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

Кафедра математических и информационных технологий

Гарифуллин Шамиль Раифович

# Генерация зависимых языков по спецификации пользователя

Магистерская диссертация

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

Научный руководитель:
Исаев В. И.

Рецензент:
Подкопаев А. В.

Санкт-Петербург
2017

SAINT PETERSBURG ACADEMIC UNIVERSITY
Higher education centre

Department of Mathematics and Information Technology

Shamil Garifullin

# Specification based generation of languages with dependent types

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Alexander Omelchenko

Scientific supervisor:
Valeriy Isaev

Reviewer:
Anton Podkopaev

Saint Petersburg
2017

# Оглавление

# Введение

Теории зависимых типов были впервые подробно рассмотрены Мартин-Лёфом в 1980 году в курсе лекций, прочтенных им в Падове, которые были расширены в книгу [16]. Одна из основных его мотиваций заключалась в том, чтобы предложить альтернативу теории множеств для формализации математики. Основным преимуществом теории типов является ...

Это называется соответствием Карри-Говарда-Ламбека[7] — когда типы языка соответствуют утверждениям в соответствующей логике, а выражения этих типов доказательствами соответствующих утверждений. Достоинство данного подхода заключается в том, что проверка доказательств перекладывается на алгоритм проверки типов соответсвующего языка программирования, а это автоматизированный процесс.

Наиболее известными примерами языков, которые пользуются соответствием Карри-Говарда для доказательства математических утверждений являются Coq[6] и Agda[1]. Например, доказательство теоремы о четырех красках было завершено в 2005 году с помощью Coq[24].

Другая область применения теории типов — это верификация программ. Так как использование произвольных выражений языка на уровне типов значительно расширяет экспрессивность системы типов языка, можно задавать произвольные ограничения на входные и выходные данные функций языка. Таким способом можно описывать и формально верифицировать даже большие системы, например существует формально верифицированная версия Standard ML[17] под названием CakeML[5]. Пример относительно простого доказательства — корректности функции filter на Agda приведен в Приложении A.

При реализации языков с зависимыми типами возникает ряд типичных задач, основной из которых является реализация функции проверки типов. Однако для её реализации необходимо реализовывать функцию вычисления выражений и проверки их на равенство, для этого необходимо уметь совершать манипуляции с выражениями языка — а значит их тоже реализовывать. Основными манипуляциями являются работа со связываниями: абстракция по переменной и подстановка выражения вместо переменной (подробнее про реализацию в подразделе). Цель данной работы заключается в том, чтобы реализовать приложение, которое по спецификации зависимого языка (подробнее о языке спицификации в Разделе 4) генерирует исходный код на каком-либо языке (для этих целей мы используем Haskell[12]), осуществляющий проверку типов заданного языка.

С одной стороны такое приложение упростит реализацию языков с зависимыми типами, так как, благодаря ему, можно будет избежать реализации фактически шаблонного кода. С другой стороны, позволит эксперементировать с различными вариациями теорий типов.

Поэтому основной задачей данной работы является определение языка спецификации зависимых языков, с дальнейшей генерацией представления АСД этого языка и функций работы с деревом в виде модуля Haskell[12]. *АСД* — абстрактное синтаксическое дерево, по любому выражению языка можно составить дерево, узлами этого дерева будут конструкции языка, а потомками узла выражения к которым конструкция применяется. Генерируемый модуль содержит функции проверки типов, вычисления выражений, работы с контекстами и стандартных операций над выражениями (такие как подстановка, абстракция и проверка на равенство).

Текст работы состоит из трёх основных разделов. Каждый из них мы обсудим подробнее в соответствующем подразделе введения.

## Языки с зависимыми типами

Все зависимые языки состоят из некоторых конструкций языка, с помощью которых пишутся все выражения в языке.

Например, язык булевых выражений Bool состоит из четырёх конструкций: тип Bool, константы true и false, и конструкция if-then-else. Нас интересуют только типизированные языки, поэтому для каждой конструкции должны быть прописаны правила типизации. В нашем языке true имеет тип Bool, false имеет тип Bool, if-then-else принимает четыре аргумента — выражение типа Bool, тип возвращаемого выражения и два выражения, имеющих тип равный возвращаемому. Формально языки задаются через правила вывода(см. Раздел 4).

Также необходимо задавать правила вычисления языка, их принято называть правилами редукции языка. Для Bool их всего два, а именно для конструкции if-then-else мы возвращаем либо ветку then, либо ветку else в зависимости от первого её аргумента.

Выше был представлен пример обычного, независимого языка Bool. Чтобы из него сделать зависимый язык Bool, нужно модифицировать конструкцию if-then-else. Теперь вторым её аргументом будет функция, возвращающая тип возвращаемого выражения в зависимости от переданного ей аргумента типа Bool. Тогда станет возможной конструкция вида: if-then-else(t, f, True, 1), которая будет возвращать либо True типа Bool, либо 1 типа Int в зависимости от истинности первого аргумента конструкции. Более подробное введение в зависимые языки можно найти в [16].

## Определение языка спецификаций

Наша цель научиться записывать правила типизации формально и по такому описанию генерировать код, который бы осуществлял проверку типов для соответствующего языка.

Формализация языка происходит путем описания его спецификации. В спецификации задаются возможные уровни выражений языка, например в Haskell это типы, термы и виды. Затем описываются его конструкции — описываются уровни аргументов и возвращаемого выражения. Также описываются правила типизации каждой конструкции и правила редукции языка. Правила типизации и редукции содержат переменные языка спецификации, которые мы впредь называем *метапеременными*, так как они являются переменными нашего мета-языка — языка спецификации. Все метапеременные, содержащиеся в правилах, должны быть аннотированы уровнями выражений, для проверки спецификации.

Затем описание проходит проверку на корректность (определение языка спецификации и проверки описаны в Разделе 4). Эта проверка должна исключать как просто некорректно записанные языки, так и языки для которых генерация кода будет проблематичной или невозможной. Поэтому важной частью этой подзадачи является огрничения множества языков, которые возможно специфицировать в нашем языке.

Стоит отметить, что спецификация позволяет задавать нестабильные теории (см. Раздел 4 для более подробного описания). Это ограничение полезно для определенных теоретических применений теории типов, которые мы не будем обсуждать в данной работе, так как они выходят за ее рамки. Одним из применений нестабильности является [29].

## Реализация

После проверок спецификации (описанных в Разделе 4.1) строится структура хранящая информацию о правилах вывода, редукциях и конструкциях языка. С её помощью происходит кодогенерация представления термов языка и функций проверки типов и вычисления.

В дальнейшем мы понимаем вычисление как переписывание термов согласно редукциям языка, пока не получим терм к которому ни одна редукция неприменима — этот процесс называется приведением терма в *нормальную форму*.

Нормализация генерируется по правилам редукции, описанным в спецификации, функция проверки типов — по правилам вывода. Неявно подразумевается, что в языке есть отношение эквивалентности на термах, которое порождается отношением редукции. Это выражается в том, что сравнение термов (которое сравнивает их с точностью до этого отношения эквивалентности) сначала нормализует термы, а потом сравнивает их нормальные формы.

Так как типы зависимого языка могут включать в себя произвольные термы, проверка типов является задачей тесно связанной с вычислением языка. Например, если наша функция принимает только списки длинны числа фибоначчи, а нам передана конкатенация списков длины 2 и 3 то, чтобы понять является ли это число числом фибоначчи, нам нужно его вычислить $2+3 => 5$, также вычислить первые несколько чисел фибоначчи, положим числа фибоначчи определены как список $[1, 1, 2, 3, 5, 8, ...]$, затем вычислить предикат принадлежности $5 \in fibs$ и только тогда мы можем вызвать функцию сравнения термов. Также стоить заметить что термы могут иметь достаточно сложную нормальную форму и нам может прийтись сравнивать АСД этих термов.

Поэтому написание функции проверки типов языка становится достаточно ёмкой задачей — мы должны попутно реализовывать функцию нормализации. Однако общий алгоритм проверки типов не сильно отличается от языка к языку — всегда нужно рекурсивно проверять АСД на удовлетворение правилам вывода, таким образом его можно генерировать по спецификации, при наличии достаточного количества ограничений на последнюю (подробнее в Разделе 2.1).

Как упоминалось выше, правильно выбранное представление может значительно упростить генерацию кода. Также существуют варианты представления, дающие больше гарантий на корректность составления термов языка, благодаря более строгой типизации. Рассмотрены несколько вариантов представления (см. Раздел 5 для подробного обсуждения этих вариантов):

1. Обычное именованное (переменные представляются в виде строк)

2. Обычные индексы де Брейна[8] (переменные явяются целыми числами, указывающими на место их связывания)

3. Индексы де Брейна с использованием полиморфной рекурсии[19]

У первых двух способов представления есть недостатки. В первом необходимо вводить $\alpha$-эквивалентность на термах — *$\alpha$-эквивалентными* называются термы, которые отличаются только в именовании связанных переменных. Во втором варианте возникают сложности с работой под связываниями переменных — а именно, так как процесс проверки типов рекурсивен, мы должны гарантировать корректность подтермов при рекурсином вызове, однако .

Также в каждом из этих случаев легко допустить ошибку при работе с термами. Третий вариант является модификацией второго, в которой совершать ошибки при работе с индексами сложнее из-за проверок на уровне типов, таким образом код пользователя получается имеет больше гарантий корректности.

Также третий подход позволяет с большей легкостью генерировать операции над термами: равенство проверяется непосредственно, подстановки и абстракция тоже не составляют больших усилий.

# 1. Постановка задачи

Целью данной работы является разработка и реализация языка для спецификации языков программирования с зависимыми типами. Ключевые задачи:

- Сужение множества возможных спецификаций зависимых языков для возможности генерации алгоритма проверки типов.
- Реализация генерации структур данных представления АСД языка и функций манипуляции этими структурами.
- Реализация генерации функций приведения термов специфицированного языка в нормальную форму и проверки типов.

# 2. Языки с зависимыми типами

Во многих языках программирования возникают ошибки связанные с доступом за границу массива. Аналогом этого в Haskell является взятие первого элемента в списке.

```
head :: [a] -> a
head (x:_) = x
head [] = error "No head!"
```

Такие ситуациях обычно решаются с помощью механизма исключений или его аналогов. Однако эту проблему можно решить иначе, наложив на вход дополнительные ограничения. А именно — не принимать некорректные входные данные.

```
head :: {n : N} -> Vec a (suc n) -> a
head (x:_) = x
```

Здесь тип явно специфицирует, что функция не принимает термы типа 'Vec a 0'. Языки с зависимыми типами позволяют типам зависеть от выражений языка, именно это позволяет описать тип списков фиксированной длины.

Как правило, программисты все равно проверяют какие-то ограничения перед вызовом функции или обладают дополнительной информацией, на основе которой они пишут код так, как они его пишут. В зависимых языках мы можем писать программы, где передача этого знания будет явно требоваться компилятором, что позволяет не допускать такого рода ошибки.

Этот способ обобщается, и можно доказывать корректность работы алгоритмов, например функции filter в Приложении A.

## 2.1. Проверка типов в зависимых языках

Выражения нашего языка мы будет называть *термами* — они строятся индуктивно из применений конструкций языка к другим конструкциям или переменным. Обычно в тексте под термом подразумевается выражения определенного уровня языка, а именно все, что не является типом (или видом). Однако 'терм' может означать 'любого выражения языка', обычно в тексте такое значение можно отличить от предыдущего тем, что в этом значении оно используется в выражении *терм языка.*

Языки с зависимыми типами обычно задаются через формализм написаний правил вывода (для описания языка целиком см. Раздел 4). Рассмотрим пример правила вывода:

$$\frac{\Gamma, x : S \vdash T\ type \qquad \Gamma, \vdash f : \Pi(S, T) \qquad \Gamma \vdash t : S}{\Gamma \vdash app(T, f, t) : T[x := t]}$$

Введём несколько определений, которые будем использовать дальше во всем тексте. Все переменные, являясь переменными мета-языка описания языков называются *метапеременными.* Каждая конструкция вида "... $\vdash$ ..." называется *суждением.* Все что левее символа $\vdash$ называется *контекстом.* Все суждения, что находятся выше черты, называется *предпосылками.* Суждение под чертой называется *заключением* правила вывода.

Само правило является правилом вывода применения зависимой функции. Конструкция языка $\Pi$ принимает в качестве аргумента терм и в зависимости от аргумента возвращает тип, привычные нам независимые функции через $\Pi$ выражаются как константные функции, так как всегда возвращают один и тот же тип.

Правила вывода можно представлять как узлы дерева вывода, где заключение является предком всех предпослок. Проверка типов в любом языке это обход АСД и происходит так: мы имеем некоторые аргументы внутри примитива, которые мы используем для составления узлов-потомков (предпосылок). На этих узлах вызываем функцию вывода типов в возможно расширенном контексте (конечно, мы должны для каждого расширения контекста проверять его корректность) рекурсивно. Если потомки составлены корректно, то получаем некие типы, которые можем использовать в проверке равенств в предпосылках и возрате типа примитива.

В зависимых языках все точно так же, однако проверка на равенство должна происходить после нормализации термов. Нормализацию мы применяем только после того, как убедимся, что термы корректно составлены. Получается, что нормализация тесно связана с проверкой типов.

Давайте разберём алгоритм на примере выше.

1. Чтобы проверить терм $app(T, f, t)$ (и вернуть его тип $T[x := t]$), поочередно проверяем предпосылки.

2. Вызываемся рекурсивно на терме $t$ и, если не произошло ошибки, получаем тип $S$.

3. Расширяем контекст типом $S$ и проверяем "тип" $T$ (на самом деле мы проверяем тип $T$ на определенность).

4. Вызываемся рекурсивно на $f$ — получаем его тип. Теперь нужно проверить равенство его нормальной формы нормальной форме типа $\Pi(S, T)$, который мы строим из имеющихся метапеременных.

5. Если мы дошли до этой стадии, значит все определено корректно и мы возвращаем тип $T[x := t]$

Можно заметить, что мы должны уметь корректно выполнять подстановки и проверять термы на равенство, равенство подразумевается до $\alpha$-эквивалентности. $\alpha$-

*эквивалентными* называются термы, которые отличаются только в именовании связанных переменных. Поэтому при реализации языка мы должны заботиться о выборе корректного представления, чтобы подстановка и равенство не требовали больших усилий при их написании.

# 3. Обзор аналогов

Построение языков программирования с зависимыми типами по спецификации является задачей достаточно специфичной. Ниже перечислены некотороые инструменты, применяемые в похожих ситуациях. Таких, как изучение формальных систем, языков программирования и их реализация.

## 3.1. BNFC

Похожим на программу описанную в дипломной работе средством разработки является BNFC[3]. Эта утилита позволяет генерировать фронтенд компилятора по аннотированной грамматике языка в форме Бэкуса-Наура[9].

Программа генерирует лексический анализатор, синтаксический анализатор и вывод структур на экран языка заданного в спецификации. Также она генерирует абстрактное синтаксическое дерево и заготовку для написания редукций, представленную в виде большой конструкции switch или её аналогов.

Генерирует представления на C, C++, C#, Haskell, Java и OCaml.

## 3.2. PLT/Redex

PLT/Redex[20] — встроенный DSL на языке Racket, созданный для спецификации и изучения операционных семантик языков программирования. Он используется для спецификации языков программирования, в том числе и с зависимыми типами.

Из отличительных черт: позволяет случайным образом тестировать цикличность редукций или иные свойства языка, задаваемые пользователем в DSL. Также позволяет визуализировать порядок редукций.

Однако описание языков с зависимыми типами не является лишь спецификацией, а требует ещё и реализации пользователя[15]. Некоторую сложность составляют подстановки — проблема, обойденная в данной дипломной работе благодаря использованию представления термов языка в виде индексов де Брейна.

## 3.3. Twelf

Twelf[23] является реализацией LF[18]. Эта программа используется для спецификации и доказательств свойств логик и языков программирования.

В спецификации задаются высказывания языка (используется принцип “высказывания в качестве типов”[11]) и некоторые операции над ним в виде отношений на языке Twelf. Затем доказываются свойства вида $\forall\Sigma$ специфицированного языка.

Таким образом, в Twelf можно доказывать свойства спецификаций языков или приводить спецификации к форме, в которой выполняются интересующие нас, как разработчика, свойства. Затем можно использовать программу, описанную в данной дипломной работе, для реализации языка.

# 4. Определение языка спецификаций

Как мы уже писали в Разделе 2, языки с зависимыми типами обычно задаются через правила вывода. Ниже представлены правила вывода для языка Bool с зависимыми типами.

$$\frac{}{\Gamma \vdash Bool} \qquad \frac{}{\Gamma \vdash True : Bool} \qquad \frac{}{\Gamma \vdash False : Bool}$$

$$\frac{\Gamma, x : Bool \vdash T \qquad \Gamma \vdash t : Bool \qquad \Gamma \vdash a : T[x := True] \qquad \Gamma \vdash b : T[x := False]}{\Gamma \vdash if(t, T, a, b) : T[x := t]}$$

Для полноты определения языка нужно определить правила для работы с контекстами и правила эквивалентности типов:

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash}, \, x \notin \Gamma \qquad \frac{\Gamma \vdash}{\Gamma \vdash x : A}, \, x : A \in \Gamma$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash a : B}$$

Рис. 1: Правила, которые мы считаем верными во всех языках

Эти четыре правила всегда верны во всех языках, которые мы встречали в литературе, поэтому у нас они верны по умолчанию и их не нужно указывать в языке спецификации. Также подразумевается рефлексивность, симметричность, транзитивность и конгруэнтность равенства.

После описания правил вывода, если мы хотим описать, как этот язык вычислять, мы записываем правила редукции. Ниже представлены оба правила редукции для языка Bool.

$$\frac{\Gamma, x : Bool \vdash T \qquad \Gamma \vdash a : T[x := True] \qquad \Gamma \vdash b : T[x := False]}{\Gamma \vdash if(True, T, a, b) \equiv a : T[x := True]}$$

$$\frac{\Gamma, x : Bool \vdash T \qquad \Gamma \vdash a : T[x := True] \qquad \Gamma \vdash b : T[x := False]}{\Gamma \vdash if(False, T, a, b) \equiv a : T[x := False]}$$

Чем более явно все описано/аннотировано, тем легче реализовывать язык. Поэтому было принято решение типизировать язык спецификации. Чтобы отличать метатипы (типы мета-языка) от типов специфицируемого языка, они носят название *сортов*. В любом языке явно выделяются сорты, которые могут зависеть от термов, и сорты простые.

```
DependentSorts:
  tm, ty
FunctionalSymbols:
  if: (tm, 0)*(ty, 1)*(tm, 0)*(tm,0) -> tm
  bool: ty
  true : tm
  false : tm
```

Listing 1: Конструкции и сорты языка Bool, описанные в языке спецификации

Давайте опишем наш язык Bool уже в языке спецификации.

Спецификация имеет следующие разделы: простые сорты (может отсутствовать), зависимые сорты, конструкции, правила вывода и редукции.

Пользователь должен описать все зависимые и независимые сорты. Термы всегда могут зависеть от термов, поэтому они должны идти в раздел зависимых сортов (они имеют специальный идентификатор 'tm'). Типы могут быть зависимыми, могут быть и независимыми, это происходит по выбору пользователя (идентификатор 'ty'). Например, в языке Bool выше мы сделали выбор — типы зависимые. Также в языке спецификации есть возможность вводить какие-то ещё сорта, которые нужны пользователю.

Затем пользователь задает конструкции, для этого он описывает сорта каждого из аргументов и количество переменных, которые этот аргумент связывает. Затем он пишет сорт возвращаемого выражения.

На вставке 1 представлена частичная спецификация языка Bool. Запись '(сорт, n)' — описывает сорт и связывание, то есть у этого аргумента может в контексте быть на $n$ переменных больше, чем в контексте, где наша конструкция используется.

Привычным примером конструкции со связыванием служит $\lambda$-абстракция из нетипизированного $\lambda$-исчисления, эта конструкция имеет такую сигнатуру $(tm, 1) \to tm$, так как связывает дополнительную переменную внутри себя. Это сделано для того, чтобы показать, что термы внутри $\lambda$ могут иметь контекст шире, чем снаружи.

После этого пользователь должен задать правила вывода, по одной на каждую конструкцию (причина объяснена в Разделе 4.1, в Пункте 6). В языке спецификации каждое правило вывода и редукция начинаются с подразделе forall, в которой пользователь описывает сорта всех метапеременных, которые он будет использовать в правиле вывода/редукции. В этой подразделе требуется именование каждой дополнительной переменной, которая может находится в контексте терма, обозначаемого метапеременной. Это сделано для возможности коррекции простых ошибок пользователя.

Явное указание увеличенных контекстов метапеременных позволяет проверять наличие нужных переменных в контексте, в котором они используются. А указание сор-

```
Axioms:
  T-Bool =
    |--- |- bool def
  T-Fls =
    |--- |- false : bool
  T-Tr =
    |--- |- true : bool
  T-If =
    forall x.T : ty, t : tm, a : tm, b : tm
      |- t : bool, x : bool |- T def,
      |- a : T[x:=true], |- b : T[x:=false]
      |------------------------------------------
      |- if(t, x.T, a, b) : T[x:=t]
```
Listing 2: Правила вывода языка Bool, описанные в языке спецификации

тов — корректность применения функциональных символов. Стоит отметить, что в подразделе forall указывается минимальный набор переменных в контексте метапеременной, то есть все эти переменные (в дополнение к общему для всех) всегда должны находится в контексте метапеременной, на момент её использования, но контекст может быть и шире.

А далее идет описание предпосылок и заключения, также как и в описании через обычные правила вывода выше.

Также стоить отметить, что отступы в языке значительны, а именно, названия правила вывода должны быть на уровень ниже названия раздела, подраздел forall на уровень ниже названия, суждения на уровень ниже подраздела forall (если её нет — на уровень ниже имени). Давайте разберем нотацию вставки 2.

Черта в языке спецификации заменена на '|---', количество дефисов от трёх и больше. Сделано для удобства пользователя, во вставке можно увидеть оба предполагаемых варианта использования. Предпосылки записываются через запятую, запись '_ $\vdash$ ...' не означает, что контекст пуст, а подразумевает '$\Gamma \vdash$ ...', просто не нужно прописывать '$\Gamma$' во всех правилах.

'def' оначает 'определено' и все суждения вида '$\Gamma \vdash T$', где нет отношения типизации справа от '$\vdash$', в нашем языке записываются как '$\vdash Tdef$'. '$\equiv$' записывается как обычное равенство. Типизация и подстановки записываются так же, как обычно. Ограничение на то, что связывание должно быть прописано явно, даёт нам возможность проверить корректность написания подстановок.

Осталось описать редукции языка, это сделано во вставке 3. Единственный незнакомый нам символ здесь это '=>', который означает 'редуцируется к'. В языке спецификации все редукции направленные (описано в Разделе 4.1, в Пункте 2), символ '=>' отражает направление редуцирования.

Также в языке есть возможность помечать правила вывода стабильными отно-

```
Reductions:
  Tr-If =
    forall x.T : ty, t : tm, a : tm, b : tm
      x : bool |- T def,
      |- a : T[x:=true], |- b : T[x:=false]
      |------------------------------------------
      |- if(true, x.T, a, b) => a : T[x:=true]
  Fls-If =
    forall x.T : ty, t : tm, a : tm, b : tm
      x : bool |- T def,
      |- a : T[x:=true], |- b : T[x:=false]
      |------------------------------------------
      |- if(false, x.T, a, b) => a : T[x:=false]
```

Listing 3: Редукции языка Bool, описанные в языке спецификации

```
Axioms:
  ...
  [bool]
  T-If =
    ...
```

Listing 4: Bool-стабильность правила вывода ‘If’, описанная в языке спецификации

сительно конечного числа типов, тогда правило вывода применимо, только если все переменные входящие в терм имеют эти типы. Если список типов пуст, то производится проверка терма на отсутствие свободных переменных в терме. Эта возможность языка продемонстрирована на вставке 4.

## 4.1. Ограничения на спецификации, налагаемые языком

Если не налагать никаких ограничений на спецификации, то пользователь может написать спецификацию, для которой мы не сможем сгенерировать функцию проверки типов. Поэтому вводятся следующие ограничения на спецификации языков:

1. Запрещено перекрытие переменных, которые уже есть в контексте. Это чисто стилистическое ограничение, которое не вносит никаких ограничений на сами языки.

2. Запрещено равенство в заключении правил вывода. Это сделано для определенности каждого шага в проверке типов определяемого языка и связано с тем, что, если мы видим равенство, не ясно в какую сторону идти при редуцировании. Поэтому мы заставляем пользователя пользоваться редукциями, а не равенствами в заключении. Поэтому все редукции направленные.

3. Все аргументы в конструкцию языка в заключении правила вывода должны быть метапеременными — случай содержащий не только метапеременные требует дальнейшего исследования. Ещё и с теми же контекстами, что и в forall — если бы контексты были шире, то сразу же приходилось бы проверять их на возможность удаления лишних переменных.

4. В заключении контекст не должен быть расширен — это ограничение связано с тем, что иначе смысл правила вывода становится странным. А именно: конструкция применима только при введении переменных в контекст.

5. Если в заключении правила вывода написана конструкция возвращающая сорт термов, она обязана быть проаннотирована типом (нельзя просто написать $\vdash f(\ldots)\ def$). Так как иначе становится неясно какой тип возвращать при выводе типа данной конструкции.

6. Правило ввывода для каждой конструкции всегда одно, иначе появляется недетерминированность в проверке типов. Не играет особой роли, так как в данной ситуации возможно сделать недетерминированность при проверке типов. Однако в ходе эксплуатации не возникало нужды в обратном. Понадобилось бы более тщательное обдумывание последствий отсутствия данного ограничения. В будущем возможны изменения.

7. Подстановки разрешены только в метапеременные — в принципе, это слабое ограничение, которое облегчает жизнь при реализации, не ограничивая пользователя. Нам, как разработчикам, не нужно ещё и на уровне мета-языка заботиться о подстановках.

8. Все метапеременные, используемые в предпосылках, должны либо присутствовать в метапеременных заключения, или же должны быть типами какой-либо предпосылки. Иначе попросту не ясно откуда брать эти метапеременные при проверке типов.

9. Если в конструкции языка встречаются метапеременные с контекстами $x_1 \ldots x_k.T$, то должна существовать предпосылка вида $x_1 : S_1 \ldots x_k : S_k \vdash T$ в правиле вывода конструкции. Это сделано для того, чтобы не передавать типы контекстов метапеременных конструкции явно, а выводить их из таких условий. Предпосылки такого вида мы называем *определениями метапеременной.*

10. Если метапеременная является типом предпосылки и не встречается в аргументах функционального символа, то она может использоваться только справа от двоеточия. Таким образом избегаются ситуации связанные с порядком проверки предпосылок языка. А именно: если у нас есть $x : S \vdash t : T,\ x : T \vdash r : S,$

то нужно строить граф зависимостей для предпосылок и использовать порядок полученный в результате его топологической сортировки в генерации кода. (Аналогично с 6.2).

11. Все переменные контекстов определения метапеременных (то есть предпосылки в правиле вывода конструкции) могут использовать только метапеременные левее внутри конструкции в заключении — это связано с тем, что иначе могут возникнуть циклы в определениях метапеременных: S тип с аргументом типа R, R тип с аргументом типа S, S тип с аргументом типа R и т.д.

12. Из-за ослабления условия на метапеременные в Пункте 8, порядок метапеременных неочевиден. Решение данной проблемы и (11) описано в Разделе 6.2.

13. Редукции не учитывают предпосылок при приведении в нормальную форму — предполагается что они не конфликтуют с правилами вывода и проверок корректности термов в правилах вывода достаточно.

14. В редукциях все метапеременные справа от '=>' должны встречаться и слева от него. Иначе непонятно откуда взять эти метаперемнные при формировании правой части редукции.

15. Подстановка запрещена слева от '=>'. Это сделано для возможности сопоставления с образцом при генерации функции приведения в нормальную форму. Также не очень понятно, как восстанавливать исходную метапеременную, до подстановки.

16. Все редукции всегда стабильны. Иначе требует дальнейшего исследования, так как появится требование передачи контекста в функцию нормализации.

## 4.2. Проверки корректности спецификации языка

Все ограничения выше проверяются при обработке спецификации языка.

Также тривиальными проверками, осуществляемыми после парсинга языка, являются:

- Все метапеременные, используемые в правилах вывода/редукциях находятся в контексте, включающем их контекст, описанный в подразделе forall.

- Проверка того, что сорты используемых выражений совпадают с сортами аргументов конструкций.

- Подстановка осуществляется в переменные, которые есть в свободном виде в метапеременной.

- Контексты метапеременных содержат все их переменные.
- Все конструкции языка имеют ассоциированное правило вывода.

# 5. Представление термов

В этом разделе описаны возможные представления термов специфицированного языка и обоснован выбор представления в виде Индексов де Брейна[8] с полиморфной рекурсией. Также описана генерация представления термов.

## 5.1. Традиционные индексы де Брейна

В работе подразумевается реализация языков программирования через описание АСД на Haskell. При реализации функциональных языков одной из первых проблем встающих перед программистом является выбор представления АСД. Также нужно описывать абстракцию терма по переменной, инстанциирование переменной терма каким-то другим термом и проверку термов на равенство, и многие задачи и ошибки в реализации связаны именно с этими операциями.

Одной из задач представления термов является сравнение $\alpha$-эквивалентных термов. *$\alpha$-эквивалентными* называются термы, которые отличаются только в именовании связанных переменных. Например, следующие три терма $\alpha$-эквивалентны:

```
λ x y → y (x z)
λ y x → x (y z)
λ a b → b (a z)
```

Одним из возможных способов представления термов является представление переменных в виде строк. С использованием такого подхода первый приведенный выше терм записывается в виде [Lam "x" (Lam "y" (App "y" (App "x" "z")))]. Проверка равенства этого терма второму терму [Lam "y" (Lam "x" (App "x" (App "y" "z")))] не тривиальна.

Другой проблемой такого представления термов является захват свободных переменных при подстановке. Предположим, мы подставляем первый терм ниже в переменную "z" во втором.

```
λ x → y
λ y → z
λ y → λ x → y = λ y x → y
```

Очевидно, что подставлять в переменную так наивно нельзя, так как "y" стала связанной, хотя не была таковой в первоначальном терме.

Ключевым замечанием является то, что переменные в функциональных языках являются "указателями" на место их связывания — этаким индексом в контекст — и не несут никакой дополнительной информации.

Результат использования этого наблюдения называется индексами де Брейна. А именно: для каждой связанной переменной мы просто пишем расстояние от неё до ближайшего связывания.

Если переписать термы с альфа эквивалентностью выше, то для всех трех термов получим [λ λ→ 1 (2 z)], и проверка на альфа-эквивалентность превращается в проверку на равенство.

Также решается проблема захвата свободных переменных, а именно:

```
λ→  y
λ→  z
λ→  λ→  y = λ λ→  y
```

Как видно "y" остался свободным.

Это представление значительно лучше удовлетворяет нашим требованиям разработчика языков. Мы перешли от [Lam "y" (Lam "x" (App "x" (App "y" "z")))] к [Lam (Lam (App 1 (App 2 "z")))].

Однако общей проблемой обоих представлений является нетипизированность переменных — никто не контролирует построение термов вида [Lam (Lam (App 123 (App 23 "z")))]. Решение этой проблемы описано в разделе 5.2.

## 5.2. Индексы де Брейна на уровне типов

В нашем описании индексов де Брейна в Секции 5.1 мы упомянули, что наивное их использование склонно к ошибкам и не использует систему типов Haskell.

Эту проблему можно решить с помощью полиморфной рекурсии[4]. По сути, каждый раз когда мы абстрагируемся по переменной в представлении де Брейна, мы добавляем единицу ко всем связанным переменным внутри терма. Ключевым наблюдением является то, что мы можем добавлять единицу оборачивая терм в Maybe. Например:

```
data Term a
  = Var a
  | App (Term a) (Term a)
  | Lam (Term (Maybe a))
```

Стоит сказать, что, если добиться некоторой абстрактности представления, это позволит нам генерировать меньше кода. Идея состоит в определении для представлении термов представителя класса Monad, смыслом операции bind будет применение функций к переменным. Через неё можно выразить подстановку.

Поэтому метод выше не очень удобен при кодогенерации, так как представитель подстановка и абстракция будет сильно зависеть от определения Term и нам придется генерировать много кода, специфичного для данного представления.

В той же статье предложен способ превращения этого паттерна программирования в трансформер монад. В последующем коде Maybe заменен на Var, в соответствии со своей семантикой, но отличие только в названии. Все представители классов у Var работают так же, как и у Maybe (Alternative, Functor, Monad и проч.). Также можно заметить, что Scope есть трансформер монад MaybeT.

```
data Var a = B | F a
newtype Scope f a = Scope { fromScope :: f (Var a) }

instance Monad f => Monad (Scope f) where
  return = Scope . return . F
  Scope m >>= f = Scope $ m >>= varAppWithDefault (return B)
     (fromScope . f)

instance MonadTrans Scope where
  lift = Scope . liftM F
```

Теперь мы можем написать общие функции абстрагирования по переменной и подстановки в самую внешнюю переменную терма.

```
abstract :: (Functor f, Eq a) => a -> f a -> Scope f a
abstract x xs = Scope (fmap go xs) where
  go y = y <$ guard (x /= y)

instantiate :: Monad f => f a -> Scope f a -> f a
instantiate x (Scope xs) = xs >>= go where
  go B = x
  go (F y) = return y
```

Функция abstract при совпадении с абстрагируемой переменной, пользуясь определением Alternative Var, возвращает B – что означает связанную переменную. Иначе она, пользуясь определением Applicative Var, возвращает pure, что есть F, то есть повышает индекс переменной.

Функция instantiate просто подставляет в B переменную (так как она наиболее внешняя), иначе понижает индекс переменной.

Теперь при генерации кода нам всего лишь понадобится определить гораздо более простую монаду подстановок для АСД термов, а со связываниями разбирается наш трансформер Scope. Представление и представитель класса Monad, которое нужно генерировать, выглядят теперь так:

```
data Term a
  = Var a
  | App (Term a) (Term a) (Scope Term a)
  | Lam (Term a) (Scope Term a)
```

```
instance Monad Term where
  Var v1 >>= f = f v1
  App v1 v2 >>= f = App (v1 >>= f) (v2 >>= f)
  Lam v1 v2 >>= f = Lam (v1 >>= f) (v2 >>>= f)

(>>>=) :: (Monad f) => Scope f a -> (a -> f b) -> Scope f b
m >>>= f = m >>= lift . f
```

Где оператор (>>>=) работает как обычная композиция с lift нашего трансформера Scope.

Стотить отметить, что этот метод с некоторыми улучшениями использован в библиотеке bound[25]. В подходе выше нужно каждую переменную оборачивать в F индивидуально, в библиотеке использован способ оборачивания поддеревьев АСД терма.

Однако в нашем случае эта оптимизация неприменима, так как ещё одной задачей, которую должно решать представление — является возможность сопоставления с образцом. Например, если у нас есть искусственная редукция вида $app[\lambda(A, x.\lambda(B, y.t)], r) => t[x := r]$, в функции приведения в нормальную форму происходит сопоставление с образцом, а именно левая часть функции нормализации выглядит как-то так nf (App (Lam _ (Scope (Lam _ t)))r)= ... — основная важность этого примера в том, что наше представление позволяет заходить под связывание при сопоставлении с образцом, чего нет в библиотеке bound.

## 5.3. Представление термов

Таким образом использование полиморфной рекурсии для выражения индексов де Брейна дает нам такие преимущества:

- Проверка корректности построения термов на уровне типов (невозможно написать терм Lam 123 в пустом контексте, так как $\lambda$ захватывает только одну переменную).
- Можно абстрагировать это представление, превратив Scope в трансформер монад. Тогда нам остается лишь определить представителя класса Monad для нашего представления термов (bind работает как подстановка), что делается крайне просто с точки зрения кодогенерации.
- Абстрактное представления дает нам обобщенные функции abstract и instantiate, которые абстрагируют переменную и инстанциируют самую внешнюю связную переменную соответственно. Таким образом решается проблема представления подстановок.

- Можно определить обобщенные Show и Eq — не теряем простоты использования более простого представления без полиморфной рекурсии.

- С помощью механизма Deriving Haskell можно получить представителя классов Functor, Traversable и Foldable. Что дает нам функции toList — список свободных переменных терма — и traverse — применить аппликативную функцию к переменным терма.

# 6. Реализация

В этом разделе описана реализация языка спецификации языков с зависимыми типами.

## 6.1. Парсер генераторы

В ходе всей работы использовались генераторы лексических и синтаксических анализаторов alex[2] и happy[10].

Решение использовать именно генераторы синтаксических анализаторов, а не парсер комбинаторы[30] или другие методы синтаксического анализа было обусловлено тем, что прогнозировались частые изменения грамматики вместе с эволюцией языка.

```
Axiom   :   Header '=' '\t' Forall '\t'
            Premise '|---' JudgementNoEq '/t' '/t'
              { Axiom (snd $1) (fst $1) $4 $6 $8 }
        |   Header '=' '\t'
            Premise '|---' JudgementNoEq '/t'
              { Axiom (snd $1) (fst $1) [] $4 $6 }
```

Listing 5: Часть спецификации синтаксического анализатора

Все изменения связанные с грамматикой языка проводились на уровне спецификации АСД. По вставке 5 можно судить, что изменения грамматики непосредственно ложатся на изменения спецификации синтаксического анализатора.

Стоить заметить, что в языке спецификации отступы значительны. Это известная проблема реализации лексического/синтаксического анализа — так как такая грамматика не является контекстно-свободной. В работе была решена с помощью монадического лексического анализатора, который преобразовывал отступы в аналог открывающих и закрывающих скобок.

## 6.2. Проверка корректного использования метапеременных

В Разделе 4 описывался язык и ограничения, налагаемые на спецификации.

Все правила вывода соответствуют конструкции, которую они описывают, и у каждой конструкции есть своё правило вывода. Правила вывода определяют корректность составления конструкции. Конструкция, определяемая правилом вывода, пишется в заключении. При проверке типов мы идем от заключения к предпослыкам, поэтому метапеременные внутри конструкции в заключении передаются в функцию проверки типов вместе с самой конструкцией.

Важно отметить, что язык не обязывает пользователя явно передавать все метапеременные, используемые в правиле вывода, внутри конструкции в заключении. Поэтому метапеременные могут быть не только аргументами определяемой конструкции, но и типами термов предпосылок, но ничем больше — так как их попросту неоткуда будет взять при проверке типов.

Здесь описан алгоритм проверки использования метапеременных в контекстах других метапеременных при их определении. А если конкретнее — проверки того, что метапеременные не используют метапеременных переданных правее в конструкции, которую мы определяем. Это связано с тем, что иначе может возникнуть цикличность в определениях метапеременных.

Общим способом отслеживания ацикличности зависимостей является построение графа зависимостей и проверка его на ацикличность. Ниже будет пояснено почему выбран именно этот способ. Этот алгоритм используется для проверки каждого правила вывода.

Итак, на шаге инициализации алгоритма добавляем все пары из отношения переменных "переменная x находится правее переменной y" ребрами в граф зависимостей переменных (это нужно для проверки ацикличности в дальнейшем).

Вначале рассмотрим алгоритм в предположении того, что все метапеременные переданы внутри конструкции. Тогда единственные места, где должна проводится проверка — это определения метапеременных. То есть предпосылки вида $x_1 : tm_1 \ldots x_k : tm_k \vdash T$. В предпосылке выше из метапеременной $T$ будут исходить стрелки во все метапеременные $tm_i$.

Если же добавить в рассмотрение предпосылки вида: $x_1 : tm_1 \ldots x_k : tm_k \vdash t : T$, которые определяют $T$ и $t$, то мы ещё и добавляем стрелку из $t$ в $T$, так как $T$ используется в определении $t$.

Вообще говоря, все варианты выше линеаризуемы и можно проверять строгий порядок, а не частичный. Приведем последний возможный случай — случай из-за которого введен граф зависимостей — $x_1 : tm_1 \ldots x_k : tm_k \vdash tm : T$. Здесь мы ставим стрелки аналогично первому варианту, но сама метапеременная T не имеет фиксированной позиции в списке аргументов конструкции языка из заключения, так как $tm$ не является метапеременной, вводимой с помощью $T$.

Итак, мы построили граф зависимостей одних метапеременных от других. Для проверки корректности правила вывода мы делаем топологическую сортировку и проверяем, что наш граф является DAG'ом.

## 6.3. Модуль проверки корректности спецификации

Вся проверка корректности проходит внутри монады SortCheckM, которая является стэком монад StateT и Either. Понятно, что Either используется для обработки ошибок.

А State нужен, так как в ходе работы алгоритма постепенно заполняется таблица определений языка спецификации.

```
data SymbolTable = SymbolTable {
  depSorts       :: Set AST.SortName
, simpleSorts    :: Set AST.SortName
, funSyms        :: Map AST.Name AST.FunctionalSymbol
, axioms         :: Map AST.Name Axiom
, reductions     :: Map AST.Name Reduction
, iSymAxiomMap   :: Map AST.Name AST.Name -- intro axioms of
   funSyms
}
```

Listing 6: Структура заполняемая модулем проверки спецификации

Эта структура содержит:

- Множества всех зависимых и независимых сортов термов
- Таблицу определений всех конструкций, которые содержат их арности и сорта
- Таблицы правил вывода и редукции, содержащие их определения
- Таблицу соответствия конструкций их правилу вывода

Изначально заполняются множества зависимых и независимых сортов. Затем происходит проверка и заполнение определения функций.

Сами правила вывода и редукции, ввиду однопроходности синтаксического анализатора, могут быть заполнены изначально некорректно. Все 0-арные конструкции языка и все метапеременные синтаксическим анализатором распознаются как переменные. Это поправляется на этапе рекурсивного обхода переменных. При встрече переменной сперва просматривается таблица конструкций, затем метапеременных правила вывода/редукции. Если ни там, ни в другом месте ничего не находится считается, что это переменная и проверяется на отсутствие перекрытия других переменных из контекста.

Затем проводятся проверки описанные в Разделе 4.1. Эти проверки достаточно очевидно переводятся в код, поэтому описывать здесь их не сочтено нужным.

## 6.4. Генерация кода

Генерация кода происходит с использованием библиотеки haskell-src-exts[27], которая дает нам функции генерации и манипуляции АСД Haskell.

```
emptyCtx :: (Show a, Eq a) => Ctx a
emptyCtx x = Left $ "Variable not in scope: " ++ show x

consCtx :: (Show a, Eq a) => Type a -> Ctx a -> Ctx (Var a)
consCtx ty ctx B = pure (F <$> ty)
consCtx ty ctx (F a)  = (F <$>) <$> ctx a

checkT :: (Show a, Eq a) => Ctx a -> Type a -> Term a -> TC ()
checkT ctx want t = do
  have <- infer ctx t
  when (nf have /= nf want) $ Left $
    "type mismatch, have: " ++ (show have) ++ " want: " ++ (show
       want)
```

Listing 7: Проверка типов и контексты

В виду того, что мы выбрали представление в виде Индексов де Брейна с полиморфной рекурсией, большинство кода для работы с представлением языка и контекстами не зависит от самого языка. От нас требуется только генерация определений 4ех сущностей:

- Представления специфицированного языка
- Представителя класса Monad для представления языка
- Функции infer — вывода типов терма
- Функции nf — приведения в нормальную форму/вычисления терма

Поэтому, мы просто модифицируем написанный от руки модуль LangTemplate.

Всё остальное либо генерируется с помощью Template Haskell[22] — представители классов Traversable[21], Functor, Foldable (Foldable дает нам функцию toList, которая возвращает свободные переменные терма, Traversable позволяет применять функции swap, rem и add к переменным внутри терма), либо написано от руки с вызовами внутри себя функций nf или infer.

Например, именно так работает функция проверки типа: в неё передается контекст, тип и терм; функция вызывает infer от контекста и терма и затем сравнивает нормальные формы типа, переданного в качестве аргумента, и типа, полученного при помощи вызова функции infer (код функции написан во вставке 7).

Представители классов Eq и Show получаются с помощью механизма DeriveEq1, DeriveShow1[26] — так как Term имеет вид $* \to *$, для него можно определить только представителей высших классов[31].

Затем мы просто пишем определения представителей, независящие от представления (см. вставку 8)

```
instance Eq a => Eq (Term a) where (==) = eq1
instance Show a => Show (Term a) where showsPrec = showsPrec1
```

Listing 8: Определение представителей классов Eq и Show для представления АСД

## 6.5. Структура модуля генерации кода

Генерация кода происходит внутри монады GenM, которая является стэком монад: ReaderT SymbolTable (StateT CodeGen (ErrorM)).

```
data CodeGen = Gen{
, decls :: [Decl]
}
```

Listing 9: Структура используемая при кодогенерации

Так как структура заполняемая модулем проверки спецификации не меняется на этапе кодогенерации, она находится внутри монады ReaderT. При кодогенерации происходит генерация деклараций языка Haskell, которые хранятся в виде списка, затем все эти декларации будут добавлены в модуль опредения языка.

Во всей программе происходит генерация определений 4ех сущностей:

- Представления специфицированного языка
- Представителя класса Monad для представления языка
- Функции infer — вывода типов терма
- Функции nf — приведения в нормальную форму/вычисления терма

Из описанных выше сущностей, только генерация последних двух представляет интерес. Так как остальные две, хоть и являются концептуально сложными для понимания, очень просты в генерации.

Из-за того, что функции infer и nf определяются при помощи сопоставления с образцом конструкций языка (или переменных), и каждое сопоставление с образцом считается отдельной декларацией АСД Haskell, можно считать генерацию каждой декларации в некотором смысле отдельной задачей. А именно — все переменные введенные внутри одной декларации могут быть переиспользованы внутри другой.

Поэтому для генерации каждой отдельной декларации функций infer и nf создается внутренняя монада BldRM, которая определена как StateT Q (ErrorM).

```
data Q = Q {
  _count :: Int,
  _doStmts :: [Stmt],
  _juds :: Juds,
```

```
  _metas :: Map.Map MetaVar [(Ctx, Exp)],
  _foralls :: Map.Map MetaVar Sort,
  _funsyms :: Map.Map AST.Name FunctionalSymbol
}

data Juds = Juds {
  _metaTyDefs :: [(MetaVar, Judgement)],
  _notDefsTy :: [(Term, Judgement)],
  _otherJuds :: [Judgement]
}
```

Listing 10: Структура используемая при кодогенерации функций infer и nf

Сама структура Q содержит всю информацию, нужную для генерации определения отдельной декларации функции infer и nf. Так как весь код, который генерируется будет исполнятся внутри монады, можно просто сгенерировать список выражений Haskell, а затем приписать сверху do, и таким образом будет обеспечен порядок выполнения выражений.

Для создания свободных переменных Haskell используется простой счетчик, так как вероятность появления большого количества переменных внутри одной декларации мала.

Все предпосылки делятся на три типа:

1. Вводящие метапеременные — меняют таблицу метапеременных. Удобно иметь соответствующую метапеременную, вводимая данным выражением.

2. Имеющие тип у терма заключения, а значит нужно ещё строит этот тип и проверять его на равенство выведенному. Терм, на равенство которому будет проведена проверка, хранится рядом для удобства.

3. Остальные — просто нужно проверить на определенность.

Это хранится в структуре Q в виде трёх списков предпосылок.

Также существует таблица, где для каждой метапеременной написан её контекст и терм языка Haskell, который ей соответствует в коде. Это нужно для генерации всех других термов специфицированного языка.

Хранится таблица всех метапеременных из подраздела forall и конструкций языка, так как в предпослыках вида ...|− T def и ...|− f (...) def нам нужно знать сорт метапеременной или сорт возвращаемый нашей конструкции, чтобы вернуть его из функции infer.

Для простоты реализации использована библиотека lens[28]. Что позволяет писать следующие функции в Haskell, например при манипуляции State:

```
FRule =
    forall S : ty, t : tm, T : ty
      x:S, y:S |- t : T,
      x:T |- t : bool,
      |- gf(S, (x z).rf(T, (y r).T)) : rf(S, (x z).T)
      |----------------
      |- ff(S, t) def
```

Listing 11: Искусственное правило вывода для конструкции ff

```
appendStmt :: Stmt -> BldRM ()
-- Modify part of the State using a function
appendStmt st = doStmts %= (++ [st])

genCheckMetaEq :: BldRM ()
genCheckMetaEq = do
  ms <- gets _metas
  -- Replaces metas inside State Monad
  metas <~ sequence (genMetaEq <$> ms)
```

А именно: doStmts %= (++ [st]) — позволяет менять состояние, как будто бы мы применили мутирующую функцию к глобальной переменной. А код metas <~ sequence (genMetaEq <$> ms) выглядит так, будто мы просто присваиваем результат монадического вычисления в “глобальную” переменную metas.

## 6.6. Построение термов

Внутри кодогенерации функций nf и infer мы должны уметь строить термы языка, который мы проверяем/редуцируем. В функции infer это связано с тем, что при проверке предпосылок и возврате типа конструкции мы должны уметь строить произвольные термы специфицированного языка. В функции nf это связано с тем, что правая часть редукции может содержать произвольные термы языка.

Итак, на данной стадии работы алгоритма у нас имеется ассоциативный массив сопоставляющий метаперенные с выражениями на Haskell, которые им соответствуют (это всегда просто переменные языка). Рассмотрим дальнейший ход действий на примере. Предположим дана аксиома ff (см. вставку 11).

На момент вызова у нас есть S и t в пустом контексте. Мы уже отсортировали предпосылки по трем группам описанным в 6.5. Сперва нам нужны предпослыки вводящие метапеременные. Поэтому первой предпосылкой которую мы проверяем является x:S, y:S |– t : T. Чтобы поучить терм T мы должны вызвать функцию вывода типов в контексте, который нам передан, расширенном двумя вхождениями типа S. Ещё мы должны проверить, что эти расширения определены, то есть вызывать функции infer в постепенно увеличивающихся контекстах с термом, который мы хотим добавить в контекст в качестве аргумента.

Для этого мы должны расширить контекст терма t. Это является ограничением, наложенным на нас нашим же представлением, так как иначе у нас не сойдутся типы. Также могло случится так, что мы должны были бы переставить наши переменные в контексте.

Затем, получив нашу переменную в увеличенном контексте (в forall она имеет контекст меньшей длины), до того как мы добавим переменную, которая является её представителем в коде Haskell, мы должны уменьшить её контекст до того, что указан в forall[1].

Затем мы проверяем вторую предпослыку, аналогично описанному выше способу. В третей предпосылке мы должны строить терм `gf(S, (x z).rf(T, (y r).T))`. Это делается рекурсивно внутри монады кодогенерации, чтобы мы имели доступ к нашему ассоциативному массиву метапеременных. В данном примере нам потребуется из x.T получить (x z).T и (x z y r).T. Затем все аналогично.

Но при получении типа мы должны проверить его на равенство типу `rf(S, (x z).T)`. Который мы строим аналогично предыдущему описанию, затем вызываем функцию проверки равенства типов, на терме полученном при выводе типа `gf(S, (x z).rf(T, (y r).T))` и построенном из метапеременных терма `rf(S, (x z).T)` (см. вставку 14).

Одной из проблем индексов де Брейна является их жесткая привязка к порядку переменных в контексте. Действительно чтобы переставить аргументы терма `[Lam "y" (Lam "x" (App "x" (App "y" (App "y" "y"))))]` мы всего лишь меняем их местами в моменты их связывания и получаем `[Lam "x" (Lam "y" (App "x" (App "y" (App "y" "y"))))]`. Однако схожая операция для представления с использованием индексов де Брейна выливается в обход всего терма(!) [Lam (Lam (App 1 (App 2 (App 2 2))))] превращается в [Lam (Lam (App 2 (App 1 (App 1 1))))].

Но если уж пользователь так написал спецификацию, что мы имеем терм с другим порядком переменных или терм с большим их количеством, то мы должны поменять эти переменные местами и даже попытаться удалить лишние переменные.

Например, чтобы привести "(x y z).T" к "(z x).T". Мы должны удалить "y" и переставить "x" и "z" местами.

Так же мы поступаем при возможном расширении контекста нашей метапеременной, например имеем "S" и хотим построить "Lam A x.S" — здесь нужна метаперемен-

[1]Не обязателен тот же порядок контекста, тк мы все равно о нём заботимся во время построения термов, то есть мы можем хранить в массиве представление метапеременной (z y x).T вместо (x y z).T. Главное чтобы это было указано в структуре, которую мы храним.

```
swap1'2 :: Var (Var  a) -> Identity (Var (Var  a))
swap1'2 (B ) = pure (F (B ))
swap1'2 (F (B )) = pure (B)
swap1'2 x = pure x

rem2 :: Var (Var a) -> TC (Var a)
rem2 B = pure B
rem2 (F B) = Left "There is var at 2"
rem2 (F (F x)) = pure (F x)

add2 :: Var a -> Identity (Var (Var a))
add2 B = pure $ B
add2 (F x) = pure $ F (F x)
```

Listing 12: Примеры функций

ная “x.S”, мы получаем её добавляя переменную в её контекст.

Решение предлагаемое в данной работе состоит из композиций операций swap_i’j, remove_i и add_i. Каждая операция выполняет traverse терма, который мы меняем. Примеры во вставке 12.

Решение не является оптимальным, так как можно пройти по всему терму единожды и применить все эти операции сразу, но сложность генерации/написания такого кода возрастает значительно.

Для решения этой задачи написан модуль Solver[2].

По сути мы либо имеем больший контекст и из него получаем меньший, либо наоборот. Хотим делать меньше swap’ов.

Рассмотрим случай приведения большего контекста к меньшему, “[x, y, z]” к “[y, x]”. Мы идем справа налево, так как наиболее близкая связанная переменная наиболее правая. Удаляем те переменные которых нет в контексте к которому мы хотим прийти, таким образом обеспечиваем меньше вызовов к разным функциям rem[3]. Затем просто применяем алгоритм insertion sort на оставшихся контекстах. На количестве сгенерированных функций swap это не отразится.

## 6.7. Вывод типов и нормализация

Сам infer работает как описано в Разделе 2.1. Мы последовательно строим каждую предпосылку и вызываем функцию вывода типов или проверки типа выражения на

[2]Стоит отметить что функции swap, rem и add должны быть сгенерированы и для этого ведется подсчёт в монаде кодогенерации путем записи максимального индекса. Следовательно функция swap дороже, так как мы генерируем $C_2^i$ функций. Именно поэтому алгоритм пытается использовать как можно меньше разных функций.

[3]Мы не можем удалить переменную из контекста, если она присутствует в терме. Монада TC обеспечивает обработку ошибок удаления.

```
If-then =
    forall t : tm, t1 : tm, t2 : tm, x.A : ty
      x : bool |- A def,    -- (1)
      |- t1 : A[x:=true],   -- (2)
      |- t2 : A[x:=false],  -- (3)
      |- t : bool           -- (4)
      |--------------------------------
      |- if(x.A, t, t1, t2) : A[x:=t]

infer ctx (If v1 v2 v3 v4) = do
    -- check (4) ---------------------------------------------
    v5 <- infer ctx v2
    checkEq Bool v5
    -- check (3) ---------------------------------------------
    v6 <- infer ctx v4
    checkEq (instantiate False (toScope (fromScope v1))) v6
    -- check (2) ---------------------------------------------
    v7 <- infer ctx v3
    checkEq (instantiate True (toScope (fromScope v1))) v7
    -- check (1) ---------------------------------------------
    checkT ctx TyDef Bool
    checkT (consCtx Bool ctx) TyDef (fromScope v1)
    -- return ------------------------------------------------
    pure (instantiate v2 (toScope (fromScope v1)))
```

Listing 13: Пример правила вывода и части сгенерированной функции infer, соответствующей этому правилу

равенство типу. Термы, которые мы передаем в эти функции, строим последовательно на основе переданных нам в конструкции или полученных при вызове функции вывода типов (подробно описано в Разделе 6.6).

Стоит отметить, что порядок или количество переменных метапеременных, которые нам переданы, могут отличаться от порядка и вида контекста, в котором наша метапеременная должна быть в момент её использования. Эту проблему мы решаем приводя метапеременные к контексту данному в подразделе forall нашего правила вывода методом, описанным в Разделе 6.6.

Функция nf пытается сопоставиться с образцом на терме, если это не выходит, то данная редукция неприменима. Поэтому нужен способ отслеживать какие редукции уже были опробованы, а какие нет. Это делается с помощью специальной структуры данных Cnt, описанной во вставке 15. Эта структура служит в качестве целого числа, которое мы инициализируем количеством редукций применимых к нашей кострукции языка, и каждая несовпавшая редукция вызывает функцию nf с меньшим и меньшим числом, если же мы доходим до нуля, то мы истощили набор применимых редукций — а значит терм находится в нормальной форме.

```
FRule =
    forall S : ty, t : tm, T : ty
      x:S, y:S |- t : T, -- (1)
      x:T |- t : bool,   -- (2)
      |- gf(S, (x z).rf(T, (y r).T)) : rf(S, (x z).T) -- (3)
      |------------------
      |- ff(S, t) def

 infer :: (Show a, Eq a) => Ctx a -> Term a -> TC (Type a)
 infer ctx (Ff v1 v2) = do
     -- check (1) -----------------------------------------
     checkT ctx TyDef v1
     checkT (consCtx v1 ctx) TyDef (rt add1 v1)
     v3 <- infer (consCtx (rt add1 v1) (consCtx v1 ctx))
             (rt add1 (rt add1 v2))
     -- check (3) -----------------------------------------
     v4 <- pure (nf v3) >>= traverse rem1 >>= traverse rem1
     v5 <- infer ctx
             (Gf v1
                (toScope2
                   (Rf (rt add1 (rt add1 v4))
                      (toScope2 (rt add1 (rt add1 (rt add1 (rt
                         add1 v4))))))))
     checkEq (Rf v1 (toScope2 (rt add1 (rt add1 v4)))) v5
     -- check (2) -----------------------------------------
     checkT ctx TyDef v4
     v6 <- infer (consCtx v4 ctx) (rt add1 v2)
     checkEq Bool v6
     -- return --------------------------------------------
     pure TyDef
```

Listing 14: Искусственный пример случая несоответствия контекстов: контекст t нужно сократить до использования в предпосылке.

```
data Cnt = Bot | U (Cnt)
  deriving(Eq, Show)

nf :: (Show a, Eq a) => Term a -> Term a
nf (If v1 v2 v3 v4)
  = nf' (U (U Bot)) (If (nf1 v1) (nf v2) (nf v3) (nf v4))

nf':: (Show a, Eq a) => Cnt -> Term a -> Term a
nf' (U (U _)) al@(If (Scope v1) True v2 v3)
  = case
      do v4 <- pure v1
         v5 <- pure v2
         v6 <- pure v3
         pure v5
      of
        Left _ -> nf' (U Bot) al
        Right x -> nf x
nf' (U _) al@(If (Scope v1) False v2 v3)
  = case
      do v4 <- pure v1
         v5 <- pure v2
         v6 <- pure v3
         pure v6
      of
        Left _ -> nf' Bot al
        Right x -> nf x
nf' _ x = x
```

Listing 15: Приведение в нормальную форму пытается применить все редукции данного функционального символа

Стоит отметить, что такое сопоставление с образцом невозможно с использованием библиотеки bound[25], поэтому был написан модуль SimpleBound с обычными, а не обобщенными, индексами де Брейна.

# Заключение

В рамках данной работы достигнуты следующие результаты:

- Определен язык спецификаций зависимых языков с дальнейшей возможностью генерации алгоритма проверки типов.
- Реализована генерации структур данных представления языка с использованием индексов де Брюйна на уровне типов и функций манипуляции этими структурами.
- Реализованы генерация функций приведения термов специфицированного языка в нормальную форму и проверки типов.

Существует несколько направлений развития данной работы:

- Можно реализовать поддержку определения функций над термами языка. Что даст возможность работать с языком, как это делается обычно, а именно:
  - Определяем некие константы (термы без свободных переменных).
  - Определяем функци, внутри которых можем использовать конструкции языка и функции определенные ранее.
  - Пишем функцию main, которая выполняется нашим языком (все функции и константы должны проходить проверку типов).
- Генерировать ещё и синтаксический анализатор специфицированного языка, чтобы пользователь все действия описанные выше проделывал в отдельном текстовом файле.
- Дать пользователю определять функции на уровне языка спецификации. Чтобы изолировать общие паттерны определения языка в отдельную функцию. Это предлагается для ещё большего удобства работы с языком спецификаций.
- Поддержать возможность композиции спецификации языков — тогда можно будет собирать языки из частей как предложено в [14]. Например: отдельно определяем языки с $\Sigma$, с $\Pi$, с $Bool$, $Nat$ и т.д. И просто добавляем их наверху спецификации. Затем определяем редукции и как они интерактируют между собой, добавляем своих функциональных символов и готово.
- Использовать в качестве IR[13] не АСД на Haskell, а что-то более оптимизированное под выполнение языков.

# Список литературы


[1] Agda programming language. — Access mode: http://wiki.portal.chalmers.se/agda/pmwiki.php (online; accessed: 25.05.2017).

[2] Alex: A lexical analyser generator for Haskell. — Access mode: https://www.haskell.org/alex/ (online; accessed: 25.05.2017).

[3] BNF Converter. — Access mode: http://bnfc.digitalgrammars.com/ (online; accessed: 25.05.2017).

[4] Bird Richard S., Paterson Ross. De Bruijn Notation As a Nested Datatype // J. Funct. Program. — 1999. — Jan. — Vol. 9, no. 1. — P. 77–91. — Access mode: http://dx.doi.org/10.1017/S0956796899003366.

[5] CakeML: A Verified Implementation of ML. — Access mode: https://cakeml.org/ (online; accessed: 25.05.2017).

[6] The Coq proof assistant. — Access mode: https://coq.inria.fr/ (online; accessed: 25.05.2017).

[7] Curry-Howard correspondence. — Access mode: https://goo.gl/TwG1Sk (online; accessed: 25.05.2017).

[8] De Bruijn N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem // INDAG. MATH. — 1972. — Vol. 34. — P. 381–392.

[9] Forsberg Markus, Ranta Aarne. The labelled bnf grammar formalism.

[10] Happy, The Parser Generator for Haskell. — Access mode: https://www.haskell.org/happy/ (online; accessed: 25.05.2017).

[11] Harper Robert, Honsell Furio, Plotkin Gordon. A Framework for Defining Logics // J. ACM. — 1993. — Jan. — Vol. 40, no. 1. — P. 143–184. — Access mode: http://doi.acm.org/10.1145/138027.138060.

[12] Haskell. An advanced, purely functional programming language. — Access mode: https://www.haskell.org/ (online; accessed: 25.05.2017).

[13] Intermediate representation. — Access mode: https://en.wikipedia.org/wiki/Intermediate_representation (online; accessed: 25.05.2017).

[14] Isaev Valery I. Algebraic Presentations of Dependent Type Theories. — arxiv : math.LO, cs.LO, math.CT/http://arxiv.org/abs/1602.08504v3.

[15] LambdaPi in PLT/Redex. — Access mode: https://github.com/racket/redex/blob/master/redex-examples/redex/examples/pi-calculus.rkt (online; accessed: 25.05.2017).

[16] Martin-Löf Per, Sambin Giovanni. Intuitionistic type theory. — Bibliopolis Napoli, 1984. — Vol. 9.

[17] Milner Robin, Tofte Mads, Harper Robert. The definition of Standard ML // The MIT Press, Massachusetts and London, England. — 1990. — Vol. 199. — P. l.

[18] Pfenning Frank. Logical Frameworks—A Brief Introduction // Proof and System-Reliability / Ed. by Helmut Schwichtenberg, Ralf Steinbrüggen. — Dordrecht : Springer Netherlands, 2002. — P. 137–166. — ISBN: 978-94-010-0413-8. — Access mode: http://dx.doi.org/10.1007/978-94-010-0413-8_5.

[19] Polymorphic recursion. — Access mode: https://en.wikipedia.org/wiki/Polymorphic_recursion (online; accessed: 25.05.2017).

[20] Run Your Research: On the Effectiveness of Lightweight Mechanization / Casey Klein, John Clements, Christos Dimoulas et al. // SIGPLAN Not. — 2012. — Jan. — Vol. 47, no. 1. — P. 285–296. — Access mode: http://doi.acm.org/10.1145/2103621.2103691.

[21] Support for deriving Functor, Foldable, and Traversable instances. — Access mode: https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/DeriveFunctor (online; accessed: 25.05.2017).

[22] Template Haskell. — Access mode: https://wiki.haskell.org/Template_Haskell (online; accessed: 25.05.2017).

[23] The Twelf Project. — Access mode: http://twelf.org/wiki/Main_Page (online; accessed: 25.05.2017).

[24] Weisstein Eric W. Four-Color theorem. — 2002.

[25] bound: Making de Bruijn Succ Less. — Access mode: https://hackage.haskell.org/package/bound (online; accessed: 25.05.2017).

[26] deriving-compat: Backports of GHC deriving extensions. — Access mode: https://hackage.haskell.org/package/deriving-compat (online; accessed: 25.05.2017).

[27] haskell-src-exts: Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer. — Access mode: https://hackage.haskell.org/package/haskell-src-exts (online; accessed: 25.05.2017).

[28] lens: Lenses, Folds and Traversals. — Access mode: `https://hackage.haskell.org/package/lens` (online; accessed: 25.05.2017).

[29] nLab. stable (infinity,1)-category. — Access mode: `https://goo.gl/oRCpVP` (online; accessed: 25.05.2017).

[30] parsec: Monadic parser combinators. — Access mode: `https://hackage.haskell.org/package/parsec` (online; accessed: 25.05.2017).

[31] prelude-extras: Higher order versions of Prelude classes. — Access mode: `https://hackage.haskell.org/package/prelude-extras` (online; accessed: 25.05.2017).

# Приложения

## A. Доказательство корректности функции filter

Ниже показан пример доказательства того, что функция filter выдает подсписок исходного списка. Код написан на Agda[1].

```
data _in_ {A : Set} : List A → List A → Set where
    nil : [] in []
    larger : {y : A} {xs ys : List A} → xs in ys → xs in (y ::
       ys)
    cons   : {x : A} {xs ys : List A} → xs in ys → (x :: xs) in
       (x :: ys)
```

Listing 16: Определяем предикат означающий “список xs является подсписком ys”

```
filter' : {A : Set} → (A → Bool) → List A → List A
filter' p [] = []
filter' p (x :: xs) = if p x then x :: filter' p xs else filter'
   p xs

filterLess : {A : Set} → (p : A → Bool) → (xs : List A) →
   filter' p xs in xs
filterLess p [] = nil
filterLess p (x :: xs) with p x
filterLess p (x :: xs) | false = larger (filterLess p xs)
filterLess p (x :: xs) | true = cons (filterLess p xs)
```

Listing 17: Докажем, что filter xs подсписок xs для любого списка xs

## B. Ссылка на исходный код

Имплементация работы описанной в дипломе находится на репозитории на гитхаб: `github.com/esengie/fpl-exploration-tool`

# C. Спецификация $\lambda\Pi$ с булевыми выражениями и сгенерированный код

lambdaPi.fpl

```
DependentSorts :
  tm, ty
FunctionalSymbols :
  bool : ty
  false : tm
  true : tm
  if : (ty,1)  lam : (ty,0)  app : (tm,0)  pi : (ty,0)Axioms :
  Tr =
    |--- |- true : bool
  Fls =
    |--- |- false : bool
  Bool =
    |--- |- bool def
  If-then =
    forall t : tm, t1 : tm, t2 : tm, x.A : ty
      x : bool |- A def, |- t1 : A[x:=true],
      |- t2 : A[x:=false], |- t : bool
      |--- |- if(x.A, t, t1, t2) : A[x:=t]
  K-Pi =
    forall T1 : ty , x.T2 : ty
      x : T1 |- T2 def |--- |- pi(T1, x.T2) def
  TAbs =
    forall S : ty , x.T : ty , x.t : tm
      x : S |- t : T |--- |- lam(S , x.t) : pi(S , x.T)
  TApp =
    forall t1 : tm , t2 : tm , S : ty, x.T : ty
      |- t1 : pi(S, x.T) , |- t2 : S , x : S |- T def |---- |- app(t1 , t2, x.T)
Reductions :
  Beta =
    forall x.b : tm, A : ty, a : tm, z.T : ty
       |--- |- app(lam(A , x.b), a, z.T) => b[x := a] -- : T[z:=a]
  IfRed1 =
    forall x.A : ty, f : tm , g : tm
      |--- |- if(x.A, true, f, g) => f : A[x:=true]
```

```
  IfRed2 =
    forall x.A : ty, f : tm , g : tm
      |--- |- if(x.A, false, f, g) => g : A[x:=true]
```

---

Lang.hs -- удалены import'ы и лишние функции

```
{-# LANGUAGE TemplateHaskell #-}
import SimpleBound

type TC = Either String
type Ctx a = a -> TC (Type a)

data Term a = Var a
            | TyDef
            | App (Term a) (Term a) (Scope Type a)
            | Bool
            | False
            | If (Scope Type a) (Term a) (Term a) (Term a)
            | Lam (Type a) (Scope Term a)
            | Pi (Type a) (Scope Type a)
            | True
type Type = Term

deriveEq1 ''Term
deriveShow1 ''Term
instance Eq a => Eq (Term a) where (==) = eq1
instance Show a => Show (Term a) where showsPrec = showsPrec1

deriveTraversable ''Term

instance Functor Term where
        fmap = fmapDefault
instance Foldable Term where
        foldMap = foldMapDefault

instance Monad Term where
        Var v1 >>= f = f v1
        App v1 v2 v3 >>= f = App (v1 >>= f) (v2 >>= f) (v3 >>>= f)
```

```
        Bool >>= f = Bool
        False >>= f = False
        If v1 v2 v3 v4 >>= f
          = If (v1 >>>= f) (v2 >>= f) (v3 >>= f) (v4 >>= f)
        Lam v1 v2 >>= f = Lam (v1 >>= f) (v2 >>>= f)
        Pi v1 v2 >>= f = Pi (v1 >>= f) (v2 >>>= f)
        True >>= f = True
        TyDef >>= f = TyDef

checkT :: (Show a, Eq a) => Ctx a -> Type a -> Term a -> TC ()
checkT ctx want t
  = do have <- infer ctx t
       when (nf have /= nf want) $ Left $
          "type mismatch, have: " ++ (show have) ++ " want: " ++ (show want)
checkEq :: (Show a, Eq a) => Term a -> Term a -> TC ()
checkEq want have
  = do when (nf have /= nf want) $ Left $
           "Terms are unequal, left: " ++
             (show have) ++ " right: " ++ (show want)

emptyCtx :: (Show a, Eq a) => Ctx a
emptyCtx x = Left $ "Variable not in scope: " ++ show x

consCtx :: (Show a, Eq a) => Type a -> Ctx a -> Ctx (Var a)
consCtx ty ctx B = pure (F <$> ty)
consCtx ty ctx (F a) = (F <$>) <$> ctx a

infer :: (Show a, Eq a) => Ctx a -> Term a -> TC (Type a)
infer ctx (Var v1) = ctx v1
infer ctx TyDef = throwError $ "Can't have TyDef : TyDef"
infer ctx al@(App v1 v2 v3)
  = do v4 <- infer ctx v2
       v5 <- pure (nf v4)
       v6 <- infer ctx v1
       checkEq (Pi v5 (toScope (fromScope v3))) v6
       checkT ctx TyDef v5
       checkT (consCtx v5 ctx) TyDef (fromScope v3)
       infer ctx v1
       infer ctx v2
```

```
       pure (instantiate v2 (toScope (fromScope v3)))
infer ctx al@Bool = pure TyDef
infer ctx al@False = pure Bool
infer ctx al@(If v1 v2 v3 v4)
  = do v5 <- infer ctx v2
       checkEq Bool v5
       v6 <- infer ctx v4
       checkEq (instantiate False (toScope (fromScope v1))) v6
       v7 <- infer ctx v3
       checkEq (instantiate True (toScope (fromScope v1))) v7
       checkT ctx TyDef Bool
       checkT (consCtx Bool ctx) TyDef (fromScope v1)
       infer ctx v2
       infer ctx v3
       infer ctx v4
       pure (instantiate v2 (toScope (fromScope v1)))
infer ctx al@(Lam v1 v2)
  = do checkT ctx TyDef v1
       v3 <- infer (consCtx v1 ctx) (fromScope v2)
       v4 <- pure (nf v3)
       pure (Pi v1 (toScope v4))
infer ctx al@(Pi v1 v2)
  = do checkT ctx TyDef v1
       checkT (consCtx v1 ctx) TyDef (fromScope v2)
       pure TyDef
infer ctx al@True = pure Bool

nf :: (Show a, Eq a) => Term a -> Term a
nf (Var v1) = Var v1
nf TyDef = TyDef
nf (App v1 v2 v3) = nf' (U Bot) (App (nf v1) (nf v2) (nf1 v3))
nf Bool = Bool
nf False = False
nf (If v1 v2 v3 v4)
  = nf' (U (U Bot)) (If (nf1 v1) (nf v2) (nf v3) (nf v4))
nf (Lam v1 v2) = Lam (nf v1) (nf1 v2)
nf (Pi v1 v2) = Pi (nf v1) (nf1 v2)
nf True = True
```

```
nf' :: (Show a, Eq a) => Cnt -> Term a -> Term a
nf' (U _) al@(App (Lam v1 (Scope v2)) v3 (Scope v4))
  = case
      do v5 <- pure v1
         v6 <- pure v4
         v7 <- pure v3
         v8 <- pure v2
         pure (instantiate v7 (toScope v8))
      of
        Left _ -> nf' Bot al
        Right x -> nf x
nf' (U (U _)) al@(If (Scope v1) True v2 v3)
  = case
      do v4 <- pure v1
         v5 <- pure v2
         v6 <- pure v3
         pure v5
      of
        Left _ -> nf' (U Bot) al
        Right x -> nf x
nf' (U _) al@(If (Scope v1) False v2 v3)
  = case
      do v4 <- pure v1
         v5 <- pure v2
         v6 <- pure v3
         pure v6
      of
        Left _ -> nf' Bot al
        Right x -> nf x
nf' _ x = x

rt f x = runIdentity (traverse f x)
nf1 x = (toScope $ nf $ fromScope x)

data Cnt = Bot | U (Cnt)
  deriving (Eq, Show)
```