Лекція 2. Математичні основи функціонального програмування.

λ-обчислення

На протяжении последних 400 лет, центральным понятием математики является понятие *функции*. Математические функции выражают связь между параметрами (входом) и результатом (выходом) некоторого процесса. Так как вычисление это тоже процесс, имеющий вход и выход, функция – вполне подходящее средство задания вычислений. Программа есть формальное описание конкретного вычисления. Если мы игнорируем процедурные детали («как» вычисления делаются) и только принимаем во внимание результат («что» вычисления сделали), то мы можем рассматривать программу как «черный ящик», который получает какой-то вход и возвращает какой-то выход. Другими словами, программа может быть рассмотрена как математическая функция.

Мы можем думать о программах, процедурах и функциях в языке программирования, как если бы они представляют математическое понятие функции:

- в случае программы, x представляет вход и y представляет выход;
- в случае процедуры или функции, x представляет параметры, а y представляет возвращаемую величину.

В любом случае мы можем ссылаться на x как «вход» и на y как «выход». Поэтому функциональный взгляд на программирование не делает различие между программой, процедурой или функцией. Но он, однако, всегда различает входную и выходную величины.

В языке программирования мы также должны различать определение функции и применение функции: первое описывает, как функция будет вычисляться, используя формальные параметры, в то время как функциональное применение есть вызов описанной функции, использующий фактические параметры или аргументы.

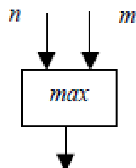
С абстрактной точки зрения мы можем рассматривать функции как «черные ящики». Рассматривая функции как черный ящик, мы можем использовать его как конструктивный блок, и с помощью объединения (когда выход одного ящика подается на вход другого) таких ящиков можно порождать более сложные операции. Такой процесс «объединения» ящиков называется *функциональной композицией*.

Пусть дана математическая функция с действительными числами max: $\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$, определяющая наибольшее значение из двух чисел:

$max(m,n) = m$, если $m>n$,

$= n$ в противном случае.

Ее представление в виде черного ящика очень просто:



Можно использовать max как блок для более сложной функции. Предположим, требуется построить функцию, которая находила бы максимум не из двух чисел, а из трех. Эту новую функцию (назовем ее $max3$) определим следующим образом:

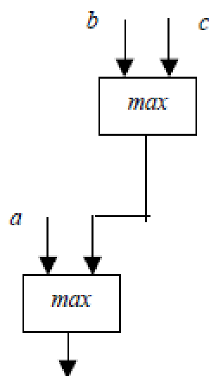
$max3\ (a,b,c) = a$, если $a \geq b$ и $a>c$ или $a \geq c$ и $a>b$,

b, если $b \geq a$ и $b>c$ или $b \geq c$ и $b>a$,

c, если $c \geq a$ и $c>b$ или $c \geq b$ и $c>a$,

a в противном случае.

Это довольно неудобное определение. Гораздо более элегантный способ определения функции $max3$ состоит в использовании уже определенной функции max:



Это можно записать следующим образом:
$max3(a,b,c) = max(a, max(b,c))$.

Поскольку функция обеспечивает детерминированное отображение тройки чисел в число, то ее можно рассматривать как черный ящик с тремя входами и одним выходом.

Таким образом, задав множество предварительно определенных черных ящиков–функций, называемых *встроенными функциями* или *примитивами*, для выполнения простых операций, подобных базовым арифметическим, можно построить новые функции, т.е. новые черные ящики для выполнения более сложных операций с помощью этих примитивов. Далее, эти новые функции можно использовать как блоки для построения еще более сложных функций и т.д.

Программа на функциональном языке состоит из множества определений функций и выражения, чья величина рассматривается как результат программы. Математической моделью функциональных языков является лямбда–исчисление. Примерами функциональных языков являются Clean, Haskell и др. Многие другие языки, такие как Лисп, имеют чисто функциональное подмножество, но также содержат нефункциональные конструкции.

Чтобы понять решение на императивном языке, мы должны понять, что машина будет делать при выполнении каждого оператора программы. При функциональном решении, с другой стороны, мы не должны думать о том, как программа будет выполняться на компьютере; отсутствует всякое упоминание об изменяемом состоянии программы или о последовательном выполнении инструкций. Функциональное решение является фактически формулировкой самой задачи, а не рецептом ее решения, и именно в этом смысле мы говорим о функциональной программе как о спецификации того, что нужно сделать вместо последовательности инструкций, описывающих, как это сделать. Таким образом, функциональное программирование является одним из видов декларативного программирования.

Функциональное программирование, по сравнению с другими парадигмами программирования, предлагает наиболее точный изоморфизм между постановкой задачей и программой.

В функциональных языках все вычисления выполняются через оценку выражений, чтобы выдать какие-то величины.

В математике переменные всегда представляют какие-то конкретные величины, в то время как в императивном программировании переменные ссылаются на конкретное место в памяти и так же на величину. В математике нет понятия «место в памяти», так что выражение $x = x+1$ не имеет смысла. Функциональный взгляд на программирование поэтому должен уничтожить понятие переменной, иное чем имя величины. Точно также исчезает присваивание как допустимый оператор.

Этот взгляд на функциональное программирование приводит к понятию «*чистое функциональное программирование*». Большинство функциональных языков оставляет некоторое понятие переменной и присваивания, и поэтому «не чисты», но все же возможно программировать эффективно, используя чистый подход. Haskell является чистым функциональным языком.

Одним из следствий отсутствия переменных и присваивания, является невозможность циклов: цикл должен иметь управляющую переменную, которая должна менять свое значение во время выполнения цикла, но это невозможно без переменных и присваиваний. Вместо этого мы должны использовать рекурсию.

Фундаментальное свойство математических функций, которое дает нам возможность собрать воедино черные ящики, – это *функциональность* (*прозрачность по ссылкам*).

Принцип *прозрачности по ссылкам* утверждает: «Значение целого не меняется, когда какая-то часть целого заменяется на равную часть». Выражение E является прозрачным по ссылкам, если любое подвыражение и его величина (как результат его вычисления) могут быть взаимно заменены без изменения величины E.

Это означает, что каждое выражение определяет единственную величину, которую нельзя изменить ни путем ее вычисления, ни предоставлением различным частям программы возможности совместно использовать это выражение. Вычисление выражения просто изменяет форму выражения, но не изменяет его величину. Все ссылки на некоторую величину эквивалентны самой этой величине, и тот факт, что на выражение можно ссылаться из другой части программы, никак не влияет на величину этого выражения.

Из-за отсутствия переменных и присваиваний в языке отсутствует понятие состояния функции. В императивном программировании два главных фактора действуют на внутреннее состояние процедуры:

- предыдущие вычисления (предшествующие вызову процедуры), включающие побочный эффект от предыдущих вызовов самой процедуры;
- порядок оценки параметров при вызове.

В императивных языках (например, Паскаль) функции могут ссылаться на глобальные данные и разрешается применять деструктивное (разрушающее) присваивание, что может привести к изменению значения функции при повторном ее вызове. Такие динамические изменения в величине данных часто именуются *побочными эффектами*. Благодаря им значение функции может изменяться, даже если ее аргументы и остаются без изменения всякий раз, когда к ней обращаются.

Пример нефункциональности Паскаля:

```
var flag: boolean;

    function f (n:integer):integer;
    begin
        if flag then f:=n
                  else f:= 2*n;
        flag:=not flag
    end;

    begin
       flag:=true;
       writeln(f(1)+f(2));      {будет напечатано 5}
       writeln(f(2)+f(1));      {будет напечатано 4}
    end.
```

Функция *f* не является математической («чистой») функцией. Операции, подобные этой, в математике не разрешены, поскольку математические рассуждения базируются на идее равенства и возможности замены одного выражения другим, означающим то же самое, т.е. определяющим ту же величину.

В функциональном программировании, однако, значение функции зависит только от значений ее параметров, но не от предыдущих вычислений, включая и вызовы самой функции. Величина любой функции также не может зависеть от порядка вычисления ее параметров (тем самым функциональные языки становятся подходящими для параллельных приложений).

Прозрачная по ссылкам функция без параметров должна всегда выдавать тот же самый результат, и поэтому ничем не отличается от константы. Более того, различные вызовы одной и той же функции с теми же аргументами будут всегда возвращать один и тот же результат.

В функциональном программировании имеется возможность оперировать функциями всевозможными способами, без ограничений. В частности функция сама по себе может быть рассматриваться как структура данных, которая может быть аргументом другой функции и которая может возвращаться как результат вычисления функции.

Ламбда–исчисление было изобретено Алонсом Чёрчем около 1930 г. Чёрч первоначально строил λ–исчисление как часть общей системы функций, которая должна стать основанием математики. Но из–за найденных парадоксов эта система оказалась противоречивой. Книга Чёрча [12] содержит непротиворечивую подтеорию его первоначальной системы, имеющую дело только с функциональной частью. Эта теория и есть λ–исчисление.

Ламбда–исчисление – это безтиповая теория, рассматривающая функции как *правила*, а не как графики. В противоположность подходу Дирихле (вводимому функции как множество пар, состоящих из аргумента и значения) более старое понятие определяет функцию как процесс перехода от аргумента к значению. С первой точки зрения x^2-4 и $(x+2)(x-2)$ – разные обозначения одной и той же функции; со второй точки зрения это разные функции.

Что значит «функция $5x^3+2$»? Если кто–то хочет быть точным, он вводит по этому поводу функциональный символ, например f, и говорит: «функция f: $\mathbf{R} \to \mathbf{R}$, определенная соотношением $f(x) = 5x^3+2$». При этом, очевидно, что переменную x можно здесь, не меняя смысла, заменить на другую переменную y. Ламбда–запись устраняет произвольность в выборе f в качестве функционального символа. Она предлагает вместо f выражение «$\lambda x.5x^3+2$».

Алонсо Чёрч

Кроме того, обычная запись $f(x)$ может обозначать как имя функции f, так и вызов функции с аргументом x. Для более строгого подхода это необходимо различать. В лямбда–обозначениях вызов функции с аргументом x выглядит как $(\lambda x.\ 5x^3+2)x$.

Функции как правила рассматриваются в полной общности. Например, мы можем считать, что функции заданы определениями на обычном русском языке и применяются к аргументам также описанным по–русски. Также мы можем рассматривать функции заданными программами и применяемые к другим программам. В обоих случаях перед нами *безтиповая структура*, где объекты изучения являются одновременно и функциями и аргументами. Это отправная точка безтипового λ–исчисления. В частности, функция может применяться к самой себе.

Лямбда–исчисление представляет класс (частичных) функций (λ–определимые функции), который в точности характеризует неформальное понятие эффективной вычислимости. Другими словами, λ–исчисления, наряду с другими подходами формализует понятие алгоритма [1, с. 143–146].

Лямбда–исчисление стало объектом особенно пристального внимания в информатике после того, как выяснилось, что оно представляет собой удобную теоретическую модель современного функционального программирования [8].

ЛИТЕРАТУРА

1. *Барендрегт, Хенк.* Ламбда-исчисление. Его синтаксис и семантика. — М.: Мир, 1985. — 606 с.

8. **Филд А., Харрисон П.** Функциональное программирование: Пер. с англ. — М.: Мир, 1993. — 637 с., ил.

12. The Calculi of Lambda-conversion. Front Cover. Alonzo Church. Princeton University Press, 1941

Большинство функциональных языков программирования используют λ–исчисление в качестве промежуточного кода, в который можно транслировать исходную программу. Функциональные языки «улучшают» нотацию λ–исчисления в прагматическом смысле, но при этом, в какой–то мере, теряется элегантность и простота последнего.

Изучение и понимание многих сложных ситуаций в программировании, например, таких, как автоаппликативность (самоприменимость) или авторепликативность (самовоспроизводство), сильно облегчается, если уже имеется опыт работы в λ–исчислении, где выделены в чистом виде основные идеи и трудности.

Язык ламбда–исчисления является сейчас одним из важнейших выразительных средств в логике, информатике, математической лингвистике, искусственном интеллекте и когнитивной науке.

Ламбда–исчисление есть язык для определения функций. Выражения языка называются λ–*выражениями* и каждое такое выражение обозначает функцию. Далее мы рассмотрим, как функции могут представлять различные структуры данных: числа, списки и т. д. Для некоторых λ–выражений мы будем использовать имена (или сокращенные обозначения), они будут записываться полужирным шрифтом.

Определение λ–выражений (λ–термов)

Имеется три вида λ–выражений:

- *Переменные*: x, y и т. д.
- *Функциональная аппликация*: если M и N есть произвольные λ–термы, то можно построить новый λ–терм (MN) (обозначающий применение, или *аппликацию*, оператора M к аргументу N). Например, если (<*m*>, <*n*>) обозначает функцию, представляющую пару чисел *m* и *n* и ***sum*** обозначает функцию сложения в λ–исчислении, то аппликация (***sum*** (<*m*>, <*n*>)) обозначает <*m*+*n*> (т. е. лямбда–терм, представляющий число $m+n$).
- *Абстракция*: По любой переменной x и любому λ–терму M можно построить новый λ–терм ($\lambda x.M$) (обозначающий функцию от x, определяемую λ–термом M). Такая конструкция называется *λ–абстракция***.** Например, λx. ***sum*** (x, <1>) обозначает функцию от x, которая увеличивает аргумент на 1.

Тождественная функция представляется λ–термом (λ*x*. *x*). Аппликация (λ*x*. *x*) *E* дает *E*.

Еще пример. Пусть λ–выражения <0>, <1>, <2>, … представляют числа 0, 1, 2, …, соответственно; и ***add*** есть λ–выражение, обозначающее функцию, удовлетворяющее правилу:

$$((\textbf{\textit{add}}\ <m>)\ <n>) = <m+n>.$$

Тогда (λ*x*. ((***add*** <1>) *x*)) есть λ–выражение, обозначающее функцию, что преобразует <*n*> в <*n*+1>, и (λ*x*. (λ*y*. ((***add*** *x*) *y*))) есть λ–выражение, обозначающее функцию, которая преобразует <*m*> в функцию (λ*y*. ((***add*** <*m*>) *y*)).

Символ *x* после λ называется *связанной переменной* абстракции и соответствует понятию формального параметра в традиционной процедуре или функции. Выражение справа от точки называется *телом абстракции*, и, подобно коду традиционной процедуры или функции, оно описывает, что нужно сделать с параметром, поступившим на вход функции. Мы читаем символ λ как «функция от» и точку (.) как «которая возвращает».

Переменная, расположенная не на месте связанной переменной, может быть *связанной* или *свободной*, что определяется с помощью следующих правил:

1. Переменная x оказывается свободной в выражении x.
2. Все x, имеющиеся в $\lambda x.M$, являются связанными. Если кроме x в $\lambda x.M$ есть переменная y, то последняя будет свободной или связанной в зависимости от того, свободно она или связанна в M.
3. Переменная встречающаяся в термах M или N выражения (MN) будет связанной или свободной в общем терме в зависимости от того, свободна она или связанна в M или N. Свободные (связанные) переменные – это переменные, которые, по крайней мере, один раз появляются в терме в свободном (связанном) виде.

Нам понадобится следующее определение *подстановки* терма в другой терм вместо свободного вхождения переменной. Для любых λ–термов M, N и переменной x через $[N/x]M$ обозначим результат подстановки N вместо каждого свободного вхождения x в M и замены всех λy в M таким образом, чтобы свободные переменные из N не стали связанными в $[N/x]M$. Мы употребляем запись $M \equiv N$ для обозначения того, что термы M и N совпадают.

Подстановка

a) $[N/x]x \equiv N$;

b) $[N/x]y \equiv y$, если переменная y не совпадает с x;

c) $[N/x]\,(PQ) \equiv ([N/x]P\;[N/x]Q)$;

d) $[N/x]\,(\lambda x.P) \equiv \lambda x.P$;

e) $[N/x]\,(\lambda y.P) \equiv \lambda y.[N/x]P$, если y не имеет свободных вхождений в N и x имеет свободное вхождение в P;

f) $[N/x](\lambda y.P) \equiv \lambda z.[N/x]([z/y]P)$, если y имеет свободное вхождение в N и x имеет свободное вхождение в P и z – любая переменная, не имеющая свободных вхождений в N.

Следующие примеры пояснят суть определения. Пусть $M \equiv \lambda y.yx$.

Если $N \equiv vx$, то $[(vx)/x](\lambda y.yx) \equiv \lambda y.\,[(vx)/x](yx)$ согласно (e)

$\equiv \lambda y.\ y(vx)$ согласно (a).

Если $N \equiv yx$, то $[(yx)/x](\lambda y.yx) \equiv \lambda z.\,[(yx)/x](zx)$ согласно (f)

$\equiv \lambda z.z(yx)$ согласно (a).

Введенное понятие подстановки требует должной мотивации. Для этого необходимо снова вспомнить о коллизии переменных при неаккуратной подстановке (см. 2.3.4).

Если бы пункт (f) в определении подстановки был опущен, то мы столкнулись бы со следующим нежелательным явлением. Хотя $\lambda v.x$ и $\lambda y.x$ обозначают одну и ту же функцию (константу, чье значение всегда есть x), после подстановки v вместо x они стали бы обозначать разные функции: $[v/x](\lambda y.x) \equiv \lambda y.v$, $[v/x](\lambda v.x) \equiv \lambda v.v$.

Мы рассмотрели, как λ–нотация может быть использована для представления функциональных выражений и сейчас готовы к тому, чтобы определить *правила конверсии* λ–исчисления, которые описывают, как вычислять выражение, т. е. как получать конечное значение выражения из его первоначального вида. Правила конверсии, описанные ниже, являются достаточно общими, так, что, например, когда они применяются к λ–терму, представляющему арифметическое выражение, то они моделируют вычисление этого выражения.

Правила конверсии

- **α–конверсия.** Любая абстракция вида $\lambda V.\ E$ может быть конвертирована к терму $\lambda V'.\ [V'/V]E$. Мы переименовываем связные переменные так, чтобы избежать коллизии переменных.
- **β–конверсия.** Любая аппликация вида $(\lambda V.\ E_1)\ E_2$ конвертируется к терму $[E_2/V]E_1$.
- **η–конверсия.** Любая абстракция вида $\lambda V.\ (E\ \ V)$, в которой V не имеет свободных вхождений в терме E может быть конвертировано к E.

Если какой-то λ–терм E_1 α–конвертируется к терму E_2, то это обозначается как $E_1 \rightarrow_\alpha E_2$. Аналогично определяются обозначения $\rightarrow_\beta$ и $\rightarrow_\eta$. Наиболее важным видом конверсии является β–конверсия; она единственная может использоваться для моделирования произвольного вычислительного механизма. α–конверсия применяется для технических преобразований связных переменных и η–конверсия выражает тот факт, что две функции, дающие один и тот же результат для любых аргументов, естественно считаются равными.

Обсудим более подробно правила конверсии. Лямбда–выражение, к которому может быть применено правило α–конверсии, называется *α–редексом*. «Редекс» есть абревиатура для «редуцируемое выражение». Правило α–конверсии говорит, что любая связанная переменная может быть корректно переименована. Например, $\lambda x.x \rightarrow_\alpha \lambda y.y$.

Лямбда–выражение, к которому может быть применено правило β–конверсии, называется *β–редексом*. Правило β–конверсии подобно вычислению вызова функции в языке программирования: тело E_1 функции $\lambda V.\ E_1$ вычисляется в окружении (в контексте), в котором «формальный параметр» V заменяется на «фактический параметр» E_2.

Примеры:

$$(\lambda x.\, f\, x)\, E \;\to_\beta\; f\, E$$
$$(\lambda x.\, (\lambda y.\, (\boldsymbol{add}\; x\; y)))\; \langle 3\rangle \;\to_\beta\; \lambda y.\, \boldsymbol{add}\; \langle 3\rangle\; y$$
$$(\lambda y.\, \boldsymbol{add}\; \langle 3\rangle\; y)\; \langle 4\rangle \;\to_\beta\; \boldsymbol{add}\; \langle 3\rangle\; \langle 4\rangle$$

Лямбда–выражение, к которому может быть применено правило η–конверсии, называется η–*редексом*. Правило η–конверсии выражает следующее свойство (называемое *экстенсиональностью*): две функции являются равными, если они дают одинаковый результат когда применяются к одинаковым аргументам. Действительно, $\lambda V.\ (E\ V)$ обозначает функцию, которая возвращает $[E'/V]\ (E\ V)$ когда применяется к аргументу E'. Если V не является свободной в E, то $[E'/V]\ (E\ V) = (E\ E')$. Так как термы $\lambda V.\ (E\ V)$ и E оба возвращают один и тот же результат $(E\ E')$, когда применяются к одинаковым аргументам, то, следовательно, они обозначают одну и ту же функцию.

Примеры:

$$\lambda x.\, \boldsymbol{add}\; x \;\to_\eta\; \boldsymbol{add}$$
$$\lambda y.\, \boldsymbol{add}\; x\; y \;\to_\eta\; \boldsymbol{add}\; x$$

Но следующая конверсия

$$\lambda x.\, \boldsymbol{add}\; x\; x \to_\eta \boldsymbol{add}\; x$$

не имеет место, поскольку переменная x свободна в $\boldsymbol{add}\ x$.

Более общим понятием, по сравнению с конверсией, является понятие *обобщенной конверсии* (по причинам, которые станут понятным позже, она также называется *одношаговой редукцией*). Терм M обобщенно α–конвертируется (β–конвертируется, η–конвертируется) к терму N, если последний получается из M в результате конверсии какого-то подтерма в M, являющегося α–редексом (β–редексом, η–редексом, соответственно). Обобщенная конверсии обозначается также как и просто конверсия: $M \to_\alpha N$ (аналогично и для других видов обобщенной конверсии).

В следующих примерах конвертируемые редексы подчеркнуты.

$$(\underline{(\lambda x.\ (\lambda y.\ (\mathbf{add}\ x\ y)))\ <3>)}\ <4> \to_\beta (\lambda y.\ \mathbf{add}\ \ <3>\ y)\ <4>$$

$$\underline{(\lambda y.\ \mathbf{add}\ \ <3> y)\ <4>} \to_\beta \mathbf{add}\ \ <3>\ <4>$$

Отметим, что в первой конверсии сам терм $((\lambda x.\ (\lambda y.\ (\mathbf{add}\ x\ y)))\ <3>)<4>$ не является редексом. Мы будем иногда писать последовательность конверсий, подобных двум предыдущим как:

$$(\underline{(\lambda x.\ (\lambda y.\ (\mathbf{add}\ x\ y)))\ <3>)}\ <4> \to_\beta (\lambda y.\ \mathbf{add}\ \ <3>\ y)\ <4> \to_\beta \mathbf{add}\ \ <3>\ <4>$$

В следующих двух примерах исходное исходное выражение одно и то же, но последовательности конвертируемых редексов различны (используемые редексы подчеркнуты):

1) $\underline{(\lambda f.\ \lambda x.\ f\ \langle 3\rangle\ x)\ (\lambda y.\ \lambda x.\mathbf{add}\ x\ y)}\ \langle 0\rangle$

$\rightarrow_{\beta}\ \underline{(\lambda x.(\lambda y.\ \lambda x\ .\ \mathbf{add}\ x\ y)\ \langle 3\rangle\ x)\ \langle 0\rangle}$

$\rightarrow_{\eta}\ \underline{(\lambda z.(\lambda y.\ \lambda x\ .\ \mathbf{add}\ x\ y)\ \langle 3\rangle\ z)\ \langle 0\rangle}$ – выбрали один из двух возможных редексов

$\rightarrow_{\beta}\ \underline{(\lambda y.\lambda z.\ \mathbf{add}\ z\ y)\ \langle 3\rangle}\ \langle 0\rangle$

$\rightarrow_{\beta}\ \underline{(\lambda z.\ \mathbf{add}\ z\ \langle 3\rangle)\ \langle 0\rangle}$

$\rightarrow_{\beta}\ \mathbf{add}\ \langle 0\rangle\ \langle 3\rangle$

2) $\underline{(\lambda f.\ \lambda x\ .f\ \langle 3\rangle\ x)\ (\lambda y.\ \lambda x.\ \mathbf{add}\ x\ y)}\ \langle 0\rangle$

$\rightarrow_{\beta}\ (\lambda x.\ \underline{(\lambda y.\ \lambda x.\ \mathbf{add}\ x\ y)\ \langle 3\rangle}\ x)\ \langle 0\rangle$ – выбрали один из двух возможных редексов

$\rightarrow_{\eta}\ \underline{(\lambda x.(\lambda x.\ \mathbf{add}\ x\ \langle 3\rangle)x\ )\ \langle 0\rangle}$

$\rightarrow_{\beta}\ \underline{(\lambda z.(\lambda x.\ \mathbf{add}\ x\ \langle 3\rangle)z\ )\ \langle 0\rangle}$ – снова делаем произвольный выбор

$\rightarrow_{\beta}\ \underline{(\lambda z.\ \mathbf{add}\ z\ \langle 3\rangle)\langle 0\rangle}$

$\rightarrow_{\beta}\ \mathbf{add}\ \langle 0\rangle\ \langle 3\rangle$

Заметим, что в данном случае независимо от выбора редексов мы пришли к одинаковому результату.

Нормальные формы

Если λ–выражение E получается из λ–выражения E' с помощью последовательности обобщенных конверсий, то естественно считать, что E' получается в результате «вычисления» E, которое более точно выражается через понятие *редукции*.

Определение отношения редукции

Пусть E и E' есть λ–выражения. Говорят, что терм E редуцируется к терму E' (и обозначают как $E \to E'$), если $E \equiv E'$ или существуют выражения E_1, E_2, …, E_n такие что:

1. $E \equiv E_1$
2. $E_n \equiv E'$
3. Для каждого i выполнено одно из трех отношений $E_i \to_\alpha E_{i+1}$, $E_i \to_\beta E_{i+1}$ или $E_i \to_\eta E_{i+1}$.

Последний пример предыдущего пункта говорит, что

$$(\lambda f.\ \lambda x\ .f <3> x)\ (\lambda y.\ \lambda x.\ \mathbf{add}\ x\ y)\ \ <0> \to\ \mathbf{add}\ <0> <3>$$

Три правила конверсии сохраняют значения λ–выражений, т. е. если терм E редуцируется к терму E', то выражения E и E' обозначают одну и ту же функцию. Мы определим понятие равенства λ–выражений.

Пусть E и E' есть λ–выражения. Говорят, что терм E *равен* терму E' (обозначают как $E = E'$), если $E \to E'$ или $E' \to E$. Мы требуем также, чтобы введенное отношение равенства для λ–выражений обладало свойством транзитивности. В частности, если $E_1 \to E'$ и $E_2 \to E'$, то считается также $E_1 = E_2$.

Говорят, что λ–выражение находится в *нормальной форме*, если к нему нельзя применить никакое правило конверсии. Другими словами, λ–выражение – в нормальной форме, если оно не содержит редексов. Нормальная форма, таким образом, соответствует понятию конца вычислений в традиционном программировании. Отсюда немедленно вытекает наивная схема вычислений:

```
while существует хотя бы один редекс
do преобразовывать один из редексов
end
```

(выражение теперь в нормальной форме).

Различные варианты конверсии в процессе редукции могут приводить к принципиально различным последствиям.

Пример:

$$(\lambda x.\lambda y.y)\ (\underline{(\lambda z.\ z\ z)\ (\lambda z.z\ z)})$$
$$\to (\lambda x.\lambda y.y)\ (\underline{(\lambda z.z\ z)\ (\lambda z.z\ z)})$$
$$\to (\lambda x.\lambda y.y)\ (\underline{(\lambda z.z\ z)\ (\lambda z.z\ z)})$$
$$\to \ldots$$

(бесконечный процесс редукции)

$$\underline{(\lambda x.\lambda y.y)\ ((\lambda z.\ z\ z)\ (\lambda z.z\ z))}$$
$$\to \lambda y.y$$

(редукция закончилась)

Порядок редукций (стратегия выбора редексов)

Самым левым редексом называется редекс, первый символ которого текстуально расположен в λ–выражении левее всех остальных редексов. (Аналогично определяется *самый правый редекс*.)

Самым внешним редексом называется редекс, который не содержится внутри никакого другого редекса.

Самым внутренним редексом называется редекс, не содержащий других редексов.

В контексте функциональных языков и λ–исчисления существуют два важных порядка редукций [8].

Аппликативный порядок редукций (АПР), который предписывает вначале преобразовывать самый левый из самых внутренних редексов.

Нормальный порядок редукций (НПР), который предписывает вначале преобразовывать самый левый из самых внешних редексов.

В терме $(\lambda x.\lambda y.y)\ (\underline{(\lambda z.zz)\ (\lambda z.z\ z)})$ подчеркнут самый левый из самых внутренних редексов – если его последовательно конвертировать, то редукция никогда не закончится.

В терме $\underline{(\lambda x.\lambda y.y)\ ((\lambda z\ .z\ z)\ (\lambda z\ .zz))}$ подчеркнут самый левый из самых внешних редексов – редукция в этом случае заканчивается за один шаг.

Функция $\lambda x.\lambda y.y$ – это классический пример функции, которая отбрасывает свой аргумент. НПР в таких случаях эффективно откладывает вычисление любых редексов внутри выражения аргумента до тех пор, пока это возможно, в расчете на то, что такое вычисление может оказаться ненужным.

В функциональных языках стратегии НПР соответствуют так называемые *ленивые вычисления*. «Не делай ничего, пока это не потребуется» – механизм вызова по необходимости (все аргументы передаются функции в не вычисленном виде и вычисляются только тогда, когда в них возникает необходимость внутри тела функции). Haskell – один из языков с ленивыми вычислениями.

Стратегия АПР приводит к *энергичным вычислениям*. «Делай все, что можешь»; другими словами, не надо заботиться о том, пригодится ли, в конечном счете, полученный результат. Таким образом, реализуется механизм вызова по значению (значение аргумента передается в тело функции). В языке Лисп реализуются, как правило, энергичные вычисления.

Теорема 14 (Теорема Чёрча-Россера о свойстве ромба) [1, с.78]. Если $E \to E_1$ и $E \to E_2$, то существует терм E' такой, что $E_1 \to E'$ и $E_2 \to E'$.

Следствие 1. Если $E_1 = E_2$, то существует терм E' такой, что $E_1 \to E'$ и $E_2 \to E'$.

Следствие 2. Если выражение E может быть приведено двумя различными способами к двум нормальным формам, то эти нормальные формы совпадают или могут быть получены одна из другой с помощью замены связанных переменных.

Теорема 15 (Теорема стандартизации) [1, с. 298–303]. Если выражение E имеет нормальную форму, то НПР гарантирует достижение этой нормальной формы (с точностью до замены связанных переменных).

Комбинаторы

Теория комбинаторов была развита российским математиком Шейфинкелем [15] перед тем как ламбда–обозначения были введены Чёрчем. Вскоре Карри переоткрыл эту теорию независимо от Шейфинкеля и Чёрча. Теорию комбинаторов так же как и λ–исчисление можно использовать для представления функций. Комбинаторы также обеспечивают хороший промежуточный код для обычных компьютеров: несколько лучших компиляторов для ленивых функциональных языков базируются на комбинаторах [8].

Существует два эквивалентных способа формализации теории комбинаторов:

1) внутри λ–исчисления;
2) как полностью независимую теорию.

Следуя первому подходу, определим комбинатор просто как λ–терм, не имеющих свободных переменных. Такой терм также называется замкнутым – он имеет фиксированное значение независимо от значения любой переменной. Оказывается любое λ–выражение равно некоторому выражению, построенному с помощью аппликации из переменных и двух комбинаторов ***K*** и ***S***. Операция β–конверсии может быть смоделирована с помощью простейших операций для ***K*** и ***S***.

Определения комбинаторов *K* и *S*

$$\boldsymbol{K} \equiv \lambda x\, y.\, x$$
$$\boldsymbol{S} \equiv \lambda f\, g\, x\, .\, (f\, x)\, (g\, x)$$

Из этих определения, используя β–конверсию получаем, что для любых термов E_1, E_2 и E_3 выполнено

$$\boldsymbol{K}\ E_1\ E_2 = E_1$$
$$\boldsymbol{S}\ E_1\ E_2\ E_3 = (E_1\ E_3)\ (E_2\ E_3)$$

Любой замкнутый λ–терм можно выразить с помощью аппликации (комбинации) двух комбинаторов $\boldsymbol{K}$ и $\boldsymbol{S}$, называемых *примитивными* комбинаторами. Определим комбинатор $\boldsymbol{I} \equiv \boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K}$, тогда $\boldsymbol{I}\ x \equiv (\boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K})\ x \equiv \boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K}\ x \rightarrow (\boldsymbol{K}\ x)\ (\boldsymbol{K}\ x) \equiv \boldsymbol{K}\ x\ (\boldsymbol{K}\ x) \rightarrow$ x. В силу η–конверсии, $\boldsymbol{I} = \lambda x.\ x$.

Теорема 16 (Теорема о функциональной полноте комбинаторов). Любое λ–выражение равно λ–выражению построенному из примитивных комбинаторов $\boldsymbol{K}$ и $\boldsymbol{S}$ и переменных с помощью только операции аппликации без использовании операции λ–абстракции.

Доказательство этой теоремы см. в [1, стр. 158-171] и, в более элементарном изложении, в [10, стр. 91–97].

Например, $\lambda x.\ f\,x = \boldsymbol{S}\ (\boldsymbol{K}\,f)\ (\boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K})$. Действительно, $(\boldsymbol{S}\ (\boldsymbol{K}\,f)\ (\boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K}))\ y \equiv (\boldsymbol{S}\ (\boldsymbol{K}\,f)\ \boldsymbol{I})\ y \equiv \boldsymbol{S}\ (\boldsymbol{K}\,f)\ \boldsymbol{I}\ y \rightarrow ((\boldsymbol{K}\,f)\ y)\ (\boldsymbol{I}\ y) \equiv (\boldsymbol{K}\,f\,y)\ (\boldsymbol{I}\ y) \rightarrow f\ (\boldsymbol{I}\ y) \rightarrow f\,y$. Таким образом, искомое равенство получается в силу η–конверсии. Как произвольное λ–выражение можно транслировать в комбинаторную форму см. [8, стр. 294–296].

Ламбда–исчисление как язык программирования

В описанном виде λ–исчисление является примитивным языком. Однако его можно использовать для представления объектов и структур современного языка программирования. Идея состоит в представление этих объектов и структур в виде λ–выражений таким образом, чтобы они обладали требуемыми свойствами. Например, для представления логических значений истинности *true* и *false* и логической связки «отрицание» (*not*) λ–выражения *true*, *false* и *not* должны удовлетворять свойствам:

$$not\ true = false$$
$$not\ false = true$$

Для того, чтобы представить конъюнкцию (*and*) соответствующее λ–выражение *and* должно обладать свойствами:

$$and\ true\ true = true$$
$$and\ true\ false = false$$
$$and\ false\ true = false$$
$$and\ false\ false = false$$

Подобно этому мы должны ожидать определенные свойства от λ–выражения *or*, представляющего операцию дизъюнкции *or* для логических значений. Выбор λ–выражений для представления программных объектов, на первый взгляд, может выглядеть немотивированным, однако, все предлагаемые определения согласованы так, что они могут работать в унисон.

Истинностные значения и условное выражение

Определим λ–выражения *true*, *false*, *not* и условное выражение *If* так, чтобы выполнялись следующие свойства:

$$not\ true = false$$
$$not\ false = true$$

$$If\ true\ E_1\ E_2 = E_1$$

$$If\ false\ E_1\ E_2 = E_2$$

Существуют бесконечно много различных способов для представления истинностных значений и отрицания; здесь предлагаются традиционные определения.

$$true \equiv \lambda x.\ \lambda y.\ x$$
$$false \equiv \lambda x.\ \lambda y.\ y$$
$$not \equiv \lambda t.\ t\ false\ true$$

Легко использовать правила конверсии, чтобы показать, что данные комбинаторы обладают желаемыми свойствами. Так, например,

$$\textit{\textbf{not true}} \equiv (\lambda t.\ t\ \textit{\textbf{false true}})\ \textit{\textbf{true}}$$
$$\rightarrow_\beta \textit{\textbf{true false true}} \equiv (\lambda x.\ \lambda y.\ x)\ \textit{\textbf{false true}}$$
$$\rightarrow_\beta (\lambda y.\ \textit{\textbf{false}})\ \textit{\textbf{true}}$$
$$\rightarrow_\beta \textit{\textbf{false}}$$

Условное выражение можно определить следующим образом

$$\textit{\textbf{If}} \equiv \lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z$$

Проверим, что данное определение обладает требуемыми свойствами:

$\textit{\textbf{If true}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textit{\textbf{true}}\ E_1\ E_2 \equiv (((\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textit{\textbf{true}})\ E_1)\ E_2 \rightarrow \textit{\textbf{true}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ x)\ E_1\ E_2 \rightarrow_\beta (\lambda y.\ E_1)\ E_2 \rightarrow_\beta E_1$

$\textit{\textbf{If false}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textit{\textbf{false}}\ E_1\ E_2 \equiv (((\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textit{\textbf{false}})\ E_1)\ E_2 \rightarrow \textit{\textbf{false}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ y)\ E_1\ E_2 \rightarrow_\beta (\lambda y.\ y)\ E_2 \rightarrow_\beta E_2$

Легко проверить, что конъюнкцию и дизъюнкцию можно определить так:

$$\textit{\textbf{and}} \equiv \lambda x.\ \lambda y.\ x\ y\ \textit{\textbf{false}}$$
$$\textit{\textbf{or}} \equiv \lambda x.\ \lambda y.\ (x\ \textit{\textbf{true}})\ y$$

Пары и кортежи

Следующие комбинаторы используются для представления пар в λ−исчислении.

$$fst \equiv \lambda x.\ x\ true$$
$$snd \equiv \lambda x.\ x\ false$$
$$(E_1, E_2) \equiv \lambda f.\ f\ E_1\ E_2$$

Выражение (E_1, E_2) представляет упорядоченную пару, причем доступ к первой компоненте осуществляется с помощью функции *fst*, а вторую компоненту возвращает функция *snd*. Проверим:

$fst\ (E_1, E_2)$
$\equiv (\lambda x.\ x\ true)\ (\lambda f.\ f\ E_1\ E_2)$
$\to_\beta (\lambda f.\ f\ E_1\ E_2)\ true$
$\to_\beta true\ E_1\ E_2$
$\to_\beta (\lambda x.\ \lambda y.\ x)\ E_1\ E_2$
$\to_\beta (\lambda y.E_1)\ E_2$
$\to_\beta E_1$

Пара есть структура данных с двумя компонентами. Обобщенная структура с n компонентами называется *n–кой* или *кортежем* и легко определяется через пары.

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

$(E_1, E_2, \ldots, E_n)$ есть n–кортеж с компонентами $E_1, E_2, \ldots, E_n$ и длиной n. Пары суть кортежи длиной 2. С помощью следующих выражений получаем доступ к отдельным компонентам кортежа.

$$E\downarrow 1 \equiv fst\ E$$

$$E\downarrow 2 \equiv fst\ (snd\ E)$$

$$E\downarrow i \equiv fst\ (\underbrace{snd\ (snd\ (\ldots(snd}_{i-1\ \text{раз}}\ E)\ldots))), \text{ если } i \text{ меньше длины } E$$

$$E\downarrow n \equiv \underbrace{snd\ (snd\ (\ldots(snd}_{n-1\ \text{раз}}\ E)\ldots)), \text{ если } n \text{ равно длине } E$$

Проверим, что эти определения правильно работают:

$(E_1, E_2, \ldots, E_n){\downarrow}1$
$\equiv \mathit{fst}\ (E_1, (E_2, (\ldots(E_{n-1}, E_n)\ldots)))$
$\to E_1$

$(E_1, E_2, \ldots, E_n){\downarrow}2$
$\equiv \mathit{fst}\ (\mathit{snd}\ (E_1, (E_2, (\ldots(E_{n-1}, E_n)\ldots))))$
$\to \mathit{fst}\ (E_2, (\ldots(E_{n-1}, E_n)\ldots))$
$\to E_2$

В общем случае $(E_1, E_2, \ldots, E_n){\downarrow}i = E_i$ для $1 \le i \le n$.

Числа

Существует много способов для определения чисел в виде λ–выражений; каждый способ имеет свои преимущества и недостатки [1]. Нашей целью является определить комбинатор <*n*> для представления натурального числа *n*. Необходимо определить также примитивные арифметические операции в терминах λ–выражений. Например, нам необходимы λ–выражения ***suc, pre, add*** и ***iszero***, представляющие функции следования $n \to n+1$, предшествования $n \to n-1$, сложения и тест для нуля, соответственно. Эти λ–выражения представляют числа корректно, если они обладают следующими свойствами:

suc <*n*> = <*n*+1> (для всех натуральных чисел *n*);
pre <*n*> = <*n*–1> (для всех натуральных чисел *n*>0);
add <*m*> <*n*> = <*m*+*n*> (для всех натуральных чисел *m* и *n*);
iszero <0> = ***true***
iszero (***suc*** <*n*>) = ***false***

Определим натуральные числа, следуя Чёрчу. Будем использовать обозначение $f^n\ x$ чтобы представить n–кратное применение f к x. Например, $f^3\ x \equiv f\ (f\ (f\ x)))$. Для удобства $f^0\ x$ представляет просто x. В общем виде,

$$E^0\ E_1 \equiv E_1$$
$$E^n\ E_1 \equiv \underbrace{(E\ (E\ (\ldots(E}_{n\ \text{раз}}\ E_1)\ldots)))$$

Легко видеть, что $E^n\ (E\ E_1) = E^{n+1}\ E_1 = E\ (E^n\ E_1)$.

$$<0> \equiv \lambda f\ x.\ x$$
$$<1> \equiv \lambda f\ x.\ f\ x$$
$$<2> \equiv \lambda f\ x.\ f\ (f\ x)$$
$$\ldots$$
$$<n> \equiv \lambda f\ x.\ f^n\ x$$
$$\ldots$$

Такое представление чисел позволяет легко определить примитивные арифметические функции.

$$\mathit{suc} \equiv \lambda n\, f x.\ n\ \ f (f\ \ x)$$
$$\mathbf{add} \equiv \lambda\ m\ n\, f x.\ m\, f\, (\ n\, f x)$$
$$\mathbf{iszero} \equiv \lambda n.\ n\ (\lambda\ x.\ \mathit{false})\ \mathit{true}$$

Проверим, что введенные комбинаторы обладают нужными свойствами:

$\mathit{suc}\ <n> \equiv (\lambda n\, f x.\ n\ \ f (f\ \ x))\ (\lambda f x.\ f^{n}\, x)$
$= \lambda f x.((\lambda f x.\ f^{n}\, x)\ \ f (f\ \ x))$
$= \lambda f x.((\lambda x.\ f^{n}\, x)\ \ (f\ \ x))$
$= \lambda f x.\ f^{n}\, (f\ \ x)$
$= \lambda f x.\ f^{n+1}\ x$
$= <\mathbf{n+1}>$

$\mathbf{iszero}\ <0> \equiv (\lambda n.\ n\ (\lambda x.\ \mathit{false})\ \mathit{true})\ (\lambda f x.\ x)$
$= (\lambda f x.\ x)\ (\lambda x.\ \mathit{false})\ \mathit{true}$
$= (\lambda x.\ x)\ \mathit{true}$
$= \mathit{true}$

$\mathbf{add}\ <p>\ <q> \equiv (\lambda\ m\ n\, f x.\ m\, f\, (\ n\, f x))\ <p>\ <q>$
$= (\lambda\ n\, f x.\ <p>\, f\, (\ n\, f x))\ <q>$

$$= \lambda f x.\ \langle p \rangle\, f\, (\langle q \rangle\, f\, x)$$
$$= \lambda f x.\ (\lambda g\, y.\ g^{p}\, y)\, f\, (\langle q \rangle\, f\, x)$$
$$= \lambda f x.\ (\lambda\, y.\ f^{p}\, y)\ (\langle q \rangle\, f\, x)$$
$$= \lambda f x.\ (\lambda\, y.\ f^{p}\, y)\ ((\lambda h\, z.\ h^{q}\, z)\, f\, x)$$
$$= \lambda f x.\ (\lambda\, y.\ f^{p}\, y)\ ((\lambda\, z.\ f^{q}\, z)\ x)$$
$$= \lambda f x.\ (\lambda\, y.\ f^{p}\, y)\ (f^{q}\, x)$$
$$= \lambda f x.\ f^{p}\, (f^{q}\, x)$$
$$\equiv \lambda f x.\ f^{p+q}\, x \equiv \langle p+q \rangle$$

Функция предшествования *pre* определяется с большим трудом, чем предыдущие функции. Сам Алонсо Чёрч бился несколько месяцев над тем, чтобы определить эту функцию. Чёрч так и не справился с этой задачей и уже уверился в неполноте своего вычисления, но в 1932 г. Стефен Клини, тогда молодой аспирант, нашел решение, и это было его первым математическим результатом.

Трудность в определения функции предшествования состоит в том, что, имея терм $\lambda f\,x.\ f^n\,x$, надо избавиться от первой аппликации f в f^n. Чтобы достигнуть этого, предварительно определим функцию ***prefn***, оперирующую с парами и обладающую следующими свойствами:

1) $\boldsymbol{prefn}\ f\ (\boldsymbol{true}, x) = (\boldsymbol{false}, x)$
2) $\boldsymbol{prefn}\ f\ (\boldsymbol{false}, x) = (\boldsymbol{false}, f\,x)$

Из этого следует, что

3) $(\boldsymbol{prefn}\ f)^n\ (\boldsymbol{false}, x) = (\boldsymbol{false}, f^n\ x)$
4) $(\boldsymbol{prefn}\ f)^n\ (\boldsymbol{true}, x) = (\boldsymbol{false}, f^{n-1}\ x)$ (если $n>0$)

Таким образом, последнее равенство, показывает, как можно получить $n-1$–кратную аппликацию f к x. Определим ***prefn*** следующим образом:

$$\boldsymbol{prefn} \equiv \lambda\, f\, p.(\boldsymbol{false}, (\boldsymbol{If}\ \ (\boldsymbol{fst}\ \ p)\ (\boldsymbol{snd}\ p)\ (f\ (\boldsymbol{snd}\ p))))$$

Из определения следует, что $\textbf{\textit{prefn}}\ \textbf{\textit{f}}\ (b, x) = (\textit{false}, (\textit{If}\ \ b\ x\ (f\ x))$, и, следовательно:

$\textbf{\textit{prefn}}\ \textbf{\textit{f}}\ (\textit{true}, x)$
$= (\textit{false}, (\textit{If}\ \ \textit{true}\ x\ (f\ x)))$
$= (\textit{false}, x)$

$\textbf{\textit{prefn}}\ \textbf{\textit{f}}\ (\textit{false}, x)$
$= (\textit{false}, (\textit{If}\ \ \textit{false}\ x\ (f\ x)))$
$= (\textit{false}, f\ x)$

Равенство 3 докажем по индукции:
Базис индукции, n=0:

$(\textit{prefn}\ f)^0\ (\textit{false}, x)$
$\equiv (\textit{false}, x)$
$\equiv (\textit{false}, f^0\ x)$

Индуктивный переход:

$(prefn\ f)^{n+1}\ (false, x) \equiv$
$(prefn\ f)^{n}\ ((prefn\ f)\ (false, x))$
$= (prefn\ f)^{n}\ (false, f\, x)$
$=$ (по предположению индукции) $(false, f^{n}\ (f\, x))$
$\equiv (false, f^{n+1}\ x)$

Равенство 4 следует из равенства 3:

$(prefn\ f)^{n}\ (true, x) \equiv$ (если n > 0)
$(prefn\ f)^{n-1}\ ((prefn\ f)\ (true, x))$
$= (prefn\ f)^{n-1}\ (false, x)$
$= (false, f^{n-1}\ x)$

Функция предшествования сейчас может быть определена так:

$$pre \equiv \lambda\, n\, f\, x\, .snd\ (n\ \ (prefn\ \ f)\ (true,\ x))$$

Проверим, что это правильное определение, используя экстенсиональность. Докажем, что при $n>0$ выполнено $pre\ \langle n\rangle\ f\, x = \langle n-1\rangle\ f\ \ x$, откуда будет следовать, что $pre\ \langle n\rangle \to_{\eta} \langle n-1\rangle$. Действительно,

$$\begin{aligned}
&\langle n-1\rangle\ f\ \ x \\
&\equiv (\lambda f\, x.\ f^{\,n-1}\ x)\ f\ \ x \\
&= f^{\,n-1} x
\end{aligned}$$

$$\begin{aligned}
&pre\ \langle n\rangle\ f\, x \\
&\equiv (\lambda\ n\, f\, x\ .snd\ (n\ \ (prefn\ \ f)\ (true,\ x)))\ \langle n\rangle\ f\, x \\
&= snd\ (\langle n\rangle\ \ (prefn\ \ f)\ (true,\ x)) \\
&= snd\ ((\lambda f\, x.\ f^{\,n}\ x)\ (prefn\ \ f)\ (true,\ x)) \\
&= snd\ (\ (prefn\ \ f)^{\,n}\ (true,\ x)) \\
&= snd\ (false,\ f^{\,n-1}\ x) \\
&= f^{\,n-1} x
\end{aligned}$$

Очевидно,

$$
\begin{aligned}
& pre <0> \\
& \equiv (\lambda\, n\, f\, x\, .snd\, (n\ \ (prefn\ \ f)\, (true, x)))\ <0> \\
& = \lambda\, f\, x\, .snd\, (<0>\ \ (prefn\ \ f)\, (true, x))\ \ \text{(по определению } <0>\text{)} \\
& = \lambda\, f\, x\, .snd\, ((\lambda f\, x.\ x)\, (prefn\ \ f)\, (true, x)) \\
& = \lambda\, f\, x\, .snd\, (true, x) \\
& = \lambda\, f\, x\, .x \\
& = <0>
\end{aligned}
$$