Функциональное и логическое программирование

Лекция 3. Лямбда исчисление как язык программирования

Учебные вопросы

3.1 Представление данных в лямбда-исчислении

3.2 Рекурсивные функции

3.3 Let-выражения

3.4 Достижение уровня полноценного языка программирования

Представление данных в лямбда-исчислении

Программы для своей работы требуют входных данных, поэтому мы начнём с фиксации определённого способа представления данных в виде выражений лямбда-исчисления. Далее введём некоторые базовые операции над этим представлением. Во многих случаях оказывается, что выражение s, представленное в форме, удобной для восприятия человеком, может напрямую отображаться в лямбда-выражение s'. Этот процесс получил жаргонное название «синтаксическая глазировка» («syntactic sugaring»), поскольку делает горькую пилюлю чистой лямбда-нотации более удобоваримой. Введём следующее обозначение:

$$s \triangleq s'.$$

Представление данных в лямбда-исчислении

Будем говорить, что «$s = s'$ по определению»; другая общепринятая форма записи этого отношения — $s =_{def} s'$. При желании, мы можем всегда считать, что вводим некоторое константное выражение, определяющее семантику операции, которая затем применяется к своим аргументам в обычном стиле лямбда-исчисления, абстрагируясь тем самым от конкретных обозначений. Например, выражение if E then E_1else E_2 возможно трактовать как COND E E_1 E_2, где COND — некоторая константа. В подобном случае все переменные в левой части определения должны быть связаны операцией абстракции, т. е. вместо

$$\mathrm{fst}\ p \triangleq p\ \mathrm{true}$$

(см. ниже) мы можем написать

$$\mathrm{fst} \triangleq \lambda p.\ p\ \mathrm{true}.$$

Представление данных в лямбда-исчислении

Для представления логических значений true («истина») и false («ложь») годятся любые два различных лямбда-выражения, но наиболее удобно использовать следующие:

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

Представление данных в лямбда-исчислении

Используя эти определения, легко ввести понятие условного выражения, соответствующего конструкции `? :` языка C. Отметим, что это условное *выражение*, а не *оператор* (который не имеет смысла в данном контексте), поэтому наличие альтернативы обязательно.

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

В самом деле, мы имеем:

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

и

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

Представление данных в лямбда-исчислении

Определив условное выражение, на его базе легко построить весь традиционный набор логических операций:

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

Представление данных в лямбда-исчислении

Определим представление упорядоченных пар следующим образом:

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

Использование скобок не обязательно, хотя мы часто будем использовать их для удобства восприятия либо подчёркивания ассоциативности. На самом деле, мы можем трактовать запятую как инфиксную операцию наподобие $+$. Определив пару, как указано выше, зададим соответствующие операции извлечения компонент пары как:

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

Представление данных в лямбда-исчислении

Легко убедиться, что эти определения работают, как требуется:

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

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

Представление данных в лямбда-исчислении

Построение троек, четвёрок, пятёрок и так далее вплоть до кортежей произвольной длины n производится композицией пар:

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

Всё, что нам при этом потребуется — определение, что инфиксный оператор запятая правоассоциативен. Дальнейшее понятно без введения дополнительных соглашений. Например:

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

Представление данных в лямбда-исчислении

$$
\begin{aligned}
\text{CURRY}\ f &\triangleq \lambda x\ y.\ f(x, y)\\
\text{UNCURRY}\ g &\triangleq \lambda p.\ g\ (\text{fst}\ p)\ (\text{snd}\ p)
\end{aligned}
$$

Эти специальные операции над парами нетрудно обобщить на случай кортежей произвольной длины n. Например, мы можем задать функцию-селектор выборки i-го компонента из кортежа p. Обозначим эту операцию $(p)_i$, и определим её как $(p)_1 = \text{fst}\ p$, $(p)_i = \text{fst}\ (\text{snd}^{i-1}\ p)$. Аналогичным образом возможно обобщение CURRY и UNCURRY:

$$
\begin{aligned}
\text{CURRY}_n\ f &\triangleq \lambda x_1\ \cdots\ x_n.\ f(x_1, \ldots, x_n)\\
\text{UNCURRY}_n\ g &\triangleq \lambda p.\ g\ (p)_1\ \cdots\ (p)_n
\end{aligned}
$$

Представление данных в лямбда-исчислении

Представим натуральное число n в виде[3]

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

то есть, $0 = \lambda f\ x.\ x$, $1 = \lambda f\ x.\ f\ x$, $2 = \lambda f\ x.\ f\ (f\ x)$ и т. д. Такое представление получило название *нумералов Чёрча*, хотя его базовая идея была опубликована ранее Витгенштейном [65].[4] Это представление не слишком эффективно, так как фактически представляет собой запись чисел в системе счисления по основанию 1: 1, 11, 111, 1111, 11111, 111111, ... С точки зрения эффективности можно разработать гораздо лучшие формы представления, к примеру, кортеж логических значений, который интерпретируется как двоичная запись числа. Впрочем, в данный момент нас интересует лишь принципиальная вычислимость, а нумералы Чёрча имеют различные удобные формальные свойства. Например, легко привести лямбда-выражения такой общеизвестной арифметической операции, как получение числа, следующего в натуральном ряду за данным, то есть, прибавление единицы к аргументу операции:

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

Представление данных в лямбда-исчислении

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

Представление данных в лямбда-исчислении

$$\text{ISZERO } n \stackrel{\triangle}{=} n\ (\lambda x.\ \text{false})\ \text{true}$$

поскольку

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

и

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

Представление данных в лямбда-исчислении

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

Представление данных в лямбда-исчислении

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

Представление данных в лямбда-исчислении

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

Представление данных в лямбда-исчислении

Несмотря на то, что эти операции на натуральных числах были определены достаточно легко, вычисление числа, предшествующего данному, гораздо сложнее. Нам требуется выражение PRE такое, что $\mathrm{PRE}\ 0 = 0$ и $\mathrm{PRE}\ (n+1) = n$. Оригинальное решение этой задачи было предложено Клини [34]. Пусть для заданного $\lambda f\ x.\ f^n\ x$ требуется «отбросить» одно из применений f. В качестве первого шага введём на множестве пар функцию PREFN такую, что

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

и

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

Представление данных в лямбда-исчислении

Предположив, что подобная функция существует, можно показать, что $(\mathrm{PREFN}\ f)^{n+1}(\mathrm{true}, x) = (\mathrm{false}, f^n\ x)$. В свою очередь, этого достаточно, чтобы задать функцию PRE, не испытывая особых затруднений. Определение PREFN, удовлетворяющее нашим нуждам, таково:

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

В свою очередь,

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

Рекурсивные функции

Возможность определения рекурсивных функций является краеугольным камнем функционального программирования, поскольку в его рамках это единственный общий способ реализовать итерацию. На первый взгляд, сделать подобное средствами лямбда-исчисления невозможно. В самом деле, *именование* функций представляется непременной частью рекурсивных определений, так как в противном случае неясно, как можно сослаться на функцию в её собственном определении, не зацикливаясь. Тем не менее, существует решение и этой проблемы, которое, однако, удалось найти лишь ценой значительных усилий, подобно построению функции PRE.

Рекурсивные функции

Ключом к решению оказалось существование так называемых *комбинаторов неподвижной точки*. Замкнутый терм Y называется комбинатором неподвижной точки, если для произвольного терма f выполняется равенство $f(Y\ f) = Y\ f$. Другими словами, комбинатор неподвижной точки определяет по заданному терму f его фиксированную точку, т. е. находит такой терм x, что $f(x) = x$. Первый пример такого комбинатора, найденный Карри, принято обозначать Y. Своим появлением он обязан парадоксу Рассела, чем объясняется его другое популярное название — «парадоксальный комбинатор». Мы определили

$$R = \lambda x.\ \neg(x\ x),$$

после чего обнаружили справедливость

$$R\ R = \neg(R\ R)$$

Рекурсивные функции

Таким образом, $R\ R$ представляет собой неподвижную точку операции отрицания. Отсюда, чтобы построить универсальный комбинатор неподвижной точки, нам потребуется лишь обобщить данное выражение, заменив $\neg$ произвольной функцией, заданной аргументом f. В результате мы получаем

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

Убедиться в справедливости этого определения несложно:

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

Рекурсивные функции

Однако, несмотря на математическую корректность, предложенное решение не слишком привлекательно с точки зрения программирования, поскольку оно справедливо лишь в смысле лямбда-эквивалентности, но не редукции (в последнем выражении мы применяли *обратное* бета-преобразование). С учётом этих соображений альтернативное определение Тьюринга может оказаться более предпочтительным:

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

Рекурсивные функции

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

Прежде всего, преобразуем эту функцию в эквивалентную:

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

которая, в свою очередь, эквивалентна

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

Отсюда следует, что fact представляет собой неподвижную точку такой функции F:

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

Рекурсивные функции

В результате всё, что нам потребуется, это положить $\mathrm{fact} = Y\ F$. Аналогичным способом можно воспользоваться и в случае взаимно рекурсивных функций, т. е. множества функций, определения которых зависят друг от друга. Такие определения, как

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

могут быть при помощи кортежей преобразованы в одно:

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

Рекурсивные функции

Положив $t = (f_1, f_2, \ldots, f_n)$, видим, что каждая из функций в правой части равенства может быть вычислена по заданному t применением соответствующей функции-селектора: $f_i = (t)_i$. Применив абстракцию по переменной t, получаем уравнение в канонической форме $t = F\ t$, решением которого является $t = Y\ F$, откуда в свою очередь находятся значения отдельных функций.

Let-выражения

Возможность использования безымянных функций была нами ранее преподнесена как одно из достоинств лямбда-исчисления. Более того, имена оказались необязательными даже при определении рекурсивных функций. Однако, зачастую всё же удобно иметь возможность давать выражениям имена с тем, чтобы избежать утомительного повторения больших термов. Простая форма такого именования может быть реализована как ещё один вид синтаксической глазури поверх чистого лямбда-исчисления:

$$\text{let } x = s \text{ in } t \triangleq (\lambda x.\, t)\ s$$

Let-выражения

Простой пример применения этой конструкции работает, как и ожидается:

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

Мы можем добиться как последовательного, так и параллельного связывания множества имён с выражениями. Первый случай реализуется простым многократным применением конструкции связывания, приведённой выше. Во втором случае введём возможность одновременного задания множества связываний, отделяемых друг от друга служебным словом `and`:

$$\text{let } x_1 = s_1 \text{ and } \cdots \text{ and } x_n = s_n \text{ in } t$$

Будем рассматривать эту конструкцию как синтаксическую глазурь для

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

Let-выражения

Продемонстрируем различия в семантике последовательного и параллельного связывания на примере:

$$\text{let } x = 1 \text{ in let } x = 2 \text{ in let } y = x \text{ in } x + y$$

и

$$\text{let } x = 1 \text{ in let } x = 2 \text{ and } y = x \text{ in } x + y$$

дают в результате 4 и 3 соответственно.

В дополнение к этому разрешим связывать выражения с именами, за которыми следует список параметров; такая форма конструкции `let` представляет собой ещё одну разновидность синтаксической глазури, позволяющую трактовать $f\ x_1\ \cdots\ x_n = t$ как $f = \lambda x_1\ \cdots\ x_n.\ t$. Наконец, помимо префиксной формы связывания $\text{let } x = s \text{ in } t$ введём постфиксную, которая в некоторых случаях оказывается удобнее для восприятия:

$$t \text{ where } x = s$$

Let-выражения

Например, мы можем написать так: $y < y^2$ where $y = 1 + x$.

Обычно конструкции `let` и `where` интерпретируются, как показано выше, без привлечения рекурсии. Например,

$$\text{let } x = x - 1 \text{ in } \cdots$$

связывает x с уменьшенным на единицу значением, которое уже было связано с именем x в охватывающем контексте, а не пытается найти неподвижную точку выражения $x = x - 1$.[5] В случае, когда нам требуется рекурсивная интерпретация, это может быть указано добавлением служебного слова `rec` в конструкции связывания (т. е. использованием `let rec` и `where rec` соответственно). Например,

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

Это выражение может считаться сокращённой формой записи let fact $= Y\ F$, где

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

как было показано выше.

Let-выражения

На данный момент мы ввели достаточно обширный набор средств «синтаксической глазировки», реализующих удобочитаемый синтаксис поверх чистого лямбда-исчисления. Примечательно, что этих средств достаточно для определения функции факториала в форме, очень близкой к языку ML. В связи с этим возникает вопрос, уместно ли считать лямбда-исчисление, расширенное предложенными обозначениями, практически пригодным языком программирования?

В конечном счёте, программа представляет собой единственное выражение. Однако, использование `let` для именования различных важных подвыражений, делает вполне естественной трактовку программы как множества *определений* различных вспомогательных функций, за которыми следует итоговое выражение, например:

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