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

Формальные языки с зависимыми типами могут быть использованы для доказательств свойств кода программы. Также в языке с зависимыми возможно определить типы, аналогичные сущностям некоторой области математики, в которой мы хотим доказывать теоремы, и просто писать термы, таким образом предъявляя доказательства утверждений. Это называется соответствием Карри-Говарда-Ламбека[6] — когда типы языка являются утверждениями, а термы доказательствами этих утверждений. Достоинство данного подхода заключается в том, что проверка доказательств перекладывается на алгоритм проверки типов соответсвующего языка — во многих случаях автоматизированный процесс.

Наиболее известными примерами языков, которые пользуются соответствием Карри-Говарда для доказательства математических утверждений являются Coq[5] и Agda[1]. Например, доказательство теоремы о четырех красках было завершено в 2005 году с помощью Coq[24]. Пример относительно простого доказательства корректности функции filter на Agda приведен в Приложении A.

Основной задачей данной работы является определение языка спецификации зависимых языков, с дальнейшей генерацией представления абстрактного синтаксического дерева этого языка и функций работы с деревом в виде модуля Haskell[11]. Генерируемый модуль содержит функции проверки типов, вычисления термов, работы с контекстами и стандартных операций над термами (такие как подстановка, абстракция, проверка на равенство и возможность работы под связыванием).

Текст данной 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. Например, язык булевых выражений Bool состоит из 4 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 первого аргумента конструкции. Подробнее об этом написано в [15].

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; его конструкции, с описанием уровней 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 метаязыка. 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 задать стабильность правил вывода относительно замкнутых типов, что означает, что данная конструкция применима только в случае наличия свободных переменных только указанных типов. Для формализации некоторых категорий[29] это условие является значительным.

Например, если мы хотим, чтобы конструкцию if-then-else можно было применять только, если все свободные переменные внутри неё имеют тип Bool или являются функциями из Bool в Bool, мы можем проаннотировать соответствующее правило вывода списком типов $[Bool, Bool \to Bool]$.

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.

Для работы со специфицированным языком генерируется структура, представляющая термы в языке. Существует несколько вариантов представления (см. Раздел 5 для подробного обсуждения этих вариантов):

1. Обычное именованное (переменные представляются в виде строк)
2. Обычные индексы де Брейна[7] (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 α-эквивалентность на термах — *α-эквивалентными* называются термы, которые отличаются только в именовании связанных переменных. Во втором варианте возникают сложности с работой под связываниями переменных. В каждом из этих случаев легко допустить ошибку при работе с термами. Третий вариант является модификацией второго, в которой допустить ошибку 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.

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. 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, таким образом его можно генерировать по спецификации, при наличии достаточного количества ограничений на последнюю (подробнее в Разделе 2.1).

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 AST 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 термов, 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 А.

2.1. Проверка типов в зависимых языках

Рассмотрим пример правила вывода:

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

Это правило вывода применения зависимой функции. Конструкция П принимает в качестве аргумента терм и в зависимости от аргумента возвращает тип, привычные нам независимые функции через П 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 АСТ 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 α-эквивалентности. *α-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 некоторые 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[8].

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[14]. 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[17]. Эта программа используется для спецификации и доказательств свойств логик и языков программирования.

В спецификации задаются высказывания языка (используется принцип “высказывания в качестве типов”[10]) 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

Языки с зависимыми тпами обычно определяются через правила вывода (тут можно дать ссылочки на примеры, скажем на Мартин-Лёфа). Приводишь пример с Bool. (Сможешь написать его через правила вывода? Тебе нужно написать правила для переменных, правило, которое в тапл называлось T-Conv, и правила для Bool: Bool, true, false, if, редукции) Потом говоришь, что в нашей формализации по сути нужно вбить эти же правила, кроме первых двух, так как они являются общими для всех языков. Кроме того, иногда явно выписывают правила для отношения эквивалентности на термах, но у нас они тоже неявно подразумеваются. Потом показываешь как вводить эти правила на нашем языке, поясняешь, что означает то или иное обозначение. Отмечаешь какие есть отличия

Вдохновением данной работы послужила статьи [16] и [13]. Поэтому сам язык спецификации выглядит как язык описания алгебраических теорий[1].

А именно, помимо правил вывода у нас есть сорта и функциональные символы. Каждая конструкция в языке — это функциональный символ в логике, а правила вывода и редукции — это аксиомы. Правила вывода говорят, когда некоторый функциональный символ определен. Все функциональные символы являются частичными функциями, поэтому это существенно алгебраические теории, а не просто алгебраические. Частичные потому, что например конструкция if-then-else в качестве первого аргумента принимает только терм типа Bool, иначе она не определена и проверка типов должна уведомлять пользователя о некорректности сформированного терма.

Начнем с примера описания языка с зависимыми типами (рис.1) [18, Глава 2.1]

В этом языке явно выделяются три сорта (можно думать о сортах как о метатипах): виды, термы и типы (правила связанные с видами и само их описание опущены для простоты).

Также явно выделяются примитивы языка (в дальнейшем мы называем их функциональными символами): абстракция, П-типы (функции в языках с зависимыми типами) и аппликация. Легко заметить, что во всех яхыках присутствуют подстановка, контексты, символ ‘:’ означающий, что тип терма слева есть терм справа, и связывание переменных.

Правила вида T-Conv и T-Var всегда верны в зависимых языках, поэтому у нас они есть по умолчанию. Также подразумевается рефлексивность, симметричность, транзитивность и конгруэнтность равенства.

Если принять во внимания все наблюдения выше, то так этот язык будет выгля-

[1] Алгебраическая теория это *сигнатура* — множество сортов и функциональных символов над ними — и набор аксиом — множество уравнений над термами, построенными с помощью типизированных переменных и функциональных символов. Сами функциональные символы тотальны, то есть применимы ко всем представителям данных сортов

λLF

Syntax

t	::=		*terms:*
		x	*variable*
		$\lambda x{:}T.t$	*abstraction*
		$t\ t$	*application*
T	::=		*types:*
		X	*type/family variable*
		$\Pi x{:}T.T$	*dependent product type*
		$T\ t$	*type family application*
K	::=		*kinds:*
		$*$	*kind of proper types*
		$\Pi x{:}T.K$	*kind of type families*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, x{:}T$	*term variable binding*
		$\Gamma, X{::}K$	*type variable binding*

Well-formed kinds $\boxed{\Gamma \vdash K}$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash T :: * \qquad \Gamma, x{:}T \vdash K}{\Gamma \vdash \Pi x{:}T.K} \qquad \text{(WF-PI)}$$

Kinding $\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma \qquad \Gamma \vdash K}{\Gamma \vdash X :: K} \qquad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash T_2 :: *}{\Gamma \vdash \Pi x{:}T_1.T_2 :: *} \qquad \text{(K-PI)}$$

$$\frac{\Gamma \vdash S :: \Pi x{:}T.K \qquad \Gamma \vdash t : T}{\Gamma \vdash S\ t : [x \mapsto t]K} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T :: K \qquad \Gamma \vdash K \equiv K'}{\Gamma \vdash T :: K'} \qquad \text{(K-CONV)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma \qquad \Gamma \vdash T :: *}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash S :: * \qquad \Gamma, x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S.t : \Pi x{:}S.T} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : \Pi x{:}S.T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : [x \mapsto t_2]T} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \equiv T' :: *}{\Gamma \vdash t : T'} \qquad \text{(T-CONV)}$$

Рис. 1: Язык с лямбдой и П-типами

деть в нашем языке спецификации[2]:

```
DependentSorts:
  tm, ty
FunctionalSymbols:
  lam: (ty, 0)*(tm, 1) -> tm
  app: (tm, 0)*(tm, 0)*(ty, 1) -> tm
  pi : (ty, 0)*(ty, 1) -> ty
Axioms:
  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
```

[2]Важно понимать, что запись _ ⊢ не означает, что контекст пуст, если слева ничего не написано, это эквивалентно записи Γ ⊢.

```
            |- 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]
```

Типизирование метапеременных позволяет проверять правильность применения функциональных символов и наличие нужных переменных в контексте. Именованные переменные служат для определения порядка переменных в контексте и не несут какой-то дополнительной информации.

Также в язык была добавлена проверка на c-стабильность — можно помечать аксиомы типами, тогда аксиома применима, только если все переменные входящие в терм являются представителями этих типов. Если список типов пуст, то производится проверка терма на отсутствие свободных переменных.

4.1. Ограничения на спецификации, налагаемые языком

Если рассматривать спецификации как произвольные существенно алгебраические теории, то пользователь может написать спецификацию, для которой мы не сможем сгенерировать функцию проверки типов. Поэтому вводятся следующие ограничения на спецификации языков:

1. Запрещено равенство в заключении аксиом для определенности каждого шага в проверке типов определяемого языка Это связано с тем, что, если мы видим равенство, не ясно в какую сторону идти при редуцировании. Поэтому мы обязываем пользователя пользваться редукциями.

2. Если в заключении аксиомы написан функциональный символ возвращающий сорт термов, он обязан также иметь тип (нельзя просто написать $\vdash f(\ldots)def$). Так как иначе становится неясно какой тип возвращать при выводе типа данного функционального символа.

3. Определения функциональных символов всегда одно, иначе появляется недетерминированность в проверке типов. Не играет особой роли, так как в данном случае можно сделать недетерминированность в проверке. Однако в ходе эксплуатации не возникало нужды в обратном. Понадобилось бы более тщательное обдумывание последствий отсутствия данного ограничения. В будущем вохможны изменения.

4. Подстановки разрешены только в метапеременные — в принципе, это слабое ограничение, которое облегчает жизнь при реализации, не ограничивая пользователя. Нам не нужно ещё и на метауровне заботиться о подстановках.

5. Все метапеременные, используемые в предпосылках, должны либо присутствовать в метапеременных заключения, или же должны быть типами какой-либо предпосылки. Иначе не ясно откуда брать эти метапеременные при проверке типов. Получается, что нужно будет считать, что высказывание с метапеременной верно для любого представителя сорта этой метапеременной.

6. Если в функциональном символе встречаются метапеременные с контекстами $x_1 \dots x_k.T$ должна существовать предпосылка вида $x_1 : S_1 \dots x_k : S_k \vdash T$. Это сделано для того, чтобы не передавать типы контекстов метапеременных функционального символа явно, а выводить их из таких условий.

7. 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).

8. Все переменные контекстов определения метапеременных могут использовать только метапеременные левее внутри функционального символа 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.

9. Из-за ослабления условия на метапеременные в Пункте 5, порядок метапеременных неочевиден. Решение данной проблемы и (8) описано в Разделе 6.2.

10. Редукции не учитывают предпосылок при приведении в нормальную форму — предполагается что они не конфликтуют с аксиомами и проверки в аксиомах достаточно.

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

12. Подстановка запрещена слева от '=>'. Это сделано для возможности сопоставления с образцом при генерации функции приведения в нормальную форму.

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

14. Все аргументы в функциональный символ в заключении аксиомы должны быть метапеременными — случай содержащий не только метапеременные требует дальнейшего исследования. Ещё и с теми же контекстами, что и в forall (не расширенный контекст, не существенное ограничение).

15. В заключении контекст не должен быть расширен — это ограничение связано с тем, что иначе смысл аксиомы становится странным. А именно: функциональный символ применим только при введении переменных в контекст.

Также у нашего языка есть ограничения, налагаемые существенно алгебраическими теориями:

- Все используемые метапеременные должны иметь аннотацию (сорт), то есть присутствовать в секции forall аксиомы/редукции.

- Мы явно специфицируем все сорты, которые используем.

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 секции forall.

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

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

- Контексты метапеременных содержат все их метапеременные.

- Все функциональные символы 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[7] с полиморфной рекурсией.

5.1. Традиционные индексы де Брейна

При реализации функциональных языков одной из первых проблем встающих перед программистом является выбор представления АСТ[3]. Также нужно описывать подстановки, и многие задачи и ошибки в реализации связаны с подстановками.

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

λ x y $\rightarrow$ y (x z)
λ y x $\rightarrow$ x (y z)
λ a b $\rightarrow$ 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 захвата свободных переменных при подстановке. Предположим, мы подставляем первый терм ниже в переменную "z" во втором.

λ x $\rightarrow$ y
λ y $\rightarrow$ z
λ y $\rightarrow$ λ x $\rightarrow$ y = λ y x $\rightarrow$ 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.

[3]В работе подразумевается реализация языков программирования через описание АСТ на Haskell.

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

Также решается проблема захвата свободных переменных, а именно:

$\lambda\to$ y
$\lambda\to$ z
$\lambda\to$ $\lambda\to$ y = λ $\lambda\to$ 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))
```

Однако этот метод не очень удобен при кодогенерации, так как instance Monad будет зависет от определения Term. В той же статье предложен способ превращения этого паттерна программирования в трансформер монад. В коде Maybe заменен на Var, в соответствии со своей семантикой, но отличие только в названии. Также можно заметить, что Scope и 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
```

При кодогенерации нам всего лишь понадобится определить гораздо более простую монаду подстановок для АСТ термов, которые 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
```

Этот метод использован в библиотеке bound[25]. В виду того, что нам часто приходится заходить внутрь связываний (это необходимо при приведении в нормальную форму), обобщенные индексы де Брейна используемые в bound нам не подходят. Это связано с тем, что мы не можем просто сопоставлять с образцом, нам нужно вызывать функцию fromScope, которая работает нетривиально. При реализации описанной выше fromScope соответствует сопоставлению с образцом на терме.

Использование полиморфной рекурсии для выражения индексов де Брейна имеет дополнительные преимущества:

- Проверка корректности построения термов на уровне типов (невозможно написать терм Lam 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 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.

- 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.

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

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[9].

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 1: Часть спецификации синтаксического анализатора

Все изменения связанные с грамматикой языка проводились на уровне спецификации AST.

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.

```
data LangSpec = LangSpec {
  stabilities      :: Stab
, depSortNames     :: [SortName]
, simpleSortNames  :: [SortName]
, funSyms          :: [FunctionalSymbol]
, axioms           :: [Axiom]
, reductions       :: [Reduction]
}

data Axiom = Axiom {
  name        :: Name
, stab        :: Stab
, forallVars  :: [(MetaVar, Sort)]
, premise     :: [Judgement]
, conclusion  :: Judgement
}

data Judgement =
  Statement {
  jContext    :: [(VarName, Term)]
, jTerm  :: Term
, jType  :: Maybe Term     -- def as maybe
} |
  Equality {
  jContext    :: [(VarName, Term)]
, jLeft   :: Term
, jRight   :: Term
, jType  :: Maybe Term -- equality t1 = t2 : Maybe t3
}

data Term = Var VarName
          | Meta MetaVar
          | FunApp Name [(Ctx, Term)]
          | Subst Term VarName Term
    deriving (Eq)
```

Listing 2: АСТ языка спецификации

Синтаксический анализатор выдает АСТ языка спецификации, которое идет на вход алгоритму проверки спецификации[4].

6.2. Проверка корректного использования метапеременных

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

Здесь описан алгоритм проверки использования метапеременных в контекстах других метапеременных. А если конкретнее — проверки того, что метапеременные не используют метапеременных переданных правее в функциональном символе, который мы определяем.

Так как язык не обязывает пользователся явно передавать типы переменных метапеременных, используемых в функциональных символах, метапеременные могут быть не только аргументами определяемого функционального символа, но и типами термов предпосылок.

Вначале рассмотрим алгоритм в предположении того, что все метапеременные переданы нам в функциональный символ. Тогда единственные места, где должна проводится проверка — это определения метапеременных. То есть предпосылки вида $x_1 : tm_1 \dots x_k : tm_k \vdash T$.

Давайте строить граф зависимости и проверять его на ацикличность. В предпосылке выше из T будет исходить стрелки во все метапеременные tm_i.

Если же добавить предпослыки 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 другие варианты линеаризуемы и можно проверять строгий порядок, а не частичный. Приведем последний возможный случай — случай из-за которого введена топологическая сортировка — $x_1 : tm_1 \dots x_k : tm_k \vdash tm : T$. Здесь мы ставим стрелки аналогично первому варианту, но сама метапеременная T не имеет фиксированной позиции в списке аргументов функционального символа из заключения.

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.

[4]Можно сравнить с гораздо более структурированной структурой (см. вставку 3), выдаваемой алгоритмом на выходе.

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

```
data SymbolTable = SymbolTable {
  stabs          :: AST.Stab
, 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 3: Структура заполняемая модулем проверки спецификации

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

Сами аксиомы и редукции, ввиду однопроходности синтаксического анализатора могут быть заполнены изначально некорректно. Все 0-арные функциональные символы и все метапеременные синтаксическим анализатором распознаются как переменные. Это поправляется на этапе рекурсивного обхода переменных. Сперва просматривается таблица функциональных символов затем метапеременных аксиомы-/редукции. Если ни там, ни в другом месте ничего не находится считается, что это переменная и проверяется на перекрытие других переменных.

Затем проводятся проверки описанные в Разделе 4.1. Эти проверки достаточно очевидно ложатся на код, поэтому описывать здесь их не имеет особого смысла.

6.4. Генерация кода

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

Так как большинство кода используемого для проверки не зависит от специфицированного языка, мы просто модифицируем написанный от руки модуль LangTemplate. В нем нужно определить функции приведения в нормальную форму и вывода типов. Также нужно определить тип данных термов и определить монадическое действие на типе данных термов.

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

```
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 4: Проверка типов и контексты

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

Listing 5: Определение представителей классов Eq и Show для представления АСТ

Представители классов Eq и Show получаются с помощью механизма DeriveEq1, DeriveShow1[26] — так как Term имеет видами $* \to *$, для него можно определить только представителей высших классов[31]. Затем мы просто пишем определения независящие от представления (см. вставку 5)

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 6: 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.

При 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],
```

```
  _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 7: Структура используемая при кодогенерации функции infer

Для простоты реализации так же как и в модуле проверки корректности спецификации (Раздел 6.3) использована библиотека lens[28]. Что позволяет писать следующие функции в Haskell, например при манипуляции State (акцент на использовании кода, выглядящего императивно):

```
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)
```

Структура Q содержит всю информацию, нужную для генерации определения функций infer и nf. Так как весь код, который генерируется будет исполнятся внутри монады с возможностью обработки ошибок, можно просто сгенерировать список выражений Haskell, а затем приписать сверху do, таким образом будет обеспечен порядок выполнения выражений.

Для создания свободных переменных 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.

```
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 8: Искусственное правило вывода для конструкции ff

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 секции forall и функциональных символов, так как в предпослыках вида ...|– T def и ...|– f (...) def нам нужно знать сорт метапеременной или сорт возвращаемый нашим функциональным символом, чтобы вернуть его из функции infer.

6.6. Построение термов

На данной стадии работы алгоритма у нас имеется ассоциативный массив метаперенных и связанных с ними переменных внутри функции Haskell. Рассмотрим дальнейший ход действий на примере. Предположим дана аксиома ff (см. вставку 8).

На момент вызова у нас есть S и t в пустом контексте. Мы уже отсортировали предпосылки, так как нам нужны предпослыки вводящие метапеременные. 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[5].

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 11).

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 метапеременная "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 9.

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.

[5]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 9: Примеры функций

Для решения этой задачи написан модуль Solver[6].

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[7]. Затем просто применяем алгоритм insertion sort на оставшихся контекстах. На количестве сгенерированных функций swap это не отразится.

[6]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.

[7]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,
      |- t1 : A[x:=true],
      |- t2 : A[x:=false],
      |- t : bool
      |------------------------------------
      |- if(x.A, t, t1, t2) : A[x:=t]

infer ctx (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)))
```

Listing 10: Пример правила вывода и части сгенерированной функции infer, соответствующей этому правилу

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

Стоит отметить, что порядок или количество переменных метапеременных, которые у нас есть, могут отличаться от порядка и вида контекста, в котором наша метапеременная должна быть. Эту проблему мы решаем приводя метапеременные к контексту данному в секции forall нашего правила вывода методом, описанным в Секции 6.6.

Функция nf пытается сопоставиться с образцом на терме, если это не выходит, то данная редукция неприменима. 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.

```
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

 infer ctx (Ff v1 v2) = do
     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))
     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
     checkT ctx TyDef v4
     v6 <- infer (consCtx v4 ctx) (rt add1 v2)
     checkEq Bool v6
     infer ctx v2
     pure TyDef
```

Listing 11: Искусственный пример случая несоответствия контекстов: контекст t нужно сократить до использования в предпосылке.

```
nf (If v1 v2 v3 v4)
  = nf' (U (U Bot)) (If (nf1 v1) (nf v2) (nf v3) (nf v4))

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

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 [13]. Например: отдельно определяем языки с Σ, с Π, 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[12] не АСТ 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] The Coq proof assistant. — Access mode: `https://coq.inria.fr/` (online; accessed: 25.05.2017).

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

[7] 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.

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

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

[10] 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`.

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

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

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

[14] 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).

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

[16] Palmgren E., Vickers S.J. Partial Horn logic and cartesian categories // Annals of Pure and Applied Logic. — 2007. — Vol. 145, no. 3. — P. 314 – 353. — Access mode: http://www.sciencedirect.com/science/article/pii/S0168007206001229.

[17] 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.

[18] Pierce Benjamin C. Advanced Topics in Types and Programming Languages. — The MIT Press, 2004. — ISBN: 0262162288.

[19] Ploymorphic 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 13: 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 14: 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
```

```haskell
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)
```