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 λ-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

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 связывание имен, то в программах на них зачастую возникают проблема коллизий в именах переменных, функций и других конструкций.

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 α-эквивалентности – эквивалентности в поведении конструкций с разными именами формальных параметров. Пример такой системы – это λ-исчисление, лежащее в основе функциональных языков программирования. Вычислительная семантика лямбда-исчисления обеспечивается посредством операции подстановки, которая обладает некоторыми базовыми свойствами.

Существуют различные представления этой системы, по-разному решающие описанные выше проблемы. Например, в именованном представлении у переменных есть имена и нам нужно рассматривать термы с точностью до имен параметров функций. Есть неименованное представление, в котором вместо имен переменных используются их позиции в контексте, или расстояние до конструкции, в которой произошло связывание. Есть и более общее представление – монадическое, когда термы наделяются структурой монады, известного объекта в теории категорий. Свойства монады в таком представлении являются аналогами свойств термов в других представлениях.

В работе делается попытка формализовать различные представления этой системы и понять, с каким из них наиболее удобно работать.

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

Во второй главе будут описаны три представления λ-термов. Для каждого из представлений мы определим характерные свойства и покажем, почему они верны. Кроме того, мы покажем, что эти представления эквивалентны между собой.

В третьей главе предполагается описание деталей реализации. Мы опишем язык, с помощью которого предлагается формализовать рассмотренные во второй главе представления а так же тонкие моменты, с которыми пришлось столкнуться в ходе выполнения работы.

1. Описание предметной области

1.1. λ-исчисление

Историческая справка Лямбда-исчисление – это формальная система, придуманная в 30-ых годах прошлого века Алонзо Черчём [6] с целью анализа и формализации понятия вычислимости. В 60-ых годах Питером Ландином была опубликована работа [19], в которой выдвигалась идея о том, что λ-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 λ-исчисления широко используются в мире функционального программирования.

Неформальное описание λ-термов Мы формально определим λ-термы во второй главе, здесь же мы просто скажем, что термы λ-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, λ-исчисление является очень мощной формальной системой, в частности, Шейнфинкелем и Карри в работах [27, 10] введен в рассмотрение базис из двух термов(комбинаторов) $S = \lambda fgx.fx(gx)$ и $K = \lambda xy.x$, 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 λ-термов состоит в том, что являясь каким-никаким, но 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 – [23].

1.2. Теории типов

Изначально, в λ-исчислении не вводилось никаких правил типизации, однако в дальнейшем появилось множество типизированных вариаций. Барендрегтом в [3] описан так называемый λ-куб, который наглядно классифицирует восемь различных систем типизации лямбда-исчисления.

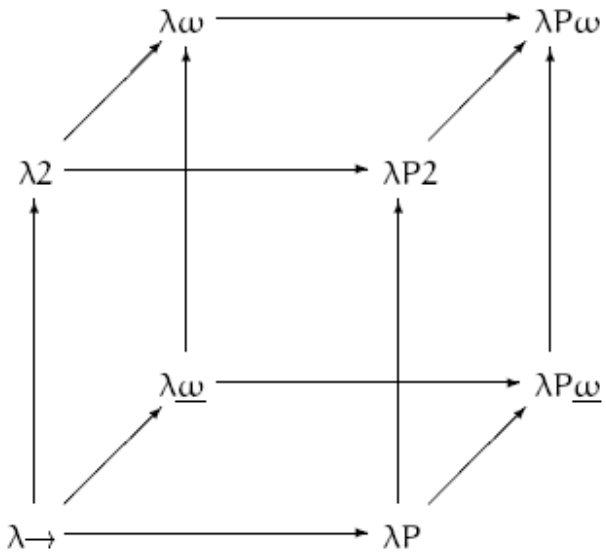

Рис. 1: Лямбда-куб

База куба – просто типизированное λ-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 $\lambda\underline{\omega}$(операторы над типами)
3. Типы, которые зависят от термов – система λP(зависимые типы)

1.3. Соответствие Карри-Говарда

Соответствие Карри-Говарда [15] 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 λ-терм соответствующего типа, чтобы доказать исходное утверждение. Для наглядности некоторые соответствия сведены в таблицу:

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

Существует две известные теории с зависимыми типами – исчисление конструкций(Calculus Of Constructions), представленное Тьерри Коканом в [8] и интуиционистская теория типов Мартин-Лёфа(Martin-Löf Type Theory), описанная в [20]. 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 это.

1.4. Постановка задачи

Целью работы является формализация различных представлений λ-термов. Для достижения этой цели необходимо решить следующие задачи:

1. Определить интересующие нас представления лямбда-термов:
   (a) Именованное
   (b) Неименованное
   (c) Монадическое

2. Для каждого представления определить операцию подстановки

3. Для каждого представления доказать свойства операции подстановки:

   (a) Унитальность(наличие правой и левой единицы)

   (b) Ассоциативность

4. Установить соответствие между описанными в первом пункте представлениями

Операция подстановки является фундаментальной операцией над термами. С её помощью вводятся отношения, которые наделяют λ-исчисление вычислительной семантикой, например β-редукция. Унитальность и ассоциативность – её основные свойства, поэтому мы хотим формализовать именно эту операцию.

1.5. Существующие решения

Стоит отметить, что задача формализации λ-исчисления довольно популярна, в связи с чем существует довольно много её решений. Один из примеров – [25], в котором, в частности, формализовано чистое λ-исчисление. Для него, как и для работ [28, 1, 4, 22, 16] характерно использование неименованного представления термов через индексы Де Брауна. Работы [21, 7, 12, 13, 26, 29] 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. Мы, в свою очередь, хотим формализовать *различные представления* чистого лямбда-исчисления и установить между ними соответствие.

Более того, наше решение будет использовать отличную от других теорию типов. В разделе 3 мы подробнее опишем язык, построенный на основе этой теории и опишем, чем она отличается от других известных теорий. Здесь же мы просто скажем, что в этом языке есть конструкции, которых нет в других языках программирования с зависимыми типами, либо ими сложно пользоваться(речь идет, например, о фактор-типах).

2. Представления λ-термов

В этой главе мы опишем три представления λ-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 соответствие между всеми представлениями.

2.1. Именованное представление термов

Термы λ-исчисления(λ-термы) в именованном представлении конструируются из переменных путем применения 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 λ-термов Λ определяется индуктивно, согласно следующим правилам:

$$\frac{v \in \mathcal{V}}{v \in \Lambda}$$

$$\frac{M \in \Lambda \qquad N \in \Lambda}{MN \in \Lambda}$$

$$\frac{M \in \Lambda \qquad v \in \mathcal{V}}{\lambda v.M \in \Lambda}$$

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(i.e 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:

$$\lambda x.x$$

$$\lambda xy.x$$

$$(\lambda x.f(xx))(\lambda x.f(xx))$$

Переменная x после абстракции $\lambda x.M$ называется *связанной*. Соответственно, до абстракции она была *свободной*. Формально, множества $FV(T)$ свободных и $BV(T)$

связанных переменных терма T определяются индуктивно следующим образом:

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

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

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:

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

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. Проблема захвата переменной состоит в том, что при подстановке в λ-абстракцию свободные переменные подставляемого терма могут оказаться захваченными абстракцией.

Эту проблему можно решить если принять соглашение Барендрегта о том, что имена связанных переменных всегда выбирать так, чтобы они отличались от имен свободных. В примере выше, например, мы можем переименовать связанную переменную y в свежую z и поведении абстракции $\lambda z.x$ не изменится. Тогда подстановку можно использовать без каких-либо оговорок о свободных и связанных переменных. Пример выше превратится в:

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

Как мы уже установили выше, мы можем переименовывать связанные переменные в абстракциях и их поведение при применении к аргументам не изменится. Более того, имена связанных переменных не играют для нас никакой роли. Поэтому, как правило,

λ-термы и рассматривают с точностью до имен параметров абстракций. Здесь мы поступим точно так же. Формально, на множестве именованных термов Λ можно задать отношение $=_\alpha \in \Lambda \times \Lambda$, которое называется α-эквивалентностью и определяется как минимальное отношение конгруэнтности, порожденное следующими правилами:

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

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

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

В частности, мы можем рассматривать фактор-множество $\Lambda/_{=_\alpha}$. Наконец, сформулируем и докажем(неформально) свойство ассоциативности для подстановки – лемму о подстановке.

***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. Индукция по терму T. База индукции – 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.

Случай аппликации тривиален, рассмотрим случай абстракции $\lambda z.T$. Рассмотрим возможные случаи:

1. $z \equiv x$. Левая часть – $(\lambda x.T)[x \mapsto N][y \mapsto M] = (\lambda x.T)[y \mapsto M] \overset{x \notin FV(M)}{=} \lambda x.T[y \mapsto M]$. Правая часть – $(\lambda x.T)[y \mapsto M][x \mapsto N[y \mapsto M]] \overset{x \notin FV(M)}{=} (\lambda x.T[y \mapsto M])[x \mapsto N[y \mapsto M]] = \lambda x.T[y \mapsto M]$.

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:

$$(\lambda y'.T[y \mapsto y'])[x \mapsto N][y \mapsto M] =$$
$$\lambda y'.T[y \mapsto y'][x \mapsto N][y \mapsto M] \overset{\text{IH}}{=}$$
$$\lambda y'.T[y \mapsto y'][y \mapsto M][x \mapsto N[y \mapsto M]]$$

Правая часть:

$$(\lambda y'.T[y \mapsto y'])[y \mapsto M][x \mapsto N[y \mapsto M]] =$$
$$\lambda y'.T[y \mapsto y'][y \mapsto M][x \mapsto N[y \mapsto M]]$$

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:

$$(\lambda y'.T[y \mapsto y'])[x \mapsto N][y \mapsto M] =$$
$$\lambda y'.T[y \mapsto y'][x \mapsto N][y \mapsto M] \overset{\text{IH}}{=}$$
$$\lambda y'.T[y \mapsto y'][y \mapsto M][x \mapsto N[y \mapsto M]]$$

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:

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

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:

$$\lambda y.T[x \mapsto N[y \mapsto M]] =_\alpha$$
$$\lambda y'.T[x \mapsto N[y \mapsto M]][y \mapsto y'] \overset{\text{IH}}{=}$$
$$\lambda y'.T[y \mapsto y'][x \mapsto N[y \mapsto M][y \mapsto y']]$$

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 уже мы уже видели в предыдущем разделе, имена формальных параметров λ-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(De Bruijn) – [11]. В нем вместо имен переменных используются числовые индексы, показывающие сколько лямбд назад была захвачена переменная. Например комбинатор $S = \lambda fgx.fx(gx)$, записанный в таком представлении будет иметь следующий вид: $\lambda(\lambda(\lambda 31(21)))$.

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

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

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

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

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: $\lambda(\lambda(\lambda v_{3,2} v_{3,0}(v_{3,1} v_{3,0})))$.

Такое представление термов удобно потому что α-эквивалентность сводится к самому обычному равенству 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 $S_0, \ldots S_{n-1} \in \Lambda_k$. Тогда $subst(T, S_{n-1}, \ldots, S_0) \in \Lambda_k$ определяется следующим образом:

$$subst(v_{n,i}, S_{n-1}, \ldots, S_0) = S_i$$
$$subst(T_1 T_2, S_{n-1}, \ldots, S_0) = subst(T_1, S_{n-1}, \ldots, S_0)\ subst(T_2, S_{n-1}, \ldots, S_0)$$
$$subst(\lambda T, S_{n-1}, \ldots, S_0) = \lambda subst(T, w(S_{n-1}), \ldots, w(S_0), v_{n+1,0})$$

Операция $w(T)$ работает следующим образом. Пусть терм $T \in \Lambda_n$, тогда терм

$w(T) \in \Lambda_{n+1}$ и определен как:

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

Сформулируем и докажем вспомогательную лемму, которая пригодится нам далее:

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

Доказательство. Индукция по структуре терма T.

1. База индукции – $v_{n,i}$. Левая часть равна, по определению подстановки:

   $$subst(v_{n+1,i+1}, w(S_{n-1}), \ldots, w(S_0), v_{m+1,0}) = w(S_i)$$

   Правая часть:

   $$w(subst(v_{n,i}, S_{n-1}, \ldots, S_0)) = w(S_i)$$

2. Случай аппликации снова тривиален. Рассмотрим случай абстракции λT. Левая часть вычислится в:

   $$subst(w(\lambda T), w(S_{n-1}), \ldots, w(S_0), v_{m+1,0}) =$$
   $$subst(\lambda w(T), w(S_{n-1}), \ldots, w(S_0), v_{m+1,0}) =$$
   $$\lambda subst(w(T), w(w(S_{n-1})), \ldots, w(w(S_0)), v_{m+2,1}, v_{m+2,0}) \overset{\text{IH}}{=}$$
   $$\lambda w(subst(T, w(S_{n-1}), \ldots, w(S_0), v_{m+1,0}))$$

   Правая часть вычисляется в:

   $$w(subst(\lambda T, S_{n-1}, \ldots, S_0)) =$$
   $$w(\lambda subst(T, w(S_{n-1}), \ldots, w(S_0), v_{m+1,0})) = \lambda w(subst(T, w(S_{n-1}), \ldots, w(S_0), v_{m+1,0}))$$

■

Сформулируем и докажем наличие правой единицы у *subst*:

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

Доказательство. Индукция по структуре терма T:

База индукции для случая $T = v_{n,i}$ тривиальна, как и случай аппликации. Рассмотрим случай абстракции λT.

$$subst(\lambda T, v_{n,n-1}, \dots, v_{n,0}) =$$
$$\lambda subst(T, w(v_{n,n-1}), \dots, w(v_{n,0}), v_{n+1,0}) =$$
$$\lambda subst(T, v_{n+1,n}, \dots, v_{n+1,1}, v_{n+1,0}) \overset{\text{IH}}{=}$$
$$\lambda T$$

■

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. Это утверждение точно так же доказывается индукцией по структуре терма T. База индукции – случай, когда терм представляет собой переменную $v_{n,i}$. Тогда левая часть вычисляется в $subst(T_i, S_{m-1}, \dots S_0)$, ровно как и правая.

Случай аппликации снова тривиален, рассмотрим случай абстракции λT. Вычислим левую часть:

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

Вычислим теперь правую часть:

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

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 λ-термов, описанный в [5]. Библиотека **Bound** [18] для языка **Haskell**, например, использует именно монадическое представление.

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

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

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

$$\frac{M \in \Lambda_{V \coprod \{*\}}}{\lambda M \in \Lambda_V}$$

Здесь $\{*\}$ – это произвольное одноэлементное множество, а $\coprod$ – операция размеченного объединения множеств. По определению $A \coprod B$ состоит из элементов $inl(a)$ и $inr(b)$, 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 так же удобно для компьютерной реализации за счет того, что проверку корректности построения термов можно выполнять на уровне типов.

Пусть у нас есть функция $f : V \to W$, тогда мы можем задать функцию F_f из Λ_V в Λ_W рекурсией по терму $T \in \Lambda_V$:

1. $v \mapsto f(v)$
2. $M\ N \mapsto F_f(M)\ F_f(N)$
3. $\lambda M \mapsto \lambda F_{f'(f)}(M)$. Заметим, что просто так отобразить терм M с помощью функции f мы не можем, так как ее домен не совпадает со множеством, которым параметризован тип терма M. Поэтому мы построим по f функцию $f'(f) : V \coprod \{*\} \to W \coprod \{*\}$. Устроена она будет следующим образом:
   (a) $f'(f)(inl(x)) = inl(f(x))$
   (b) $f'(f)(inr(*)) = inr(*)$

Знакомый с языком программирования **Haskell** или теорией категорий читатель узнает, что мы задали структуру функтора. Интуитивно, действие этого функтора – это переименование переменных. Покажем, что это действительно функтор, именно, что он уважает тождественное отображение и композицию отображений.

***Утверждение* 4.** *Для любого $T \in \Lambda_V$ верно, что $F_{id_V}(T) = T$*

Доказательство. Индукция по терму T. База тривиальна, равно как и случай аппликации, покажем, что утверждение верно и для случая лямбды. Вспомогательная функция $f'(f)$ устроена следующим образом:

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

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 А, в функции `fmap-respects-id` ■

***Утверждение* 5.** *Для любого $T \in \Lambda_V$ и $f : V \to W$, $g : W \to X$ верно, что $F_{g \circ f}(T) = F_g(T) \circ F_f(T)$*

Доказательство. Снова индукция по терму T. Случай переменной снова тривиален, случай аппликации напрямую следует из индукционной гипотезы, рассмотрим случай абстракции. Посмотрим, во что вычислится левая часть:

$$F_{g \circ f}(\lambda M) = \lambda F_{f'(g \circ f)}(M)$$

где

$$f'(g \circ f)(inl(v)) = inl(g(f(v)))$$
$$f'(g \circ f)(inr(*)) = inr(*)$$

Правая, в свою очередь:

$$F_g(F_f(\lambda M)) = F_g(\lambda F_{f'(f)}(M)) =$$
$$\lambda F_{g'(g)}(F_{f'(f)}(M))$$

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

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

Вспомогательная функция $g'(g) : W \coprod \{*\} \to X \coprod \{*\}$ 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 лямбды утверждение верно. Формальное доказательство этого утверждения можно увидеть в приложении А, в функции `fmap-respects-comp` ■

Мы пойдем еще дальше и зададим структуру монады. Существует много способов определить её, мы воспользуемся тем, который принят в языке программирования **Haskell**. Именно, мы определим две операции: монадическую единицу $\texttt{return} : V \to \Lambda_V$ и монадическое связывание $\texttt{bind} : \Lambda_V \to (V \to \Lambda_W) \to \Lambda_W$. Кроме этого, мы покажем, что они удовлетворяют монадическим законам.

Монадическая единица устроена очень просто – это переменная. Связывание же, принимает на вход так называемую стрелку Клейсли [30] $k : V \to \Lambda_W$ и определяется индуктивно по структуре терма T:

1. $v \mapsto k(v)$
2. $M\ N \mapsto (\texttt{bind}(M, k))\ (\texttt{bind}(N, k))$
3. $\lambda M \mapsto \lambda(\texttt{bind}(M, k'(k)))$, где $k'(k) : V \coprod \{*\} \to \Lambda_{W \coprod \{*\}}$ и определяется следующим образом:
   (a) $k'(k)(inl(v)) = F_{inl}(k(v))$
   (b) $k'(k)(inr(*)) = \texttt{return}(inr(*))$

Сформулируем и докажем теперь свойства этих двух операций.

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

Доказательство. Индукция по структуре терма T:

1. База индукции, $T = v$. Имеем, что $\texttt{bind}(v, \texttt{return}) = \texttt{return}(v) = v$.
2. Случай аппликации напрямую следует из предположения индукции.
3. Пусть теперь $T = \lambda M$. Имеем, что $\texttt{bind}(\lambda M, \texttt{return}) = \lambda(\texttt{bind}(M, k'(\texttt{return})))$. Вспомогательное отображение $k'(return)$ устроено следующим образом:
   (a) $k'(\texttt{return})(inl(v)) = F_{inl}(\texttt{return}(v)) = \texttt{return}(inl(v))$
   (b) $k'(\texttt{return})(inr(*)) = \texttt{return}(inr(*))$

Заметим, что оно ведет себя так же, как и `return` на $V \coprod \{*\}$, 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 $k : V \to \Lambda_V$ верно $\texttt{bind}(\texttt{return}(v), k) = k(v)$*

Доказательство. Утверждение тривиально 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 монадный закон, сформулируем и докажем несколько технических лемм, которые помогут нам в его доказательстве.

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

Доказательство. Индукция по структуре терма T с тривиальными случаями переменной и аппликации. Рассмотрим случай абстракции λM. Нам нужно показать, что $\lambda\texttt{bind}(F_{f'(f)}(M), k'(k)) = \lambda\texttt{bind}(M, k'(k \circ f))$. По индукционной гипотезе мы знаем, что $\texttt{bind}(F_{f'(f)}(M), k'(k)) = \texttt{bind}(M, k'(k) \circ f'(f))$, 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* $T \in \Lambda_V$, $f : W \to U$ *и* $k : V \to \Lambda_W$ *верно* $F_f(\texttt{bind}(T, k)) = \texttt{bind}(T, (x \mapsto F_f(k(x))))$.

Доказательство. Снова индукция по структуре терма T. Случай переменной и аппликации снова тривиален, поэтому рассмотрим случай абстракции λM.

Нам нужно показать, что $\lambda F_{f'(f)}(\texttt{bind}(M, k'(k))) = \lambda\texttt{bind}(M, k'(x \mapsto F_f(k(x))))$. По индукционной гипотезе мы знаем, что $F_{f'(f)}(\texttt{bind}(M, k'(k))) = \texttt{bind}(M, (x \mapsto F_{f'(f)}(k'(k)(x))))$. Покажем теперь, что стрелки Клейсли $k'(x \mapsto F_f(k(x)))$ и $x \mapsto F_{f'(f)}(k'(k)(x))$ ведут себя одинаково на всех входах:

1. $inr(*)$. Обе стрелки вычисляются в $\texttt{return}(inr(*))$
2. $inl(v)$. Надо показать, что $F_{f'(inl)}(F_f(k(v))) = F_{f'(f)}(F_{inl}(k(v)))$. Так как Λ_V – функтор и он уважает композицию отображений имеем, что нужно показать $F_{f'(inl) \circ f}(k(v)) = F_{f'(f) \circ inl}(k(v))$. Для этого в свою очередь нужно снова показать, что два отображения $f'(inl) \circ f$ и $f'(f) \circ inl$ ведут себя одинаково на всех входах, но это очень легко понять, просто взглянув на определение 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* $f : V \to \Lambda_W$ *верно* $\texttt{bind}(F_{inl}(t), k'(g)) = F_{inl}(\texttt{bind}(T, f))$.

Доказательство. Снова индукция по структуре терма T, случай переменной и аппликации тривиален, рассмотрим случай абстракции λM.

Левая часть вычислится в:

$$\lambda F_{f'(inl)}(\texttt{bind}(M, k'(f)))$$

Правая в:

$$\lambda \texttt{bind}(F_{f'(inl)}(M), k'(k'(f)))$$

По лемме 2 имеем, что $\texttt{bind}(F_{f'(inl)}(M), k'(k'(f))) = \texttt{bind}(M, k'(k'(f)) \circ f'(inl))$. По лемме 3 имеем $F_{f'(inl)}(\texttt{bind}(M, k'(f))) = \texttt{bind}(M, (x \mapsto F_{f'(inl)}(k'(f)(x))))$. Заметим теперь, что стрелки Клейсли $k'(k'(f)) \circ f'(inl)$ и $x \mapsto F_{f'(inl)}(k'(f)(x))$ 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`. ■

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

Доказательство. Индукция по терму T:

1. Рассмотрим случай, когда $T = v$. Тогда левая часть вычисляется в $\texttt{bind}(f(v), g)$, ровно как и правая.

2. Случай для аппликации следует напрямую из индукционной гипотезы.

3. Рассмотрим случай абстракции λM. Посмотрим, во что вычисляется левая часть:

$$\texttt{bind}(\texttt{bind}(\lambda M, f), g) = \texttt{bind}(\lambda \texttt{bind}(M, k'(f)), g) =$$
$$\lambda \texttt{bind}(\texttt{bind}(M, k'(f)), k'(g))$$
где
$$k'(g)(inl(w)) = F_{inl}(g(w))$$
$$k'(g)(inr(*)) = \texttt{return}(inr(*))$$
и
$$k'(f)(inl(v)) = F_{inl}(f(v))$$
$$k'(f)(inr(*)) = \texttt{return}(inr(*))$$

Правая часть, в свою очередь, вычисляется в:

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

Чтобы воспользоваться индукционной гипотезой, необходимо показать, что $k'(x \mapsto \texttt{bind}(f(x), g)) : V \coprod \{*\} \to \Lambda_{U \coprod \{*\}}$ ведет себя так же как и $x \mapsto \texttt{bind}(k'(f)(x), k'(g))$.

Для этого мы просто покажем, что они возвращают одинаковый результат на всех входах.

Рассмотрим два случая, как могут выглядеть входные данные:

(a) $inr(*)$. Обе части вычисляются в $\texttt{return}(inr(*))$.

(b) $inl(v)$. Левая часть вычисляется в:

$$F_{inl}(\texttt{bind}(f(v), g))$$

Правая:

$$\texttt{bind}(F_{inl}(f(v)), f'(g))$$

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

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

Отметим наконец, что действие функтора мы проинтерпретировали, как переименование переменных. Действие монадического связывания можно, в таком случае, проинтерпретировать как подстановку. Это утверждение не столь очевидно, но если рассмотреть сигнатуру `bind` и обратить внимание на то, что функцию $k : V \to \Lambda_W$ можно задать в виде списка пар (V, Λ_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.

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 Γ – это не содержащий дубликатов список переменных $x_1, \ldots x_n$, $x_i \in \mathcal{V}$.

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

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

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

2. $x_1, \ldots, x_n \vdash MN \mapsto \Phi(x_1, \ldots, x_n \vdash M)\ \Phi(x_1, \ldots, x_n \vdash N)$

3. $x_1, \ldots, x_n \vdash \lambda x.M \mapsto \lambda\Phi(x_1, \ldots, x_n, x \vdash M)$

Покажем, что такое преобразование уважает отношение α-эквивалентности, введенное в разделе 2.1.

***Утверждение* 9.** *Пусть* $T_1, T_2 \in \Lambda$, $\Gamma_1 \vdash T_1$, $\Gamma_2 \vdash T_2$ *и* $T_1 =_\alpha T_2$. *Тогда* $\Phi(\Gamma_1 \vdash T_1) = \Phi(\Gamma_2 \vdash T_2)$.

Доказательство. Индукция по термам T_1 и T_2. Так как мы знаем, что они α-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).

База индукции для случая двух переменных тривиальна, как и случай для двух аппликаций. Рассмотрим случай, когда $T_1 = \lambda x.M$, $T_2 = \lambda y.M'$. Так как они α-эквивалентны, то мы знаем, что $M' =_\alpha M[x \mapsto y]$ и $y \notin FV(M)$. Мы знаем, что $\Gamma_1 \vdash \lambda x.M$ и $\Gamma_2 \vdash \lambda y.M'$, следовательно $\Gamma_1, x \vdash M$ и $\Gamma_2, y \vdash M'$. Кроме того, так как $y \notin FV(M)$, то $\Gamma_1, y \vdash M[x \mapsto y]$. По посылке из правила альфа-эквивалентности для лямбд мы знаем, что $M' =_\alpha M[x \mapsto y]$, следовательно по индукционной гипотезе $\Phi(\Gamma_2, y \vdash M') = \Phi(\Gamma_1, y \vdash M[x \mapsto y])$. Осталось заметить, что $\Phi(\Gamma_1, y \vdash M[x \mapsto y]) = \Phi(\Gamma_1, x \vdash M)$ и получить доказательство исходного утверждения. ■

Легко сконструировать преобразование в обратную сторону – из неименованных термов в именованные. Оно принимает на вход терм $T' \in \Lambda_n$ и возвращает пару из контекста Γ и терма $T \in \Lambda$, определенного в нем. Рассмотрим три случая, необходимые для того, чтобы рекурсивно задать это преобразование, назовем его Ψ.

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

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

2. В случае аппликации $M\ N$ 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 π_1 и π_2 – первая и вторая проекция для пар соответственно.

3. Для абстракции λT действуем примерно так же. Вызываемся рекурсивно от T и получаем терм, который определен в расширенном на одну переменную контексте. Так как контекст расширяется путем добавления переменной в конец, то мы точно знаем, по какой переменной абстрагироваться:

$$\lambda T \mapsto take(n, \pi_1(\Psi(T))) \vdash \lambda\ last(\pi_1(\Psi(T)))\ \pi_2(\Psi(T))$$

Здесь $take(n, xs)$ – операция, берущая первые n элементов из списка xs, а $last(xs)$ – операция, возвращающая последний элемент в списке.

Легко заметить, что эти два преобразования взаимно-обратны. Действительно, индукция по структуре терма настолько прямолинейна, что мы не будем приводить ее здесь.

Еще легче сконструировать преобразования между неименованными и монадическими термами. Для примера покажем преобразование Δ из Λ_n в $\Lambda_{\overline{n}}$, где $\overline{n}$ – это множество $\{0, 1, \ldots, n-1\}$. Действительно, рассмотрим три случая:

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

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

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

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

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

$$f : \overline{n+1} \to \overline{n} \coprod \{*\}$$
$$0 \mapsto inr(*)$$
$$i \mapsto inl(i-1)$$

Окончательно:

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

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

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:

$$f^{-1} : \overline{n} \coprod \{*\} \to \overline{n+1}$$
$$inl(i) \mapsto i + 1$$
$$inr(*) \mapsto 0$$

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

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.

3.1. Описание языка Vclang

Для реализации всех описанных во второй главе представлений руководителем был предложен язык **Vclang**. В этом подразделе мы кратко опишем его отличия от известных систем автоматического доказательства теорем, например **Agda**.

Язык **Vclang** разрабатывается в компании JetBrains с февраля 2015 года в рамках исследования в группе гомотопической теории типов и зависимых типов[1]. 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**).

С точки зрения синтаксиса, **Vclang** очень похож на **Agda**, кроме использования Unicode-обозначений и 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. Метапеременные
4. Проверка на тотальность

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 тотальными. Проверка на тотальность 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.

[1] https://research.jetbrains.org/ru/groups/group-for-dependent-types-and-hott

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

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

Листинг 1: 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:

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

Листинг 2: Доказательство симметричности равенства в **Agda**

В **Vclang** равенство определяется с помощью так называемых «путей». Формально эта теория вводится в статье [17], 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 [9].

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.

```
\data Int
  | pos Nat
  | neg Nat
```

Листинг 3: 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:

```
\data Int
  | pos Nat
  | neg Nat
\with
  | neg zero => pos zero
```

Листинг 4: Тип целых чисел с условием

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

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

Листинг 5: 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. Аналогом этой конструкции в **Agda** являются *сетоиды* – множества с введенным на них отношением эквивалентности. В нашем случае, можно обойтись и формализовывать именованное представление без фактор-типов и использовать сетоиды, но тогда вместо доказательстве равенства типов(эквивалентности между различными представлениями) нам придется вводить и доказывать отношение эквивалентности сетоидов. Отношение равенства предпочтительней по многим причинам, в частности, для него верен принцип Лейбница(принцип подстановки эквивалентных).

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

```
-- I is for index
\data ITerm (n : Nat)
  | IVar (Fin n)
  | IApp (ITerm n) (ITerm n)
  | ILam (ITerm (suc n))
```

Листинг 6: Тип данных, кодирующий термы в неименованном представлении.

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

Единственный тонкий момент, связанный с реализацией этого представления – это преобразование неименованных термов в именованные. Если определять его напрямую, как описывалось в разделе 2.2, то уже в случае аппликации выяснится несоответствие типов. Рассмотрим пример:

```
\function
psi
  {n : Nat}
  (t : ITerm n) : \Sigma (env : Ctx n) (t' : NTerm) (proof : TermInCtx
  ↪ env t') <= \elim t
    | IVar i       => (gen_env n, NVar (env !! i), VarInCtx (i, idp))
    | IApp s1 s2   => ((psi s1).1, NApp ((psi s1).2) ((psi s2).2),
    ↪ AppInCtx ((psi s1).3) {?})
    | ILam t       => (rtail ((psi t).1), NLam (rhead ((psi t).1))
    ↪ ((psi t).2), LamInCtx (transport (\lam a => TermInCtx a t')
    ↪ (inv (snoc-lemma ((psi t).1))) ((psi t).3)))
```

Листинг 7: Вариант определения функции, переводящей неименованный терм в именованный.

Мы хотим вернуть доказательство того, что аппликация `NApp t1 t2` определена в контексте `env1`, для этого нам, очевидно, нужны доказательства, что оба аппликанта определены в `env1`. Для первого аппликанта оно естественным образом получается из рекурсивного вызова. Для второго же мы можем только получить доказательство, что `t2` определен в контексте `env2`.

Чтобы решить эту проблему заметим, что нам не очень важно, в каком контексте будет определен получившийся на выходе терм. Поэтому мы можем поступить иначе. Мы определим вспомогательную функцию, которая будет принимать на вход контекст и возвращать пару из терма и доказательства, что он определен в этом контексте.

Тогда основная функция будет генерировать контекст заданной длины и вызывать от него вспомогательную. Например:

```
\function
psi'
    {n : Nat}
    (env : Ctx n)
    (t : ITerm n) : \Sigma (t' : NTerm) (p : TermInCtx env t') <=
    ↪ \elim t
        | IVar i        => (NVar (env !! i), VarInCtx (i, idp))
        | IApp t1 t2    => (NApp (psi' env t1).1 (psi' env t2).1,
        ↪ AppInCtx (psi' env t1).2 (psi' env t2).2)
        | ILam t        => (NLam n (psi' (extend env n) t).1, LamInCtx
        ↪ (psi' (extend env n) t).2)


\function
psi
    {n : Nat}
    (t : ITerm n) : \Sigma (env : Ctx n) (t' : NTerm) (proof :
    ↪ TermInCtx env t') => (gen_env n, (psi' (gen_env n) t).1, (psi'
    ↪ (gen_env n) t).2)
```

Листинг 8: Вариант определения функции, переводящей неименованный терм в именованный, не вызывающий ошибки проверки типов.

3.3. Реализация монадического представления термов

Термы в монадическом представлении кодируются следующим типом данных:

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

Листинг 9: Тип данных, кодирующий термы в монадическом представлении.

Для абстракции можно воспользоваться типом `Maybe V`, нетрудно убедиться, что эти типы изоморфны между собой.

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 фукнцию `bind'` : $FTerm(V + n) \to (V \to FTerm\ W) \to FTerm(W + n)$, 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 10: 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 11: Тип $V + n$

Функции `fmap-telescope` и `twistT` являются в некотором роде аналогами `fmap` и `sequenceA` из класса типов Traversable в **Haskell** [14] 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:

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

Листинг 12: Вариант определения преобразования монадического терма в неименованный.

Однако в случае абстракции становится неочевидно, что этот код завершается. Поэтому в качестве примера определим тип `Fun n` очень похожий на `Fin n`(на самом деле – они изоморфны друг другу):

```
\function
Fun
    (n : Nat) : \Type <= \elim n
        | zero  => Empty
        | suc n => Either (Fun n) Unit
```

Листинг 13: Тип `Fun n`.

Тогда преобразование в него определяется довольно просто:

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

Листинг 14: Работающий вариант определения преобразования монадического терма в неименованный.

3.4. Реализация именованного представления термов

Термы в этом представлении кодируются следующим типом данных:

```
-- N is for named
\data NTerm
    | NVar Nat
    | NApp NTerm NTerm
    | NLam Nat NTerm
```

Листинг 15: 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. В дальнейшем, мы пользуемся им для определения, например, операции подстановки.

Альфа-эквивалентность определяется как следующий индуктивный предикат:

```
\data AlphaEq (t1 t2 : NTerm)
  | AlphaEq (NVar x) (NVar y)           => VarEq (x = y)
  | AlphaEq (NApp t1 s1) (NApp t2 s2)   => AppEq (AlphaEq t1 t2)
  ↪ (AlphaEq s1 s2)
  | 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 16: 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»:

```
\function
VarInCtx {n : Nat} (env : Ctx n) (x : Nat) => \Sigma (i : Fin n) (p :
 ↪  env !! i = x)

\data TermInCtx {n : Nat} (env : Ctx n) (t : NTerm)
  | TermInCtx {n} env (NVar x)     => Variable (p : VarInCtx env x)
  | TermInCtx {n} env (NApp t1 t2) => Application (p : TermInCtx env
   ↪  t1) (q : TermInCtx env t2)
  | TermInCtx {n} env (NLam x t)   => Abstraction (p : TermInCtx
   ↪  (extend env x) t)
```

Листинг 17: 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»:

```
\data AlphaEq
    {n : Nat}
    (env1 : Ctx n)
    (t1 : NTerm)
    (p1 : TermInCtx env1 t1)
    (env2 : Ctx n)
    (t2 : NTerm)
    (p2 : TermInCtx env2 t2)
        | AlphaEq env1 (NVar x) (Variable p) env2 (NVar y) (Variable
         ↪  q)                          => VarEq (p : p.1 = q.1)
        | 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 18: 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:

```
\function
nsubst
    (t : NTerm)
    (y : Nat)
    (s : NTerm) : NTerm <= \elim t
        | NVar x     => \case x =? y | inr _ => s | inl _ => (NVar x)
        | NApp t1 t2 => NApp (nsubst t1 y s) (nsubst t2 y s)
        | NLam x t   => \let
                          | frv => gen-fresh-var t
                        \in \case isFreeDec x s | inr _ => (NLam frv
                        ↪  (nsubst (nsubst t x (NVar frv)) y s)) |
                        ↪  inl _ => (NLam x (nsubst t y s))
```

Листинг 19: 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:

```
\function
nsubst'
    (t : NTerm)
    (ps : List (\Sigma Nat NTerm)) : NTerm <= \elim t
        | NVar x        => lookup ps x
        | NApp t1 t2    => NApp (nsubst' t1 ps) (nsubst' t2 ps)
        | NLam x t      =>  \let
                                | ts => (NLam x t) :-: (map (\lam (p :
                                ↪  \Sigma Nat NTerm) => p.2) ps)
                                | x' => gen-fresh-var ts
                            \in NLam x' (nsubst' t ((x, NVar x') :-:
                            ↪  ps))

\function
nsubst
    (t : NTerm)
    (y : Nat)
    (s : NTerm) : NTerm => nsubst' t (singleton (y, s))
```

Листинг 20: Вариант определения операции подстановки с помощью параллельной подстановки.

Вспомогательная функция `nsubst'` 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, завершается. Однако доказательство леммы о подстановке все еще представляет большие трудности из-за того, что необходимо рассматривать большое количество случаев и конструкции в типах становятся громоздкими. Один из способов решить эту и ряд вышеперечисленных проблем – использовать идеи, аналогичные тем, что применяются в номинальных множествах – [24].

Заключение

В ходе выполнения работы была рассмотрена с новой точки зрения задача формализации лямбда-исчисления. Мы описали различные представления лямбда-термов, определили фундаментальную операцию подстановки и описали её свойства.

В отличии от других решений этой задачи, мы установили соответствие между различными представлениями лямбда-термов. Мы формализовали неименованное, монадическое и часть именованного представления термов. Мы выяснили, что с неименованным и монадическим представлением очень удобно работать в том смысле, что формальные определения без особых усилий перекладываются на язык доказателя теорем. Для этого мы использовали экспериментальный язык, построенный на новой теории типов, которая, в частности, позволяет описывать некоторые конструкции гораздо более удобным образом, нежели, например, **Agda**.

Однако, работа не лишена недостатков. Технические проблемы в самом языке и сложность именованного представления для формализации привели к тому, что свойство ассоциативности для операции подстановки осталось неформализованным. В качестве решения этой проблемы можно использовать идеи, описанные в теории номинальных множеств. Мы оставим эти идеи, как направление дальнейшего развития работы.

Исходный код опубликован в публичном Git-репозитории и доступен по ссылке `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] Bird Richard S, Paterson Ross. De Bruijn notation as a nested datatype // Journal of functional programming. — 1999. — Vol. 9, no. 01. — P. 77–91.

[6] Church Alonzo. An unsolvable problem of elementary number theory // American journal of mathematics. — 1936. — Vol. 58, no. 2. — P. 345–363.

[7] Coquand Thierry. An algorithm for type-checking dependent types // Science of Computer Programming. — 1996. — Vol. 26, no. 1-3. — P. 167–177.

[8] Coquand Thierry, Huet Gérard. The calculus of constructions // Information and computation. — 1988. — Vol. 76, no. 2-3. — P. 95–120.

[9] Cubical type theory: a constructive interpretation of the univalence axiom / Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg // arXiv preprint arXiv:1611.02108. — 2016.

[10] Curry Haskell B. Grundlagen der kombinatorischen Logik // American journal of mathematics. — 1930. — Vol. 52, no. 4. — P. 789–834.

[11] 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.

[12] 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.

[13] Gordon Andrew, Melham Tom. Five axioms of alpha-conversion // Theorem proving in higher order logics. — 1996. — P. 173–190.

[14] Hackage. Data.Traversable. — Access mode: https://goo.gl/JZ2xfa (online; accessed: 2017-06-02).

[15] 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.

[16] Huet Gérard. Residual theory in λ-calculus: A formal development // Journal of Functional Programming. — 1994. — Vol. 4, no. 03. — P. 371–394.

[17] Isaev Valery. Model Structures on Categories of Models of Type Theories // arXiv preprint arXiv:1607.07407. — 2016.

[18] Kmett Edward A. "bound: Making de Bruijn Succ Less". — Access mode: https://goo.gl/TKhGDn (online; accessed: 2017-05-27).

[19] Landin Peter J. The mechanical evaluation of expressions // The Computer Journal. — 1964. — Vol. 6, no. 4. — P. 308–320.

[20] Martin-Löf Per. An intuitionistic theory of types: Predicative part // Studies in Logic and the Foundations of Mathematics. — 1975. — Vol. 80. — P. 73–118.

[21] McKinna James, Pollack Robert. Some lambda calculus and type theory formalized // Journal of Automated Reasoning. — 1999. — Vol. 23, no. 3. — P. 373–409.

[22] Nipkow Tobias. More Church-Rosser proofs (in isabelle/hol) // Automated Deduction—CADE-13. — 1996. — P. 733–747.

[23] Norell Ulf. Towards a practical programming language based on dependent type theory. — Citeseer, 2007. — Vol. 32.

[24] Pitts Andrew M. Nominal sets: Names and symmetry in computer science. — Cambridge University Press, 2013.

[25] Sakaguchi Kazuhiko. A Formalization of Typed and Untyped λ-Calculi in SSReflect-Coq and Agda2. — Access mode: https://goo.gl/WtcdZE (online; accessed: 2017-05-23).

[26] Sato Masahiko. Theory of symbolic expressions, I // Theoretical Computer Science. — 1983. — Vol. 22, no. 1-2. — P. 19–55.

[27] Schönfinkel Moses. Über die Bausteine der mathematischen Logik // Mathematische Annalen. — 1924. — Vol. 92, no. 3. — P. 305–316.

[28] Shankar Natarajan. A mechanical proof of the Church-Rosser theorem // Journal of the ACM (JACM). — 1988. — Vol. 35, no. 3. — P. 475–522.

[29] Stoughton Allen. Substitution revisited // Theoretical Computer Science. — 1988. — Vol. 59, no. 3. — P. 317–325.

[30] Wiki Haskell. Arrow tutorial. — Access mode: `https://goo.gl/bIiCsy` (online; accessed: 2017-05-29).

A. 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
```