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 **4**

1. Постановка задачи **8**

2. Языки с зависимыми типами **9**

2.1. Проверка типов в зависимых языках 9

3. Обзор аналогов **12**

3.1. BNFC 12

3.2. PLT/Redex 12

3.3. Twelf 12

4. Определение языка спецификаций **14**

4.1. Ограничения на спецификации, налагаемые языком 18

4.2. Проверки корректности спецификации языка 20

5. Реализация **21**

5.1. Генераторы синтаксических анализаторов 21

5.2. Проверка корректного использования метапеременных 22

5.3. Модуль проверки корректности спецификации 23

5.4. Представление выражений 24

5.4.1. Традиционные индексы де Брейна 24

5.4.2. Индексы де Брейна на уровне типов 25

5.4.3. Построение выражений 28

5.4.4. Представление выражений 31

5.5. Генерация кода 32

5.6. Структура модуля генерации кода 34

5.7. Вывод типов и нормализация 37

5.7.1. Пример генерации функции вывода типов 37

5.7.2. Пример генерации функции нормализации 40

Заключение **42**

Список литературы **43**

Приложения **46**

A. Доказательство корректности функции filter 46

B. Ссылка на исходный код 46

C. Спецификация $\lambda\Pi$ с булевыми выражениями и сгенерированный код . . 47

Введение

Теории зависимых типов были впервые введены Мартин-Лёфом в семидесятых годах прошлого века[16], затем была написана книга [17]. Одна из основных его мотиваций заключалась в том, чтобы предложить альтернативу теории множеств для формализации математики. Сама теория типов обладает рядом преимуществ перед теорией множеств (описаны в статье [27], а преимущества теорий зависимых типов описаны в [14]), благодаря которым формализация математики становится проще.

Наиболее известными примерами языков с зависимыми типами, которые используются для доказательства математических утверждений являются Coq[6] и Agda[1]. Например, доказательство теоремы о четырех красках было завершено в 2005 году с помощью Coq[7].

Другая область применения теорий зависимых 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[21] под названием CakeML[4]. Пример относительно простой верификации — корректности функции filter на Agda приведен в Приложении А.

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 (подробнее про реализацию в подразделе про реализацию и Разделе 5). 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[10]), 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 для более близкого знакомства с ними.

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[10]. *АСД* — абстрактное синтаксическое дерево, по любому выражению языка можно составить дерево, узлами этого дерева будут конструкции языка, а потомками узла выражения к которым конструк-

ция 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 затем 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.

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.

Такое описание языка достаточно неформально, как языки задаются через формальные правила вывода см. Разделы 2 и 4.

Выше был представлен пример обычного, независимого языка Bool. Чтобы из него сделать зависимый язык Bool, нужно модифицировать конструкцию if-then-else, чтобы она могла принимать зависимые типы. Теперь вторым её аргументом будет функция, возвращающая тип возвращаемого выражения в зависимости от переданного ей аргумента типа Bool — тип зависит от выражения языка, переданного в него. 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 [17].

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 — для них объявляются уровни

аргументов и возвращаемого выражения. Также задаются правила типизации каждой конструкции и правила редукции языка.

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 огрничение 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 её рамки. Одним из применений нестабильности является формализация [32].

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 выражений языка и функций проверки типов и вычисления.

В дальнейшем мы понимаем вычисление как переписывание выражений согласно редукциям языка, пока не получим выражение к которому ни одна редукция неприменима — этот процесс называется приведением выражения 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 выражениях, которое порождается отношением редукции. Это выражается в том, что сравнение выражений (которое сравнивает их с точностью до этого отношения эквивалентности) сначала нормализует выражения, а потом сравнивает их нормальные формы.

Так как типы зависимого языка могут включать в себя произвольные выражения, проверка типов является задачей тесно связанной с вычислением языка. Например, если наша функция принимает только списки длины 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 выражений. Также стоить заметить, что выражения могут иметь достаточно сложную нормальную форму и нам может прийтись сравнивать АСД этих выражений.

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 выражений языка, благодаря более строгой типизации. Рассмотрены несколько вариантов представления (см. Раздел 5.4 для подробного обсуждения этих вариантов):

1. Обычное именованное (переменные представляются в виде строк)
2. Обычные индексы де Брейна[15] (переменные явяются целыми числами, указывающими на место их связывания)
3. Индексы де Брейна с использованием полиморфной рекурсии[18]

У первых двух способов представления есть недостатки. В первом необходимо вводить α-эквивалентность на выражениях — *α-эквивалентными* называются выражения, которые отличаются только в именовании связанных переменных. Также в каждом из этих случаев легко допустить ошибку при работе с выражениями. 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.

В дополнение, третий подход позволяет с большей легкостью генерировать операции над выражениями: 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 выражений 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 проблемы обычно решаются с помощью использования механизма исключений или его аналогов. Однако эту проблему можно решить иначе, просто 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 произвольных выражений языка, именно это и 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 А.

2.1. Проверка типов в зависимых языках

Выражения нашего языка строятся индуктивно из применений конструкций языка к другим конструкциям или переменным. Обычно в тексте под *термами* подразумеваются выражения определенного уровня языка, а именно все выражения, которые не являются типами (или видами).

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

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

Введём несколько определений, которые будем использовать дальше во всем тексте. Все переменные, являясь переменными мета-языка описания языков называются

метапеременными — таким образом мы отличаем их от переменных языка, который мы специфируем. Каждая конструкция вида '... $\vdash$ ...' называется *суждением*. Все, что находится левее символа '$\vdash$', называется *контекстом*. Все суждения, что находятся выше черты, называется *предпосылками*. Суждение под чертой называется *заключением* правила вывода.

Само правило, представленное выше, является правилом вывода применения зависимой функции. Конструкция языка Π принимает в качестве аргумента терм и в зависимости от аргумента возвращает тип, привычные нам независимые функции через Π 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 конструкции, которые мы используем для составления узлов-потомков (предпосылок). На этих узлах вызываем функцию вывода типов в возможно расширенном контексте (каждое расширение контекста должно проверяться на корректность, так как типы в контексте могут зависеть от типов в контексте до них) рекурсивно. Если потомки составлены корректно, то получаем типы потомков, 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 выражений. Нормализацию мы применяем только после того, как убедимся, что выражения корректно составлены. Получается, что нормализация тесно связана с проверкой типов.

Разберём работу алгоритма проверки типов на примере правила вывода 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 его тип S.

3. Расширяем контекст типом S и рекурсивно вызываемся на типе T, таким образом проверяя корректность его определения.

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

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

Можно заметить, что мы должны уметь корректно выполнять подстановки и проверять выражения на равенство, равенство подразумевается до α-эквивалентности. *α-эквивалентными* называются выражения, 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[13]. *Фронтендом* называется комбинация представления АСД и синтаксического и лексического анализаторов языка.

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 языка C или её аналогов.

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

3.2. PLT/Redex

PLT/Redex[22] — 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[12]. Некоторую сложность составляют подстановки — проблема, обойденная в данной дипломной работе благодаря использованию представления выражений языка в виде индексов де Брейна.

3.3. Twelf

Twelf[25] является реализацией LF[8]. Эта программа используется для спецификации и доказательств свойств логик и языков программирования.

В спецификации задаются высказывания языка (используется принцип “высказывания в качестве типов”[20]) и некоторые операции над языком 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 свойства интересующие нас. 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:

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

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

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

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

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

Давайте опишем теперь язык Bool в языке спецификации, попутно поясняя обозначения используемые в нём.

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.

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

Сперва пользователь должен описать все зависимые и независимые сорты. Термы всегда могут зависеть от термов, поэтому они всегда находятся в разделе зависимых сортов (термы имеют специальный идентификатор ‘tm’). Типы могут быть

```
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

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 дополнительные сорты, нужные пользователю — в языке Bool это не понадобится.

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

На вставке 1 представлена частичная спецификация языка Bool. Запись '(сорт, n)' — описывает сорт и связывание аргумента. Что означает, что у этого аргумента может в контексте быть на n переменных больше, чем в контексте, где наша конструкция используется. Разберем это на примере знакомого нам правила вывода:

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

Сорт типа T в аргументе конструкции app '(ty, 1)', так как в правиле выше этот тип определен в контексте шире на единицу, чем стандартный контекст Γ. Спецификация требует, чтобы это было прописано явно, для корректной работы с контекстами.

Ещё одним привычным примером конструкции со связыванием служит λ-абстракция из нетипизированного λ-исчисления, эта конструкция имеет такую сигнатуру '$(tm, 1) \to tm$' в языке спецификации, так как связывает дополнительную переменную внутри себя — выражения внутри конструкции λ могут иметь контекст на одну переменную шире, чем снаружи.

После описания сортов и конструкций языка пользователь должен задать правила вывода, по одной на каждую конструкцию (причина объяснена в Разделе 4.1, в Пункте 5).

Четыре правила вывода в Списке правил представленные ниже всегда верны во всех языках, которые мы встречали в литературе, поэтому в языке спецификации они верны по умолчанию и их не нужно указывать в языке спецификации.

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

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

Правилом вывода конструкции — называется правило вывода такое, что в его заключении наиболее внешней является конструкция ей соответсвуюшая. В языке Bool есть по одному правилу вывода на каждую конструкцию.

В языке спецификации каждое правило вывода и редукции начинаются с подраздела forall, в котором пользователь описывает сорты всех метапеременных, которые он будет использовать в правиле вывода/редукции. В этом подразделе требуется именование каждой дополнительной переменной, которая может находится в контексте метапеременной. Это сделано для возможности коррекции простых ошибок пользователя — например использование метапеременных в меньшем контексте, чем он должен быть. Например в правиле вывода для 'app' подраздел forall выглядел бы так:

```
forall f:tm, t:tm, S:ty, x.T:ty
```

Явное указание увеличенных контекстов метапеременных позволяет проверять наличие нужных переменных в контексте, в котором они используются. А указание сортов — корректность применения функциональных символов. Стоит отметить, что в подразделе forall указывается минимальный набор переменных в контексте метапеременной, то есть на момент её использования в правиле все эти переменные всегда должны находится в контексте метапеременной, однако контекст может быть и шире. То есть мы могли указать в forall выше 'T : ty', однако проверка подстановки бы нас остановила — так как переменной 'x' не было бы в контексте метапеременной. Но если бы в заключении правила не было подстановки, то такая запись была бы корректной — это бы просто означало, что тип T не является зависимым в данном правиле.

Далее в определении правила вывода идет описание предпосылок и заключения, также как и в описании через обычные правила вывода, которое мы писали выше.

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 его нет — на уровень ниже имени). Давайте разберем нотацию вставки 2.

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

'def' оначает 'определено' и все суждения вида 'Γ ⊢ T', где нет отношения типизации справа от '⊢', в нашем языке записываются как '⊢ $T\ def$'.

'≡' записывается как обычное равенство. Также подразумевается рефлексивность, симметричность, транзитивность и конгруэнтность равенства. Типизация и подста-

```
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

```
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

новки 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'. Так как редукции всегда направленные, символ '=>' отражает направление редуцирования.

Также в языке есть возможность помечать правила вывода стабильными относительно конечного числа типов, тогда правило вывода применимо, только если все переменные входящие в выражение имеют эти типы. Если список типов пуст, то производится проверка выражения на отсутствие свободных переменных в выражении. Эта возможность языка продемонстрирована на вставке 4. Таким образом при проверке корректности конструкции, первым делом нужно проверять, что все свободные

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

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

переменные внутри выражения языка имеют типы из списка.

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, они заменены редукциями.

3. В заключении правила вывода может быть только конструкция языка с метапеременными в качестве аргументов. Не должно быть подстановок или метапеременных на верхнем уровне слева от отношения типизации. Неясно как работать с подстановками в заключениях правил вывода, так как на этапе проверки типов знание того, что “нам передается что-то с выполненной в него подстановкой куда-то”, не дает нам никакой дополнительной информации, в сравнении с отсутствием этого знания. Также неясно, что делать с метапеременной переданной в заключении, так как она будет подходить подо все выражения сорта этой метапеременной. Поэтому разрешены только конструкции языка в заключении.

4. Все аргументы в конструкцию языка в заключении правила вывода должны быть метапеременными — случай содержащий не только метапеременные требует дальнейшего исследования. Метапеременные должны быть с теми же контекстами, что и в forall — если бы контексты были шире, то в алгоритме проверки типов сразу же приходилось бы проверять выражения, соответствующие метапеременным, на возможность удаления лишних переменных.

5. Правило вывода для каждой конструкции всегда одно, иначе появляется недетерминированность в проверке типов — неясно какое правило вывода выбрать

при проверке типов. В ходе эксплуатации не возникало нужды в обратном. Понадобилось бы более тщательное обдумывание последствий отсутствия данного ограничения. В будущем возможны изменения.

6. 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.

7. 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.

8. 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 (языка спецификации) заботиться о подстановках.

9. Все метапеременные, используемые в предпосылках, должны либо присутствовать в метапеременных заключения, или же должны быть типами какой-либо предпосылки. Иначе попросту неясно откуда брать выражения, соотвествующие этим метапеременным, при проверке типов языка.

10. Если в заключении правила вывода во вводимой 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 в конструкции явно, а выводить их из такого рода предпосылок. Предпосылки такого вида мы называем *определениями метапеременной*.

11. Если метапеременная является типом предпосылки и не встречается в аргументах конструкции, 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 5.2). Так как иначе нам неоткуда будет взять выражение соответствующее этой метапеременной при проверке типов.

12. Все переменные контекстов определения метапеременных (то есть предпосылки, описанные в Пункте 10) могут использовать только метапеременные, переданные 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. A

13. Из-за ослабления условия на метапеременные в Пункте 9, порядок метапеременных неочевиден. Решение данной проблемы и (12) описано в Разделе 5.2.

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

15. В редукциях все метапеременные справа от '=>' должны встречаться и слева от него. Иначе непонятно откуда взять выражения соответствующие этим метаперемнным при формировании правой части редукции.

16. Подстановка запрещена слева от '=>'. Это сделано для возможности сопоставления с образцом при генерации функции приведения в нормальную форму. Также как и в Пункте 3 не очень понятно, как восстанавливать исходную метапеременную до подстановки.

4.2. Проверки корректности спецификации языка

Все ограничения выше проверяются при обработке спецификации языка.

Также тривиальными проверками, осуществляемыми после синтаксического разбора 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. Реализация

В этом разделе описана реализация языка спецификации языков с зависимыми типами. Подразумевается, что читатель знаком с Haskell на уровне прочтённой книги [5].

5.1. Генераторы синтаксических анализаторов

В ходе всей работы использовались генераторы лексических и синтаксических анализаторов alex[2] и happy[9].

Решение использовать именно генераторы синтаксических анализаторов, а не комбинаторы синтаксических анализаторов[33] 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. При изменении шрамматики мы меняем их в спецификации напрямую, в крайних случаях ещё приходится менять представление АСД языка, возвращаемого анализатором.

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.

Сам синтаксический анализатор очень простой и однопроходный. Поэтому все, что выглядит как переменная, считается переменной. Для того, чтобы отделить метапеременные и конструкции нулевой арности применяется второй проход по структуре, выдаваемой первым проходом.

5.2. 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 предпосылкам, 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 выражений предпосылок, но ничем больше — так как выражения им соответствующие попросту неоткуда будет взять при проверке типов.

В данном разделе 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 $x_1 : tm_1, \dots x_k : tm_k \vdash T$. В предпосылке выше из метапеременной T будут исходить стрелки во все метапеременные tm_i.

Если же добавить в рассмотрение предпосылки вида: $x_1 : tm_1, \dots x_k : tm_k \vdash t : T$, 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 — $x_1 : tm_1, \dots x_k : tm_k \vdash tm : T$. 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.

5.3. Модуль проверки корректности спецификации

Вся проверка корректности проходит внутри монады SortCheckM, которая является стэком монад StateT и Either.

Монада Either используется для обработки ошибок. А монада State нужна, 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
-- rules of constructs
, iSymAxiomMap   :: Map AST.Name AST.Name
}
```

Listing 6: Структура заполняемая модулем проверки спецификации

Эта структура содержит:

- Множества всех зависимых и независимых сортов выражений
- 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 конструкций.

Сами правила вывода и редукции, ввиду однопроходности синтаксического анализатора (упоминалось в Разделе 5.1), 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 их здесь мы не сочли нужным.

5.4. Представление выражений

В этом разделе описаны возможные представления выражений специфицированного языка и обоснован выбор представления, использованного для генерации кода. Также описана генерация представления выражений.

5.4.1. Традиционные индексы де Брейна

Стоит отметить, что в работе подразумевается реализация языков программирования через непосредственное описание АСД на Haskell.

При реализации функциональных языков одной из первых задач встающих перед программистом является выбор представления АСД. Также нужно описывать абстракцию выражения языка по переменной, подстановку в переменные выражения и проверку на равенство выражений, и многие задачи и ошибки в реализации связаны именно с этими операциями.

Одной из задач представления выражений является сравнение α-эквивалентных выражений. *α-эквивалентными* называются выражения, которые отличаются только в именовании связанных переменных. Например, следующие три выражения α-эквивалентны:

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

Одним из возможных способов представления выражений является представление переменных в виде строк. С использованием такого подхода первый приведенное выше выражение записывается в виде [Lam “x” (Lam “y” (App “y” (App “x” “z”)))]. Проверка равенства этого выражения второму выражению [Lam “y” (Lam “x” (App “x” (App “y” “z”)))] не тривиальна.

Другой проблемой такого представления выражений является захват свободных переменных при подстановке. Предположим, мы подставляем первое выражение 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 выражении.

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 места её связывания.

Если переписать выражения из примера с α-эквивалентностью выше, то для всех трёх выражений получим [$\lambda\ \lambda\rightarrow\ \ 1\ (2\ \ ``\mathrm{z}'')$], и проверка на α-эквивалентность превращается в обычную проверку на равенство.

При представлении с помощью индексов де Брейна также решается проблема захвата свободных переменных. Если переписать пример выше, использованный для объяснения этой задачи:

$\lambda\rightarrow\ \ \mathrm{y}$
$\lambda\rightarrow\ \ \mathrm{z}$
$\lambda\rightarrow\ \ \lambda\rightarrow\ \ \mathrm{y}\ =\ \lambda\ \ \lambda\rightarrow\ \ \mathrm{y}$

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 выражений вида [Lam (Lam (App 123 (App 23 ‘‘z’’)))] — где индексы указывают за пределы возможных связываный. Решение этой проблемы описано в разделе 5.4.2.

5.4.2. Индексы де Брейна на уровне типов

В нашем описании индексов де Брейна в Секции 5.4.1 мы упомянули, что наивное их использование склонно к ошибкам и не использует систему типов Haskell.

Эту проблему можно решить с помощью полиморфной рекурсии[19]. По сути, каждый раз когда мы абстрагируемся по переменной в представлении де Брейна, мы добавляем единицу ко всем свободным переменным внутри выражения. Ключевым наблюдением является то, что мы можем добавлять единицу оборачивая выражение 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 представления выражений представителя класса Monad, смыслом операции bind будет применение функций к переменным. Через неё можно выразить подстановку, при этом ниже будет показано, каким образом можно выделить связывания в свой модуль так, чтобы выражения заботились только о подстановке в переменные, а связывания обрабатывались бы в коде этого модуля.

Поэтому метод выше не очень удобен при кодогенерации, так как функции подстановки и абстракции будут сильно зависеть от определения Term и нам придется генерировать много кода, специфичного для каждого представления (полный пример кода для данного представления можно увидеть на `github.com/esengie/cath_lec/blob/master/lec7/tasks7.hs`).

В той же статье[19] 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 (прочитать, как именно это работает для Maybe и MaybeT, можно в книге[5]).

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

toScope :: f (Var a) -> Scope f a
toScope = Scope

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 связанную переменную выражения.

```
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 другой переменной внутри выражения, так как появилось ещё одно связывание между переменной и местов её связывания.

Функция instantiate просто подставляет в B переменную (так как она наиболее внешняя), иначе понижает индекс переменной, так как одно связывание между ней и местов её связывания исчезло.

Теперь при генерации кода нам всего лишь понадобится определить гораздо более простую монаду подстановок для АСД выражений, 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.

Также стоит описать роль функций toScope и fromScope, в виду их частого использования в реализации. Они предназначены для обертки выражений в тип данных Scope и выноса их из него (для удобства введены несколько функций $fromScope_i$ и $toScope_i$, которые являются композициями исходных). Эти функции служат для вхождения под связывания и выхода из них. Тип '$Term\ a \to Term\ (Var\ a)$' — означает, что выражение переходит в тип с контекстом большим на единицу. Например при наличии у нас метапеременной $\Gamma \vdash (xyz).T$, fromScope2 переведёт её вот в такой вид $\Gamma, x, y \vdash z.T$ — важным моментом здесь является, что контекст Γ должен быть расширен соответствующим образом. Для этого есть функция consCtx, которая

имеет тип '$Ctx\ a \to Ctx\ (Var\ a)$', которая занимается переносом связывания в контекст. Таким образом тип выражения следит за количеством свободных переменных в нём — переменные просто переходят в контекст при вхождении под связывания этих переменных.

Стоить отметить, что этот метод с некоторыми оптимизациями использован в библиотеке bound[28]. В подходе выше нужно каждую переменную оборачивать в F индивидуально, в библиотеке использован способ оборачивания поддеревьев АСД выражений.

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 не позволяет сделать библиотека bound.

5.4.3. Построение выражений

Одной из проблем индексов де Брейна является их жесткая привязка к порядку переменных в контексте. Действительно, чтобы переставить аргументы выражения 'Lam ‘‘y’’ (Lam ‘‘x’’ (App ‘‘x’’ (App ‘‘y’’ (App ‘‘y’’ ‘‘y’’))))', нужно всего лишь поменять их местами в моменты 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 выражения(!), и 'Lam (Lam (App 1 (App 2 (App 2 2))))' превращается в 'Lam (Lam (App 2 (App 1 (App 1 1))))'. То есть, все единицы и двойки в выражении поменялись местами.

Но пользователь мог так написать спецификацию языка, что в момент проверки типов или применения редукций у нас есть выражение представляющее метапеременную с переставленным порядком переменных (или с большим или меньшим их количеством), чем в выражении, которое мы строим (пример такого правила обсуждается в Разделе 5.7). Тогда мы должны поменять эти переменные местами, добавить недостающие и даже попытаться удалить лишние переменные.

Разберем пример с удалением переменной, чтобы привести '(y z x).T' к '(y x).T' нужно удалить переменную 'z'. Поясним удаление переменной подробнее, предположим нам дано выражение 'App ‘‘x’’ (App ‘‘y’’ (App ‘‘y’’ ‘‘y’’))' в контексте '[y, z, x]', то есть оно как раз соответсвует нашей метаперемнной '(y z x).T'. Что в пред-

ставлении с использованием индексов де Брейна превращается в ‘App 1 (App 3 (App 3 3))’.

При построении какого-то выражения может понадобиться удалить вторую переменную — для этого нужно понизить индексы связанные за ней на единицу, получим ‘App 1 (App 2 (App 2 2))’ — выражение соответствующее ‘(y x).T’. Однако, если удаляемая переменная есть внутри выражения, построение выражения вынуждено сообщить об ошибке.

При возможном расширении контекста метапеременной алгоритм сходен алгоритму удаления. Например, имеем ‘S’ и требуется построить ‘Lam A x.S’ — здесь понадобится метапеременная ‘x.S’, получаем её путем добавления единицы ко всем переменным которые являются связанными за ней.

Итак, предлагаемое решение состоит из композиций операций $swap_{i,j}$, $remove_i$ и add_i. Где индексы означают переменные, над которыми совершается операция.

Решение не является оптимальным, так как можно пройти по всему выражению единожды и применить все эти операции сразу, но сложность генерации/написания такого кода возрастёт. Поэтому было принято решение остановиться на таком алгоритме, этот момент в реализации обладает потенциалом к оптимизации.

Задача о контекстах

Можно дать алгоритмические постановки этих двух задач — уменьшения контекста и увеличения контекста — абстрагировавшись от технических деталей, связанных с манипуляцией выражений.

Формализуем задачу увеличения контекстов: при подаче на вход двух списков xs и ys таких, что все элементы xs находятся в списке ys, выдать две последовательности применений функций $swap_{i,j}$ и add_i соответственно, которые нужно применить к xs (вначале все $swap_{i,j}$, затем все add_i), чтобы получить ys.

Постановка для удаления и перестановок дается аналогично.

Для решения этой задачи написан модуль Solver[1].

По сути, мы либо имеем больший контекст и из него получаем меньший, либо наоборот. В первом случае мы удаляем все переменные, которые нам не нужны оптимальным способом, затем приводим один контекст к другому аналогом сортировки вставками. Таким образом нам понадобится меньше операций swap. Во втором случае вначале производятся все опреции swap, затем добавляются все недостающие переменные, начиная с конца — с конца по той же причине, что удаление идет в начало.

[1]Стоит отметить, что при генерации реализации языка функции $swap_{i,j}$, $remove_i$ и add_i должны быть сгенерированы, и для этого ведется подсчёт в монаде кодогенерации путем записи максимального номера использованной функции. Именно поэтому алгоритм пытается использовать как можно меньше разных функций.

Рассмотрим случай приведения большего контекста к меньшему, ‘[x, y, z]’ к ‘[y, x]’. Алгоритм идет 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, так как при удалении первого элемента мы можем удалить следующий первый элемент и т.д. Затем просто применяем аналог алгоритма сортировки вставками на оставшихся контекстах — находим первый элемент целевого контекста и ставим его на место, затем второй и т.д.

Техническая часть задачи о контекстах

Осталось описать решение технической стороны задачи манипуляций контекстами выражений в генерируемом языке.

В представлении через индексы де Брейна с полиморфной рекурсией нам понадобится функция ‘traverse’ выражения, который подвергается изменениям. Она применяет аппликативную операцию ко всем внутренним элементам выражения. Если определять представителя класса Traversible через механизм Deriving Haskell[23], то это превращается в обычный обход синтаксического дерева, с применением функции к переменным — чего мы и хотим (более подробное описание представлено в [28]). Примеры функций $swap_{i,j}$, $remove_i$ и add_i приведены во вставке 7. Для того, чтобы сразу получать результат функций $swap_{i,j}$ и add_i написана вспомогательная функция ‘rt’, определяемая так: ‘rt f x = runIdentity (traverse f x)’, так как эти функции всегда применимы к выражению, в отличие от удаления переменной под номерем i.

Если посмотреть на вставку 7 внимательнее, можно заметить, что удаление переменной из выражения, присутствующего в нем, приводит к ошибке. Монада Either обеспечивает обработку ошибок удаления.

Также можно отметить, раз у нас есть Traversable, то мы можем определить представителей Functor и Foldable через функций fmapDefault и foldMapDefault соответственно. Эти функции есть в классе Traversible. Представитель Foldable даёт нам функцию toList, которая просто выдает список свободных переменных выражения.

```
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) -> Either String (Var a)
rem2 B = pure B
rem2 (F B) = Left "There is a 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 7: Примеры функций $swap_{i,j}$, $remove_i$ и add_i

```
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

5.4.4. Представление выражений

В итоге, было выбрано представление описанное в Разделе 5.4.2.

Осталось только описать, как можно реализовать равенство для выражений вида 'Term a'. Как видим, вид этого типа в Haskell '$* \to *$', и для него можно определить только представителей высших классов[34] Eq1 и Show1.

Так как равенство выражений просто структурное, его возможно сгенерировать с помощью Template Haskell[24]. Представители классов Eq и Show получаются с помощью механизма DeriveEq1, DeriveShow1[29].

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

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

- Проверка корректности построения выражений на уровне типов (невозможно написать выражение λ 123 в пустом контексте, так как λ захватывает только одну переменную).
- Можно абстрагировать это представление, превратив Scope в трансформер монад. Тогда нам остается лишь определить представителя класса Monad для нашего представления выражений (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 связанную 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 выражения — и traverse — применить аппликативную функцию к переменным выражения.

Маленькой деталью реализации, оставшейся за кадром, которую стоит упомянуть, является наличие видов всех сортов, которые не являются термами. Это сделано для того, чтобы функция вывода типов могла вернуть типы не термов. А вывод уже типов видов приводит к ошибке.

5.5. Генерация кода

Генерация кода происходит с использованием библиотеки haskell-src-exts[30], которая дает нам функции генерации и манипуляции АСД Haskell.

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 четырёх сущностей:

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

Из описанных выше сущностей, не была описана генерация только последних двух (остальные описаны в Разделе 5.4.4).

Так как большинство сущностей не зависит от реализуемого языка, мы просто модифицируем написанный от руки модуль LangTemplate, в котором уже написаны функции работы с контекстами, функции проверки типов (см. вставку 9) и проверки выражений на равенство. Также написаны заглушки для четырех сущностей, указанных выше.

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

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 9: Проверка типов и контексты

Проверка типов специфицированного языка происходит внутри монады TC, определенной как 'Either String', чтобы можно было сообщать пользователю об ошибках.

Например, так работает функция проверки типов: 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'.

Функция увеличения контекста 'consCtx' используется при проверке типов во время вхождения под связывания, а значит должна увеличивать индексы де Брейна контекста по мере продвижения под связывания выражения. Увеличиваются индексы с помощью fmap. Интерес представляет возвращаемый тип, но это всего лишь означает, что функция теперь может возвращать типы переменных с большими индексами де Брейна — чего мы их хотим при вхождении под связывания. Работает в паре с функциями fromScope и toScope, описанными в Разделе 5.4.2.

Так же работает рекурсивный вызов с вхождением под связывание выражения в функциях 'infer' или 'nf' — увеличение контекста отражается на типах. $Term\ a \to Term\ (Var\ a)$ — означает, что выражение переходит в выражение типа, контекст которого длиннее на единицу.

Всё остальное генерируется с помощью Template Haskell[24] — представители классов Traversable[23], Functor, Foldable (Foldable дает нам функцию toList, которая возвращает свободные переменные выражения, Traversable позволяет применять функции swap, rem и add к переменным внутри выражения). Также представители классов Eq и Show (описано в Разделе 5.4.4).

5.6. 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 10: 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.

Из-за того, что функции infer и nf определяются при помощи сопоставления с образцом конструкций языка (также нужно сопоставляться с образцом с переменными), и каждое сопоставление функции 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 внутренная монада BldRM, которая определена как StateT Q (ErrorM).

```
data Q = Q {
  _count :: Int,
  _doStmts :: [Stmt],
  _metas :: Map.Map MetaVar [(Ctx, Exp)],
  -- These three, below are used for infer only
  _juds  :: Juds,
  _foralls :: Map.Map MetaVar Sort,
  _funsyms :: Map.Map AST.Name FunctionalSymbol
}

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

Listing 11: 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 TC (Either Left), 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. Эти переменные используются для хранения результатов предыдущих вычислений внутри функции.

В структуре ‘Q’ существует таблица, где для каждой метапеременной написана переменная языка Haskell, которая ей соответствует в генерирующемся коде. Каждая метапеременная находится в контексте описанном в подразделе forall правила вывода/редукции. Это нужно для генерации всех других выражений специфицированного языка так, как описано в Разделе 5.4.3.

Поле ‘juds’ нужно только при генерации функции ‘infer’. В нем хранятся предпосылки правила вывода конструкции трёх видов:

1. Вводящие метапеременные — меняют таблицу метапеременных. Соответствующую метапеременную, вводимую данным выражением, удобно хранить вместе с предпосылкой, в которой она вводится. Примером такой предпосылки является ‘$x : S \vdash t : T$’, если метапеременная T не встречается в конструкции в заключении правила вывода.

2. Имеющие какой-то тип у терма справа от ‘$\vdash$’, а значит нужно ещё строит этот тип и проверять его на равенство выведенному. Тип, на равенство которому будет проводиться проверка, хранится рядом для удобства. Пример такой предпосылки: ‘$x : S \vdash tm : type$’, где $type$ какое-то выражение.

3. Остальные — предпосылки, которые просто нужно проверить на определенность. Примером такой предпосылки является ‘$x : S \vdash t\ def$’.

Также в ‘Q’ Хранится таблица всех метапеременных из подраздела forall и конструкций языка, так как в предпослыках вида $\Gamma \vdash T\ def$ и $\Gamma \vdash f(...)\ def$ нам нужно знать сорт метапеременной или сорт возвращаемый нашей конструкцией, чтобы вернуть его из функции ‘infer’.

Для простоты реализации использована библиотека lens[31]. Что позволяет писать функции выглядящие императивно в Haskell, например при манипуляции State во вставке 12.

А именно: doStmts %= (++ [st]) — позволяет менять состояние, как будто бы мы применили мутирующую функцию к глобальной переменной. А код metas $<\tilde{}$

```
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)
```

Listing 12: Примеры “императивного” кода внутри монады State с использованием библиотеки lens

sequence (genMetaEq <$> ms) выглядит так, будто мы просто присваиваем результат монадического вычисления в “глобальную” переменную metas.

```
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
```

Listing 13: Искусственное правило вывода для конструкции ff

5.7. 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 равенство типу. Выражения, которые мы передаем в эти функции, строим последовательно, на основе переданных нам в конструкции в заключении, или полученных при вызове функции вывода типов (подробнее об этом ниже).

Внутри кодогенерации функций nf и infer мы должны уметь строить выражения 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 выражения специфицированного языка. В функции nf это связано с тем, что правая часть редукции может содержать произвольные выражения языка.

5.7.1. Пример генерации функции вывода типов

Итак, на данной стадии работы алгоритма у нас имеется ассоциативный массив сопоставляющий метаперенные с переменными Haskell, которые им соответствуют. Рассмотрим дальнейший ход действий на примере. Предположим нам дано искусственное правило вывода ff (см. вставку 13).

На момент вызова у нас есть S и t в общем для всех контексте ctx. Предположим, мы уже отсортировали предпосылки по трем группам описанным в 5.6. Сперва нам необходимо сгенерировать проверку предпосылок вводящих метапеременные, так как эти метапеременные могут быть использованы в других предпосылках (они не могут быть использованы в предпосылках, задающие другие метапеременные, так как это ограничение нашего языка спецификации, см. Раздел 4.1, Пункт 11). А после этого уже проверяем остальные предпосылки.

(Ниже описан ход работы одного шага функции ‘infer’ для правила ff и того, как работает генерация её кода. Дальнейший текст лучше читать имея вставку 14 перед глазами.)

Поэтому первой предпосылкой которую мы проверяем является ‘x:S, y:S |- t : T’. Чтобы получить выражение соответствующее (x y).T, мы должны вызвать функцию вывода типов в контексте, который нам передан, расширенном двумя вхождени-

ями типа S. Для этого мы должны расширить контекст выражения t. Это является ограничением, наложенным на нас нашим же представлением, так как иначе у нас не сойдутся типы.

Правда мы должны сперва проверить, что эти расширения контекста определены, то есть проверять определенность каждого типа, который мы добавляем в контекст – в данном случае это выльется в два вызова функции проверки типа и один вызов функции ‘infer’, именно это и происходит в ‘check (1)’ во вставке 14. ‘rt add1’ просто добавляет связывание между выражением и переменными контекста как описано в Разделе 5.4.3.

Затем, получив переменную ‘v3’, соответствующую нашей метапеременной в увеличенном контексте $\Gamma, x, y \vdash T$, до того как мы добавим её в таблицу метапеременных кодогенерации, мы должны уменьшить её контекст до того контекста, что указан в forall, а если это невозможно — то оповестить об ошибке проверки типов. Что и происходит в соответствующей строке кода.

После проверки первой мы проверяем вторую предпослыку, аналогично описанному выше способу. В третей предпосылке мы должны построить выражение ‘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’, что мы и делаем.

После этого все аналогично первому примеру, кроме последней неоговоренной детали — 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 типе полученном при выводе типа ‘gf(S, (x z).rf(T, (y r).T))’ и построенном из метапеременных типа ‘rf(S, (x z).T)’ (см. вставку 14, пункт под названием ‘equality check’). В конце работы ‘infer’ мы выдаем тип нашей конструкции.

```
rt f x = runIdentity (traverse f x)
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))

    -- (x y).T --> T --------------------------------------------

    v4 <- pure (nf v3) >>= traverse rem1 >>= traverse rem1

    -- check (2) ------------------------------------------------

    checkT ctx TyDef v4
    v6 <- infer (consCtx v4 ctx) (rt add1 v2)
    checkEq Bool v6

    -- check (3) ------------------------------------------------

    v5 <- infer ctx
            (Gf v1
               (toScope2
                  (Rf (rt add1 (rt add1 v4))
                     (toScope2 (rt add1 (rt add1 (rt add1 (rt
                        add1 v4)))))))))

    -- equality check -------------

    checkEq (Rf v1 (toScope2 (rt add1 (rt add1 v4)))) v5

    -- return ---------------------------------------------------

    pure TyDef
```

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

```
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]
```

Listing 15: Правила редукций для конструкции If

5.7.2. Пример генерации функции нормализации

Функция nf пытается сопоставиться с образцом на выражении, если это не выходит, то данная редукция неприменима. Поэтому нужен способ отслеживать, какие редукции уже были опробованы, а какие нет. Это делается с помощью структуры данных Cnt:

```
data Cnt = Bot | U (Cnt)
```

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 (Bot), то мы истощили набор применимых редукций — а значит выражение находится в нормальной форме.

Стоит отметить, что такое сопоставление с образцом невозможно с использованием библиотеки bound[28], поэтому был написан модуль SimpleBound с обычными, а не обобщенными, индексами де Брейна (это было упомянуто в Разделе 5.4.2).

Рассмотрим работу алгоритма на примере редукций конструкции ‘If’ (см. вставку 15).

(Дальнейший текст лучше читать имея вставку 16 перед глазами.)

Если у конструкции есть редукции, то функция nf вызывает функцию nf’, в которую передает количество возможных редукций для данной конструкции. Также она передает нормализованные внутренние выражения для того, чтобы сопоставление с образцом было корректным (так как в работе алгоритма и при описании редукций языка подразумевается нормальная форма внутренних конструкций).

Строки 6 и 15 соответствуют сопоставлению с образцом левой части редукции. Случай, когда выражение находится в нормальной форме учтен на строке 24.

Внутри каждой функции происходит попытка построения правой части редукции таким же способом, как это происходит в функции вывода типов infer. То есть так же внутри монады ‘TC’. Но теперь в случае ошибки мы просто пытаемся применить следующую редукцию к выражению.

Это можно заметить на строках 13 и 22 — в случае ошибки рекурсивно вызывается функция nf’ с понижением счетчика на единицу.

```
1  nf :: (Show a, Eq a) => Term a -> Term a
2  nf (If v1 v2 v3 v4)
3    = nf' (U (U Bot)) (If (nf1 v1) (nf v2) (nf v3) (nf v4))
4
5  nf' :: (Show a, Eq a) => Cnt -> Term a -> Term a
6  nf' (U (U _)) al@(If (Scope v1) True v2 v3)
7    = case
8        do v4 <- pure v1
9           v5 <- pure v2
10          v6 <- pure v3
11          pure v5
12       of
13         Left _ -> nf' (U Bot) al
14         Right x -> nf x
15 nf' (U _) al@(If (Scope v1) False v2 v3)
16   = case
17       do v4 <- pure v1
18          v5 <- pure v2
19          v6 <- pure v3
20          pure v6
21       of
22         Left _ -> nf' Bot al
23         Right x -> nf x
24 nf' _ x = x
```

Listing 16: Приведение в нормальную форму пытается применить все редукции данного функционального символа

```
nf (Lam v1 v2) = Lam (nf v1) (nf1 v2)

nf1 x = (toScope $ nf $ fromScope x)
```

Listing 17: Приведение в нормальную форму конструкции, у которой нет редукций

В случае применимости редукции мы вызываем функцию nf на правой части редукции. Стоит отметить отличие возвращаемого значения строк 14 и 23 от возвращаемого значения в строке 24. Если нет применимых редукций, то выражение в нормальной форме. Если есть, то правая часть редукции может быть редуцирована дальше.

Во вставке 17 показан пример того, что происходит, если у конструкции нет редукций — нормализуются выражения внутри и возвращается модифицированное выражение.

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 Брейна на уровне типов и функций манипуляции этими структурами.
- Реализованы генерация функций приведения выражений специфицированного языка в нормальную форму и проверки типов.

Существует несколько направлений развития данной работы:

- Реализовать поддержку определения функций над выражениями языка. Что даст возможность работать с языком, как это делается обычно, а именно:
  - Определяем некие константы (выражения без свободных переменных).
  - Определяем функции, 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 все действия, описанные выше, пользователь проделывал в отдельном текстовом файле.
- Дать пользователю возможность 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 спецификаций языков — тогда можно будет собирать языки из отдельных частей, как предложено в [26]. Например: отдельно определяем библиотеку языков с Σ, с Π, с $Bool$, Nat и т.д. Затем просто добавляем нужные в список наследования спецификации языка. Добавляем свои конструкции, определяем дополнительные редукции, определяющие интеракции этих языков между собой и дополнительных конструкций, и работаем с результирующим языком.
- Использовать в качестве промежуточного представления[11] 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] CakeML: A Verified Implementation of ML. — Access mode: https://cakeml.org/ (online; accessed: 25.05.2017).

[5] Christofer Allen, Julie Moronuki. Haskell Programming from first principles. — Gumroad, 2017. — Access mode: http://haskellbook.com.

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

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

[8] Frank Pfenning. Logical Frameworks—A Brief Introduction // Proof and System-Reliability / Ed. by Schwichtenberg Helmut, Steinbrüggen Ralf. — 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.

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

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

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

[12] 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).

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

[14] Michael Shulman. Homotopy Type Theory: A synthetic approach to higher equalities // arXiv preprint arXiv:1601.05035. — 2016.

[15] Nicolaas Govert De Bruijn. 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.

[16] Per Martin-Löf. An intuitionistic theory of types. — 1974.

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

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

[19] Richard Bird, Ross Paterson. 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.

[20] Robert Harper, Furio Honsell, Gordon Plotkin. 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.

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

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

[23] 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).

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

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

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

[27] William Farmer. The seven virtues of simple type theory // Journal of Applied Logic. — 2008. — Vol. 6, no. 3. — P. 267–286.

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

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

[30] 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).

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

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

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

[34] 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 18: 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 19: Докажем, что filter xs подсписок xs для любого списка xs

B. Ссылка на исходный код

Реализация работы, описанной в дипломной работе, 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) : T[x:=t2]
```

```
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)
```
