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:
аспирант Исаев В. И.

Рецензент:
аспирант 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:
PhD student Valeriy Isaev

Reviewer:
PhD student Anton Podkopaev

Saint-Petersburg
2017

Оглавление

Введение

Формальные языки с зависимыми типами могут быть использованы для доказательств свойств кода программы. Также возможно ввести типы, аналогичные сущностям области математики, в которой мы хотим доказывать теоремы и просто писать термы, таким образом предъявляя доказательства утверждений. Плюс данного подхода заключается в том, что проверка доказательств перекладывается на алгоритм проверки типов — во многих случаях автоматизированный процесс.

Однако может возникнуть ситуация, что конструкции, которыми мы хотим пользоваться, не существуют в языке программирования. Поэтому, если мы хотим переложить верификацию наших высказываний на алгоритм проверки типов, приходится писать свой язык программирования и уже в нем доказывать утверждения.

Решение описанной проблемы — генерация зависимых языков по спецификации конструкций, которые мы хотим от нашего языка, является темой данной работы.

Зависимые языки состоят из конструкций языка и их семантика задается с помощью правил вывода и редукций. В работе использована "типизация" конструкций[1], так же как и в [17].

В спецификация задаются метатипы метапеременных (сорта) и конструкций, сами конструкции, правила вывода каждой конструкции и редукции языка. Всегда есть сорт зависимых термов и сорт (зависимых) типов. Например STLC[14] будет задан так:

[1]Это сделано ещё и потому, что работа основана на [8]. В статье предложено описано языков в качестве существенно лгебраических теорий. У специфицируемого языка есть сигнатура и аксиомы, что транслируется в типизироанные конструкции и правила вывода и редукции.

```
Dependent sorts:
  tm
SimpleSorts:
  ty
FunctionalSymbols:
  lam: (ty, 0)*(tm, 1) -> tm
  app: (ty, 0)*(tm, 0)*(tm, 0) -> tm
  arrow: (ty, 0)*(ty, 0) -> ty
Axioms:
  TAbs =
    forall S : ty , T : ty , x.t : tm
      x : S |- t : T |--- |- lam(S , x.t) : arrow(S, T)
  TApp =
    forall t1 : tm , t2 : tm , S : ty, x.T : ty
      |- t1 : arrow(S, T), |- t2 : S
      |-------------------------------
      |- app(T, t1 , t2) : T
  IArrow =
    forall T1 : ty , T2 : ty
      |--- |- arrow(T1, T2) def
Reductions:
  Beta =
    forall x.b : tm, A : ty, a : tm, T : ty
       |--- |- app(T, lam(A, x.b), a) => b[x := a]
```

Также можно задать c-стабильность правил вывода относительно замкнутых типов, что означает, что данная аксиома применима только в случае наличия свободных переменных только специфицированных типов. Например, если мы хотим, чтобы лямбду можно было применять только, если все свободные переменные внутри неё имеют тип Bool или стрелок из Bool в Bool, мы аннотируем наше правило вывода типом Bool:

```
FunctionalSymbols:
  lam: (ty, 0)*(tm, 1) -> tm
  app: (ty, 0)*(tm, 0)*(tm, 0) -> tm
  arrow: (ty, 0)*(ty, 0) -> ty
  bool: ty
Axioms:
  IBool =
    |--- |- Bool def
  [bool, arrow(bool, bool)]
  TAbs =
    forall S : ty , T : ty , x.t : tm
      x : S |- t : T |--- |- lam(S , x.t) : arrow(S, T)
```

Конечно у нас появится новая конструкция и правило вывода соответствующая нашему новоиспеченному типу. Ещё вероятно мы захотим его населить и т.д.

После проверок спецификации (описанных в Секции 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 нормализации.

Было выбрано представление структур данных в виде индексов де Брейна через полиморфную рекурсию (подробнее в Секции 5.4). Само представление де Брейна имеет ряд преимуществ: альфа-эквивалентность превращается в проверку на равенство и нет проблем с избеганием захвата переменных (подробнее в Секции2.2).

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

- Проверка корректности построения термов на уровне типов (нельзя писать Lam 123, тк лямбда 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 (работает как подстановка), что делается комически просто.
- Абстрагирование представления дает нам функции abstract и instantiate, которые абстрагируют переменную и инстанциируют самую внешнюю связную переменную соответственно.
- 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 более простого представления.

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

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

Listing 1: Представление STLC и представитель класса Monad

Затем происходит генерация функций infer и nf, так как остальные функции — проверки типа терма, печати терма, функции управления контекстами — не зависят от специфицируемого языка.

Функция nf сопоставляет терм с образцом, сгенерированном для каждой левой части редукции. Если происходит совпадение — строит правую часть совпавшей редукции. Если нет совпадения то возвращает конструкцию, которую приняла, предварительно применив себя же ко всем внутренним термам.

Функция infer сопоставляет терм с образцом, наличие правил вывода для всех конструкций гарантирует совпадение хоть с одним образцом. Затем проверяет каждую предпосылку отдельно и возвращает тип терма, который ей был передан, в соответствии с правилом вывода конструкции (если терм переданный не является представителем сорта термов возвращается аналог кайнда * для данного сорта).

Для построения правой части мы должны уметь строить термы из имеющихся у нас метапеременных, это описано в деталях в Секции 5.7.

Затем используется библиотека Haskell.src.exts[20] для замены структур данных и функций заглушек в написанном от руки модуле LangTemplate.

Полученный код компилируется Haskell и на нем можно описывать термы специфицированного языка, его контекст и выводить тип терма или приводить его в нормальную форму.

1. Постановка задачи

Целью данной работы является дизайн и имплементация языка для спецификации языков программирования с зависимыми типами. Ключевые задачи, которые решает работа:

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

2. Зависимые языки

Языки с зависимыми типами позволяют типам зависеть от термов, то есть мы, например, можем иметь тип списков фиксированной длины. Это позволяет нам описывать ограничения налагаемые на использование функций, которые мы пишем.

Одной из наиболее частых ошибок при программировнии на языке вида Haskell является взятие первого элемента пустого списка.

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

Которая легко решается, если мы можем иметь термы языка в типе.

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

Здесь тип явно специфицирует, что функция не принимает термы типа 'Vec a 0'

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

2.1. Проверка типов в зависимых языках

Рассмотрим пример:

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

Если считать, что заключение правила вывода, то проверка типов в любом языке 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[2] 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. То есть имеем, что нормализация тесно связана с проверкой типов. Более того проверка типов невозможна без нормализации термов.

Действительно, чтобы понять что $2 + 3 = 5$, мы должны провести вычисления и убедиться в этом.

[2]Конечно, мы должны для каждого расширения контекста проверять его корректность.

2.2. Индексы де Брейна

При реализации функциональных языков одной из самых сложных частей является написание подстановок. Большинство проблем и ошибок в реализации тоже связано с ней.

Одной из таких проблем является сравнение альфа-эквивалентных термов. Альфа-эквивалентными называются термы, которые отличаются только в именовании связанных переменных. Например, следующие три терма альфа-эквивалентны:

$\lambda\ \mathrm{x}\ \mathrm{y}\ \rightarrow\ \mathrm{y}\ (\mathrm{x}\ \mathrm{z})$
$\lambda\ \mathrm{y}\ \mathrm{x}\ \rightarrow\ \mathrm{x}\ (\mathrm{y}\ \mathrm{z})$
$\lambda\ \mathrm{a}\ \mathrm{b}\ \rightarrow\ \mathrm{b}\ (\mathrm{a}\ \mathrm{z})$

Понятно, что мы сталкиваемся с проблемами при использовании переменных в виде строк, например первый терм сверху выглядел бы как [Lam "x" (Lam "y" (App "y" (App "x" "
И проверка равенства этого терма терму [Lam "y" (Lam "x" (App "x" (App "y" "z")))]
занятие, склонное к ошибкам.

Другой проблемой такого представления термов является избегание захвата переменных при подстановке. Положим, мы подставляем первый терм ниже в переменную "z" во втором.

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

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

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

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

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

Также решается проблема избегания захвата переменных, а именно:

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

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

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 секции 5.4.

3. Обзор аналогов

Построение языков программирования с зависимыми типами по спецификации является задачей достаточно специфичной. Ниже перечислены некотороые инструменты, применяемые в похожих ситуациях (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[5].

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[13] — встроенный DSL на языке Racket, созданный для спецификации и изучения операционных семантик языков программирования. 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 спецификация языков с зависимыми типами занимает столько же усилий, как если бы пользователь писал реализацию языка в Haskell[9]. Большую сложность составляют подстановки — проблема, обойденная в данной дипломной работе благодаря использованию представления термов языка в виде индексов де Брейна.

3.3. Twelf

Twelf[17] является реализацией LF[11]. Используется для спецификации и доказательств свойств логик и языков программирования.

В спецификации задаются высказывания языка (используется принцип "высказывания в качестве типов"[7]) 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

дизайнер, свойства. Затем можно использовать программу описанную в данной дипломной работе для реализации языка.

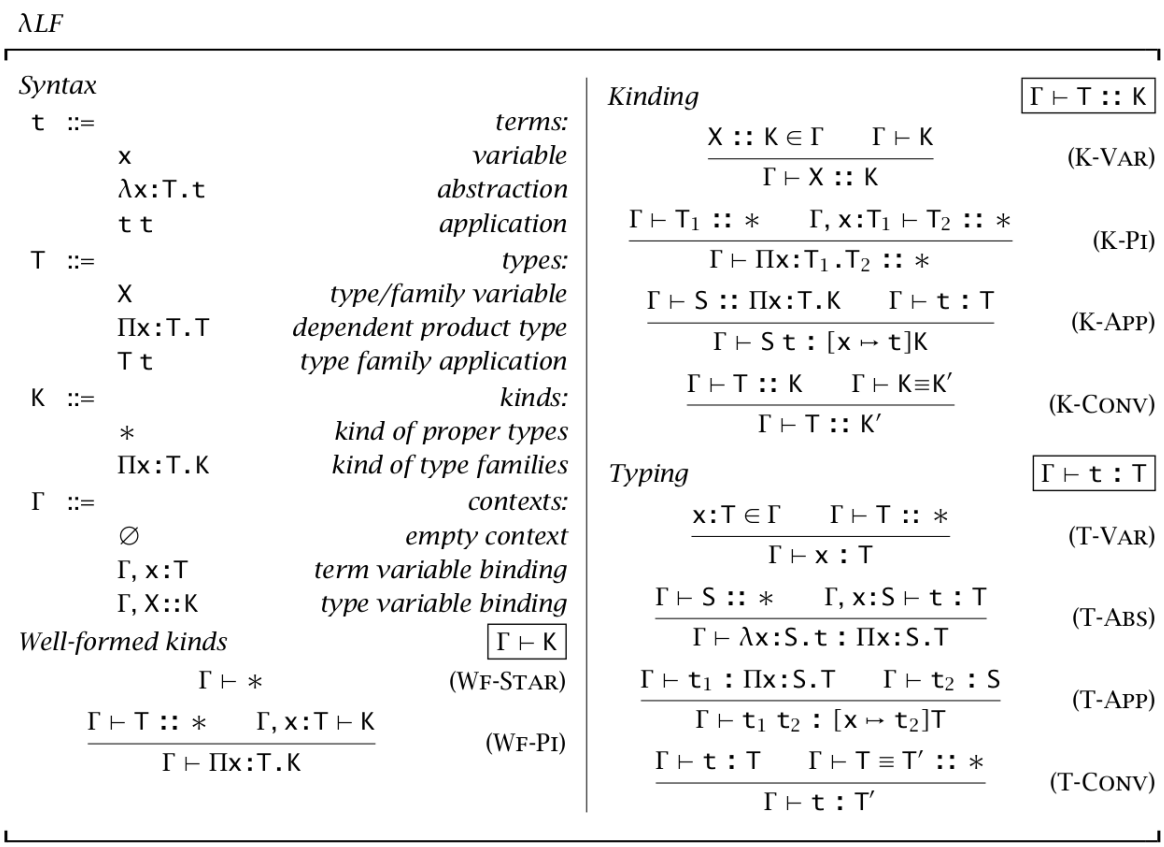

Рис. 1: Язык с лямбдой и П-типами

4. Определение языка спецификаций

Вдохновением данной работы послужила статьи [10] и [8]. Поэтому сам язык спецификации выглядит как язык описания алгебраических теорий.[3]

Начнем с примера описания языка с зависимыми типами (рис.1) [12, Глава 2.1]

В данном нас явно выделяются три сорта (можно думать о сортах как о метатипах): кайнды, термы и типы (правила связанные с кайндами и само их описание опущены для простоты).

Также явно выделяются примитивы языка[4]: абстракция, пи-типы (стрелки в языках без зависимых типов) 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,

[3]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.

[4]В дальнейшем мы называем их функциональными символами.

транзитивность и конгруэнтность равенства.

Если принять во внимания все наблюдения выше то так этот язык будет выглядеть в нашем языке спецификации[5]:

```
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
           |- 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[6].

[5]Важно понимать, что запись $\_ \vdash$ не означает, что контекст пуст, если слева ничего не написано, это эквивалентно записи $\Gamma \vdash$.

[6]Если список типов пуст, то производится проверка 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 тайпчекер. 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 $x_1 \ldots x_k.T$ должна существовать предпосылка вида $x_1 : S_1 \ldots x_k : S_k \vdash T$. 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 5.2).

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...

9. Из-за ослабления условия на метапеременные в Пункте 5, порядок метапеременных неочевиден. Решение данной проблемы и (8) описано в Секции 5.2.

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 с не только метапеременными требует дальнейшего исследования. Ещё и с теми же аргументами, что и в forall (не расширенный котекст, не существенно).

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. Реализация

В этой секции описана реализация языка спецификации языков с зависимыми типами.

5.1. Парсер генераторы

В ходе всей работы использовались генераторы лексических и синтаксических анализаторов alex[2] и happy[6].

Решение использовать именно генераторы синтаксических анализаторов, а не парсер комбинаторы[22] 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 2: 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 3: ACT языка спецификации

Синтаксический анализатор выдает ACT языка спецификации, которое идет на

вход алгоритму проверки спецификации[7].

5.2. Проверка корректного использования метапеременных

В секции 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 пользователься 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).

Остался случай из-за которого введена топологическая сортировка (все другие случаи линеаризуемы и можно проверять строгий порядок, а не частичный) — случай $x_1 : tm_1 \dots x_k : tm_k \vdash tm : T$. Здесь мы ставим стрелки аналогично первому случаю, 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.

5.3. 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.

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

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

```
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 4: Структура заполняемая модулем проверки спецификации

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

Сами аксиомы и редукции, в виду 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 Секции 4.1. Эти проверки достаточно очевидно ложатся на код, поэтому описывать здесь их не имеет особого смысла.

5.4. Индексы де Брейна на уровне типов

В нашем описании индексов де Брейна в Секции 2.2 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[8].

```
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

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 ADT 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
```

[8]в коде Maybe заменен на Var, в соответствие со своей семантикой.

Этот метод использован в библиотеке bound[18]. В виду того, что нам часто приходится заходить внутрь контекстов[9] обобщенные индексы де Брейна используемые в bound нам не подходят. Это связано с тем, что мы не можем просто паттернматчиться, нам нужно вызывать функцию fromScope, которая работает нетривиально. При реализации описанной выше fromScope соответствует паттерматчингу на терме.

5.5. Генерация кода

Генерация кода происходит с использованием библиотеки haskell-src-exts[20], которая дает нам функции генерации и манипуляции ACT 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[16] — instance Traversable[15], Functor, Foldable[10], 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 5: Проверка типов и контексты

Представители классов Eq и Show получаются с помощью механизма DeriveEq1, DeriveShow1[19] — так как Term имеет кайнд $* \to *$, для него можно определить только представителей высших классов[23]. Затем мы просто пишем определения независящие от представления:

[9] Необходимо при приведении в нормальную форму.

[10] Foldable дает нам функцию toList, которая возвращает свободные переменные терма, Traversable позволяет применять функции swap, rem и add к переменным обходя весь терм.

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

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 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 (Секция 5.3) использована библиотека lens[21]. Что позволяет писать следующие функции в Haskell, например при манипуляции State (использование кода, выглядещего 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 вряд ли появится так уж много переменных внутри одной декларации.

Все предпосылки делятся на три типа:

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

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

3. Остальные — просто нужно проверить на определенность.

Это хранится в структуре Q в виде трёх списков предпосылок.

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

Хранится таблица всех метапеременных из 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.

5.7. Построение термов

На данной стадии работы алгоритма у нас имеется ассоциативный массив метапеременных и связанных с ними переменных внутри функции Haskell. Давайте рассмотрим пример, что нужно делать дальше, положим нам дана аксиома ff (см. вставку 8).

```
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: Исскуственное правило вывода для конструкции ff

На момент вызова у нас есть S и t в пустом контексте. Мы уже отсортировали предпосылки (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[11].

Для этого мы должны расширить контекст терма t. Это является ограничением, наложенным на нас нашим же представлением (иначе у нас не сойдутся типы, все логично). 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[12].

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 10).

Одной из проблем индексов де Брейна является их жесткая привязка к порядку переменных в контексте. Действительно чтобы переставить аргументы терма [Lam "y" (Lam "x"
мы всего-лишь меняем их местами в моменты их связывания и получаем [Lam "x" (Lam "y" (Ap
Однако схожая операция для представления с использованием индексов де Брейна

[11]Расширенном двумя вхождениями типа S – мы еще 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.

[12]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.

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 функций:

```
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)
```

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

Для решения этой задачи написан модуль Solver[13].

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 прий-

[13]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.

ти, таким образом обеспечиваем меньше вызовов к разным функциям rem[14]. Затем просто применяем алгоритм insertion sort на оставшихся контекстах. На количестве сгенерированных функций swap это не отразится.

5.8. Вывод типов и нормализация

Сам infer работает как описано в Секции 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 Секции 5.7).

```
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 9: Пример правила вывода и части сгенерированной функции infer, соответствующей этому правилу

Стоит отметить, что порядок или количество переменных метапеременных которые у нас есть могут отличаться от порядка и вида контекста в котором наша ме-

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

тапеременная должна быть. Эту проблему мы решаем приводя метапеременные к контексту данному в секции forall правила вывода, методом описанным в Секции 5.7.

```
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 10: Искусственный пример случая несоответствия контекстов (контекст t нужно сократить до использования в предпосылке)

Функция nf пытается паттернматчиться на терме, если это не выходит, то данная редукция неприменима[15].

[15]Такой паттернматчинг невозможен с использованием библиотеки bound[18], поэтому был написан модуль SimpleBound с обычными, а не обобщенными, индексами де Брейна.

```
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 11: 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 тайпчекера.
- 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 какие-то константы (термы без свободных переменных).
  - Определяем несколько функций, внутри которых можем использовать конструкции языка и функции определенные ранее.
  - Пишем функцию main, которая выполняется нашим языком (понятно что 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 [8]. Например: отдельно определяем языки с Σ, с Π, с $Bool$, Nat и т.д. И просто добавляем их наверху спецификации. Затем определяем редукции как они интерактируют между собой, добавляем своих функциональных символов и готово.

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

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

[7] 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.

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

[9] 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).

[10] 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.

[11] 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.

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

[13] 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.

[14] Simply Typed Lambda Calculus. — Access mode: https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus (online; accessed: 25.05.2017).

[15] 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).

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

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

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

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

[20] 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).

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

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

[23] 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 :: y
```

Listing 12: 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 x
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 13: 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 :
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

```
Тут типо код
```