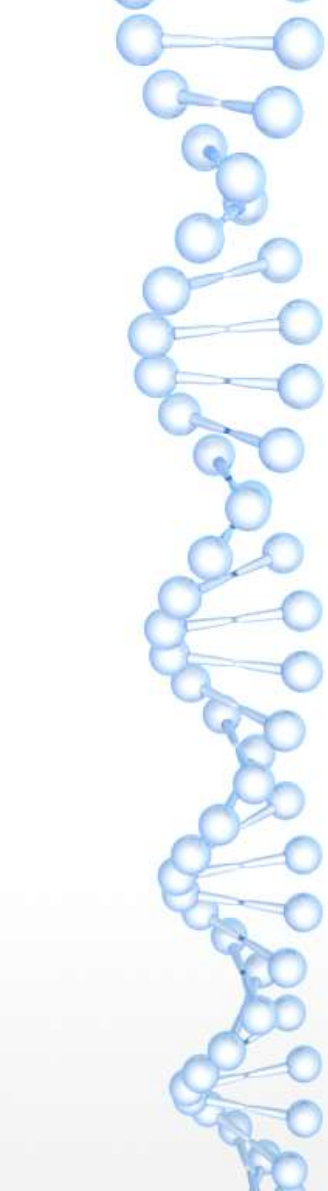
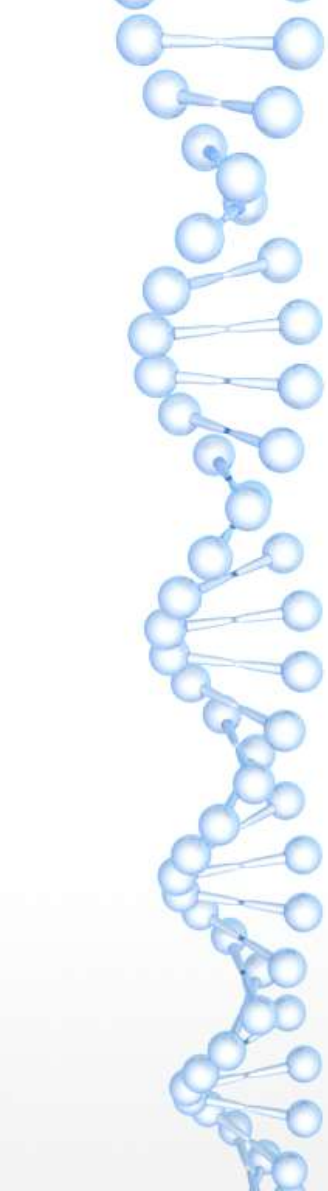


Функциональное и логическое программирование

Лекция 7. Лямбда исчисление как язык программирования



Учебные вопросы

1 Представление данных в лямбда-исчислении. Рекурсивные функции

2 Let-выражения. Достижение уровня полноценного языка программирования

3 Списки Чёрча

4 Типизированное лямбда исчисление

5 Полиморфизм

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:

$$s \stackrel{\triangle}{=} s'.$$

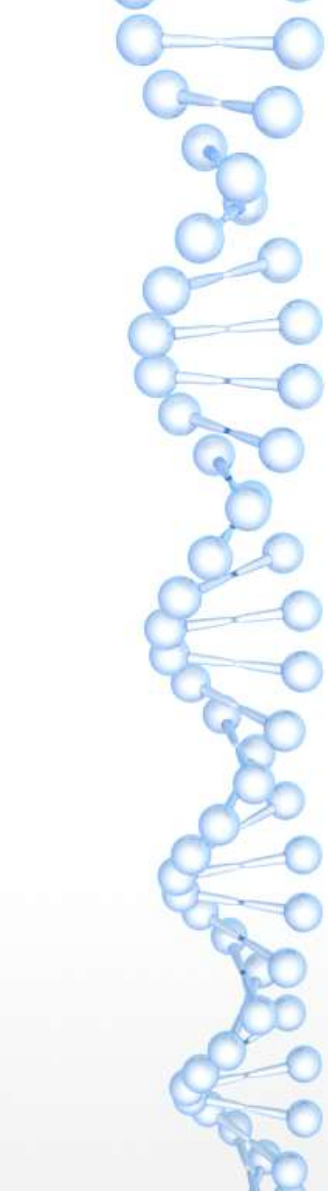
Представление данных в лямбда-исчислении

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

$$\text{fst } p \stackrel{\triangle}{=} p \text{ true}$$

(см. ниже) мы можем написать

$$\text{fst} \stackrel{\triangle}{=} \lambda p.\ p \text{ true}.$$



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:

$$\text{true} \stackrel{\triangle}{=} \lambda x\ y.\ x$$

$$\text{false} \stackrel{\triangle}{=} \lambda x\ y.\ y$$

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.

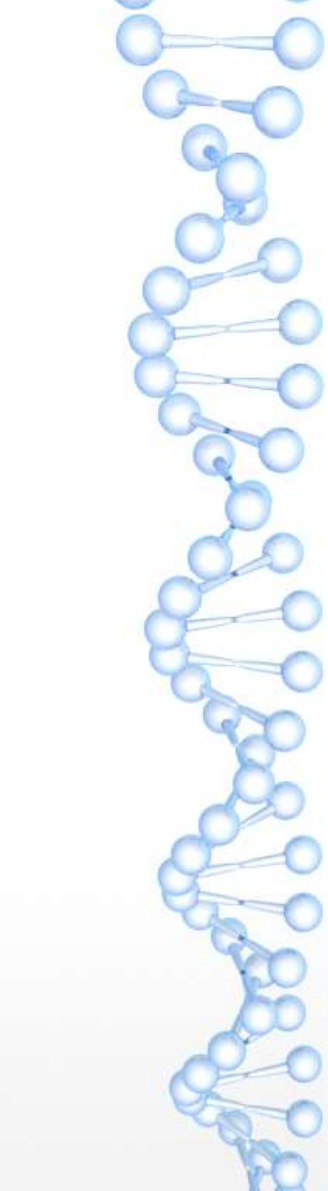
$$\text{if } E \text{ then } E_1 \text{ else } E_2 \stackrel{\triangle}{=} E\ E_1\ E_2$$

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

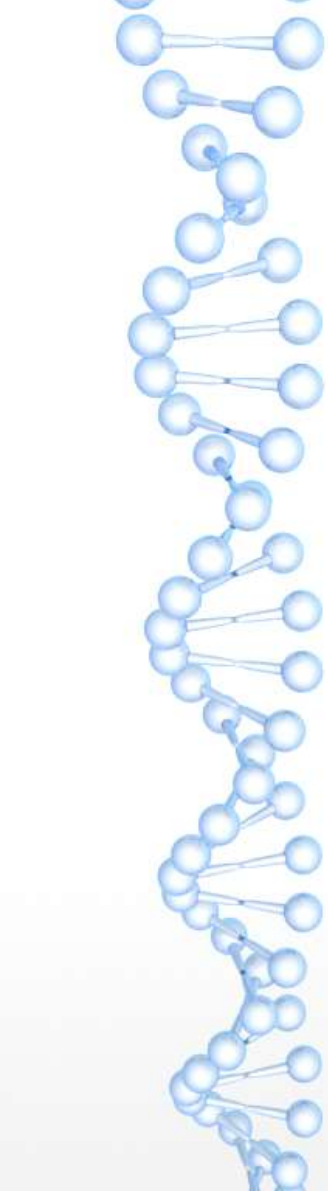
$$\begin{aligned} \text{if false then } E_1 \text{ else } E_2 &= \text{false } E_1\ E_2 \\ &= (\lambda x\ y.\ y)\ E_1\ E_2 \\ &= E_2 \end{aligned}$$



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:

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

Представление данных в лямбда-исчислении

Легко убедиться, что эти определения работают, как требуется:

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

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:

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

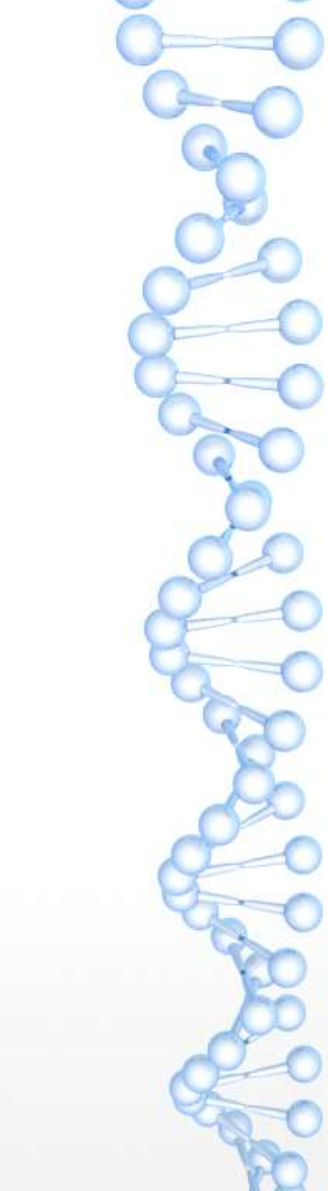
Представление данных в лямбда-исчислении

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

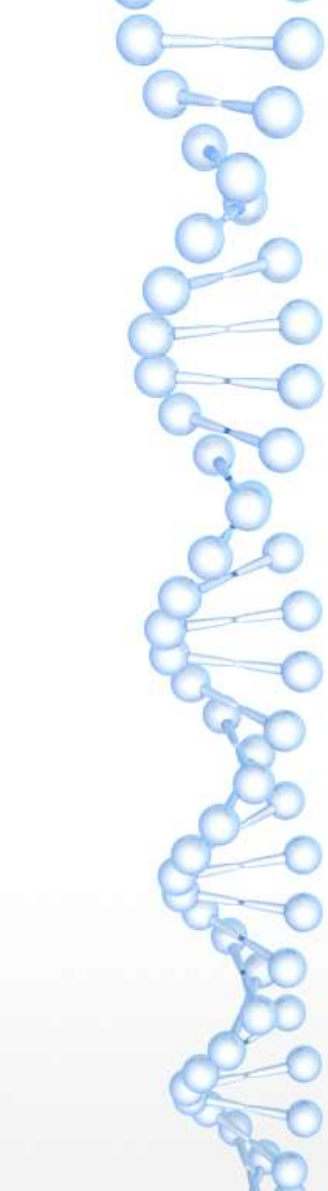
$$\text{ISZERO}\ n \stackrel{\triangle}{=} n\ (\lambda x.\ \text{false})\ \text{true}$$

поскольку

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

и

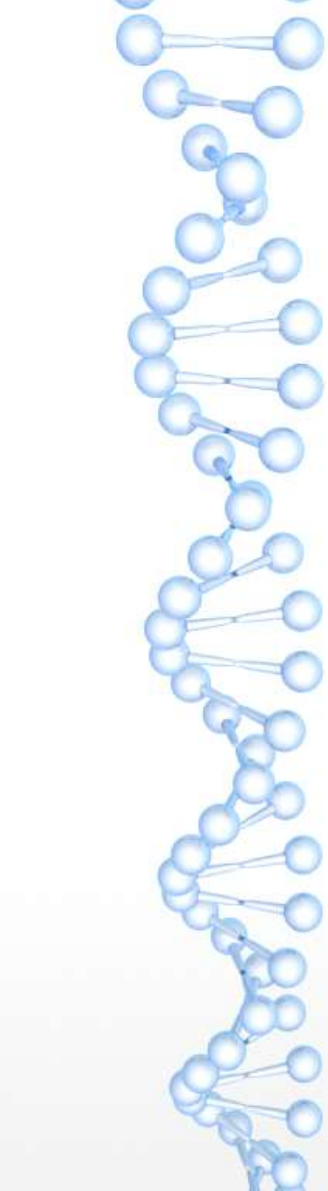
$$\begin{aligned}
\text{ISZERO}\ (n+1) &= (\lambda f\ x.\ f^{n+1}x)(\lambda x.\ \text{false})\text{true} \\
&= (\lambda x.\ \text{false})^{n+1}\ \text{true} \\
&= (\lambda x.\ \text{false})((\lambda x.\ \text{false})^{n}\ \text{true}) \\
&= \text{false}
\end{aligned}$$



Представление данных в лямбда-исчислении

$$m + n \triangleq \lambda f\ x.\ m\ f\ (n\ f\ x)$$

$$m * n \triangleq \lambda f\ x.\ m\ (n\ f)\ x$$



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 PRE $0 = 0$ и 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

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

и

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

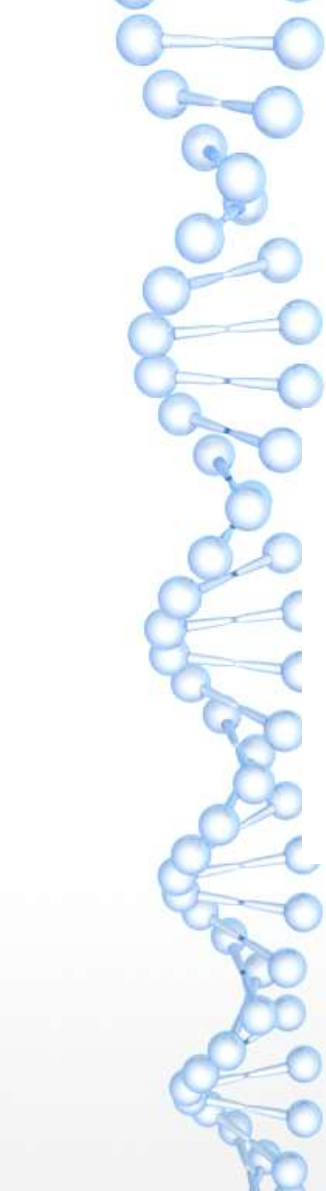
Представление данных в лямбда-исчислении

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

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

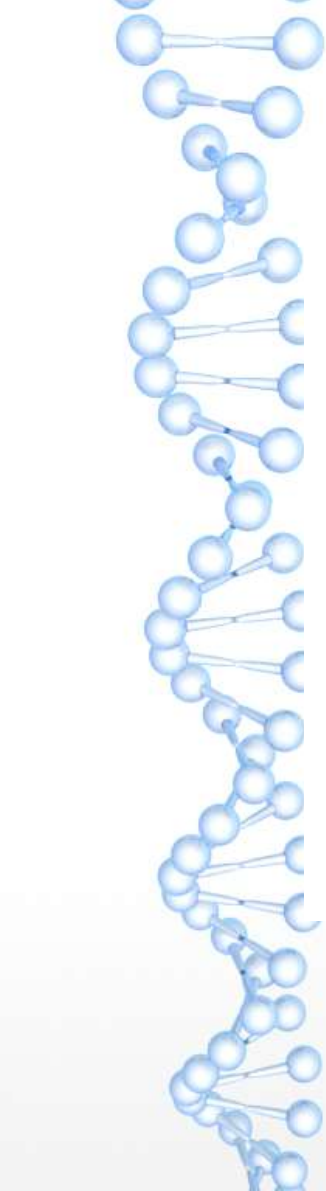
В свою очередь,

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



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.



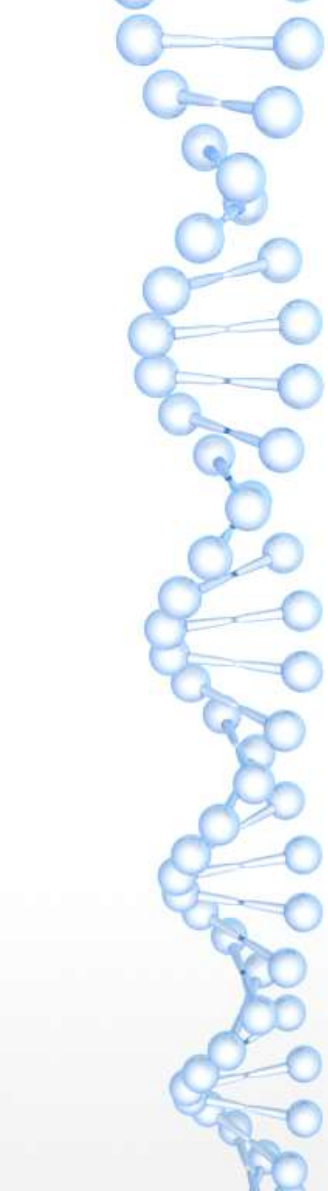
Рекурсивные функции

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

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



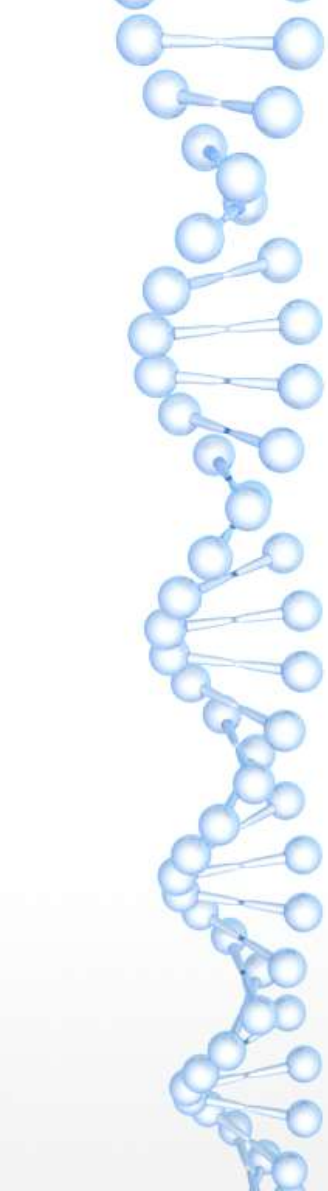
Рекурсивные функции

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

$$Y \stackrel{\triangle}{=} \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))$$

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

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

Прежде всего, преобразуем эту функцию в эквивалентную:

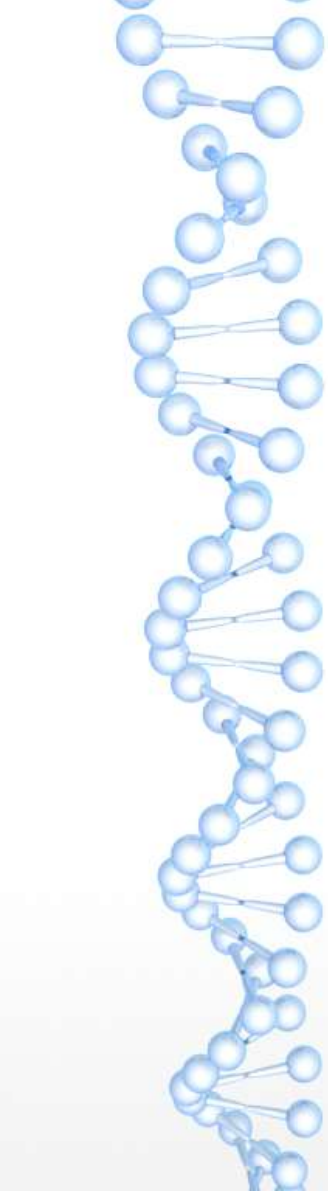
$$\text{fact} = \lambda n.\ \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

которая, в свою очередь, эквивалентна

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

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

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



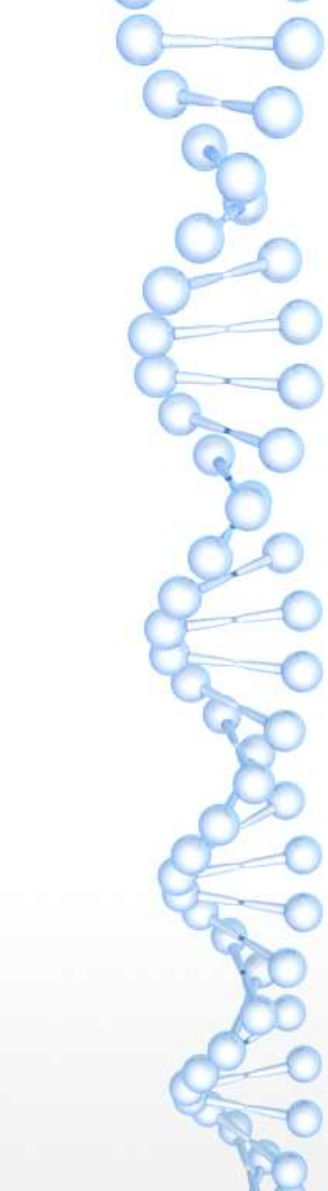
Рекурсивные функции

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

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

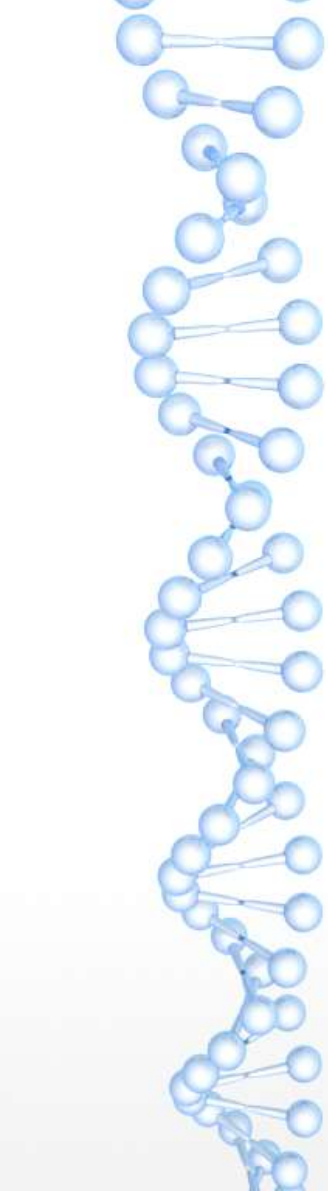
могут быть при помощи кортежей преобразованы в одно:

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



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.

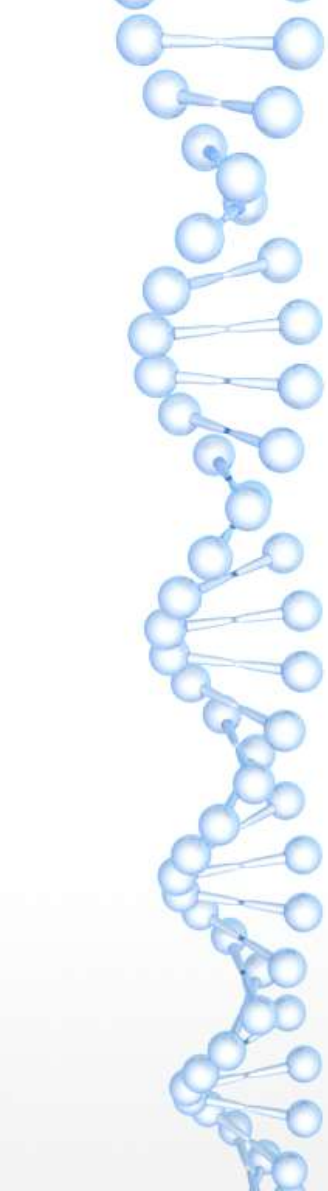


Let-выражения

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:

$$\text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n) \text{ in}$$
$$\ldots$$
$$fact(6)$$



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:

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

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,

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

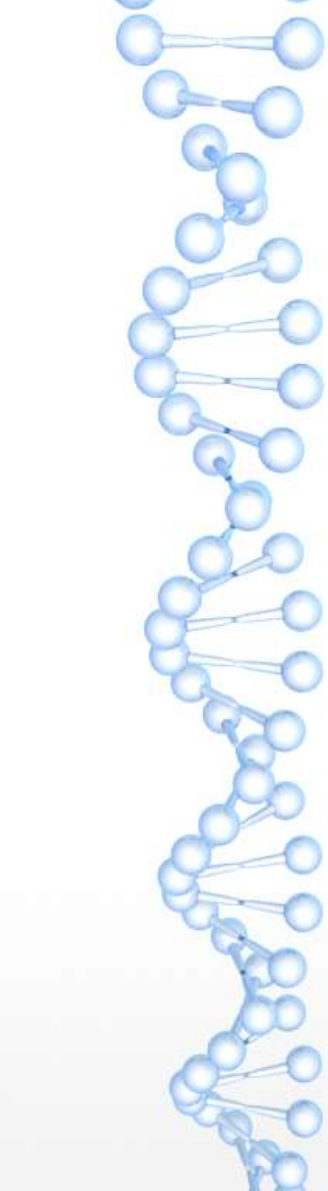
связывает x с уменьшенным на единицу значением, которое уже было связано с именем x в охватывающем контексте, а не пытается найти неподвижную точку выражения $x = x - 1$.[5] В случае, когда нам требуется рекурсивная интерпретация, это может быть указано добавлением служебного слова `rec` в конструкции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



Списки Чёрча

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.

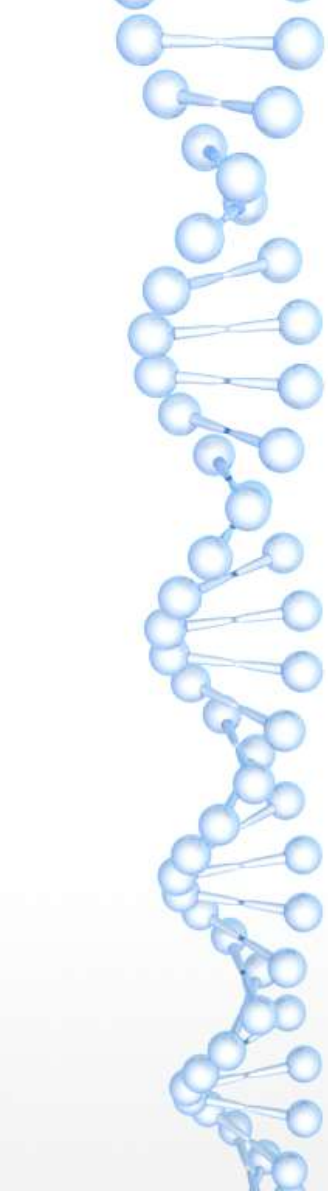
В качестве альтернативы представим списки через операцию образования пар. Этот способ кодирования легче понять, потому что он ближе к прикладным реализациям. Список $[x_1, x_2, \ldots, x_n]$ будет представлен как $x_1 :: x_2 :: \ldots :: x_n :: nil$. Чтобы операции были максимально простыми, мы будем 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.

Списки Чёрча

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

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

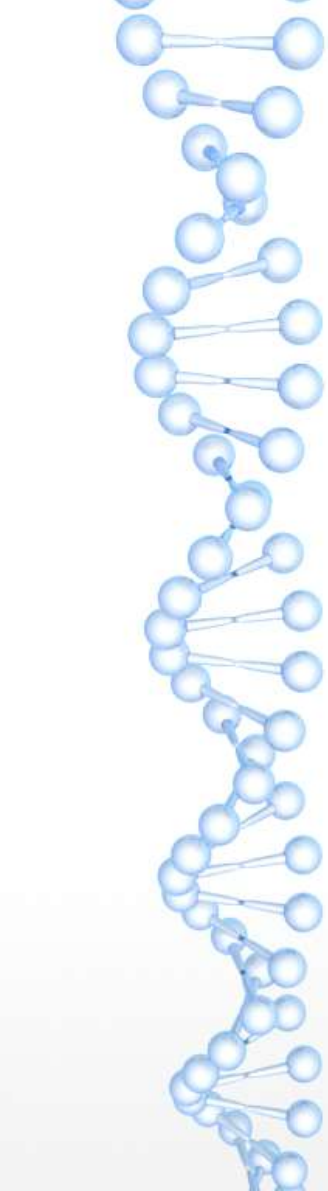
$$
\begin{aligned}
\mathbf{null}\ \mathbf{nil} &= \mathbf{true}\,, \\
\mathbf{null}\ (\mathbf{cons}\ M\ N) &= \mathbf{false}\,, \\
\mathbf{hd}\ (\mathbf{cons}\ M\ N) &= M\,, \\
\mathbf{tl}\ (\mathbf{cons}\ M\ N) &= N\,.
\end{aligned}
$$



Списки Чёрча

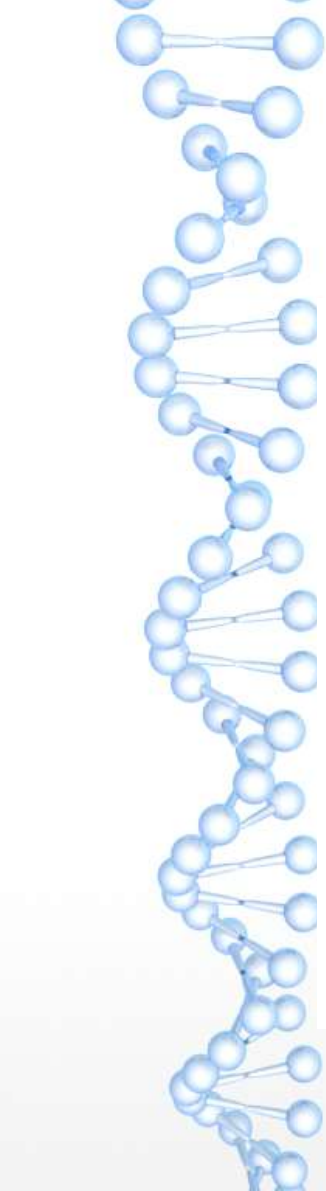
Обратите внимание, что **null nil** = **true** вышло действительно случайно, в то время как другие соотношения являются следствием использования наших операций над парами.

Также заметьте, что ключевые характеристические соотношения **hd** (**cons** M N) = M и **tl** (**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.



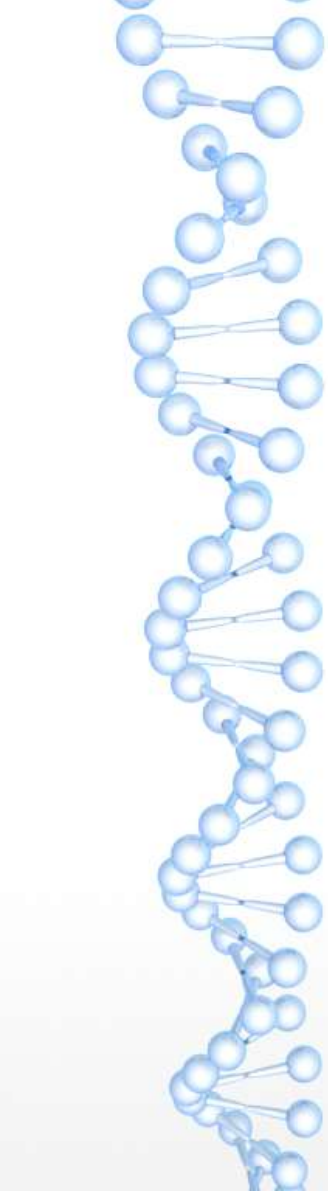
Списки Чёрча

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



Типизированное лямбда исчисление

В первом приближении расширение лямбда-исчисления понятием типа не представляет особого труда, но в итоге, как будет показано, потребуется куда больше усилий. Основная идея состоит в том, что каждому терму назначается *тип*, после чего выражение $s\ t$, т. е. применение терма s к терму t, допустимо исключительно для совместимых типов, то есть в случае, когда типы s и t имеют вид $\sigma \to \tau$ и σ соответственно. Результирующий терм будет иметь при этом тип τ. Такую типизацию принято называть *сильной*.[4] Терм t *обязан* иметь тип σ, подтипы и преобразования не допускаются. Такой подход составляет резкий контраст с некоторыми языками программирования, например, с языком C, в котором функция, ожидающая аргумент типа `float` либо `double`, принимает также значения типа `int`, выполняя автоматическое преобразование. Аналогичные понятия подтипов и преобразований возможно задать и в рамках лямбда-исчисления, но их освещение завело бы нас слишком далеко.



Типизированное лямбда исчисление

Введём для отношения «t имеет тип σ» обозначение $t : \sigma$. Подобная запись традиционно используется математиками при работе с функциональными пространствами, поскольку $f : \sigma \to \tau$ обозначает функцию f, отображающую множество σ во множество τ. Будем считать типы множествами, которые содержат соответствующие объекты, и трактовать $t : \sigma$ как $t \in \sigma$. Однако, несмотря на то, что мы предлагаем читателям также воспользоваться этой удобной аналогией, типизированное лямбда-исчисление будет в дальнейшем рассматриваться исключительно как формальная система, свободная от каких-либо интерпретаций.

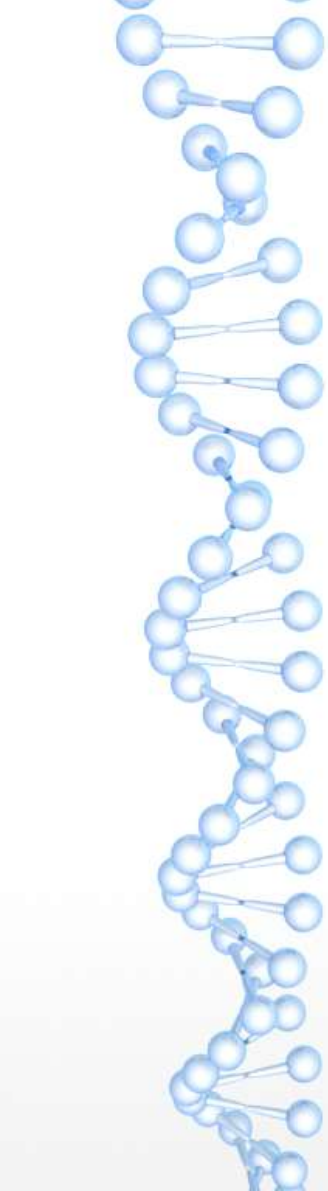
Типизированное лямбда исчисление

Начнём формализацию строгим определением понятия типа. Предположим, что у нас имеется некоторое множество *примитивных типов*, в которое входят, например, типы `bool` и `int`. Составные типы могут быть определены при помощи *конструктора типа функции*. Формально, индуктивное определение множества типов Ty_C, основанного на множестве примитивных типов C, выглядит так:

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

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

Например, в рамках данного определения допустимы типы int, $bool \to bool$ либо $(int \to bool) \to int \to bool$. Будем считать операцию «$\to$» правоассоциативной, т. е. полагать выражение $\sigma \to \tau \to \upsilon$ равным $\sigma \to (\tau \to \upsilon)$. Такая трактовка естественно согласуется с другими синтаксическими правилами, касающимися каррирования.



Типизированное лямбда исчисление

Следующим нашим шагом будет расширение системы типов в двух направлениях. Во-первых, введём наравне с примитивными типами (которые выполняют роль констант) так называемые *переменные типа,* которые впоследствии лягут в основу полиморфизма. Во-вторых, разрешим использование множества конструкторов других типов, помимо типа функции. Например, в дальнейшем нам понадобится конструктор $\times$ для типа декартова произведения. Как следствие, наше индуктивное определение должно быть дополнено ещё одним выражением:

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{}$$

Типизированное лямбда исчисление

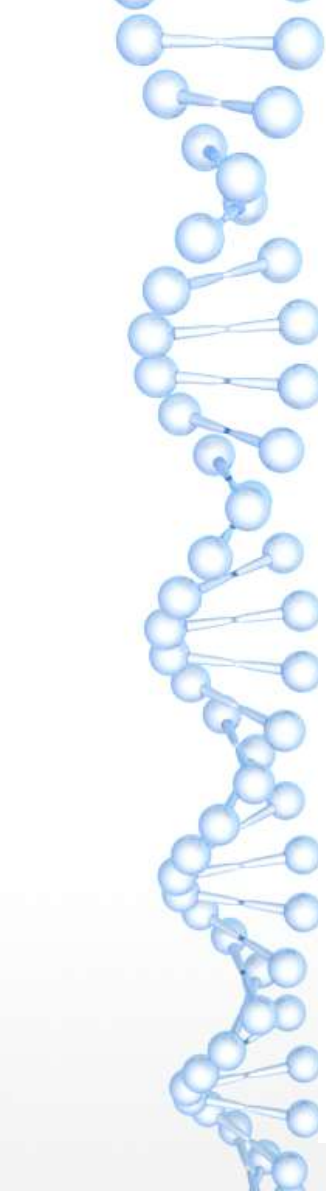
Известны два основных подхода к определению типизированного лямбда-исчисления. Один из них, разработанный Чёрчем, подразумевает *явное* указание типов. Каждому терму при этом назначается единственный тип. Другими словами, в ходе построения термов каждому нетипизированному терму, которые были рассмотрены ранее, в дополнение указывается тип. Типы констант являются предопределёнными, но типы переменных могут быть произвольными. Точные правила построения типизированных термов приведены ниже:

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

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

$$\frac{s : \sigma \to \tau \quad t : \sigma}{s\ t : \tau}$$

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



Типизированное лямбда исчисление

В то же время, некоторые формальные аспекты назначения типов по Карри оказываются достаточно сложными. Отношение типизируемости не может быть задано в отрыве от некоторого *контекста,* представляющего собой конечное множество утверждений относительно типов переменных. Обозначим через

$$\Gamma \vdash t : \sigma$$

утверждение «в контексте Γ терму t может быть назначен тип σ». (Если это утверждение справедливо при пустом контексте, выражение сокращается до $\vdash t : \sigma$ или даже до $t : \sigma$.) Элементы множества Γ имеют вид $v : \sigma$ т. е. сами по себе являются утверждениями относительно типов отдельных переменных, обычно тех, которые входят в терм t. Будем полагать, что контекст Γ не содержит противоречивых утверждений о типе некоторой переменной; при желании, мы можем рассуждать о нём как о частичной функции, отображающей индексное множество переменных во множество типов. Использование нами символа $\vdash$ соответствует его роли в традиционной логике, где $\Gamma \vdash \phi$ принято трактовать как «утверждение ϕ следует из множества посылок Γ».

Типизированное лямбда исчисление

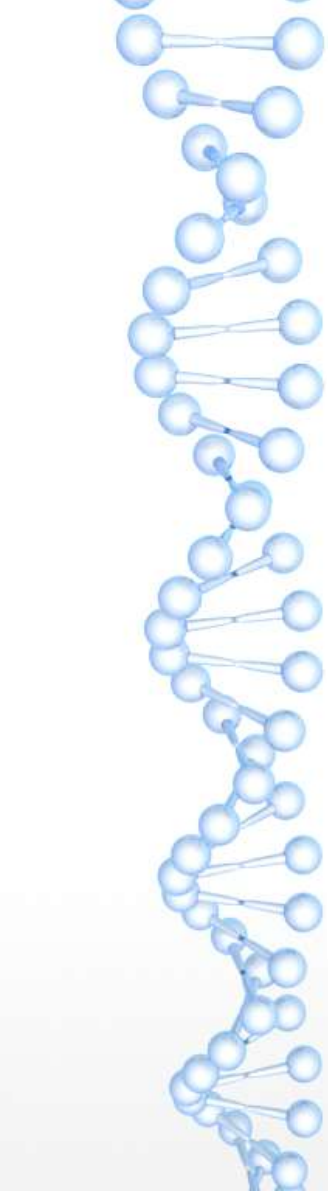
Формулировка правил назначения типов выражениям достаточно естественна. Прежде, чем мы приведём эти правила, напомним ещё раз, что $t : \sigma$ следует интерпретировать как «t *может* иметь тип σ».

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

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

$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash s\, t : \tau}$$

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



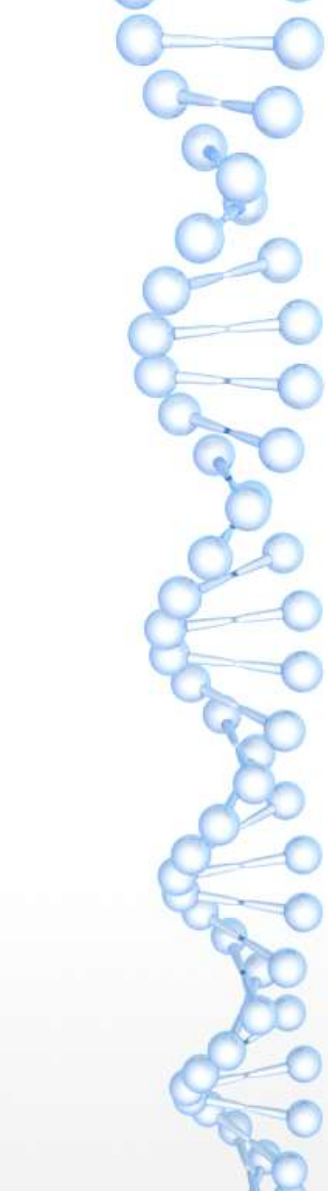
Типизированное лямбда исчисление

Ещё раз повторим, что эти выражения следует понимать как индуктивное определение отношения типизируемости, так что терм может иметь тип лишь тогда, когда последний выводим при помощи упомянутых выше правил. В качестве примера рассмотрим процедуру вывода типа тождественной функции. Согласно правилу типизации переменных, мы имеем:

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

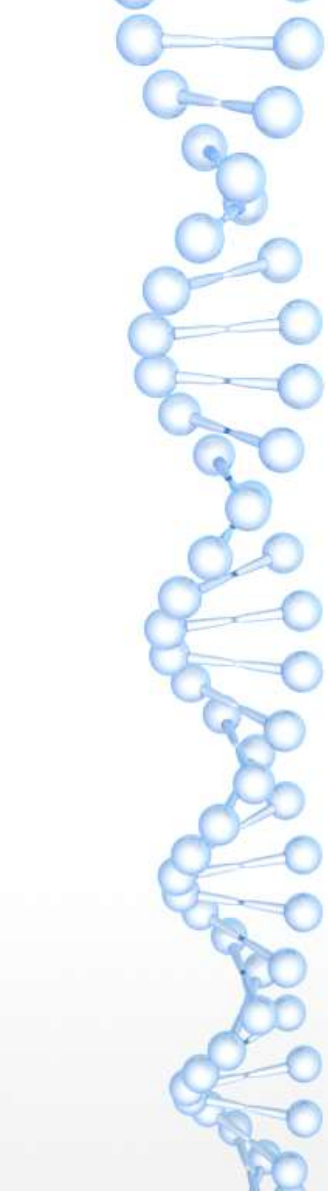
откуда, применив последнее правило, получаем:

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



Типизированное лямбда исчисление

Применив установленное ранее соглашение о пустых контекстах, мы можем сократить это выражение до $\lambda x.\, x : \sigma \to \sigma$. На данном примере также хорошо видна как важная роль контекстов в типизации по Карри, так и их необязательность в рамках типизации по Чёрчу. Опуская контекст, мы можем вывести $x : \tau$ для произвольного типа τ, после чего, согласно последнему правилу, получаем $\lambda x.\, x : \sigma \to \tau$ — налицо различие с интуитивной трактовкой тождественной функции! Эта проблема не возникает в ходе типизации по Чёрчу, поскольку в её рамках либо обе переменные имеют тип σ, откуда получаем $\lambda x.\, x : \sigma \to \sigma$, либо эти переменные на самом деле различны (поскольку различны их типы, которые считаются неотъемлемой частью терма). В последнем случае тип выражения в действительности будет равен $\lambda x : \sigma.\, (x : \tau) : \sigma \to \tau$, но это обосновано тем, что данное выражение альфа-эквивалентно $\lambda x : \sigma.\, (y : \tau) : \sigma \to \tau$. Поскольку в ходе типизации по Карри термы не содержат в себе типов явно, нам требуется некоторый механизм связывания между собой одинаковых переменных.

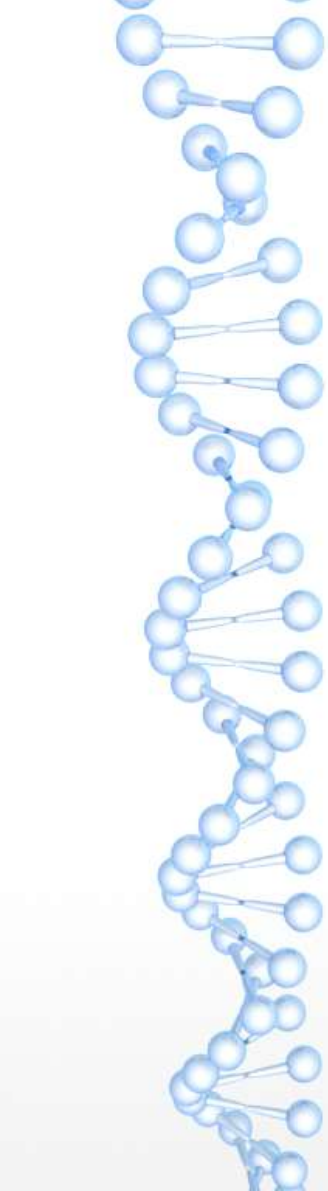


Типизированное лямбда исчисление

Очевидное сходство структуры термов типизированного и нетипизированного лямбда-исчисления порождает естественное желание применить в типизированном случае аппарат формальных преобразований, разработанный для нетипизированного. Однако, нам потребуется предварительно доказать, что тип выражения в ходе преобразований не изменяется (такое свойство называется *сохранением типа*). Убедиться в этом не представляет труда; мы рассмотрим краткое изложение доказательства для случая η-преобразования, предоставив остальные читателю в качестве упражнения. Прежде всего, докажем две леммы, которые весьма очевидны, но тем не менее, требуют формального обоснования. Во-первых, покажем, что добавление новых элементов в контекст не влияет на типизируемость:

Типизированное лямбда исчисление

Лемма 4.1 (О монотонности) *Если* $\Gamma \vdash t : \sigma$ *и* $\Gamma \subseteq \Delta$, *то справедливо* $\Delta \vdash t : \sigma$.
Доказательство: *Применим индукцию по структуре* t. *Зафиксировав* t, *докажем приведённое выше утверждение для всевозможных* Γ *и* Δ, *поскольку в ходе шага индукции для абстракций эти множества изменяются. Если* t — *переменная, справедливо* $t : \sigma \in \Gamma$, *из чего следует и* $t : \sigma \in \Delta$, *откуда получаем требуемое. Если* t — *константа, желаемый вывод очевиден, поскольку множество констант и их типов не зависит от контекста. В случае терма* t, *имеющего вид комбинации термов* s *и, для некоторого типа* τ *выполняется* $\Gamma \vdash s : \tau \to \sigma$ *и* $\Gamma \vdash u : \tau$. *Согласно индуктивному предположению,* $\Delta \vdash s : \tau \to \sigma$ *и* $\Delta \vdash u : \tau$, *откуда также получаем требуемое. Наконец, если терм* t *представляет собой абстракцию* $\lambda x.\, s$, *то согласно последнему правилу типизации* σ *имеет вид* $\tau \to \tau'$, *а также справедливо, что* $\Gamma \cup \{x : \tau\} \vdash s : \tau'$. *Так как* $\Gamma \subseteq \Delta$, *мы получаем* $\Gamma \cup \{x : \tau\} \subseteq \Delta \cup \{x : \tau\}$, *откуда по индуктивному предположению* $\Delta \cup \{x : \tau\} \vdash s : \tau'$. *Применяя правило типизации абстракций, получаем требуемое.* □



Типизированное лямбда исчисление

Во-вторых, элементы контекста, представляющие переменные, которые не являются свободными в заданном терме, могут игнорироваться.

Лемма 4.2 *Если* $\Gamma \vdash t : \sigma$, *то справедливо также* $\Gamma_t \vdash t : \sigma$, *где* Γ_t *содержит исключительно свободные переменные терма* t *(*$\Gamma_t = \{x : \alpha \mid x : \alpha \in \Gamma \text{ и } x \in FV(t)\}$*).*

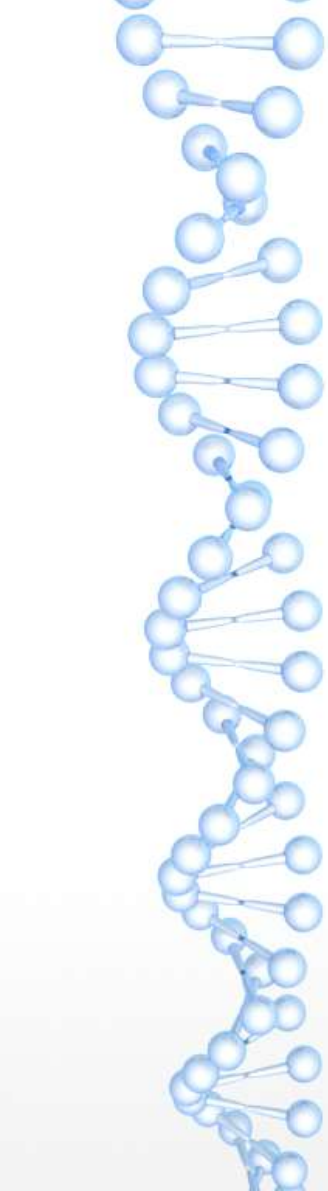
Типизированное лямбда исчисление

Доказательство: *Аналогично предыдущей лемме, докажем наше утверждение для произвольного контекста* Γ *и соответствующего ему* Γ_t *путём структурной индукции по* t. *Если* t — *переменная, то* $\Gamma \vdash t : \sigma$ *требует наличия в контексте элемента* $x : \sigma$. *Согласно первому правилу типизации* $\{x : \sigma\} \vdash x : \sigma$, *что и требуется. Тип константы не зависит от контекста, так что лемма справедлива и в этом случае. Если терм* t *представляет собой комбинацию термов вида* s u, *то для некоторого* τ *справедливо* $\Gamma \vdash s : \tau \to \sigma$ *и* $\Gamma \vdash u : \tau$. *Согласно индуктивному предположению,* $\Gamma_s \vdash s : \tau \to \sigma$ *и* $\Gamma_u \vdash u : \tau$. *Согласно лемме о монотонности, получаем* $\Gamma_{su} \vdash s : \tau \to \sigma$ *и* $\Gamma_{su} \vdash u : \tau$, *поскольку* $FV(s\ u) = FV(s) \cup FV(u)$. *Применив правило вывода типа комбинации термов, получаем* $\Gamma_{su} \vdash t : \sigma$. *Наконец, если* t *имеет вид* $\lambda x.\ s$, *это подразумевает* $\Gamma \cup \{x : \tau\} \vdash s : \tau'$, *где* σ *имеет форму* $\tau \to \tau'$. *Согласно индуктивному предположению,* $(\Gamma \cup \{x : \tau\})_s \vdash s : \tau'$, *откуда* $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \vdash (\lambda x.\ s) : \sigma$. *Теперь нам требуется лишь отметить, что* $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \subseteq \Gamma_t$ *и ещё раз применить лемму о монотонности.* $\square$

Типизированное лямбда исчисление

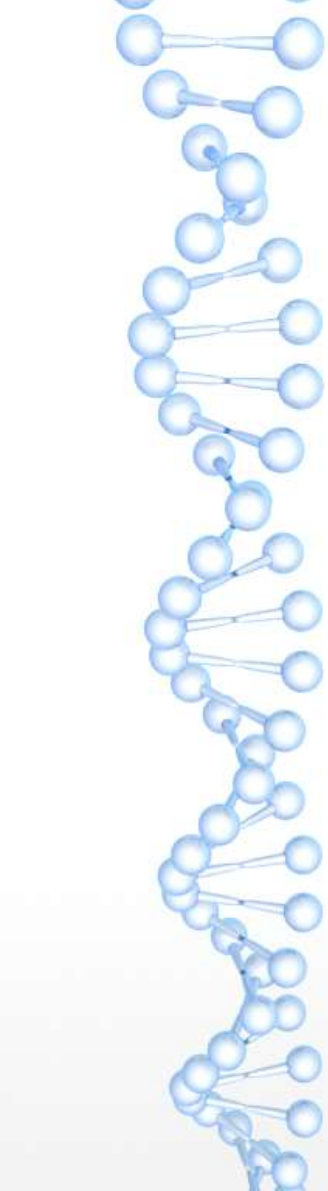
Теорема 4.3 (О сохранении типа) *Если* $\Gamma \vdash t : \sigma$ *и* $t \underset{\eta}{\longrightarrow} t'$, *то из этого следует, что* $\Gamma \vdash t' : \sigma$.

Доказательство: *Поскольку по условию теоремы терм* t *является* η*-редексом, он должен иметь структуру* $(\lambda x.\ t\ x)$, *причём* $x \notin FV(t)$. *Следовательно, его тип может быть выведен лишь из последнего правила типизации, при этом* σ *имеет вид* $\tau \to \tau'$, *и справедливо* $\{x : \tau\} \vdash (t\ x) : \tau'$. *Дальнейший анализ требует применения правила вывода типа комбинаций. Поскольку контекст может содержать не более одного утверждения о типе каждой переменной, справедливо* $\{x : \tau\} \vdash t : \tau \to \tau'$. *Так как по условию* $x \notin FV(t)$, *то применив лемму 4.2, получаем* $\vdash t : \tau \to \tau'$, *что и требовалось.* $\square$



Типизированное лямбда исчисление

Собрав воедино результаты аналогичных доказательств для других преобразований, получаем, что если $\Gamma \vdash t : \sigma$ и $t \longrightarrow t'$, то выполняется также $\Gamma \vdash t' : \sigma$. Важность этого вывода в том, что если бы правила вычислений, применяемые в ходе исполнения программы, могли изменять типы выражений, это подорвало бы основы статической типизации.



Полиморфизм

Типизация по Карри предоставляет в наше распоряжение разновидность *полиморфизма,* позволяя назначить заданному терму различные типы. Следует различать схожие понятия полиморфизма и *перегрузки.* Оба они подразумевают, что выражение может иметь множество типов. Однако, в случае полиморфизма все эти типы структурно связаны друг с другом, так что допустимы любые из них, удовлетворяющие заданному образцу. Например, тождественной функции можно назначить тип $\sigma \to \sigma$, или $\tau \to \tau$, либо даже $(\sigma \to \tau) \to (\sigma \to \tau)$, но все они имеют одинаковую структуру. С другой стороны, суть перегрузки в том, что заданная функция может иметь различные типы, структура которых может различаться, либо же допустимо лишь ограниченное множество типов. Например функции + может быть позволено иметь тип $int \to int \to int$ либо $float \to float \to float$, но не $bool \to bool \to bool$.[6] Ещё одним близким понятием являются подтипы, представляющие собой более жёсткую форму перегрузки. Введение подтипов позволяет трактовать некоторый тип как подмножество другого. Однако, этот подход на практике оказывается куда сложнее, чем кажется на первый взгляд.[7]

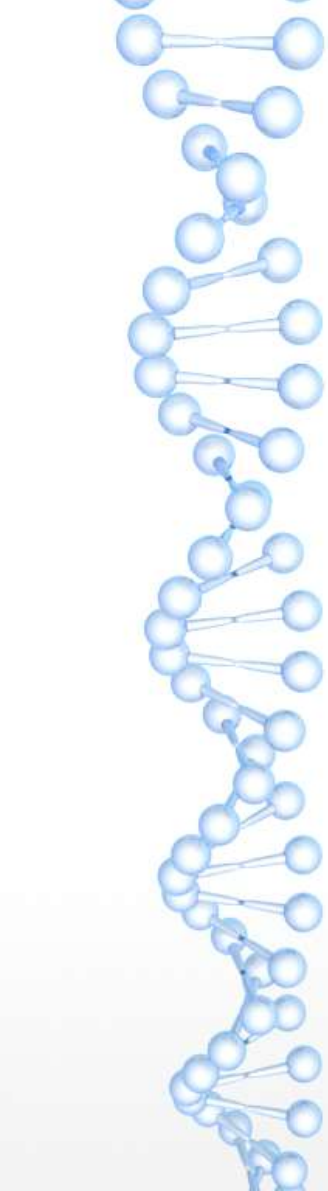
Полиморфизм

К сожалению, определённая выше система типов накладывает некоторые нежелательные ограничения на полиморфизм. Например, следующее выражение абсолютно корректно:

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

Докажем, что это выражение может быть типизировано согласно нашим правилам. Предположим, что константам можно назначить типы в пустом контексте, и что мы можем двукратно применить правило типизации комбинации термов для назначения типа `if` (принимая во внимание, что выражение вида if b then t_1 else t_2 является всего лишь сокращённой записью для COND $b\ t_1\ t_2$).

$$\dfrac{\dfrac{\dfrac{\{x : bool\} \vdash x : bool}{\vdash (\lambda x.\ x) : bool \to bool} \quad \dfrac{}{\vdash \text{true} : bool}}{\vdash (\lambda x.\ x) \text{ true} : bool} \quad \dfrac{\dfrac{\dfrac{\{x : int\} \vdash x : int}{\vdash (\lambda x.\ x) : int \to int} \quad \dfrac{}{\vdash 1 : int}}{\vdash (\lambda x.\ x)\ 1 : int} \quad \dfrac{}{\vdash 0 : int}}{\vdash \text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0 : int}$$



Полиморфизм

Два экземпляра тождественной функции получают типы $bool \to bool$ и $int \to int$ соответственно. Далее рассмотрим другое выражение:

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

Согласно нашим определениям, это всего лишь удобный способ записи для

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

Нетрудно убедиться, что тип этого выражения не может быть выведен в рамках наших правил. Мы имеем *единственный* экземпляр тождественной функции, которому должны назначить единственный тип. Подобное ограничение на практике неприемлемо, поскольку функциональное программирование предполагает частое использование `let`. Если правила типизации не будут изменены, многие выгоды полиморфизма окажутся потерянными. Нашим решением будет отказ от трактовки конструкции `let` как сокращённой записи в пользу реализации её как примитива языка, после чего ко множеству правил типизации следует добавить новое правило:

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[s/x] : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau}$$

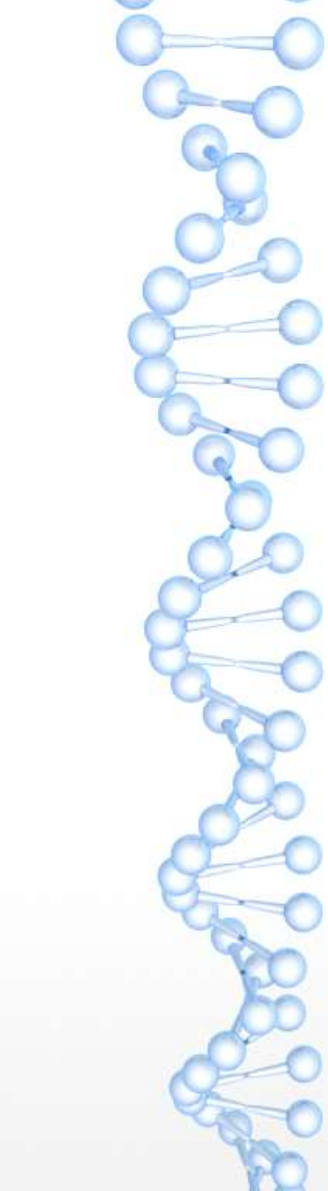
Полиморфизм

Это правило, которым вводится понятие *let-полиморфизма*, демонстрирует, что по крайней мере с точки зрения типизации, `let`-связанные переменные трактуются как простые подстановки соответствующих выражений вместо их имён. Дополнительная посылка $\Gamma \vdash s : \sigma$ требуется исключительно для того, чтобы гарантировать существование корректного типа выражения s, причём точное значение этого типа нас не интересует. Цель данного ограничения в том, чтобы избежать ошибочных выводов о существовании корректных типов для таких термов, как

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

Теперь мы в состоянии вывести тип нашего проблемного выражения, пользуясь приведёнными выше правилами:

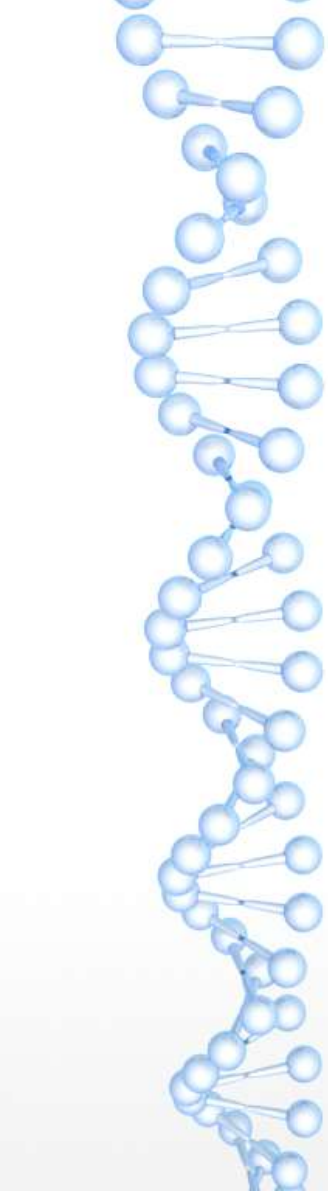
$$\dfrac{\dfrac{\dfrac{}{\{x : \sigma\} \vdash x : \sigma}}{\vdash \lambda x.\ x : \sigma \to \sigma} \quad \dfrac{\textbf{См. выше}}{\vdash \text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0 : \mathit{int}}}{\vdash \text{let } I = \lambda x.\ x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0 : \mathit{int}}$$



Нормализация

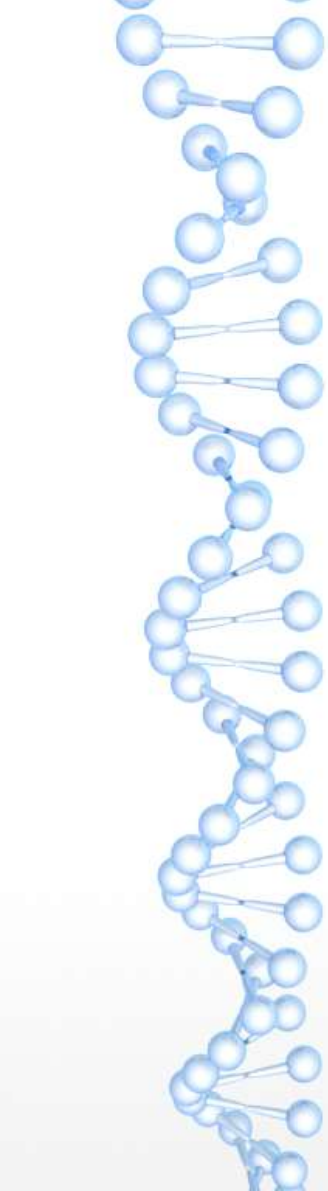
Теорема 4.5 (О сильной нормализации) *Любой типизируемый терм имеет нормальную форму, а любая возможная последовательность редукций, которая начинается с типизируемого терма, завершается.*[10]

На первый взгляд преимущества очевидны — функциональная программа, удовлетворяющая нашей дисциплине типов, может вычисляться в произвольном порядке, при этом процесс редукции всегда конечен и приводит к единственной нормальной форме. (Единственность следует из теоремы Чёрча-Россера, которая остаётся справедливой и в случае типизированного лямбда-исчисления.) Однако, возможность реализации незавершимых функций необходима для обеспечения Тьюринг-полноты,[11] в противном случае мы более не в состоянии определить произвольные вычислимые функции, более того — даже не всё множество всюду определённых функций.



Нормализация

Мы бы могли пренебречь этим ограничением, если бы оно позволяло нам использовать все функции, представляющие практический интерес. Однако, это не так — класс всевозможных функций, представимых в рамках типизированного лямбда-исчисления, оказывается весьма узким. Швихтенберг показал, что класс представимых функций на основе нумералов Чёрча ограничен всевозможными полиномами либо кусочными функциями на их основе [58]. Отметим, что этот результат имеет сугубо интенсиональную природу, то есть, определяется свойствами заданного представления чисел, а выбор другого представления ведет к другому классу функций. В любом случае, для универсального языка программирования этого недостаточно.



Нормализация

Поскольку все определимые функции являются всюду определёнными, мы, очевидно, не в состоянии давать произвольные рекурсивные определения. В самом деле, оказывается, что обычные комбинаторы неподвижной точки не поддаются типизации; очевидно, что тип $Y = \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))$ не существует, поскольку x применяется к самому себе, будучи связанным лямбда-абстракцией. Для восстановления Тьюринг-полноты введём альтернативный способ задания произвольных рекурсивных функций, не принося в жертву типизацию. Определим полиморфный оператор рекурсии, всевозможные типы которого имеют вид

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

и дополнительное правило редукции, согласно которому для произвольной функции $F : (\sigma \to \tau) \to (\sigma \to \tau)$ мы имеем

$$Rec\ F \longrightarrow F\ (Rec\ F)$$

Начиная с этого момента будем полагать, что рекурсивные определения вида `let rec` отображаются на эти операторы рекурсии.