} = \mathbf{fst} \\
\texttt{LET}\quad & \mathbf{hd} = \lambda z.\, \mathbf{fst}\ (\mathbf{snd}\ z) \\
\texttt{LET}\quad & \mathbf{tl} = \lambda z.\, \mathbf{snd}\ (\mathbf{snd}\ z)
\end{aligned}$$

Следующие свойства легко проверить; они выполняются для любых выражений M и N:

$$\begin{aligned}
\mathbf{null}\ \mathbf{nil} &= \mathbf{true}\,, \\
\mathbf{null}\ (\mathbf{cons}\ M\ N) &= \mathbf{false}\,, \\
\mathbf{hd}\ (\mathbf{cons}\ M\ N) &= M\,, \\
\mathbf{tl}\ (\mathbf{cons}\ M\ N) &= N\,.
\end{aligned}$$

Обратите внимание, что $\mathbf{null}\ \mathbf{nil} = \mathbf{true}$ вышло действительно случайно, в то время как другие соотношения являются следствием использования наших операций над парами.

Также заметьте, что ключевые характеристические соотношения $\mathbf{hd}\ (\mathbf{cons}\ M\ N) = M$ и $\mathbf{tl}\ (\mathbf{cons}\ M\ N) = N$ выполняются для любых M и N, в том числе даже для таких выражений, которые не имеют нормальных форм! Таким образом, **pair** и **cons** — "ленивые" операторы, то есть они не вычисляют свои аргументы. Следовательно, после введения рекурсивных определений мы сможем с их помощью работать даже с бесконечными списками.

Упражнение 2.13. Измените кодировку списков, чтобы получить кодировку натуральных чисел.

Упражнение 2.14. Рассмотрим следующее альтернативное определение списков, предложенное Маэрсоном:

$$
\begin{aligned}
\textbf{nil} &= \lambda cn.\ n \\
\textbf{cons} &= \lambda xlcn.\ c\ x\ (l\ c\ n) \\
\textbf{head} &= \lambda l.\ l\ (\lambda xy.\ x)\ \textbf{nil} \\
\textbf{tail} &= \lambda l.\ \textbf{snd}\ (l\ (\lambda xp.\ (\textbf{cons}\ x\ (\textbf{fst}\ p),\ \textbf{fst}\ p))\ (\textbf{nil},\ \textbf{nil})) \\
\textbf{append} &= \lambda l_1 l_2.\ l_1\ \textbf{cons}\ l_2 \\
\textbf{applists} &= \lambda L.\ L\ \textbf{append nil} \\
\textbf{map} &= \lambda fl.\ l\ (\lambda x.\ \textbf{cons}\ (f\ x))\ \textbf{nil} \\
\textbf{length} &= \lambda l.\ l\ (\lambda x.\ \textbf{suc})\ \textbf{0} \\
\textbf{tack} &= \lambda xl.\ l\ \textbf{cons}\ (\textbf{cons}\ x\ \textbf{nil}) \\
\textbf{reverse} &= \lambda l.\ l\ \textbf{tack nil} \\
\textbf{filter} &= \lambda lp.\ l\ (\lambda x.\ (p\ x)\ (\textbf{cons}\ x)\ (\lambda y.\ y))\textbf{nil}\,.
\end{aligned}
$$

Почему Маэрсон называет их “списками Чёрча”? Что делают все эти функции?

2.5. Рекурсивные вызовы

Уже построенные нами структуры описывают всё же достаточно простой набор функций. Для получения полноценного языка программирования, эквивалентного всему классу частично-рекурсивных функций, нам необходимо научиться описывать рекурсию. 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 в лямбда-исчислении так называемых операторов неподвижной точки. Замкнутое[1] лямбда-выражение **Fix** называется *оператором неподвижной точки*[2] (*fixed-point operator*), если для любого лямбда-выра-

[1] То есть не содержащее свободных переменных.

[2] Также используют термин *комбинатор неподвижной точки*.

жения E выполняется:

$$\mathbf{Fix}\ E\ =\ E\ (\mathbf{Fix}\ E)$$

Таким образом, оператор неподвижной точки по заданной функции f (лямбда-выражению E в нашем случае) возвращает неподвижную точку этой функции, то есть такое x (лямбда-выражение **Fix** E), при котором $f(x) = x$.

Осталось придумать такое выражение **Fix**. Известно, что существует бесконечно много операторов неподвижной точки. Первый из них, найденный Карри, обычно обозначается как $\mathbf{Y}$ и определяется следующим образом:

$$\texttt{LET}\quad \mathbf{Y}\ =\ \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Часто его называют “парадоксальным комбинатором”.

Убедимся в том, что у любого выражения существует неподвижная точка, а $\mathbf{Y}$ действительно является оператором неподвижной точки:

$$\begin{array}{lcll}
\mathbf{Y}\ E & = & (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ E & (\text{определение } \mathbf{Y}) \\
 & = & (\lambda x.\ E\ (x\ x))\ (\lambda x.\ E\ (x\ x)) & (\beta\text{-конверсия}) \\
 & = & E\ ((\lambda x.\ E\ (x\ x))\ (\lambda x.\ E\ (x\ x))) & (\beta\text{-конверсия}) \\
 & = & E\ (\mathbf{Y}\ E) & (\text{см. второй шаг})
\end{array}$$

Таким образом, действительно

$$\mathbf{Y}\ E\ =\ E\ (\mathbf{Y}\ E)\,.$$

Хотя с математической точки зрения все выкладки в проверке оператора $\mathbf{Y}$ верны, с вычислительной стороны такое определение вызывает трудности, поскольку использует лямбда-равенство,

а не только лямбда-редукции (на последнем шаге мы использовали *обратную* β*-конверсию*). По этой причине можно предпочесть следующее определение оператора неподвижной точки, принадлежащее Тьюрингу:

$$\texttt{LET} \quad \mathbf{T} \;=\; (\lambda xy.\; y\;(x\;x\;y))\;(\lambda xy.\; y\;(x\;x\;y))$$

Упражнение 2.15. Покажите, что $\mathbf{T}$ является оператором неподвижной точки.

Упражнение 2.16. Покажите, что выражение

$$\mathbf{Y}_1 \;=\; \mathbf{Y}\;(\lambda yf.\; f\;(y\;f))$$

является оператором неподвижной точки.

Покажем, как оператор неподвижной точки может помочь в определении рекурсивных функций. Рассмотрим в качестве примера функцию факториала. Мы хотим определить функцию **fact**, которая обладает следующим свойством (записано в синтаксисе "обычного" языка программирования):

$$\mathbf{fact}(\mathtt{n}) \;=\; \mathtt{if}\;(\mathtt{n} = 0)\;\mathtt{then}\;\mathtt{1}\;\mathtt{else}\;(\mathtt{n} * \mathbf{fact}(\mathtt{n} - \mathtt{1}))\,.$$

То же свойство в терминах чисел Чёрча и соответствующих введённых нами функций будет выглядеть как:

$$\mathbf{fact} \;=\; \lambda n.\; \mathbf{ifthenelse}\;(\mathbf{iszero}\;n)\;\mathbf{1}\;(\mathbf{mul}\;n\;(\mathbf{fact}\;(\mathbf{pre}\;n)))\,.$$

Это выражение, в свою очередь, эквивалентно

$$\mathbf{fact} \;=\; (\lambda fn.\; \mathbf{ifthenelse}\;(\mathbf{iszero}\;n)\;\mathbf{1}\;(\mathbf{mul}\;n\;(f\;(\mathbf{pre}\;n))))\;\mathbf{fact}\,.$$

Отсюда можно заключить, что **fact** является неподвижной точкой некоторого выражения F следующего вида:

$$F \;=\; \lambda fn.\; \mathbf{ifthenelse}\;(\mathbf{iszero}\;n)\;\mathbf{1}\;(\mathbf{mul}\;n\;(f\;(\mathbf{pre}\;n)))\,.$$

Действительно, согласно нашим выкладкам, **fact** $=$ F (**fact**), что и соответствует понятию неподвижной точки. В определении выражения F не упоминается в явном виде **fact**, так что мы можем однозначно построить его с помощью применения оператора неподвижной точки к выражению F:

$$\textbf{fact} \;=\; \mathbf{Y}\; F\,.$$

Введённое нами определение рекурсивной функции **fact** конструктивно и само не содержит рекурсию (в правой части нет упоминаний **fact**).

Упражнение 2.17. Покажите, что **fact 3** $=$ **6**.

Любую рекурсивную функцию можно рассматривать как неподвижную точку какой-то "нерекурсивной" функции, и таким образом она может быть выражена как лямбда-выражение (с использованием **Y**). В частности, можно переопределить при помощи рекурсии операции и предикаты для натуральных чисел. Например, умножение **mul** можно описать следующими лямбда-выражениями:

$$\texttt{LET}\quad \textbf{mulfn} \;=$$
$$(\lambda f m n.\ (\textbf{ifthenelse}\ (\textbf{iszero}\ m)\ \textbf{0}\ (\textbf{add}\ n\ (f\ (\textbf{pre}\ m)\ n))))$$

$$\texttt{LET}\quad \textbf{mul} \;=\; \mathbf{Y}\ \textbf{mulfn}$$

Здесь выражение **mulfn** описывает структуру рекурсивного умножения в "нерекурсивном" виде (поскольку f — не конкретная функция умножения, а некая абстрактная функциональная переменная), а выражение **mul** получается из **mulfn** чисто механически при помощи оператора **Y**.

Упражнение 2.18. Дайте рекурсивное определение операции разности натуральных чисел.

Упражнение 2.19. Постройте лямбда-выражение для предиката равенства **eq**, обладающего следующим свойством:

$$\mathbf{eq}\ m\ n\ =\ \mathbf{ifthenelse}\ (\mathbf{iszero}\ m)\ (\mathbf{iszero}\ n)$$
$$(\mathbf{ifthenelse}\ (\mathbf{iszero}\ n)\ \mathbf{false}\ (\mathbf{eq}\ (\mathbf{pre}\ m)\ (\mathbf{pre}\ n)))\,.$$

Упражнение 2.20. Докажите, что следующее выражение (комбинатор Клопа) действительно является комбинатором неподвижной точки:

$$\pounds\,,$$

где $\pounds$ обозначает выражение

$$\lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)\,.$$

2.6. Именованные выражения

Возможность определять безымянные функции является преимуществом лямбда-исчисления. Выше нами было показано, что рекурсивные функции можно определить, не вводя никаких имен. Тем не менее возможность давать имена выражениям полезна, поскольку позволяет избегать переписывания именованных выражений. Простая форма именования может быть введена как форма "синтаксического сахара" над чистым лямбда-исчислением:

$$\mathbf{let}\ x = S\ \mathbf{in}\ T\ =\ (\lambda x.\ T)\ S\,.$$

Например, как и следовало ожидать:

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

Можно связать несколько выражений с переменными в последовательном или параллельном стиле. В первом случае let-конструкции вкладываются друг в друга. Во втором используется запись:

$$\mathbf{let}\ \{x_1 = S_1, x_2 = S_2, \ldots, x_n = S_n\}\ \mathbf{in}\ T\,.$$

Это выражение можно рассматривать как "синтаксический сахар" для выражения

$$(\lambda(x_1, \ldots, x_n).\ t)\ (s_1, \ldots, s_n)\,.$$

Вместо префиксной формы с **let** можно ввести постфиксный вариант:

$$T\ \mathbf{where}\ x\ =\ S\,.$$

С помощью let-нотации можно определять функции, введя соглашение, что

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

означает

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

Для определения рекурсивных функций можно ввести соглашение по использованию комбинатора неподвижной точки, т. е. $\mathbf{let} f\ =\ F\ f\ \mathbf{in}\ S$ означает $\mathbf{let} f\ =\ Y\ F\ \mathbf{in}\ S$.

Рассмотренная система "синтаксического сахара" позволяет поддерживать понимаемый человеком синтаксис для чистого лямбда-исчисления, и с ее помощью можно писать программы на языке, близком к настоящему языку программирования.

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.

3. Типизированное лямбда-исчисление

Причина введения типов в лямбда-исчисление и в языки программирования возникает с точки зрения как логики, так и программирования. Лямбда-исчисление разрабатывалось для формализации языка математики. Чёрч предполагал включить в лямбда-исчисление теорию множеств. По заданному множеству S можно определить его характеристическую функцию χ_S, такую что:

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

С другой стороны, имея унарный предикат P, можно определить множество таких x, что $P(x) = true$. Однако определение предикатов (и, следовательно, множеств) в виде произвольных лямбда-выражений может привести к противоречиям.

Рассмотрим парадокс Рассела. Определим множество R, состоящее из всех множеств, которые не содержат себя в качестве элемента:

$$R = \{x \mid x \notin x\} \,.$$

Тогда $R \in R \Leftrightarrow R \notin R$, что является противоречием. В терминах лямбда-исчисления можно определить предикат $R = \lambda x.\neg(x\ x)$ и получить противоречие $R\ R = \neg(R\ R)$. Выражение $R\ R$ является неподвижной точкой оператора отрицания.

Парадокс Рассела в лямбда-исчислении возникает из-за того, что мы применяем функцию к самой себе. Однако это необязательно приводит к парадоксу: например, функция идентичности $\lambda x.x$ или константная функция $\lambda x.y$ не приводят к противоречиям. Более четкое представление функций, обозначаемых лямбда-термами, требует точного знания области их определения и значений и применения только к аргументам, принадлежащим областям их определения. По этим причинам Рассел предложил ввести понятие типа.

Типы возникли также и в языках программирования. Одной из причин этого была эффективность: зная о типе переменной, можно генерировать более эффективный код и более эффективно использовать память.

Помимо эффективности, типы предоставляют возможность статической проверки программ. Типы можно использовать также для достижения модульности и скрытия данных.

Существуют также бестиповые языки, как императивные, так и функциональные. Некоторые языки являются слабо типизированными, когда компилятор допускает некоторое несоответствие в типах и сам делает необходимые преобразования. Существуют языки (например, Python), которые выполняют проверку типов динамически, т. е. во время выполнения программы, а не во время компиляции.

Модифицируем лямбда-исчисление, введя в него понятие типа. Каждый лямбда-терм должен иметь тип, причем терм S можно применить к терму T в комбинации $S\ T$, только если их типы правильно соотносятся друг с другом, т. е. S имеет тип функции $\sigma \to \tau$ и T имеет тип σ. В результате выражение $S\ T$ имеет тип τ. Это свойство называется *сильной типизацией*. Приведение типов не допускается.

Мы будем использовать запись вида $T :: \sigma$ для утверждения "T имеет тип τ".

3.1. Базовые типы

Предположим, что 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 типов Ty_C, основывающихся на множестве базовых типов C:

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

$$\frac{\sigma \in C, \tau \in C}{(\sigma \to \tau) \in Ty_C}$$

Предполагается, что функциональная стрелка $\to$ ассоциативна вправо, т. е. $\sigma \to \tau \to \nu$ означает $\sigma \to (\tau \to \nu)$.

Можно расширить систему типов двумя способами. Во-первых, можно ввести понятие переменных типа, являющихся средством для реализации полиморфизма. Во-вторых, можно ввести дополнительные конструкторы типов, помимо функциональной стрелки, например конструктор $\times$ для типа пары значений. В этом случае необходимо добавить правило:

$$\frac{\sigma \in C, \tau \in C}{(\sigma \times \tau) \in Ty_C}$$

Можно ввести именованные конструкторы произвольной арности. Будем использовать запись $Con(a_1, \ldots, a_n)$ для применения n-арного конструктора Con к аргументам a_i.

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

3.2. Типизация по Чёрчу и по Карри

Существуют два основных подхода к определению типизированного лямбда-исчисления. Первый подход, принадлежащий Чёрчу, это *явная типизация*. Каждому терму сопоставляется единственный тип. Это означает, что в процессе конструирования тер-

мов нетипизированные термы модифицируются с помощью дополнительной характеристики — типа. Для констант этот тип задан заранее, но переменные могут иметь любой тип. Правила корректного формирования термов выглядят 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 тип, 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 SML или Haskell понятию полиморфизма, а во-вторых, позволяет не задавать типы явным образом.

Определим понятие типизируемости по отношению к *контексту*, т. е. конечному набору предположений о типах переменных. Будем записывать:

$$\Gamma \vdash T :: \sigma$$

чтобы обозначить, что "в контексте Γ терм T может иметь тип σ". Будем употреблять запись $\vdash T :: \sigma$ или просто $T :: \sigma$, если суждение о типизации выполняется в пустом контексте. Элементы контекста Γ имеют вид $v :: \sigma$, т. е. они представляют собой

предположения о типах переменных. Будем предполагать, что в Γ нет противоречивых предположений.

Индуктивным определением отношения типизируемости являются следующие формальные правила:

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

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

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

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

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:

$$\varnothing \ \vdash \ \lambda x.x :: \sigma \to \sigma\,.$$

В силу соглашения о пустых контекстах получаем $\lambda x.x :: \sigma \to \sigma$. Можно показать, что все преобразования лямбда-термов сохраняют типы.

3.3. Полиморфизм

Система типов по Карри обеспечивает *полиморфизм*, в том смысле, что терм может иметь различные типы. Необходимо различать концепции полиморфизма и *перегрузки*. Оба этих термина означают, что выражение может иметь несколько типов. Однако

в случае полиморфизма все типы сохраняют струкурное сходство. Например, функция идентичности может иметь тип $\sigma \to \sigma$ или $\tau \to \tau$ или $(\sigma \to \tau) \to (\sigma \to \tau)$. При перегрузке функция может иметь различные типы, не связанные друг с другом структурным сходством.

let-полиморфизм. Рассмотренная система типов приводит к некоторым ограничениям на полиморфизм. Например, приемлемо следующее выражение:

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

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

Рассмотрим выражение:

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

Согласно определению, это иной способ записи для

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

Однако это выражение не может быть типизировано. В нем присутствует единственный экземпляр функции идентичности, и он должен иметь единственный тип.

Для преодоления этого ограничения добавим правило типизации, в котором let-конструкция рассматривается как первичная:

$$\frac{\Gamma \vdash S :: \sigma,\ \Gamma \vdash T[x := S] :: \tau}{\Gamma \vdash \mathbf{let}\ x = S\ \mathbf{in}\ T :: \tau}$$

Это правило определяет так называемый *let-полиморфизм*.

Наиболее общие типы. Некоторые выражения не имеют типа, например $\lambda f.\ f\ f$ или $\lambda f.\ (f\ \mathbf{true}, f\ \mathbf{1})$. Такая ситуация достаточно распространена на практике, поскольку многие алгоритмы могут работать с данными любых типов (например, алгоритм определения длины списка не зависит от типа элементов этого списка).

Имеет место 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.

Введем понятие *переменных типа*. Типы могут быть сконструированы с помощью применения конструкторов типа либо к константам типа, либо к переменным типа. Будем использовать буквы α и β для обозначения переменных типа, а буквы σ и τ — для обозначения произвольных типов. Определим понятие замены переменной типа на некоторый тип. Будем использовать ту же нотацию, что и при подстановке термов. Например:

$$(\sigma \to \mathit{bool})[\sigma := (\sigma \to \tau)] \ = \ (\sigma \to \tau) \to \mathit{bool}\,.$$

Расширим это определение, добавив параллельные подстановки:

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

Можно рассматривать типовые константы как 0-арные конструкторы, т. е. считать, что int задается как int (). Имея определение подстановки, можно считать, 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:

$$
\begin{aligned}
\alpha &\preceq \alpha\,, \\
\alpha \to \alpha &\preceq \beta \to \beta\,, \\
\alpha \to \mathit{bool} &\preceq (\beta \to \beta) \to \mathit{bool}\,, \\
\beta \to \alpha &\preceq \alpha \to \beta\,, \\
\alpha \to \alpha &\not\preceq (\beta \to \beta) \to \beta\,.
\end{aligned}
$$

Имеет место

Теорема 3.1. *Каждый типизируемый терм имеет некоторый основной тип, т. е. если $T :: \tau$, то существует некоторый σ, такой что $T :: \sigma$ и для любого σ' из $T :: \sigma'$ следует $\sigma \preceq \sigma'$.*

Доказательство этой теоремы конструктивно: оно дает конкретную процедуру для поиска основного типа. Эта процедура известна как *алгоритм Хиндли-Милнера*. Все реализации языков программирования типа ML включают в себя вариант этого алгоритма. Выражения в них могут быть сопоставлены их основному типу либо отвергнуты как нетипизируемые.

3.4. Сильная нормализация

Теорема 3.2. *Каждый типизируемый терм имеет нормальную форму и каждая возможная последовательность редукций, начинающаяся с типизируемого терма, завершается.*

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 создавать незавершающиеся программы существенна для полноты по Тьюрингу для определения произвольных вычислимых и полных функций.

Чтобы обеспечить полноту по Тьюрингу, добавим способ определения произвольных рекурсивных функций, которые по опре-

делению являются правильно типизированными. Для этого определим полиморфный оператор рекурсии для всех типов вида:

$$Rec \ : \ ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau$$

Кроме того, добавим правило редукции:

$$Rec\ F \ \to \ F\ (Rec\ F)$$

для любого $F \ : \ (\sigma \to \tau) \to (\sigma \to \tau)$. Будем считать, что рекурсивные определения функций интерпретируются с использованием этих операторов рекурсии.

Заключение

Функциональный подход к программированию, реализующий модель лямбда-исчисления, имеет ряд преимуществ перед более традиционным императивным подходом. Функциональные программы более точно соответствуют фундаментальным математическим объектам и, следовательно, позволяют проводить более строгие рассуждения. Установить значение императивной программы, т. е. функции, вычисление которой она реализует, в общем случае довольно трудно. Значение функциональной программы может быть выведено практически непосредственно.

Функции в функциональных программах являются функциями в математическом смысле. Из этого следует, что вычисление любого выражения не может иметь никаких побочных эффектов и порядок вычисления его подвыражений не оказывает влияния на результат. Функциональные программы легко поддаются распараллеливанию, поскольку отдельные компоненты выражений могут вычисляться одновременно.

Основной принцип функционального программирования — это сам принцип функциональности: при вызове одной и той же функции с одними и теми же значениями аргументов всегда возвращается один и тот же результат. В "обычных" языках программирования этот принцип нарушен, так как там функция может обращаться к глобальным переменным или общей памяти.

Второй важнейший принцип — отсутствие искусственного управления процессом вычислений. Конечно, если аргументом функции является другая функция, то она будет вычислена раньше, чем вызвавшая её функция. Однако других способов задания последовательности действий (например, циклов) в чистой функциональной программе быть не может.

Следование этим принципам приводит к довольно интересным результатам. Например, в чистых функциональных языках нет понятия переменной, следовательно, отсутствует и оператор при-

сваивания. Нет операторных скобок, операторов цикла, механизмов работы с памятью. Синтаксис очень беден, однако и программа получается, как правило, более красивая и компактная.

К плюсам функционального подхода можно отнести:

- близость кода программы её спецификации (по сути дела программа представляет собой уточнённую спецификацию);
- отсутствие побочных эффектов от использования оператора присваивания и работы с памятью;
- отсутствие привязки к архитектуре (программа максимально абстрактна, поэтому её можно откомпилировать для любой конкретной ЭВМ);
- возможность так называемых “ленивых” (отложенных) вычислений, при которых непосредственно в ходе выполнения программы отсекаются избыточные вызовы подпрограмм;
- простота работы с динамическими рекурсивными структурами данных (списками, деревьями);
- реализация так называемого параметрического (“истинного”) полиморфизма, при котором одна и та же функция может работать с аргументами разных типов;
- работа с потенциально бесконечными типами данных;
- удобство анализа программы формальными математическими методами (например, при её верификации).

Литература

[1] Барендрегт, Х. Ламбда-исчисление. Его синтаксис и семантика / Х. Барендрегт. — М.: Мир, 1985. — 680 с.

[2] Зыков, С. В. Современные языки программирования. Ч. I : Функциональный подход к программированию. / С. В. Зыков. — М. : МИФИ, 2003. — 230 с.

[3] Непейвода, Н. Н. Стили и методы программирования / Н. Н. Непейвода. — М. : ИнтУИТ, 2012. — 320 с.

[4] Отт, А. Обзор литературы о функциональном программировании / А. Отт. — Практика функционального программирования, вып. 1, 2009. — С. 93–114.

[5] Рогозин, О. В. Функциональное и рекурсивно-логическое программирование / О. В. Рогозин. — М. : Евразийский открытый институт, 2009. — 139 с.

[6] Филд, А. Функциональное программирование / А. Филд, П. Харрисон. — М. : Мир, 1993. — 637 с.

[7] Gordon, M. Introduction to Functional Programming / M. Gordon. — Cambridge University Web Archive, 1996. URL: `https://www.cl.cam.ac.uk/archive/mjcg/Teaching/FuncProg/FuncProg.html`

[8] Harrison, J. Introduction to Functional Programming / J. Harrison. — Cambridge University Web Archive, 1998. URL: `https://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/`

[9] Paulson, L. C. Foundations of Functional Programming / L. C. Paulson. — Cambridge University Web Archive, 2000. URL: `https://www.cl.cam.ac.uk/~lp15/papers/Notes/Founds-FP.pdf`

Учебное издание

Башкин Владимир Анатольевич

Лямбда-исчисление

Учебно-методическое пособие

Редактор, корректор Л. Н. Селиванова
Компьютерный набор, вёрстка В. А. Башкин

Подписано в печать 15.11.2018. Формат 60×84/16.
Усл. печ. л. 2,9. Уч.-изд. л. 2,0.
Тираж 3 экз. Заказ .

Оригинал-макет подготовлен в редакционно-издательском отделе ЯрГУ

Ярославский государственный университет им. П. Г. Демидова.

150003, Ярославль, ул. Советская, 14.