

Федеральное государственное бюджетное  
образовательное учреждение высшего образования  
«Санкт-Петербургский национальный исследовательский  
Академический университет Российской академии наук»  
Центр высшего образования

Кафедра математических и информационных технологий

Жаворонков Эдгар Андреевич

# Реализация сравнения различных представлений $\lambda$ -термов

Магистерская диссертация

Допущена к защите.  
Зав. кафедрой:  
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:  
каф. МиИТ, преп. Исаев В. И.

Рецензент:  
ООО «ИнтеллиДжей Лабс», программист Березун Д. А.

Санкт-Петербург  
2017

# Оглавление

<b>Введение</b>	<b>3</b>
<b>1. Описание предметной области</b>	<b>5</b>
1.1. $\lambda$ -исчисление . . . . .	5
1.2. Теории типов . . . . .	5
1.3. Соответствие Карри-Говарда . . . . .	6
1.4. Постановка задачи . . . . .	7
1.5. Существующие решения . . . . .	8
<b>2. Представления <math>\lambda</math>-термов</b>	<b>10</b>
2.1. Именованное представление термов . . . . .	10
2.2. Неименованное представление термов . . . . .	14
2.3. Монадическое представление термов . . . . .	17
2.4. Преобразования между представлениями . . . . .	23
<b>3. Особенности реализации</b>	<b>27</b>
3.1. Описание языка Vclang . . . . .	27
3.2. Реализация неименованного представления термов . . . . .	30
3.3. Реализация монадического представления термов . . . . .	32
3.4. Реализация именованного представления термов . . . . .	34
<b>Заключение</b>	<b>38</b>
<b>Список литературы</b>	<b>39</b>
<b>Приложение А. Пример формализации монадического представления термов</b>	<b>42</b>

# Введение

Верификация программного обеспечения используется в важнейших отраслях индустрии разработки программного обеспечения и включает в себя формальные рассуждения о программах и их свойствах. Так как языки программирования используют имена для идентификации сущностей (данные, функции и другие конструкции), то в программах зачастую возникают проблема коллизий имен. Она заключается в том, что разные сущности идентифицируются одним именем.

Языки программирования или средства для разработки пытаются сами решить эту проблему. Например анализатор кода в среде разработки может подсказать о том, что такое имя уже занято и программисту следует придумать новое. Некоторые языки разрешают имена с помощью системы модулей (или пространств имен, как например в языке C++).

Более того, иногда возникает желание рассматривать конструкции в языке программирования с точностью до имен параметров. Это полезно, например, при поиске фрагментов кода, которые производят одинаковые вычисления или подчиняются некоторому общему шаблону. Пример таких фрагментов – всевозможные циклы и обходы контейнеров.

Соответственно, для того, чтобы доказывать какие-либо свойства программ необходима формальная система, позволяющая решать проблемы коллизий имен и формально описывающая отношение так называемой  $\alpha$ -эквивалентности – эквивалентности в поведении конструкций с разными именами формальных параметров. Пример такой системы – это  $\lambda$ -исчисление, лежащее в основе функциональных языков программирования. С помощью фундаментальной операции подстановки, мы можем не только записывать лямбда-выражения<sup>1</sup>, но и сокращать их. Другими словами, мы можем осуществлять процесс вычисления. Естественно, что операция подстановки удовлетворяет некоторым базовым свойствам, которые мы опишем далее.

Существуют различные представления этой системы, по-разному решающие описанную выше проблему. Например, в именованном представлении у переменных есть имена и нам нужно рассматривать выражения с точностью до имен параметров функций. Существует неименованное представление, в котором, например, переменная – это индекс в некотором глобальном хранилище переменных. Эти идеи можно обобщить и рассуждать о выражениях, используя методы теории категорий.

В работе делается попытка установить равенство между различными представлениями этой системы. Для этого мы реализуем с помощью системы автоматического доказательства теорем эти представления, и построим между ними взаимно однозначные соответствия.

---

<sup>1</sup>Далее мы будем использовать термины «лямбда-терм», « $\lambda$ -терм», «лямбда-выражение», полагая их синонимами.

В первой главе дан анализ предметной области и краткое описание  $\lambda$ -исчисления. Обозначена цель работы и задачи, решение которых необходимо для её достижения. Кроме того, рассмотрены уже существующие решения и описаны их отличия от представленного в работе.

Во второй главе подробнее описаны три представления  $\lambda$ -термов. Для каждого из представлений определены и доказаны характерные свойства. Кроме того, показано, что эти представления равны между собой.

В третьей главе описаны детали реализации. Описан язык, с помощью которого формализованы рассмотренные во второй главе представления, а так же тонкие моменты, с которыми пришлось столкнуться в ходе выполнения работы.

# 1. Описание предметной области

## 1.1. $\lambda$ -исчисление

**Историческая справка** Лямбда-исчисление – это формальная система, придуманная в 30-ых годах прошлого века Алонзо Черчём (англ. Alonso Church) [8] с целью анализа и формализации понятия вычислимости. В 60-ых годах Питером Ландином (англ. Peter Landin) была опубликована работа [22], в которой выдвигалась идея о том, что  $\lambda$ -исчисление может использоваться для моделирования различных выражений в языках программирования того времени, что в дальнейшем привело к развитию языков в стиле **ML**. С тех пор идеи  $\lambda$ -исчисления широко используются в мире функционального программирования.

**Неформальное описание  $\lambda$ -термов** Мы формально определим  $\lambda$ -термы во второй главе, здесь же мы просто скажем, что термы  $\lambda$ -исчисления рекурсивно конструируются из переменных с помощью всего двух операций – применения функции к аргументу и создания анонимной функции. Наличие каких-либо констант здесь не предполагается. Несмотря на кажущуюся простоту,  $\lambda$ -исчисление является очень мощной формальной системой, в частности, Шейнфинкелем (нем. Moses Schönfinkel) и Карри (англ. Haskell Brooks Curry) в работах [30, 13] введен в рассмотрение базис из двух термов (комбинаторов)  $S = \lambda f g x. f x (g x)$  и  $K = \lambda x y. x$ , который примечателен тем, что обладает полнотой по Тьюрингу.

Отметим, что лямбда-термы допускают не единственный способ записи. В работе мы неоднократно будем употреблять термин «представление термов». Под ним мы будем понимать способ, которым термы кодируются на каком-либо языке. Мотивация к появлению различных представлений  $\lambda$ -термов состоит в том, что являясь языком программирования, лямбда-исчисление само столкнулось с проблемой коллизии имен переменных. Различные представления термов по-разному решают эту проблему. Как следствие, какие-то представления удобнее для восприятия человеком и неформальных рассуждений «на бумаге». Какие-то представления удобны для компьютерной реализации и используются как внутреннее представление в функциональных языках программирования или системах автоматического доказательства теорем, как например в **Agda** [26].

## 1.2. Теории типов

Изначально, в  $\lambda$ -исчислении не вводилось никаких правил типизации (встречается термин «бестиповое» или «чистое»  $\lambda$ -исчисление, англ. untyped  $\lambda$ -calculus), однако в дальнейшем появилось множество типизированных вариаций. Далее мы будем употреблять термины «чистое  $\lambda$ -исчисление, бестиповое  $\lambda$ -исчисление,  $\lambda$ -исчисление», по-

лагая их синонимами. Хенком Барендрегтом(нидерл. Hendrik Pieter Barendregt) в [3] описан так называемый  $\lambda$ -куб, который наглядно классифицирует восемь различных систем типизации лямбда-исчисления.

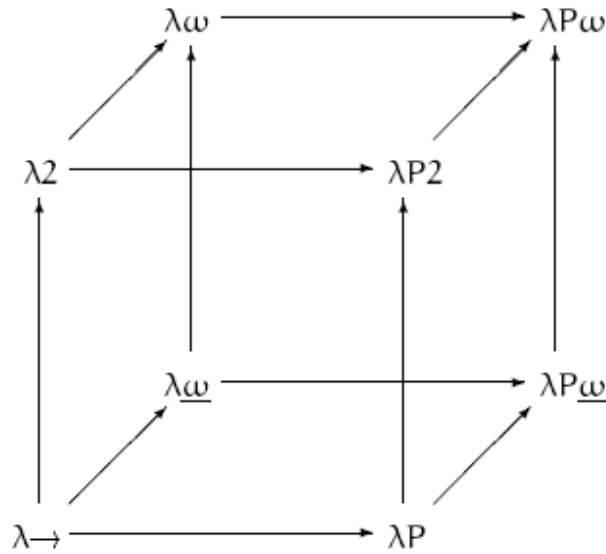


Рис. 1: Лямбда-куб<sup>2</sup>

База куба – просто типизированное  $\lambda$ -исчисление( $\lambda \rightarrow$ ), в котором термы могут зависеть только от термов. Три оси соответствуют расширениям, комбинации которых позволяют получить остальные системы типов:

1. Термы, которые зависят от типов – система  $\lambda 2$  или **System F**
2. Типы, которые зависят от типов – система  $\lambda \omega$ (операторы над типами)
3. Типы, которые зависят от термов – система  $\lambda P$ (зависимые типы)

### 1.3. Соответствие Карри-Говарда

Соответствие Карри-Говарда(англ. Curry-Howard correspondence) [18] устанавливает прямую связь между логикой и теорией типов. Логической связке соответствует конструкция в теории типов, а логическому утверждению – тип. Доказательству того факта, что утверждение истинно, соответствует тогда доказательство того факта, что соответствующий этому утверждению тип населен. Иначе говоря, мы можем предъявить  $\lambda$ -терм соответствующего типа, чтобы доказать исходное утверждение. Для наглядности некоторые соответствия сведены в таблицу:

<sup>2</sup>Изображение взято с сайта [https://en.wikipedia.org/wiki/Lambda\\_cube](https://en.wikipedia.org/wiki/Lambda_cube), автор – Денис Москвин

Высказывание $A$ :	Тип $A$ :
Истинно	Населен
Тождественная истина	$\top$ (единичный тип)
Тождественная ложь	$\perp$ (пустой тип без обитателей)
$\neg A$ (отрицание)	$A \rightarrow \perp$
$A \wedge B$ (конъюнкция)	$A \times B$ (тип-произведение)
$A \vee B$ (дизъюнкция)	$A \coprod B$ (тип-сумма)
$A \rightarrow B$ (импликация)	$A \rightarrow B$ (тип функций из $A$ в $B$ )
$\exists x.P(x)$	$\Sigma(x : A)(Pa)$ (тип зависимых пар)
$\forall x.P(x)$	$\Pi(x : A) \rightarrow (Pa)$ (тип зависимых функций)

Таблица 1: Соответствия высказываний в логике и конструкций в теории типов

Чем больше логических связок мы хотим использовать, тем более мощные теории типов нам придется использовать, чтобы доказывать эти утверждения. Так, например, если нам потребуется доказывать формулы пропозициональной логики, то мы можем обойтись просто типизированным  $\lambda$ -исчислением. Если нам понадобятся кванторы в формулах, то на помощь приходят теории с зависимыми типами. Таким образом, благодаря соответствию Карри-Говарда, процесс доказательства утверждений становится похож на программирование, следовательно, задача проверки корректности доказательств становится похожа на задачу проверки типов.

Существует две известные теории с зависимыми типами – исчисление конструкций (Calculus Of Constructions, соответствующее вершине  $\lambda P\omega$  лямбда-куба), представленное Тьерри Коканом в [11] и интуиционистская теория типов Мартин-Лёфа (Martin-Löf Type Theory), описанная в [23]. Их расширения лежат в основе таких систем интерактивного доказательства теорем (языков с зависимыми типами) как **Coq** и **Agda** соответственно. Обычно эти языки используют для верификации программ, но так как они являются полноценной логикой, то с их помощью можно формулировать и доказывать математические утверждения. Говоря «формализация», мы будем иметь в виду именно процесс формулирования и доказательства логических утверждений с помощью системы интерактивного доказательства теорем.

## 1.4. Постановка задачи

Целью работы является доказательство равенства между различными представлениями  $\lambda$ -термов. Для это мы планируем решить следующие задачи:

1. Исследовать существующие решения задачи формализации лямбда-исчисления с целью доказательства равенства между различными представлениями термов.

2. Формализовать интересующие нас представления лямбда-термов с помощью языка **Vclang**:
  - (a) Именованное
  - (b) Неименованное
  - (c) Монадическое
3. Для каждого представления реализовать операцию подстановки
4. Для каждого представления формализовать свойства операции подстановки:
  - (a) Унитарность (наличие правой и левой единицы)
  - (b) Ассоциативность
5. С помощью языка **Vclang** доказать, что описанные в первом пункте представления равны между собой

Операция подстановки является фундаментальной операцией над термами. С её помощью вводятся отношения (например  $\beta$ -редукция), которые позволяют не просто записывать  $\lambda$ -выражения, но и сокращать их, то есть вычислять. Унитарность и ассоциативность – её основные свойства, поэтому мы хотим формализовать именно эту операцию.

## 1.5. Существующие решения

Стоит отметить, что задача формализации  $\lambda$ -исчисления довольно популярна, в связи с чем существует довольно много её решений. Один из примеров – [28], в котором, в частности, формализовано чистое  $\lambda$ -исчисление. Для него, как и для работ [31, 1, 4, 25, 19] характерно использование неименованного представления термов через индексы Де Брауна. Работы [24, 10, 15, 16, 29, 32] отличаются от вышеперечисленных тем, что авторы формализуют именованное представление термов. Есть работа [2], в которой формализуется монадическое представление термов.

Вышеперечисленные работы объединяет то, что авторы рассматривают только одно представление термов (именованное, неименованное или монадическое), не устанавливая между ними никакого соответствия. Кроме того, некоторые работы формализуют типизированные вариации лямбда-исчисления. Есть, однако, работа [5], в которой реализовано сравнение именованного и неименованного представлений, но в том смысле, что авторы реализовали оба представления термов и сделали выводы о том, какое из оказалось удобнее для реализации.



Стоит отметить и работу [33], в которой средствами логического программирования на языке **Prolog** построены биекции между именованным и неименованным представлениями лямбда-термов. Однако доказательства того факта, что авторы предъявили именно биекции, в работе не приводится.

Мы, в свою очередь, хотим не просто формализовать различные представления чистого лямбда-исчисления, но и установить между ними соответствие. Именно, мы хотим доказать, что различные представления термов равны между собой в том смысле, что равны типы, соответствующие этим представлениям. С этим связан тонкий момент, потому что именованные термы рассматриваются с точностью до  $\alpha$ -эквивалентности (формально это понятие будет введено далее в разделе 2.1). То есть мы рассматриваем не просто множество термов, но множество термов, в котором  $\alpha$ -эквивалентные термы отождествлены, так называемое фактор-множество (формальное определение этого понятия, опять же будет дано в разделе 2.1).

Новизна нашего решения заключается в том, что во-первых, у нас будет не два, а три различных представления, а во-вторых, оно будет использовать отличную от других теорию типов, которая позволяет удобным образом конструировать фактор-типы. В разделе 3 мы подробнее опишем язык, построенный на основе этой теории. Мы сравним его с известными системами автоматического доказательства теорем и выделим его достоинства, которые привели нас к выбору именно этого языка.

## 2. Представления $\lambda$ -термов

В этой главе мы опишем три представления  $\lambda$ -термов: именованное, неименованное (через индексы Де Брауна) и монадическое. Мы формально опишем, как определяются термы в каждом из представлений, определим характерные свойства для каждого представления и покажем, почему они верны. Кроме того, мы установим взаимно-однозначное соответствие между всеми представлениями.

### 2.1. Именованное представление термов

Термы  $\lambda$ -исчисления ( $\lambda$ -термы) в именованном представлении конструируются из переменных путем применения термов друг к другу или создания анонимных функций.

Формально, пусть  $\mathcal{V} = \{x, y, z, \dots\}$  – счетное множество переменных. Договоримся обозначать переменные прописными буквами, а произвольные термы – заглавными. Тогда множество  $\lambda$ -термов  $\Lambda$  определяется индуктивно, согласно следующим правилам вывода:

$$\begin{array}{c} \frac{v \in \mathcal{V}}{v \in \Lambda} \\[1em] \frac{M \in \Lambda \quad N \in \Lambda}{MN \in \Lambda} \\[1em] \frac{M \in \Lambda \quad v \in \mathcal{V}}{\lambda v.M \in \Lambda} \end{array}$$

Нотация аппликации  $MN$  обозначает применение функции  $M$  ко входу  $N$ . Заметим, что так как здесь не вводится никаких правил типизации, то ничто не мешает нам применить терм к самому себе (например  $FF$ ). Нотация абстракции  $\lambda x.M$ , в свою очередь, обозначает создание анонимной функции от переменной  $x$ , которая сопоставляет конкретному значению  $x$  выражение  $M[x]$ . Здесь заметим, что терм  $M$  вовсе не обязан содержать в себе переменную  $x$ , в таком случае мы считаем абстракцию  $\lambda x.M$  константной функцией.

Некоторые примеры термов:

$$\begin{array}{c} \lambda x.x \\ \lambda xy.x \\ (\lambda x.f(xx))(\lambda x.f(xx)) \end{array}$$

Вхождение переменной  $x$  после абстракции  $\lambda x.M$  называется *связанным*. Соответственно, до абстракции вхождение этой переменной было *свободным*. Существует так называемое соглашение Барендрегта, согласно которому имена переменных в терме

выбираются уникальным образом. Тогда мы можем формально определить множества  $FV(T)$  свободных и  $BV(T)$  связанных переменных терма  $T$  следующим образом:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \\ BV(x) &= \emptyset \\ BV(MN) &= BV(M) \cup BV(N) \\ BV(\lambda x.M) &= \{x\} \cup BV(M) \end{aligned}$$

Применение абстракции к некоторому аргументу  $(\lambda x.M)N$  – это *подстановка*  $M[x \mapsto N]$  терма  $N$  вместо *свободных* вхождений переменной  $x$  в терме  $M$ . Формально, правила подстановки выглядят следующим образом:

$$\begin{aligned} x[x \mapsto N] &= N \\ y[x \mapsto N] &= y, (x \neq y) \\ (TS)[x \mapsto N] &= T[x \mapsto N]S[x \mapsto N] \\ (\lambda x.T)[x \mapsto N] &= \lambda x.T \\ (\lambda y.T)[x \mapsto N] &= \lambda y.T[x \mapsto N], (y \notin FV(N), x \neq y) \end{aligned}$$

Рассмотрим, что произойдет, если в последнем правиле условие  $y \notin FV(N)$  не выполняется:

$$(\lambda y.x)[x \mapsto y] = \lambda y.y$$

Получилось, что в результате подстановки мы превратили константную функцию  $\lambda y.x$  в тождественную. Такая ситуация носит название захвата переменной, когда при подстановке в  $\lambda$ -абстракцию свободные переменные подставляемого терма могут оказаться захваченными абстракцией.

Однако воспользовавшись соглашением Барендрегта, мы можем переименовать связанную переменную  $y$  в «свежую»  $z$  и поведение абстракции  $\lambda z.x$  не изменится. Далее мы будем под «свежей» переменной подразумевать переменную такую, что она не встречается среди переменных терма. Тогда подстановку можно использовать без каких-либо оговорок о свободных и связанных переменных. Пример выше превратится в:

$$(\lambda z.x)[x \mapsto y] = \lambda z.y$$

Мы можем переименовывать связанные переменные в абстракциях и их поведение при применении к аргументам не изменится. Из этого мы можем заключить, что их имена не играют для нас никакой роли. Поэтому, как правило,  $\lambda$ -термы и рас-

смаатривают с точностью до имен переменных, связанных абстракциями. Здесь мы поступим точно так же. Формально, на множестве именованных термов  $\Lambda$  можно задать отношение  $=_\alpha \in \Lambda \times \Lambda$ , которое называется  $\alpha$ -эквивалентностью и определяется как минимальное отношение конгруэнтности, порожденное следующими правилами вывода:

$$\frac{x \in \mathcal{V}}{x =_\alpha x}$$

$$\frac{M =_\alpha M' \quad N =_\alpha N'}{MN =_\alpha M'N'}$$

$$\frac{M[x \mapsto y] =_\alpha M' \quad y \notin FV(M)}{\lambda x.M =_\alpha \lambda y.M'}$$

В частности, мы можем рассматривать фактор-множество  $\Lambda /_{=_\alpha}$ .

**Определение.** Пусть на множестве  $A$  задано отношение эквивалентности  $\simeq$ . **Классом эквивалентности** элемента  $a \in A$  назовем подмножество  $[a]_\simeq$ , состоящее из таких  $a' \in A$ , что  $a' \simeq a$ .

**Определение.** **Каноническим представителем** класса эквивалентности  $[a]_\simeq$  назовем произвольный элемент из  $a_0 \in [a]_\simeq$

**Определение.** **Фактор-множеством**  $A /_{\simeq}$  назовем подмножество множества  $A$ , состоящее только из канонических представителей.

Наконец, сформулируем и докажем (неформально) свойство ассоциативности для подстановки – лемму о подстановке. Но перед этим сделаем небольшое замечание касательно доказательств в этой и следующих разделах.

*Замечание.* Если не указано обратного, то все доказательства будут вестись индукцией по структуре терма. Мы будем рассматривать лишь случай абстракции, так как случай переменной легко показывается по определению, а случай аппликации напрямую следует из индукционной гипотезы.

**Утверждение 1.** Для любых  $T, M, N \in \Lambda; x, y \in \mathcal{V}$ , если  $x \neq y$  и  $x \notin FV(M)$ , то верно  $T[x \mapsto N][y \mapsto M] = T[y \mapsto M][x \mapsto N[y \mapsto M]]$

*Доказательство.* База рассуждений – случай, когда  $T$  является переменной. Рассмотрим три случая:

1.  $T \equiv x$ . Левая часть –  $x[x \mapsto N][y \mapsto M] = N[y \mapsto M]$ . Правая часть –  $x[y \mapsto M][x \mapsto N[y \mapsto M]] = N[y \mapsto M]$ .
2.  $T \equiv y$ . Левая часть –  $y[x \mapsto N][y \mapsto M] = M$ . Правая часть –  $y[y \mapsto M][x \mapsto N[y \mapsto M]] = M[x \mapsto N[y \mapsto M]]$ . Так как  $x \notin FV(M)$ , то  $M[x \mapsto N[y \mapsto M]] = M$

3.  $T \equiv z \neq x, y$ . Обе части редуцируются к  $z$ .

Рассмотрим случай абстракции  $\lambda z.T$ . Рассмотрим возможные случаи:

1.  $z \equiv x$ . Левая часть  $-(\lambda x.T)[x \mapsto N][y \mapsto M] = (\lambda x.T)[y \mapsto M] \stackrel{x \notin FV(M)}{=} \lambda x.T[y \mapsto M]$ . Правая часть  $-(\lambda x.T)[y \mapsto M][x \mapsto N[y \mapsto M]] \stackrel{x \notin FV(M)}{=} (\lambda x.T[y \mapsto M])[x \mapsto N[y \mapsto M]] = \lambda x.T[y \mapsto M]$ .
2.  $z \equiv y$ . Пусть  $y \in FV(N)$  и  $y \in FV(M)$ . По соглашению Барендрегта, нам нужно переименовать  $y$  в свежую переменную  $y'$ , такую что  $y' \notin FV(N)$  и  $y' \notin FV(M)$ . Это эквивалентно тому, что мы можем осуществить подстановку  $\lambda y'.t[y \mapsto y']$ . Вычислим левую часть:

$$\begin{aligned} (\lambda y'.T[y \mapsto y'])[x \mapsto N][y \mapsto M] &= \\ \lambda y'.T[y \mapsto y'][x \mapsto N][y \mapsto M] &\stackrel{\text{IH}}{=} \\ \lambda y'.T[y \mapsto y'][y \mapsto M][x \mapsto N[y \mapsto M]] & \end{aligned}$$

Правая часть:

$$\begin{aligned} (\lambda y'.T[y \mapsto y'])[y \mapsto M][x \mapsto N[y \mapsto M]] &= \\ \lambda y'.T[y \mapsto y'][y \mapsto M][x \mapsto N[y \mapsto M]] & \end{aligned}$$

Заметим здесь, что так как  $y' \notin FV(N)$  и  $y' \notin FV(M)$ , то, очевидно, что  $y' \notin FV(N[y \mapsto M])$ , поэтому в последнем переходе обе подстановки «проваливаются» в абстракцию.

3.  $z \equiv y$ . Пусть  $y \in FV(N)$ , но теперь уже  $y \notin FV(M)$ . Аналогично предыдущему пункту, мы переименовываем  $y$  в свежую переменную  $y'$ , такую что  $y' \notin FV(N)$  и  $y' \notin FV(M)$ , осуществляя подстановку  $\lambda y'.t[y \mapsto y']$ . Вычислим левую часть:

$$\begin{aligned} (\lambda y'.T[y \mapsto y'])[x \mapsto N][y \mapsto M] &= \\ \lambda y'.T[y \mapsto y'][x \mapsto N][y \mapsto M] &\stackrel{\text{IH}}{=} \\ \lambda y'.T[y \mapsto y'][y \mapsto M][x \mapsto N[y \mapsto M]] & \end{aligned}$$

А в этом случае, заметим, что так как мы сначала заменили **все** свободные вхождения  $y$  в  $T$  на  $y'$ , а потом на место  $y$  подставили  $M$ , то вторая подстановка ничего не делает, поэтому левая часть окончательно равна  $\lambda y'.T[y \mapsto y'][x \mapsto N[y \mapsto M]]$

Правая часть тогда редуцируется в:

$$\begin{aligned} (\lambda y.T)[y \mapsto M][x \mapsto N[y \mapsto M]] &= \\ (\lambda y.T)[x \mapsto N[y \mapsto M]] &= \\ \lambda y.T[x \mapsto N[y \mapsto M]] & \end{aligned}$$

В последнем шаге вычисления, отметим, что так как  $y \in FV(N)$  и  $y \notin FV(M)$ , то  $y \notin FV(N[y \mapsto M])$ , поэтому подстановка снова заносится под абстракцию. Переименуем и здесь  $y$  в  $y'$ , тогда получим:

$$\begin{aligned} \lambda y.T[x \mapsto N[y \mapsto M]] &=_{\alpha} \\ \lambda y'.T[x \mapsto N[y \mapsto M]][y \mapsto y'] &\stackrel{\text{IH}}{=} \\ \lambda y'.T[y \mapsto y'] [x \mapsto N[y \mapsto M]][y \mapsto y'] & \end{aligned}$$

Из тех же соображений, что и выше, подстановка  $N[y \mapsto M][y \mapsto y']$  – это то же самое, что и  $N[y \mapsto M]$ , поэтому правая часть окончательно вычисляется в  $\lambda y'.T[y \mapsto y'] [x \mapsto N[y \mapsto M]]$ .

4.  $z \neq x, y$ . Доказательство аналогично предыдущим двум пунктам. ■

## 2.2. Неименованное представление термов

Как мы уже видели в предыдущем разделе, имена формальных параметров  $\lambda$ -абстракций не важны и, в целом, мы можем не обращать на них внимания. Более того, мы можем вообще отказаться от именованных переменных. Широко известен альтернативный способ записи термов через так называемые индексы Де Брауна (нидерл. Nicolaas Govert de Bruijn) – [14]. В нем вместо имен переменных используются числовые индексы, показывающие количество абстракций в дереве разбора терма, на которое нужно подняться, чтобы найти ту абстракцию, с которой данная переменная связана. Например комбинатор  $S = \lambda f g x. f x (g x)$ , записанный в таком представлении будет иметь следующий вид:  $\lambda(\lambda(\lambda 3 1(21)))$ .

Существует и альтернативный способ такого представления. Множество всех термов разбивается на так называемые «уровни» (levels) и вместо него рассматриваются множества  $\Lambda_n$ , где  $n$  – длина контекста, в котором определен терм. О контексте в котором определен терм, можно думать, как о простом списке свободных переменных терма (формальное определение контекста дано в разделе 2.4). Индуктивно, они определяются следующим образом:

$$\frac{0 \leq i < n}{v_{n,i} \in \Lambda_n}$$

$$\frac{T_1 \in \Lambda_n \quad T_2 \in \Lambda_n}{T_1 T_2 \in \Lambda_n}$$

$$\frac{T \in \Lambda_{n+1}}{\lambda T \in \Lambda_n}$$

В случае переменной, индекс  $i$  обозначает позицию переменной в контексте. Договоримся отсчитывать ее с конца контекста. Комбинатор  $S$ , например, в таком представлении будет выглядеть вот так:  $\lambda(\lambda(v_{3,2}v_{3,0}(v_{3,1}v_{3,0})))$ .

Такое представление термов удобно потому что проверка двух термов на  $\alpha$ -эквивалентности превращается в проверку двух термов на равенство и, как следствие, пропадает проблема коллизии имен переменных.

Определим операцию подстановки для таких термов. Мы определим ее в более общем случае – вместо какой-то одной переменной мы будем осуществлять подстановку во **все** переменные терма. Пусть  $T \in \Lambda_n$ , и  $S_0, \dots, S_{n-1} \in \Lambda_k$ . Тогда  $\text{subst}(T, S_{n-1}, \dots, S_0) \in \Lambda_k$  определяется следующим образом:

$$\begin{aligned} \text{subst}(v_{n,i}, S_{n-1}, \dots, S_0) &= S_i \\ \text{subst}(T_1 T_2, S_{n-1}, \dots, S_0) &= \text{subst}(T_1, S_{n-1}, \dots, S_0) \text{subst}(T_2, S_{n-1}, \dots, S_0) \\ \text{subst}(\lambda T, S_{n-1}, \dots, S_0) &= \lambda \text{subst}(T, w(S_{n-1}), \dots, w(S_0), v_{n+1,0}) \end{aligned}$$

Операция  $w(T)$  определена следующим образом. Пусть терм  $T \in \Lambda_n$ , тогда терм  $w(T) \in \Lambda_{n+1}$  и определен как:

$$\begin{aligned} w(v_{n,i}) &= v_{n+1,i+1} \\ w(T_1 T_2) &= w(T_1) w(T_2) \\ w(\lambda T) &= \lambda(w(T)) \end{aligned}$$

Сформулируем и докажем вспомогательную лемму, которая пригодится нам далее:

**Лемма 1.** Пусть  $T \in \Lambda_n$ , а  $S_{n-1}, \dots, S_0 \in \Lambda_m$ . Тогда  $\text{subst}(w(T), w(S_{n-1}), \dots, w(S_0), v_{m+1,0}) = w(\text{subst}(T, S_{n-1}, \dots, S_0))$

*Доказательство.* В случае абстракции  $\lambda T$  левая часть вычислится в:

$$\begin{aligned} \text{subst}(w(\lambda T), w(S_{n-1}), \dots, w(S_0), v_{m+1,0}) &= \\ \text{subst}(\lambda w(T), w(S_{n-1}), \dots, w(S_0), v_{m+1,0}) &= \\ \lambda \text{subst}(w(T), w(w(S_{n-1})), \dots, w(w(S_0)), v_{m+2,1}, v_{m+2,0}) &\stackrel{\text{IH}}{=} \\ \lambda w(\text{subst}(T, w(S_{n-1}), \dots, w(S_0), v_{m+1,0})) & \end{aligned}$$

Правая часть вычисляется в:

$$\begin{aligned} w(\text{subst}(\lambda T, S_{n-1}, \dots, S_0)) &= \\ w(\lambda \text{subst}(T, w(S_{n-1}), \dots, w(S_0), v_{m+1,0})) &= \lambda w(\text{subst}(T, w(S_{n-1}), \dots, w(S_0), v_{m+1,0})) \end{aligned}$$

■

Сформулируем и докажем наличие правой единицы у  $subst$ :

**Утверждение 2.** Для любого  $T \in \Lambda_n$  верно, что  $subst(T, v_{n,n-1}, \dots, v_{n,0}) = T$

*Доказательство.*

$$\begin{aligned} subst(\lambda T, v_{n,n-1}, \dots, v_{n,0}) &= \\ \lambda subst(T, w(v_{n,n-1}), \dots, w(v_{n,0}), v_{n+1,0}) &= \\ \lambda subst(T, v_{n+1,n}, \dots, v_{n+1,1}, v_{n+1,0}) &\stackrel{\text{IH}}{=} \\ \lambda T & \end{aligned}$$

■

Аналогично именованному представлению, сформулируем лемму о подстановке:

**Утверждение 3.** Пусть  $T \in \Lambda_n; T_{n-1}, \dots, T_0 \in \Lambda_m, S_{m-1}, \dots, S_0 \in \Lambda_k$ , тогда верно  $subst(subst(T, T_{n-1}, \dots, T_0), S_{m-1}, \dots, S_0) = subst(T, T'_{n-1}, \dots, T'_0)$ , где  $T'_i = subst(T_i, S_{m-1}, \dots, S_0)$ .

Заметим еще, что в таком представлении термов нет необходимости в сторонних условиях, как в лемме о подстановке для именованных термов.

*Доказательство.* Рассмотрим случай абстракции  $\lambda T$ . Вычислим левую часть:

$$\begin{aligned} &subst(subst(\lambda T, T_{n-1}, \dots, T_0), S_{m-1}, \dots, S_0) = \\ &subst(\lambda subst(T, w(T_{n-1}), \dots, w(T_0), v_{m+1,0}), S_{m-1}, \dots, S_0) = \\ &\lambda(subst(subst(T, w(T_{n-1}), \dots, w(T_0), v_{m+1,0}), w(S_{m-1}), \dots, w(S_0), v_{k+1,0}) \stackrel{\text{IH}}{=} \\ &\lambda(subst(T, subst(w(T_{n-1}), w(S_{m-1}), \dots, w(S_0), v_{k+1,0}), \dots, \\ &subst(w(T_0), w(S_{m-1}), \dots, w(S_0), v_{k+1,0}), subst(v_{m+1,0}, w(S_{m-1}), \dots, w(S_0), v_{k+1,0}))) = \\ &\lambda(subst(T, subst(w(T_{n-1}), w(S_{m-1}), \dots, w(S_0), v_{k+1,0}), \dots, \\ &subst(w(T_0), w(S_{m-1}), \dots, w(S_0), v_{k+1,0}), v_{k+1,0})) \end{aligned}$$

Вычислим теперь правую часть:

$$\begin{aligned} &subst(\lambda T, subst(T_{n-1}, S_{m-1}, \dots, S_0), \dots, subst(T_0, S_{m-1}, \dots, S_0)) = \\ &\lambda(subst(T, w(subst(T_{n-1}, S_{m-1}, \dots, S_0)), \dots, w(subst(T_0, S_{m-1}, \dots, S_0)), v_{k+1,0})) \end{aligned}$$

По лемме 1  $subst(w(T_i), w(S_{m-1}), \dots, w(S_0), v_{k+1,0}) = w(subst(T_i, S_{m-1}, \dots, S_0))$  для всех  $i = \overline{0, n-1}$ , следовательно наше утверждение верно. ■



## 2.3. Монадическое представление термов

Существует еще один способ записи  $\lambda$ -термов, описанный в [6]. Библиотека **Bound** [21] для языка **Haskell**, например, использует именно монадическое представление.

Прежде чем продолжить, мы дадим определение функтора и монады. Мы постараемся не перегружать читателя громоздкими определениями из теории категорий и попробуем дать неформальное, но интуитивно понятное определение этим двум объектам.

Чаще всего, имея на руках значение  $v$  некоторого типа  $V$ , мы можем применить к нему функцию  $f : V \rightarrow W$  и получить значение  $f(v) \in W$ . Теперь представим, что на руках у нас не просто значение, но значение в некотором контексте. О контексте, в свою очередь, можно думать как о контейнере, в котором лежит значение (например, как о списке значений или о дереве, в узлах которого содержатся значения<sup>3</sup>). Теперь так просто применить функцию к значению в контексте мы уже не можем, нам нужно научиться каким-то образом извлекать значение из контекста, применять к нему функцию и запаковывать обратно.

Функтор делает именно это. Имея на входе функцию, которую мы хотим применить к значению, и упакованное в контекст значение, он извлекает значение из контекста, применяет к нему функцию и возвращает нам запакованное значение. В языке программирования **Haskell**, в классе типов *Functor* определена операция `fmap`, которая и осуществляет все вышеперечисленное. Эта операция должна удовлетворять двум свойствам, которые еще называют «законами функтора»:

1. Пусть  $fv$  – некоторое значение  $v \in V$  в контексте  $f$ . Тогда  $\text{fmap}(id_V, fv) = fv$ , где  $id_V$  – тождественное отображение на  $V$ .
2. Пусть  $fv$  – некоторое значение  $v \in V$  в контексте  $f$ . Пусть  $f : V \rightarrow W, g : W \rightarrow U$ , тогда  $\text{fmap}(g \circ f, fv) = \text{fmap}(g, \text{fmap}(f, fv))$

Эти два закона говорят нам о том, что операция `fmap` не должна менять «структуру» контейнера. Например, она не должна порождать новых элементов в списке или переподвешивать поддерева в дереве.

Заметим, что функция, которую применяют к упакованному значению функторы, принимает на вход и возвращает неупакованное значение. Монады же, в отличие от функторов, применяют к упакованному значению функцию, которая принимает **неупакованное** значение, а возвращает **упакованное** значение. Так же как и функтор, монада возвращает значение, упакованное в контекст.

---

<sup>3</sup>Пример с контейнером, в общем случае, не самый удачный, но он выбран по причине простоты. Примеры более сложных, вычислительных контекстов, таких как ввод-вывод или работа с состоянием в функциональном стиле не сильно коррелируют с определением лямбда-термов, поэтому специально исключены из рассмотрения.

Аналогично, в языке программирования **Haskell**, в классе типов *Monad* определены две операции – **return** и **>>=** (читается как «bind», мы будем обозначать её соответственно **bind**). Операция **return** называется монадической единицей и принимает на вход значение  $v \in V$  и возвращает упакованное в контекст значение. Операция **bind** называется монадическим связыванием и принимает на вход значение  $mA$  типа  $A$  в контексте  $m$ , функцию  $k : A \rightarrow mB$  и возвращает значение  $mB$  типа  $B$  в контексте  $m$ . Функцию  $k : A \rightarrow mB$  называют еще стрелкой Клейсли [34]. Эти две операции должны удовлетворять трем свойствам, которые носят название «монадических законов»:

1. Пусть  $m$  – некоторое значение в контексте. Тогда  $\text{bind}(m, \text{return}) = m$ .
2. Пусть  $a$  – некоторое неупакованное значение типа  $A$ , а  $k : A \rightarrow mB$ . Тогда  $\text{bind}(\text{return}(a), k) = k(a)$
3. Пусть  $m$  – некоторое запакованное значение. Пусть  $f : A \rightarrow mB$ ,  $g : B \rightarrow mC$ . Тогда  $\text{bind}(\text{bind}(m, f), g) = \text{bind}(m, (x \mapsto \text{bind}(f(x), g)))$ . Здесь и далее  $x \mapsto f(x)$  обозначает анонимную функцию, сопоставляющую конкретному значению  $x$  значение  $f(x)$ .

Основная идея заключается в том, что именованное представление для термов можно обобщить и свободные переменные брать из произвольного множества  $V$ . Тогда множество термов  $\Lambda_V$  определяется индуктивно по следующим правилам вывода:

$$\frac{v \in V}{v \in \Lambda_V}$$

$$\frac{M \in \Lambda_V \quad N \in \Lambda_V}{MN \in \Lambda_V}$$

$$\frac{M \in \Lambda_V \amalg \{*\}}{\lambda M \in \Lambda_V}$$

Здесь  $\{*\}$  – это произвольное одноэлементное множество, а  $\amalg$  – операция размеченного объединения множеств. По определению,  $A \amalg B$  состоит из элементов  $\text{inl}(a)$  и  $\text{inr}(b)$ , где  $a \in A$  и  $b \in B$ . Так как для абстракции нам нужно иметь на одну свободную переменную больше, то ее можно получить, взяв размеченное объединение с произвольным одноэлементным множеством. Это представление еще более удобно для компьютерной реализации, чем описанное в предыдущем разделе, за счет того, что проверку корректности построения термов можно выполнять на уровне типов.

Пусть у нас есть терм  $T \in \Lambda_V$  и функция  $f : V \rightarrow W$ , тогда мы можем задать функцию  $\text{fmap} : \Lambda_V \rightarrow (V \rightarrow W) \rightarrow \Lambda_W$  рекурсивно:

1.  $v \mapsto f(v)$

$$2. M \ N \mapsto \mathbf{fmap}(f, M) \ \mathbf{fmap}(f, N)$$

3.  $\lambda M \mapsto \lambda \mathbf{fmap}(f'(f), M)$ . Заметим, что просто так отобразить терм  $M$  с помощью функции  $f$  мы не можем, так как ее домен не совпадает с множеством, которым параметризован тип терма  $M$ . Поэтому мы построим по  $f$  функцию  $f'(f) : V \coprod \{*\} \rightarrow W \coprod \{*\}$ . Устроена она будет следующим образом:

$$(a) \ f'(f)(\mathit{inl}(x)) = \mathit{inl}(f(x))$$

$$(b) \ f'(f)(\mathit{inr}(*)) = \mathit{inr}(*)$$

Интуитивно, действие операции  $\mathbf{fmap}$  на терм – это переименование переменных в нем. Покажем, что это определение операции  $\mathbf{fmap}$  уважает тождественное отображение и композицию отображений.

**Утверждение 4.** Для любого  $T \in \Lambda_V$  верно, что  $\mathbf{fmap}(\mathit{id}_V, T) = T$

*Доказательство.* Покажем, что утверждение верно и для случая лямбда-абстракции. Вспомогательная функция  $f'(f)$  устроена следующим образом:

$$1. \ f'(\mathit{id}_V)(\mathit{inl}(x)) = \mathit{inl}(x)$$

$$2. \ f'(\mathit{id}_V)(\mathit{inr}(*)) = \mathit{inr}(x)$$

Следовательно, оно является тождеством на  $V \coprod \{*\}$ . По индукционной гипотезе получаем, что случай для лямбды тоже верен.

Формальное доказательство этого утверждения можно увидеть в приложении A, в функции `fmap-respects-id` ■

**Утверждение 5.** Для любого  $T \in \Lambda_V$  и  $f : V \rightarrow W$ ,  $g : W \rightarrow X$  верно, что  $\mathbf{fmap}(g \circ f, T) = \mathbf{fmap}(f, \mathbf{fmap}(g, T))$

*Доказательство.* Рассмотрим случай абстракции  $\lambda M$ . Посмотрим, во что вычислится левая часть:

$$\mathbf{fmap}(g \circ f, \lambda M) = \lambda \mathbf{fmap}(f'(g \circ f), M)$$

где

$$f'(g \circ f)(\mathit{inl}(v)) = \mathit{inl}(g(f(v)))$$

$$f'(g \circ f)(\mathit{inr}(*)) = \mathit{inr}(*)$$

Правая, в свою очередь:

$$\begin{aligned} \mathbf{fmap}(g, \mathbf{fmap}(f, \lambda M)) &= \mathbf{fmap}(g, \lambda \mathbf{fmap}(f'(f), M)) = \\ &\lambda \mathbf{fmap}(g'(g), \mathbf{fmap}(f'(f), M)) \end{aligned}$$

Вспомогательная функция  $f'(f) : V \coprod \{*\} \rightarrow W \coprod \{*\}$  устроена здесь следующим образом:

1.  $f'(f)(inl(v)) = inl(f(v))$
2.  $f'(f)(inr(*)) = inr(*)$

Вспомогательная функция  $g'(g) : W \amalg \{*\} \rightarrow X \amalg \{*\}$  устроена здесь следующим образом:

1.  $g'(g)(inl(w)) = g'(g)(inl(f(v))) = inl(g(f(v)))$
2.  $g'(g)(inr(*)) = inr(*)$

Несложно увидеть, что эти две функции из левой и правой частей ведут себя одинаково, следовательно и для случая абстракции утверждение верно. Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `fmap-respects-comp` ■

Мы убедились, что такое определение термов функториально. Зададим теперь структуру монады. Монадическая единица `return` – это переменная. Связывание `bind` определяется рекурсией по структуре терма  $T$ :

1.  $v \mapsto k(v)$
2.  $M \ N \mapsto (\text{bind}(M, k)) (\text{bind}(N, k))$
3.  $\lambda M \mapsto \lambda(\text{bind}(M, k'(k)))$ , где  $k'(k) : V \amalg \{*\} \rightarrow \Lambda_W \amalg \{*\}$  и определяется следующим образом:
  - (a)  $k'(k)(inl(v)) = \text{fmap}(inl, k(v))$
  - (b)  $k'(k)(inr(*)) = \text{return}(inr(*))$

Действие монадического связывания можно, в свою очередь, проинтерпретировать как подстановку. Это утверждение не столь очевидно, но если рассмотреть сигнатуру операции `bind` и обратить внимание на то, что функцию  $k : V \rightarrow \Lambda_W$  можно задать в виде списка пар  $(v, T)$ , где  $v \in V$ ,  $T \in \Lambda_W$ , то это соответствие становится куда более явным. Монадные законы, в свою очередь, в точности описывают свойства подстановки, которые мы в явном виде задали в прошлых разделах 2.1 и 2.2 – унитарность и ассоциативность.

Сформулируем и докажем теперь монадические законы для термов.

**Утверждение 6.** Для любого  $T \in \Lambda_V$  верно  $\text{bind}(T, \text{return}) = T$ .

*Доказательство.* Индукция по структуре терма  $T$ :

1. База индукции,  $T = v$ . Имеем, что  $\text{bind}(v, \text{return}) = \text{return}(v) = v$ .
2. Случай аппликации напрямую следует из предположения индукции.

3. Пусть теперь  $T = \lambda M$ . Имеем, что  $\text{bind}(\lambda M, \text{return}) = \lambda(\text{bind}(M, k'(\text{return})))$ .  
Вспомогательное отображение  $k'(\text{return})$  устроено следующим образом:

- (a)  $k'(\text{return})(\text{inl}(v)) = \text{fmap}(\text{inl}, \text{return}(v)) = \text{return}(\text{inl}(v))$
- (b)  $k'(\text{return})(\text{inr}(*)) = \text{return}(\text{inr}(*))$

Заметим, что  $k'(\text{return})$  ведет себя так же, как и  $\text{return}$  на  $V \amalg \{*\}$ , следовательно по индукционной гипотезе получаем исходное утверждение. Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `bind-right-unit`. ■

**Утверждение 7.** Для любого  $v \in V$  и  $k : V \rightarrow \Lambda_V$  верно  $\text{bind}(\text{return}(v), k) = k(v)$

*Доказательство.* Утверждение следует из определения `bind` и того факта, что `return` – это переменная. Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `bind-left-unit`. ■

Прежде, чем формулировать последний монадический закон, сформулируем и докажем несколько технических лемм, которые помогут нам в его доказательстве.

**Лемма 2.** Для любых  $T \in \Lambda_V$ ,  $f : V \rightarrow W$  и  $k : W \rightarrow \Lambda_U$  верно  $\text{bind}(\text{fmap}(f, T), k) = \text{bind}(T, k \circ f)$ .

*Доказательство.* В случае абстракции  $\lambda M$ , нам нужно показать, что  $\lambda \text{bind}(\text{fmap}(f'(f), M), k') = \lambda \text{bind}(M, k'(k \circ f))$ . По индукционной гипотезе мы знаем, что  $\text{bind}(\text{fmap}(f'(f), M), k'(k)) = \text{bind}(M, k'(k) \circ f'(f))$ . Заметим теперь, что  $k'(k \circ f)$  и  $k'(k) \circ f'(f)$  ведут себя одинаково на всех входах, следовательно это утверждение доказано. Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `bind-fmap-comm-lhs`. ■

**Лемма 3.** Для любых  $T \in \Lambda_V$ ,  $f : W \rightarrow U$  и  $k : V \rightarrow \Lambda_W$  верно  $\text{fmap}(f, \text{bind}(T, k)) = \text{bind}(T, (x \mapsto \text{fmap}(f, k(x))))$ .

*Доказательство.* Здесь в случае абстракции  $\lambda M$  нам нужно показать, что  $\lambda \text{fmap}(f'(f), \text{bind}(M, k')) = \lambda \text{bind}(M, k'(x \mapsto \text{fmap}(f, k(x))))$ . По индукционной гипотезе мы знаем, что  $\text{fmap}(f'(f), \text{bind}(M, k')) = \text{bind}(M, (x \mapsto \text{fmap}(f'(f), k'(k)(x))))$ . Покажем теперь, что стрелки Клейсли  $k'(x \mapsto \text{fmap}(f, k(x)))$  и  $x \mapsto \text{fmap}(f'(f), k'(k)(x))$  ведут себя одинаково на всех входах:

1.  $\text{inr}(*)$ . Обе стрелки вычисляются в  $\text{return}(\text{inr}(*))$
2.  $\text{inl}(v)$ . Надо показать, что  $\text{fmap}(f'(\text{inl}), \text{fmap}(f, k(v))) = \text{fmap}(f'(f), \text{fmap}(\text{inl}, k(v)))$ .

Так как  $\Lambda_V$  – функтор и он уважает композицию отображений имеем, что нужно показать  $\text{fmap}(f'(\text{inl}) \circ f, k(v)) = \text{fmap}(f'(f) \circ \text{inl}, k(v))$ . Для этого в свою очередь нужно снова показать, что два отображения  $f'(\text{inl}) \circ f$  и  $f'(f) \circ \text{inl}$  ведут себя одинаково на всех входах, но это напрямую следует из определения  $f'$ .

Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `bind-fmap-comm-rhs`. ■

**Лемма 4.** Для любого  $T \in \Lambda_V$  и  $f : V \rightarrow \Lambda_W$  верно  $\text{bind}(\text{fmap}(\text{inl}, t), k'(g)) = \text{fmap}(\text{inl}, \text{bind}(T, f))$ .

*Доказательство.* Рассмотрим случай абстракции  $\lambda M$ .

Левая часть вычислится в:

$$\lambda \text{fmap}(f'(\text{inl}), \text{bind}(M, k'(f)))$$

Правая в:

$$\lambda \text{bind}(\text{fmap}(f'(\text{inl}), M), k'(k'(f)))$$

По лемме 2 имеем, что  $\text{bind}(\text{fmap}(f'(\text{inl}), M), k'(k'(f))) = \text{bind}(M, k'(k'(f)) \circ f'(\text{inl}))$ . По лемме 3 имеем  $\text{fmap}(f'(\text{inl}), \text{bind}(M, k'(f))) = \text{bind}(M, (x \mapsto \text{fmap}(f'(\text{inl}), k'(f)(x))))$ . Заметим теперь, что стрелки Клейсли  $k'(k'(f)) \circ f'(\text{inl})$  и  $x \mapsto \text{fmap}(f'(\text{inl}), k'(f)(x))$  ведут себя одинаково на всех входах, тогда по симметричности и транзитивности равенства получаем доказательство требуемого утверждения.

Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `bind-fmap-comm`. ■

Наконец, сформулируем аналог леммы о подстановке для именованного представления термов.

**Утверждение 8.** Для любого  $T \in \Lambda_V$ ,  $f : V \rightarrow \Lambda_W$ ,  $g : W \rightarrow \Lambda_U$  верно  $\text{bind}(\text{bind}(T, f), g) = \text{bind}(T, (x \mapsto \text{bind}(f(x), g)))$ .

*Доказательство.* Индукция по структуре терма  $T$ :

1. База индукции  $T = v$ . Тогда левая часть вычисляется в  $\text{bind}(f(v), g)$ , ровно как и правая.
2. Случай для аппликации следует напрямую из индукционной гипотезы.
3. Рассмотрим случай абстракции  $\lambda M$ . Посмотрим, во что вычисляется левая часть:

$$\begin{aligned} \text{bind}(\text{bind}(\lambda M, f), g) &= \text{bind}(\lambda \text{bind}(M, k'(f)), g) = \\ &\quad \lambda \text{bind}(\text{bind}(M, k'(f)), k'(g)) \end{aligned}$$

где

$$k'(g)(\text{inl}(w)) = \text{fmap}(\text{inl}, g(w))$$

$$k'(g)(\text{inr}(*)) = \text{return}(\text{inr}(*))$$

и

$$k'(f)(\text{inl}(v)) = \text{fmap}(\text{inl}, f(v))$$

$$k'(f)(\text{inr}(*)) = \text{return}(\text{inr}(*))$$

Правая часть, в свою очередь, вычисляется в:

$$\text{bind}(\lambda M, (x \mapsto \text{bind}(f(x), g))) = \lambda \text{bind}(M, k'(x \mapsto \text{bind}(f(x), g)))$$

Чтобы воспользоваться индукционной гипотезой, необходимо показать, что  $k'(x \mapsto \text{bind}(f(x), g)) : V \amalg \{*\} \rightarrow \Lambda_U \amalg \{*\}$  ведет себя так же как и  $x \mapsto \text{bind}(k'(f)(x), k'(g))$ . Для этого мы просто покажем, что они возвращают одинаковый результат на всех входах.

Рассмотрим два случая, как могут выглядеть входные данные:

- (a)  $\text{inr}(*)$ . Обе части вычисляются в  $\text{return}(\text{inr}(*))$ .
- (b)  $\text{inl}(v)$ . Левая часть вычисляется в:

$$\text{fmap}(\text{inl}, \text{bind}(f(v), g))$$

Правая:

$$\text{bind}(\text{fmap}(\text{inl}, f(v)), f'(g))$$

Воспользовавшись леммой 4 для терма  $f(v)$  и  $g$  получаем доказательство исходного утверждения. ■

Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `bind-assoc`.

## 2.4. Преобразования между представлениями

В этом разделе мы опишем преобразования между представлениями и начнем с преобразования именованных термов в неименованные. Очевидно, что для осуществления этого нам необходимо знать порядок на переменных в терме. Поэтому мы считаем, что кроме самого терма нам дают контекст.

**Определение.** Контекст  $\Gamma$  – это не содержащий дубликатов список переменных  $x_1, \dots, x_n$ ,  $x_i \in \mathcal{V}$ .

**Определение.** Терм  $T$  определен в контексте  $\Gamma$  тогда и только тогда, когда все свободные переменные терма  $T$  присутствуют в  $\Gamma$ . Этот факт традиционно обозначается как  $\Gamma \vdash T$ .

Итак, преобразование  $\Phi$  именованных термов в неименованные принимает на вход контекст  $\Gamma = x_1, \dots, x_n$ , терм  $T \in \Lambda$  такой что он определен в контексте  $\Gamma$  и возвращает неименованный терм  $T' \in \Lambda_n$ . Определяется оно индукцией по структуре терма  $T$ :

1.  $x_1, \dots, x_n \vdash x_i \mapsto v_{n, n-i}$

2.  $x_1, \dots, x_n \vdash MN \mapsto \Phi(x_1, \dots, x_n \vdash M) \Phi(x_1, \dots, x_n \vdash N)$
3.  $x_1, \dots, x_n \vdash \lambda x.M \mapsto \lambda \Phi(x_1, \dots, x_n, x \vdash M)$

Покажем, что такое преобразование уважает отношение  $\alpha$ -эквивалентности, введенное в разделе 2.1.

**Утверждение 9.** Пусть  $T_1, T_2 \in \Lambda$ ,  $\Gamma \vdash T_1$ ,  $\Gamma \vdash T_2$  и  $T_1 =_\alpha T_2$ . Тогда  $\Phi(\Gamma \vdash T_1) = \Phi(\Gamma \vdash T_2)$ .

*Доказательство.* Одновременная индукция по структуре термов  $T_1$  и  $T_2$ . Так как мы знаем, что они  $\alpha$ -эквивалентны, то нам нет необходимости рассматривать всевозможные комбинации термов. Поэтому рассмотрим лишь случаи, когда термы имеют общую структуру (две переменные, две аппликации или две абстракции).

1. База индукции.  $T_1 = x$ ,  $T_2 = y$ , так как они альфа-эквивалентны, то  $x \equiv y$ , а так как они определены в одинаковом контексте, то и стоят на одинаковых позициях, по определению  $\Phi$ , получаем, что база индукции верна.
2. Рассмотрим случай, когда  $T_1 = \lambda x.M$ ,  $T_2 = \lambda y.M'$ . Так как они  $\alpha$ -эквивалентны, то мы знаем, что  $M' =_\alpha M[x \mapsto y]$  и  $y \notin FV(M)$ . Мы знаем, что  $\Gamma \vdash \lambda x.M$  и  $\Gamma \vdash \lambda y.M'$ , следовательно  $\Gamma, x \vdash M$  и  $\Gamma, y \vdash M'$ . Кроме того, так как  $y \notin FV(M)$ , то  $\Gamma, y \vdash M[x \mapsto y]$ . По посылке из правила альфа-эквивалентности для абстракций мы знаем, что  $M' =_\alpha M[x \mapsto y]$ , следовательно по индукционной гипотезе  $\Phi(\Gamma, y \vdash M') = \Phi(\Gamma, y \vdash M[x \mapsto y])$ . Осталось заметить, что  $\Phi(\Gamma, y \vdash M[x \mapsto y]) = \Phi(\Gamma, x \vdash M)$  и получить доказательство исходного утверждения. ■

Сконструируем теперь преобразование в обратную сторону – из неименованных термов в именованные. Нам хотелось бы, чтобы оно принимало на вход терм  $T' \in \Lambda_n$  и возвращало пару из контекста  $\Gamma$  и терма  $T \in \Lambda$ , определенного в нем. Рассмотрим три случая, как бы мы могли задать это преобразование, назовем его  $\Psi$ .

1. В случае, когда  $T' = v_{n,i}$  мы можем просто сгенерировать  $n$  именованных переменных  $\Gamma = x_1, \dots, x_n$  и в качестве результата вернуть пару из контекста  $\Gamma$  и  $x_{n-i}$ . То есть:

$$v_{n,i} \mapsto x_1, \dots, x_n \vdash x_{n-i}$$

2. В случае аппликации  $M N$  кажется, что все еще проще. Так как она определена в том же контексте, что и оба аппликанта, то нам достаточно пары рекурсивных вызовов и мы можем взять любой контекст в окончательный результат. То есть:

$$M N \mapsto \pi_1(\Psi(M)) \vdash \pi_2(\Psi(M)) \pi_2(\Psi(N))$$



Здесь и далее  $\pi_1$  и  $\pi_2$  – первая и вторая проекция для пар соответственно. Но здесь нас поджидает неприятный момент, связанный с тем, что рекурсивные вызовы, возвращают нам два *различных* контекста, в которых определены оба аппликанта. Поэтому формально, мы не можем так определить преобразование из неименованных термов в именованные.

Для того, что бы корректно определить  $\Psi$ , заметим, что нам вообще-то не важно, в каком контексте будет определен результирующий терм. Мы знаем его длину, следовательно, мы можем сгенерировать его и подать на вход обратному преобразованию. Тогда оно вернет нам именованный терм, определенный в данном контексте. Корректное определение  $\Psi$ , выглядит следующим образом:

1.  $\Gamma, v_{n,i} \mapsto \Gamma_{n-i}$
2.  $\Gamma, MN \mapsto \Psi(\Gamma, M) \Psi(\Gamma, N)$
3.  $\Gamma, \lambda M \mapsto \lambda x' \Psi(\Gamma; x', M)$

В последнем случае, переменная  $x'$  выбирается «свежей», в том смысле, что и в разделе 2.1. Запись  $\Gamma; x'$  обозначает расширение контекста  $\Gamma$ , путем дописывания в его конец, переменной  $x'$ .

Легко заметить, что эти два преобразования взаимно-обратны. Однако необходимо оговориться, что мы установили биекцию между множеством пар из именованных термов и их контекстов.

Сконструируем преобразования между неименованными и монадическими термами. Для примера покажем преобразование  $\Delta$  из  $\Lambda_n$  в  $\Lambda_{\bar{n}}$ , где  $\bar{n}$  – это множество  $\{0, 1, \dots, n-1\}$ . Действительно, рассмотрим три случая:

1.  $v_{n,i}$ . Так как  $0 \leq i < n$ , то  $i$  и так лежит в  $\Lambda_{\bar{n}}$

$$v_{n,i} \mapsto i$$

2.  $M N$ . Вызываемся рекурсивно от обеих частей:

$$M N \mapsto \Delta(M) \Delta(N)$$

3.  $\lambda M$ . В этом случае придется воспользоваться тем, что  $\Lambda_{\overline{n+1}}$  является функтором. Сначала вызовем рекурсивно от  $M$  и получим терм, лежащий в  $\Lambda_{\overline{n+1}}$ . А затем отобразим его в  $\Lambda_{\bar{n}} \amalg \{*\}$  следующим образом:

$$f : \overline{n+1} \rightarrow \bar{n} \amalg \{*\}$$

$$0 \mapsto inr(*)$$

$$i \mapsto inl(i-1)$$

Окончательно:

$$\lambda M \mapsto \lambda F_f(\Delta(M))$$

Аналогичным образом конструируется преобразование в обратную сторону  $\Theta : \Lambda_{\bar{n}} \rightarrow \Lambda_n$ :

1.  $i \mapsto v_{n,i}$
2.  $M \ N \mapsto \Theta(M) \ \Theta(N)$
3.  $\lambda M \mapsto \lambda \Theta(F_{f^{-1}}(M))$  где:

$$\begin{aligned} f^{-1} : \bar{n} \coprod \{*\} &\rightarrow \overline{n+1} \\ inl(i) &\mapsto i+1 \\ inr(*) &\mapsto 0 \end{aligned}$$

Эти преобразования также взаимно-обратны, мы не будем приводить доказательство этого факта, полагая его очевидным.

### 3. Особенности реализации

В этом разделе мы опишем язык, с помощью которого формализовывались все представления. Для каждого представления мы опишем тонкие моменты, с которыми пришлось иметь дело в ходе выполнения работы. Для начала мы опишем как формализовывались неименованное и монадическое представление и закончим описанием трудностей, с которыми мы столкнулись при формализации именованного представления.

#### 3.1. Описание языка Vclang

Для реализации всех описанных во второй главе представлений был выбран язык **Vclang**. В этом подразделе мы кратко опишем его отличия от известных систем автоматического доказательства теорем, например **Agda**, которые привели к тому, что именно **Vclang** был выбран в качестве инструмента.

Язык **Vclang** разрабатывается в компании JetBrains с февраля 2015 года в рамках исследования в группе гомотопической теории типов и зависимых типов<sup>4</sup>. Он представляет собой функциональный язык программирования, основанный на гомотопической теории типов с типом интервала. Компилятор языка написан полностью на **Java**, что позволяет полностью использовать все преимущества платформы **JVM**, как-то кроссплатформенность, легкость в инструментировании и возможность разрабатывать модули на языках программирования, совместимых с платформой **JVM**(например на **Scala** или **Kotlin**).

**Vclang** позволяет определять и пользоваться многими знакомыми конструкциями, как то:

1. Индуктивные типы данных(как и в упрощенном синтаксисе, так и в синтаксисе обобщенных алгебраических типов данных)
2. Сопоставление с образцом
3. Метаварiables
4. Неявные аргументы функций
5. Проверка на завершаемость(termination checking)

Последний пункт означает примерно следующее. Так как задача проверки типа в такой системе должна быть разрешима, то все вычисления в типах должны завершаться. Это, в свою очередь, означает, что все функции, которые определяет программист должны завершаться на всех входах, то есть быть всюду определенными.

---

<sup>4</sup><https://research.jetbrains.org/ru/groups/group-for-dependent-types-and-hott>

Проверка на завершаемость и означает, что среди прочих особенностей, в языке есть компонент, который проверяет, действительно ли все функции завершаются на всех входах. Это приводит к тому, что конструкции приходится определять так, чтобы вычислителю было очевидно, что они завершаются. При этом с точки зрения программиста, эти конструкции из простых и понятных могут превратиться в неочевидные.

С точки зрения синтаксиса, **Vclang** имеет общие черты с **Agda** или **Haskell**, кроме нескольких особенностей. Во-первых, ключевые слова языка начинаются с обратной косой черты (`\data`, `\function` и так далее), во-вторых, **Vclang** не использует Unicode-обозначения и миксфиксные операторы, как например это делает **Agda**. Ниже приведен пример определения типа данных списков фиксированной длины (векторов):

---

```

1  \data Vector (n : Nat) (A : \Type)
2    | Vector zero A => vnil
3    | Vector (suc n) A => \infixr 5 (:~) A (Vector n A)

```

---

Листинг 1: Пример определения типа данных списков фиксированной длины в **Vclang**

Конструкторы данных и случаи при сопоставлении с образцом (как в определениях функций, так и в конструкции `\case`) отделяются прямой чертой `|`. Вместо знака равенства используется символ `=>`. Для сопоставления с образцом используется конструкция `\elim` и символ `<=`. Например, определение функции, берущей первый элемент в векторе выглядит следующим образом:

---

```

1  \function
2  head {n : Nat} {A : \Type0} (xs : Vector (suc n) A) : A <= \elim xs
3    | (:~) x _ => x

```

---

Листинг 2: Пример определения функции взятия первого элемента в векторе в **Vclang**

Больше примеров можно посмотреть в стандартной библиотеке языка, которая доступна по адресу <https://github.com/valis/vclang-lib>

Главное же отличие **Vclang** состоит в том, как в нем устроено равенство. Напомним, что в **Agda** тип-равенство определяется как тип с единственным конструктором:

---

```

1  infix 1 _≡_
2  data _≡_ {A : Set} (a : A) : A → Set where
3    refl : a ≡ a

```

---

Листинг 3: Определение типа-равенства в **Agda**

О типе  $a \equiv b$  можно в таком случае думать, как об утверждении, что  $a$  равен  $b$ . Доказательством этого утверждения будет терм `refl`. Так как `refl` является конструктором, то мы можем использовать сопоставление с образцом. Таким образом можно, например, доказывать свойства равенства:

---

```

1  sym : {A : Set} {a a' : A} → a ≡ a' → a' ≡ a
2  sym refl = refl

```

---

#### Листинг 4: Доказательство симметричности равенства в **Agda**

В **Vclang** равенство определяется с помощью так называемых «путей». Формально эта теория вводится в статье [20], здесь же мы просто скажем, что тип-равенство определяется как функция над специальным типом  $I$  (от слова «interval»). Все свойства равенства (рефлексивность, симметричность и т.д), соответственно, определяются как функции над путями. Похожим образом равенство определяется в кубической теории типов, описанной в работе [12].

Вторым отличием **Vclang** являются типы данных с условиями. Поясним на небольшом примере. Пусть у нас есть тип натуральных чисел `Nat`. Мы хотим определить тип целых чисел, как тип с двумя конструкторами: первый описывает положительные числа, второй – отрицательные.

---

```

1  \data Int
2    | pos Nat
3    | neg Nat

```

---

#### Листинг 5: Тип целых чисел. Вариант 1

У этого определения есть одна проблема – мы получили два нуля. Один со знаком плюс, второй – со знаком минус. Эту проблему очень легко решить. **Vclang** позволяет написать условие, которое говорит, как вычисляются конструкторы. Например:

---

```

1  \data Int
2    | pos Nat
3    | neg Nat
4  \with
5    | neg zero => pos zero

```

---

#### Листинг 6: Тип целых чисел с условием

Ключевое слово `\with` означает, что выражение, стоящее слева от знака `=>` вычисляется в выражение, стоящее справа.

В языках программирования существуют различные конструкции, которые в том или ином роде моделируют фактор-множества. Самый простой способ сделать это – переопределить операцию сравнения на типе. В системе автоматического доказательства теорем **Agda** аналогом фактор-множеств являются так называемые *сетойды* – множества с введенным на них отношением эквивалентности. Технически, они устроены, как записи из подлежащего типа и отношения. В **Isabelle** также есть фактор-типы, но устроены они несколько иным, довольно запутанным образом.

В **Coq** есть некоторая поддержка фактор-типов, реализованная, согласно работе [9] с помощью библиотеки-расширения **SSReflect**. Однако, есть и более ранняя работа [7], в которой авторы упоминают о расширении теории типов Calculus Of Constructions фактор-типами и показывают, что такое расширение, если реализовать его в **Coq** противоречиво.

В нашем случае, можно обойтись и формализовывать именованное представление без фактор-типов и использовать, например, сетойды, но тогда вместо доказательства равенства типов(эквивалентности между различными представлениями) нам придется вводить и доказывать отношение эквивалентности сетойдов. Отношение равенства мы считаем более предпочтительным по многим причинам, в частности, для него верен принцип Лейбница(принцип подстановки эквивалентных).

Можно возразить, что в **Agda** и **Coq** так же есть поддержка гомотопической теории типов, но **Vclang** не просто ее поддерживает – он построен на ней, именно поэтому мы выбрали его в качестве инструмента для реализации.

Тип интервала в **Vclang** и типы данных с условиями дают нам возможность элегантно описывать фактор-множества:

---

```

1   \truncated \data Quotient (A : \Set) (R : A -> A -> \Prop) : \Set
2   | classEq A
3   | quotEq (a a' : A) (R a a') I
4   \with
5   | quotEq a a' p left  => classEq a
6   | quotEq a a' p right => classEq a'
```

---

Листинг 7: Тип фактор-множества  $A/R$

Первый конструктор «вкладывает» элемент  $A$  в класс эквивалентности  $[a]_R$ , а второй – склеивает вместе элементы, которые находятся в отношении  $R$ .

### 3.2. Реализация неименованного представления термов

Как уже упоминалось в разделах 1.1 и 2.2, неименованное представление удобно для компьютерной реализации, так как, в частности, проверка термов на  $\alpha$ -эквивалентность

сводится к обычной проверке на равенство. Ниже приведен пример определения типа данных для термов в данном представлении:

---

```

1      -- I is for index
2      \data ITerm (n : Nat)
3        | IVar (Fin n)
4        | IApp (ITerm n) (ITerm n)
5        | ILam (ITerm (suc n))

```

---

Листинг 8: Тип данных, кодирующий термы в неименованном представлении.

Здесь  $\text{Fin } n$  – это тип конечных множеств на  $n$  элементов. О них можно думать как о множествах  $\bar{n} = \{0, 1, \dots, n - 1\}$ .

Покажем, например, как определяется преобразование неименованного терма в именованный.

---

```

1      \function
2      psi'
3      {n : Nat}
4      (env : Ctx n)
5      (t : ITerm n) : \Sigma (t' : NTerm) (p : TermInCtx env t') <=
6        ↪ \elim t
7        | IVar i      => (NVar (env !! i), VarInCtx (i, idp))
8        | IApp t1 t2   => (NApp (psi' env t1).1 (psi' env t2).1,
9          ↪ AppInCtx (psi' env t1).2 (psi' env t2).2)
10       | ILam t       => (NLam n (psi' (extend env n) t).1, LamInCtx
11         ↪ (psi' (extend env n) t).2)
12
13     \function
14     psi
15     {n : Nat}
16     (t : ITerm n) : \Sigma (env : Ctx n) (t' : NTerm) (proof :
17       ↪ TermInCtx env t') => (gen_env n, (psi' (gen_env n) t).1, (psi'
18         ↪ (gen_env n) t).2)

```

---

Листинг 9: Определение функции, переводящей неименованный терм в именованный.

### 3.3. Реализация монадического представления термов

Термы в монадическом представлении кодируются следующим типом данных:

---

```
1  -- F is for functor.
2  \data FTerm (V : \Set)
3      | FVar V
4      | FApp (FTerm V) (FTerm V)
5      | FLam (FTerm (Either V Unit))
```

---

Листинг 10: Тип данных, кодирующий термы в монадическом представлении.

Для абстракции можно воспользоваться типом `Maybe V`. Типы `Either V Unit` и `Maybe V` очень похожи между собой, более того, они изоморфны.

Для реализации монадического связывания можно заметить, что сигнатуру `bind` можно обобщить. Для краткости будем обозначать тип `Either V Unit` как  $V + 1$ . Если мы определим вспомогательную функцию  $\text{bind}' : FTerm(V + n) \rightarrow (V \rightarrow FTerm W) \rightarrow FTerm(W + n)$ , то `bind`, очевидно, будет определен, как её результат при  $n = 0$ . Её можно было бы определить примерно следующим образом:

---

```
1  \function
2  bind'
3      {V W : \Type}
4      (n : Nat)
5      (t : FTerm (Telescope n V))
6      (k : V -> FTerm W) : FTerm (Telescope n W) <= \elim t
7          | FVar x      => twistT n (fmap-telescope n k x)
8          | FApp t1 t2  => FApp (bind' n t1 k) (bind' n t2 k)
9          | FLam t      => FLam (bind' (suc n) t k)
10
11
12  \function
13  bind
14      {V W : \Type}
15      (t : FTerm V)
16      (k : V -> FTerm W) : FTerm W => bind' zero t k
```

---

Листинг 11: Один из вариантов определения `bind`.

Здесь `Telescope n V` – это тип  $V + n$ , определяемый рекурсивно следующим образом:



---

```

1  \function
2  Telescope
3      (n : Nat)
4      (V : \Type) : \Type <= \elim n
5          | zero  => V
6          | suc n => Either (Telescope n V) Unit

```

---

Листинг 12: Тип  $V + n$

Функции `fmap-telescope` и `twistT` являются в некотором роде аналогами `fmap` и `sequenceA` из класса типов `Traversable` в **Haskell** [17] для типа  $V + n$ . Проблема такого определения, что про него очень неудобно доказывать свойство ассоциативности `bind`. Поэтому от этого определения было решено отказаться в пользу определения, приведенного в разделе 2.3.

Второй тонкий момент связан с реализацией преобразования из монадических термов в неименованные. Мы могли бы определить его следующим образом:

---

```

1  \function
2  delta
3      {n : Nat}
4      (t : FTerm (Fin n)) : ITerm n <= \elim t
5          | FVar i      => IVar i
6          | FApp t1 t2   => IApp (delta t1) (delta t2)
7          | FLam t       => ILam (delta (fmap (\lam e => \case e | inl
              ↪ i => fsuc i | inr unit => fzero) t))

```

---

Листинг 13: Вариант определения преобразования монадического терма в неименованный.

Однако в случае абстракции становится неочевидно, что этот код завершается. Для этого, надо пояснить, как работает проверка на завершаемость в **Vclang**. Нестро-го говоря, вычислитель следит за тем, что бы при каждом вызове функции, хотя бы один из аргументов «убывал». Под словом «убывание» мы подразумеваем снятие конструктора с аргумента. Тогда, если среди всех вызовов этой функции есть аргумент, с которого снимаются конструкторы, то мы считаем, что эта функция завершается. Здесь же, мы рекурсивно вызываемся не от аргумента `t`, а от аргумента `fmap (\lam e => \case e | inl i => fsuc i | inr unit => fzero) t`, что для вычислителя равносильно тому, что аргумент, наоборот, «возрастает».

Поэтому в качестве примера определим тип `Fun n` очень похожий на `Fin n` (на самом деле – они изоморфны друг другу):

---

```

1  \function
2  Fun
3      (n : Nat) : \Type <= \elim n
4      | zero  => Empty
5      | suc n => Either (Fun n) Unit

```

---

Листинг 14: Тип Fun n.

Тогда преобразование в него определяется довольно просто:

---

```

1  \function
2  delta
3      {n : Nat}
4      (t : FTerm (Fun n)) : ITerm n <= \elim t
5      | FVar f          => IVar (to-fin f)
6      | FApp t1 t2      => IApp (delta t1) (delta t2)
7      | FLam t          => ILam (delta {suc n} t)

```

---

Листинг 15: Работающий вариант определения преобразования монадического терма в неименованный.

### 3.4. Реализация именованного представления термов

Термы в этом представлении кодируются следующим типом данных:

---

```

1  -- N is for named
2  \data NTerm
3      | NVar Nat
4      | NApp NTerm NTerm
5      | NLam Nat NTerm

```

---

Листинг 16: Тип данных, кодирующий термы в именованном представлении.

Сразу же отметим, что в имена переменных мы берем из множества натуральных чисел. Мы делаем это по ряду причин. Во-первых, в языке бедная стандартная библиотека и символов в ней нет. Во-вторых, в описании именованного представления в разделе 2.1 мы неявно предполагаем, что на множестве  $\mathcal{V}$  равенство разрешимо, а для натуральных чисел в стандартной библиотеке языка есть доказательство этого факта. В дальнейшем, мы пользуемся им для определения, например, операции подстановки.

Альфа-эквивалентность определяется как следующий индуктивный предикат:

---

```

1  \data AlphaEq (t1 t2 : NTerm)
2    | AlphaEq (NVar x) (NVar y)          => VarEq (x = y)
3    | AlphaEq (NApp t1 s1) (NApp t2 s2)  => AppEq (AlphaEq t1 t2)
      ↪ (AlphaEq s1 s2)
4    | AlphaEq (NLam x t1) (NLam y t2)    => LamEq (b : Nat) (b != x) (b
      ↪ != y) (b # t1) (b # t2) (AlphaEq (nsubst-var t1 x b) (nsubst-var
      ↪ t2 y b))

```

---

Листинг 17: Определение альфа-эквивалентности.

Обратим внимание на случай абстракции. Можно было бы определить его проще, как в разделе 2.1, но то определение имеет ряд недостатков. Предположим, что мы хотели бы доказать рефлексивность альфа-эквивалентности. В случае абстракции нам бы понадобилось свойство унитальности для подстановки. Но в доказательстве свойства унитальности так же используется рефлексивность. Получается своеобразная цепочка взаимно-рекурсивных определений, завершаемость которой крайне неочевидна для языка.

Здесь предикат  $x \# t$  означает, что переменная  $x$  является «свежей» для терма  $t$ , то есть не встречается в списке переменных этого терма. Случай для абстракции означает примерно следующее – две лямбды альфа-эквивалентны, если есть переменная  $b$ , отличная от  $x$  и  $y$ , свежая в термах  $t1$  и  $t2$ , такая что если подставить ее в тела в  $t1$  и  $t2$ , то они будут альфа-эквивалентны.

Для преобразования именованных термов в неименованные мы в явном виде определили предикат «терм определен в контексте»:

---

```

1  \function
2  VarInCtx {n : Nat} (env : Ctx n) (x : Nat) => \Sigma (i : Fin n) (p :
      ↪ env !! i = x)
3
4  \data TermInCtx {n : Nat} (env : Ctx n) (t : NTerm)
5    | TermInCtx {n} env (NVar x)          => Variable (p : VarInCtx env x)
6    | TermInCtx {n} env (NApp t1 t2)      => Application (p : TermInCtx env
      ↪ t1) (q : TermInCtx env t2)
7    | TermInCtx {n} env (NLam x t)        => Abstraction (p : TermInCtx
      ↪ (extend env x) t)

```

---

Листинг 18: Предикат «терм определен в контексте».

Имея его, можно определить отношение альфа-эквивалентности еще одним образом – на тройках «контекст, терм и доказательство того, что терм определен в

контексте»:

---

```
1  \data AlphaEq
2      {n : Nat}
3      (env1 : Ctx n)
4      (t1 : NTerm)
5      (p1 : TermInCtx env1 t1)
6      (env2 : Ctx n)
7      (t2 : NTerm)
8      (p2 : TermInCtx env2 t2)
9      | AlphaEq env1 (NVar x) (Variable p) env2 (NVar y) (Variable
10         ↪ q)                                     => VarEq (p : p.1 = q.1)
11     | AlphaEq env1 (NApp t1 s1) (Application p1 q1) env2 (NApp t2
        ↪ s2) (Application p2 q2) => AppEq (p : AlphaEq env1 t1 p1
        ↪ env2 t2 p2) (q : AlphaEq env1 s1 q1 env2 s2 q2)
        | AlphaEq env1 (NLam x t1) (Abstraction p1) env2 (NLam y t2)
        ↪ (Abstraction p2)          => LamEq (p : AlphaEq (extend
        ↪ env1 x) t1 p1 (extend env2 y) t2 p2)
```

---

Листинг 19: Еще один вариант определения альфа-эквивалентности.

О контексте здесь можно думать, как о связывании переменных, тогда очень легко показать, что такое отношение сохраняется при преобразовании именованного терма в неименованный.

Операция подстановки могла бы быть определена следующим образом:

---

```
1  \function
2  nsubst
3      (t : NTerm)
4      (y : Nat)
5      (s : NTerm) : NTerm <= \elim t
6      | NVar x      => \case x =? y | inr _ => s | inl _ => (NVar x)
7      | NApp t1 t2 => NApp (nsubst t1 y s) (nsubst t2 y s)
8      | NLam x t   => \let
9                      | frv => gen-fresh-var t
10                     \in \case isFreeDec x s | inr _ => (NLam frv
        ↪ (nsubst (nsubst t x (NVar frv)) y s)) |
        ↪ inl _ => (NLam x (nsubst t y s))
```

---

Листинг 20: Один из вариантов определения операции подстановки.

К сожалению этот вариант не проходит проверку на завершаемость в случае лямбда-абстракции. Отметим, что и в разделе 2.1 в доказательстве леммы о подстановке мы весьма небрежно использовали индукционную гипотезу. Для неформальных рассуждений это еще допустимо, но для строго формальных – уже нет. Чтобы решить эту проблему, мы определили более общий случай подстановки – параллельную подстановку:

---

```

1  \function
2  nsubst'
3      (t : NTerm)
4      (ps : List (\Sigma Nat NTerm)) : NTerm <= \elim t
5          | NVar x          => lookup ps x
6          | NApp t1 t2      => NApp (nsbst' t1 ps) (nsbst' t2 ps)
7          | NLam x t        => \let
8                                  | ts => (NLam x t) :-: (map (\lam (p :
9                                      ↪ \Sigma Nat NTerm) => p.2) ps)
10                                 | x' => gen-fresh-var ts
11                                 \in NLam x' (nsbst' t ((x, NVar x') :-:
12                                     ↪ ps))
13
14 \function
15 nsubst
16     (t : NTerm)
17     (y : Nat)
18     (s : NTerm) : NTerm => nsubst' t (singleton (y, s))

```

---

Листинг 21: Вариант определения операции подстановки с помощью параллельной подстановки.

Вспомогательная функция `nsbst'` принимает на вход помимо терма уже не просто переменную и подставляемый терм, а список пар, из переменных и термов, которые необходимо подставить на их место. Этот вариант уже, очевидно, проходит проверку на завершаемость. К сожалению, из-за технических проблем в самом языке этот вариант определения подстановки появился ближе к концу процесса выполнения работы, поэтому мы не успели доказать про него лемму о подстановке в случае абстракции. Один из способов решить эту и ряд вышеперечисленных проблем – использовать идеи, аналогичные тем, что применяются в номинальных множествах – [27].

## Заключение

В ходе выполнения работы была рассмотрена задача доказательства равенства между различными представлениями лямбда-термов. Для ее решения мы использовали экспериментальный язык, построенный на новой теории типов, которая, в частности, позволяет описывать некоторые конструкции гораздо более удобным образом, нежели, например, **Agda**.

С помощью этого языка мы реализовали различные представления термов: именованное, неименованное и монадическое. Мы реализовали фундаментальную операцию подстановки и сформулировали её свойства – унитарность и ассоциативность. Мы полностью доказали эти свойства для неименованного и монадических представлений. Для именованного представления небольшая часть доказательств не была доведена до конца в связи с техническими проблемами в самом языке. Помимо этого, мы построили преобразования между этими представлениями и доказали, что они взаимно-обратны, тем самым, показав равенство. Однако следует сделать замечание, что так как для именованного представления термов нам нужен еще и контекст, мы показали эквивалентность между парами из термов и контекстов и неименованными термами.

Один из вариантов развития работы, который может помочь в решении проблем, возникших при формализации именованного представления – это номинальные множества. Мы оставим его, как направление развития.

Исходный код опубликован в публичном Git-репозитории и доступен по ссылке [https://github.com/edgarzhavoronkov/vclang-lib/tree/lambda\\_calculus/test/LC](https://github.com/edgarzhavoronkov/vclang-lib/tree/lambda_calculus/test/LC)

## Список литературы

- [1] Altenkirch Thorsten. A formalization of the strong normalization proof for System F in LEGO // Typed Lambda Calculi and Applications. — 1993. — P. 13–28.
- [2] Altenkirch Thorsten, Reus Bernhard. Monadic presentations of lambda terms using generalized inductive types // Computer science logic / Springer. — 1999. — P. 825–825.
- [3] Barendregt HP. Lambda calculi with types, Handbook of logic in computer science (vol. 2): background: computational structures. — 1993.
- [4] Barras Bruno. Coq en coq : Ph. D. thesis / Bruno Barras ; INRIA. — 1996.
- [5] Berghofer Stefan, Urban Christian. A head-to-head comparison of de Bruijn indices and names // Electronic Notes in Theoretical Computer Science. — 2007. — Vol. 174, no. 5. — P. 53–67.
- [6] Bird Richard S, Paterson Ross. De Bruijn notation as a nested datatype // Journal of functional programming. — 1999. — Vol. 9, no. 01. — P. 77–91.
- [7] Chicli Laurent, Pottier Loïc, Simpson Carlos. Mathematical quotients and quotient types in Coq // International Workshop on Types for Proofs and Programs / Springer. — 2002. — P. 95–107.
- [8] Church Alonzo. An unsolvable problem of elementary number theory // American journal of mathematics. — 1936. — Vol. 58, no. 2. — P. 345–363.
- [9] Cohen Cyril. Pragmatic quotient types in Coq // International Conference on Interactive Theorem Proving / Springer. — 2013. — P. 213–228.
- [10] Coquand Thierry. An algorithm for type-checking dependent types // Science of Computer Programming. — 1996. — Vol. 26, no. 1-3. — P. 167–177.
- [11] Coquand Thierry, Huet Gérard. The calculus of constructions // Information and computation. — 1988. — Vol. 76, no. 2-3. — P. 95–120.
- [12] Cubical type theory: a constructive interpretation of the univalence axiom / Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg // arXiv preprint arXiv:1611.02108. — 2016.
- [13] Curry Haskell B. Grundlagen der kombinatorischen Logik // American journal of mathematics. — 1930. — Vol. 52, no. 4. — P. 789–834.

- [14] De Bruijn Nicolaas Govert. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem // *Indagationes Mathematicae (Proceedings)* / Elsevier. — Vol. 75. — 1972. — P. 381–392.
- [15] Gabbay Murdoch, Pitts Andrew. A new approach to abstract syntax involving binders // *Logic in Computer Science, 1999. Proceedings. 14th Symposium on / IEEE*. — 1999. — P. 214–224.
- [16] Gordon Andrew, Melham Tom. Five axioms of alpha-conversion // *Theorem proving in higher order logics*. — 1996. — P. 173–190.
- [17] Hackage. Data.Traversable. — Access mode: <https://goo.gl/JZ2xfa> (online; accessed: 2017-06-02).
- [18] Howard William A. The formulae-as-types notion of construction // *To HB Curry: essays on combinatory logic, lambda calculus and formalism*. — 1980. — Vol. 44. — P. 479–490.
- [19] Huet Gérard. Residual theory in  $\lambda$ -calculus: A formal development // *Journal of Functional Programming*. — 1994. — Vol. 4, no. 03. — P. 371–394.
- [20] Isaev Valery. Model Structures on Categories of Models of Type Theories // *arXiv preprint arXiv:1607.07407*. — 2016.
- [21] Kmett Edward A. "bound: Making de Bruijn Succ Less". — Access mode: <https://goo.gl/TKhGDn> (online; accessed: 2017-05-27).
- [22] Landin Peter J. The mechanical evaluation of expressions // *The Computer Journal*. — 1964. — Vol. 6, no. 4. — P. 308–320.
- [23] Martin-Löf Per. An intuitionistic theory of types: Predicative part // *Studies in Logic and the Foundations of Mathematics*. — 1975. — Vol. 80. — P. 73–118.
- [24] McKinna James, Pollack Robert. Some lambda calculus and type theory formalized // *Journal of Automated Reasoning*. — 1999. — Vol. 23, no. 3. — P. 373–409.
- [25] Nipkow Tobias. More Church-Rosser proofs (in isabelle/hol) // *Automated Deduction—CADE-13*. — 1996. — P. 733–747.
- [26] Norell Ulf. Towards a practical programming language based on dependent type theory. — Citeseer, 2007. — Vol. 32.
- [27] Pitts Andrew M. Nominal sets: Names and symmetry in computer science. — Cambridge University Press, 2013.



- [28] Sakaguchi Kazuhiko. A Formalization of Typed and Untyped  $\lambda$ -Calculi in SSReflect-Coq and Agda2. — Access mode: <https://goo.gl/WtcdZE> (online; accessed: 2017-05-23).
- [29] Sato Masahiko. Theory of symbolic expressions, I // Theoretical Computer Science. — 1983. — Vol. 22, no. 1-2. — P. 19–55.
- [30] Schönfinkel Moses. Über die Bausteine der mathematischen Logik // Mathematische Annalen. — 1924. — Vol. 92, no. 3. — P. 305–316.
- [31] Shankar Natarajan. A mechanical proof of the Church-Rosser theorem // Journal of the ACM (JACM). — 1988. — Vol. 35, no. 3. — P. 475–522.
- [32] Stoughton Allen. Substitution revisited // Theoretical Computer Science. — 1988. — Vol. 59, no. 3. — P. 317–325.
- [33] Tarau Paul. On logic programming representations of lambda terms: de Bruijn indices, compression, type inference, combinatorial generation, normalization // International Symposium on Practical Aspects of Declarative Languages / Springer. — 2015. — P. 115–131.
- [34] Wiki Haskell. Arrow tutorial. — Access mode: <https://goo.gl/bIiCsy> (online; accessed: 2017-05-29).

## А. Пример формализации монадического представления термов

В этом приложении приведен пример кода на языке Vclang, формализующий монадическое представление термов, описанное в разделе 2.3

```
\open ::Combinators
\open ::Data::Either
\open ::Data::Nat::Base
\open ::Data::Nat::Compare
\open ::Data::Unit
\open ::Paths

-- Term definition

\data FTerm (V : \Set)
  | FVar V
  | FApp (FTerm V) (FTerm V)
  | FLam (FTerm (Either V Unit))

-- Functor structure

\function
fmap
  {V W : \Set}
  (f : V -> W)
  (t : FTerm V) : FTerm W <= \elim t
    | FVar v      => FVar (f v)
    | FApp t1 t2  => FApp (fmap f t1) (fmap f t2)
    | FLam t      => FLam (fmap (map-inl f) t)

-- Monad structure

\function
return
  {V : \Set}
  (v : V) : FTerm V => FVar v

\function
bind-fun-helper
  {V W : \Set}
  (k : V -> FTerm W)
  (e : Either V Unit) : FTerm (Either W Unit) <= \elim e
    | inl v      => fmap {W} {Either W Unit} inl (k v)
    | inr unit   => FVar (inr unit)

\function
bind
  {V W : \Set}
  (t : FTerm V)
  (k : V -> FTerm W) : FTerm W <= \elim t
    | FVar x      => k x
```

```

| FApp t1 t2    => FApp (t1 `bind` k) (t2 `bind` k)
| FLam t        => FLam (t `bind` (bind-fun-helper k))

-- Functor laws and related

\function
map-inl-respects-id
  {L R : \Set}
  (e : Either L R) : map-inl (\lam x => x) e = e <= \elim e
    | inl l => idp
    | inr r => idp

\function
map-inl-respects-comp
  {A B C R : \Set}
  (f : A -> B)
  (g : B -> C)
  (e : Either A R) : map-inl g (map-inl f e) = map-inl (g `o` f) e <= \elim e
    | inl l => idp
    | inr r => idp

-- Утверждение 4 из раздела 2.3
\function
fmap-respects-id
  {V : \Set}
  (t : FTerm V) : fmap (\lam x => x) t = t <= \elim t
    | FVar v => idp
    | FApp t1 t2 => pmap2 FApp (fmap-respects-id t1) (fmap-respects-id t2)
    | FLam t => \let
      | rec => fmap-respects-id t
      | fext => funExt (\lam _ => Either V Unit) (\lam x => x) (map-inl (\lam x => x)) (\lam
        ↪ e => inv (map-inl-respects-id e))
      | rect => transport (\lam x => fmap {Either V Unit} {Either V Unit} x t = t) fext rec
    \in pmap FLam rect

-- Утверждение 5 из раздела 2.3
\function
fmap-respects-comp
  {A B C : \Set}
  (f : A -> B)
  (g : B -> C)
  (t : FTerm A) : fmap g (fmap f t) = fmap (g `o` f) t <= \elim t
    | FVar v => idp
    | FApp t1 t2 => pmap2 FApp (fmap-respects-comp f g t1) (fmap-respects-comp f g t2)
    | FLam t => \let
      | rec => fmap-respects-comp (map-inl {A} {B} {Unit} f) (map-inl {B} {C} {Unit} g) t
      | fext => funExt (\lam _ => Either C Unit) (\lam x => map-inl {B} {C} {Unit} g (map-inl
        ↪ {A} {B} f x)) (map-inl (g `o` f)) (map-inl-respects-comp f g)
      | trrec => transport (\lam x => fmap (map-inl {B} {C} {Unit} g) (fmap (\lam (e : Either
        ↪ A Unit) => map-inl {A} {B} {Unit} f e) t) = fmap x t) fext rec
    \in pmap FLam trrec

-- Helper lemmata

```

```

\function
return-right-unit-funext-helper
  {V : \Set}
  (e : Either V Unit) : return e = (bind-fun-helper return) e <= \elim e
    | inl v => idp
    | inr unit => idp

\function
bind-fun-helper-fmap-comm
  {A B : \Set}
  (f : A -> FTerm B)
  (x : Either A Unit) : bind-fun-helper (bind-fun-helper f) (map-inl inl x) = fmap (map-inl inl)
    ↪ (bind-fun-helper f x) <= \elim x
      | inl a      => inv (bind-fmap-comm-rhs-fext-helper-lemma (f a) (inl))
      | inr unit   => idp

\function
bind-fmap-comm-rhs-fext-helper-lemma-fext-helper
  {B C : \Set}
  (f : B -> C)
  (x : Either B Unit) : map-inl {Either B Unit} {Either C Unit} {Unit} (\lam (e : Either B Unit) => map-inl
    ↪ {B} {C} {Unit} f e) (map-inl {B} {Either B Unit} {Unit} inl x) = map-inl {C} {Either C Unit} {Unit}
    ↪ inl (map-inl {B} {C} {Unit} f x) <= \elim x
      | inl b => idp
      | inr unit => idp

\function
bind-fmap-comm-rhs-fext-helper-lemma
  {B C : \Set}
  (t : FTerm B)
  (f : B -> C) : fmap (map-inl {B} {C} {Unit} f) (fmap {B} {Either B Unit} inl t) = fmap {C} {Either C Unit}
    ↪ inl (fmap f t) <= \elim t
      | FVar b      => idp
      | FApp t1 t2  => pmap2 FApp (bind-fmap-comm-rhs-fext-helper-lemma t1 f)
        ↪ (bind-fmap-comm-rhs-fext-helper-lemma t2 f)
      | FLam t      => \let
        | rec      => bind-fmap-comm-rhs-fext-helper-lemma t (map-inl f)
        | comp1 => fmap-respects-comp (map-inl {_} {_} {Unit} inl) (map-inl {_} {_}
          ↪ {Unit} (map-inl {B} {C} {Unit} f)) t
        | comp2 => fmap-respects-comp (map-inl {B} {C} {Unit} f) (map-inl {C} {Either C
          ↪ Unit} {Unit} inl) t
        | fext  => funExt (\lam _ => Either (Either C Unit) Unit) (\lam x => map-inl
          ↪ {Either B Unit} {Either C Unit} {Unit} (map-inl {B} {C} {Unit} f)
          ↪ (map-inl {B} {Either B Unit} {Unit} inl x)) (\lam x => map-inl {C}
          ↪ {Either C Unit} {Unit} inl (map-inl {B} {C} {Unit} f x))
          ↪ (bind-fmap-comm-rhs-fext-helper-lemma-fext-helper f)
        | trans => comp1 *> (pmap (\lam x => fmap x t) fext) *> (inv comp2)
    \in pmap FLam trans

\function
bind-fmap-comm-rhs-fext-helper
  {A B C : \Set}
  (f : B -> C)

```

```

(k : A -> FTerm B)
(x : Either A Unit) : fmap (map-inl {B} {C} {Unit} f) (bind-fun-helper k x) = bind-fun-helper (\lam y =>
  ↪  fmap f (k y)) x <= \elim x
  | inl a => bind-fmap-comm-rhs-fext-helper-lemma (k a) f
  | inr unit => idp

-- Лемма 3 из раздела 2.3
\function
bind-fmap-comm-rhs
  {A B C : \Set}
  (t : FTerm A)
  (f : B -> C)
  (k : A -> FTerm B) : fmap f (t `bind` k) = t `bind` (\lam x => fmap f (k x)) <= \elim t
    | FVar a      => idp
    | FApp t1 t2  => pmap2 FApp (bind-fmap-comm-rhs t1 f k) (bind-fmap-comm-rhs t2 f k)
    | FLam t      => \let
      | rec => bind-fmap-comm-rhs t (map-inl {B} {C} {Unit} f) (bind-fun-helper k)
      | fext => funExt (\lam _ => FTerm (Either C Unit)) (\lam x => fmap (map-inl {B}
        ↪  {C} {Unit} f) (bind-fun-helper k x)) (\lam e => bind-fun-helper (\lam x
        ↪  => fmap f (k x)) e) (bind-fmap-comm-rhs-fext-helper f k)
      | trrec => transport (\lam u => fmap (map-inl {B} {C} {Unit} f) (bind t
        ↪  (bind-fun-helper k)) = u) (pmap (bind t) fext) rec
    \in pmap FLam trrec

\function
bind-fmap-comm-lhs-fext-helper
  {A B C : \Set}
  (f : A -> B)
  (k : B -> FTerm C)
  (x : Either A Unit) : bind-fun-helper k (map-inl {A} {B} {Unit} f x) = bind-fun-helper (\lam y => k (f y))
    ↪  x <= \elim x
    | inl a => idp
    | inr unit => idp

-- Лемма 2 из раздела 2.3
\function
bind-fmap-comm-lhs
  {A B C : \Set}
  (t : FTerm A)
  (f : A -> B)
  (k : B -> FTerm C) : fmap f t `bind` k = t `bind` (k `o` f) <= \elim t
    | FVar a      => idp
    | FApp t1 t2  => pmap2 FApp (bind-fmap-comm-lhs t1 f k) (bind-fmap-comm-lhs t2 f k)
    | FLam t      => \let
      | rec => bind-fmap-comm-lhs t (map-inl {A} {B} {Unit} f) (bind-fun-helper k)
      | fext => funExt (\lam _ => FTerm (Either C Unit)) (\lam x => bind-fun-helper k
        ↪  (map-inl {A} {B} {Unit} f x)) (\lam e => bind-fun-helper (\lam x => k (f
        ↪  x)) e) (bind-fmap-comm-lhs-fext-helper f k)
      | trrec => transport (\lam u => (fmap (map-inl {A} {B} {Unit} f) t) `bind`
        ↪  (bind-fun-helper k) = u) (pmap (bind t) fext) rec
    \in pmap FLam trrec

-- Лемма 4 из раздела 2.3
\function
bind-fmap-comm
  {A B : \Set}

```

```

(t : FTerm A)
(f : A -> FTerm B) : bind (fmap {A} {Either A Unit} inl t) (bind-fun-helper f) = fmap {B} {Either B Unit}
  ↪ inl (bind t f) <= \elim t
    | FVar v      => idp
    | FApp t1 t2  => pmap2 FApp (bind-fmap-comm t1 f) (bind-fmap-comm t2 f)
    | FLam t      => \let
      | lhs => bind-fmap-comm-lhs t (map-inl inl) (bind-fun-helper (bind-fun-helper
        ↪ f))
      | rhs => bind-fmap-comm-rhs t (map-inl inl) (bind-fun-helper f)
      | fext => funExt (\lam _ => FTerm (Either (Either B Unit) Unit)) (\lam x =>
        ↪ bind-fun-helper (bind-fun-helper f) (map-inl inl x)) (\lam x => fmap
        ↪ (map-inl inl) (bind-fun-helper f x)) (bind-fun-helper-fmap-comm f)
      | trans => lhs *> (pmap (bind t) fext) *> (inv rhs)
    \in pmap FLam trans

\function
bind-assoc-funext-helper
  {A B C : \Set}
  (f : A -> FTerm B)
  (g : B -> FTerm C)
  (x : Either A Unit) : bind ((bind-fun-helper f) x) (bind-fun-helper g) = bind-fun-helper (\lam a => bind (f
    ↪ a) g) x <= \elim x
    | inl a      => bind-fmap-comm (f a) g
    | inr unit   => idp

-- Monad laws
-- Утверждение 6 из раздела 2.3
\function
return-right-unit
  {V : \Set}
  (t : FTerm V) : t `bind` return = t <= \elim t
    | FVar v      => idp
    | FApp t1 t2  => pmap2 FApp (return-right-unit t1) (return-right-unit t2)
    | FLam t      => \let
      | rec      => return-right-unit t
      | funext   => funExt (\lam _ => FTerm (Either V Unit)) return (bind-fun-helper
        ↪ return) return-right-unit-funext-helper
      | trrec    => transport (\lam x => bind t x = t) funext rec
    \in pmap FLam trrec

-- Утверждение 7 из раздела 2.3
\function
return-left-unit
  {V W : \Set}
  (x : V)
  (k : V -> FTerm W) : ((return x) `bind` k = k x) => idp

-- Утверждение 8 из раздела 2.3
\function
bind-assoc
  {A B C : \Set}
  (f : A -> FTerm B)
  (g : B -> FTerm C)
  (t : FTerm A) : (t `bind` f) `bind` g = t `bind` (\lam x => (f x) `bind` g) <= \elim t
    | FVar v      => idp

```

```

| FApp t1 t2    => pmap2 FApp (bind-assoc f g t1) (bind-assoc f g t2)
| FLam t        => \let
    | f'      => bind-fun-helper f
    | g'      => bind-fun-helper g
    | rec     => bind-assoc f' g' t
    | fext    => funExt (\lam _ => FTerm (Either C Unit)) (\lam x => (f' x) `bind`
        ↪      g') (bind-fun-helper (\lam x => (f x) `bind` g))
        ↪      (bind-assoc-funext-helper f g)
    | trrec   => transport (\lam x => (t `bind` f') `bind` g' = t `bind` x) fext rec
\in pmap FLam trrec

```