Реферат

Пояснительная записка содержит XXX страниц (из них XX страниц приложений). Количество использованных источников – XX. Количество приложений – X.

Ключевые слова: ….

Целью данной работы является …

В первой главе проводится обзор и анализ …

Во второй главе описываются использованные и разработанные/модифицированные методы-/модели/алгоритмы ….

В третьей главе приводится описание программной реализации и экспериментальной проверки ….

В приложении А описаны основные требования к форматированию пояснительных записок к дипломам и (магистерским) диссертациям.

В приложении **??** представлена общая структура пояснительной записки.

В приложении **??** приведены некоторые дополнительные комментарии к использованию данного шаблона.

Содержание

Введение

Введение всегда содержит краткую характеристику работы по следующим аспектам:

- актуальность:
  - кто и почему в настоящее время интересуется данной проблематикой (в т.ч. для решения каких задач могут быть полезны исслелования в данной области),
  - краткая история вопроса (в формате год-фамилия-что сделал),
  - нерешенные вопросы/проблемы;
- новизна работы (что нового привносится данной работой);
- оригинальная суть исследования;
- содержание по главам (по одному абзацу на главу).

Общий объем введения должен не превышать 1,5 страниц (для ПЗ к УИРам может быть чуть меньше).

1. Анализ проблематики

Программные ошибки могут иметь серьезные последствия, поэтому, при проектировнии программ с повышенными требованиями надежности используют формальную верификацию — формальное доказательство соответствия предмета верификации его формальному описанию. Предметом здесь, например, могут являться алгоритмы, цифровые схемы.

Формальная верификация является трудозатратным процессом, который не всегда оправдан. Для упрощения данного процесса используются различные системы автоматических доказательств. Под верификацией в данном случае понимается доказательство того что программа соответствует формальному описанию и что в программе отсутствуют ошибки [1].

Существует ряд средств которые широко используются для автоматического доказательства теорем и автоматической верификации программ: ACL2, Coq, Isabelle/HOL, Twelf, PVS.

Система Coq является программным комплексом, предназначенный для формализации и проверки правильности математических рассуждений. Она представляет собой логическую среду, позволяющую описывать ма- тематические теории и в интерактивном полуавтоматическом режиме строить доказательства (формальные выводы). В основе системы ле- жат интуиционистская логика и теория типов CIC (Calculus of Inductive Constructions), что позволяет строить конструктивные доказательства и извлекать из них соответствующие алгоритмы в виде верифицированных программ (поддерживаются языки функционального программиро- вания OCaml, Haskell и Scheme). При этом правильность построенных доказательств проверяется автоматически посредством сведения к зада- че проверки правильности типизации термов в системе CIC[2].

1.1 Формальные основы системы Coq

Coq — интерактивное средство доказательства теорем, использующее свой функциональный язык Gallina с зависимыми типами. Формальной основой Coq является исчисление индуктивных конструкций (CIC), которое, в свою очередь, является расширение исчисления конструкций (CoC). Исчисление конструкций — типизированное лямбда -исчисление высшего порядка, разработанное Терри Коквандом.

1.1.1 Исчисление конструкций

Исчислений конструкций (CoC) находится на вершине лямбда-куба — наглядной классификации типизированных лямбда- исчислений с явным приписыванием типов. Куб организован в

соответствии с возможными зависимостями между типами и термами этого исчисления[3].

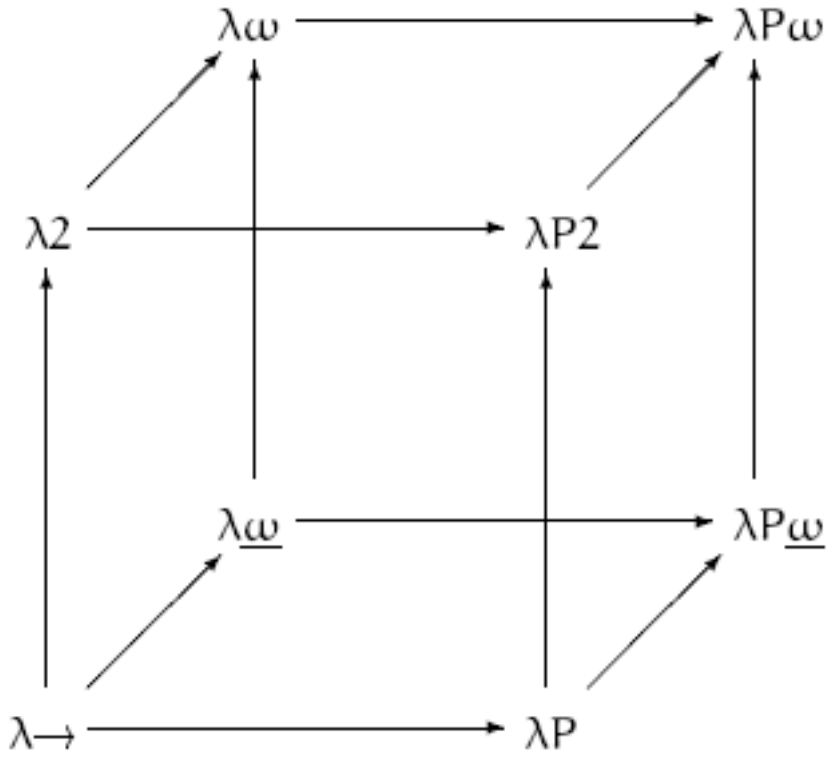

Рис. 1.1 – Лямбда-куб

Базовой вершиной куба служит система $\lambda^{\rightarrow}$, соответствующая просто типизированному лямбда-исчислению. Термы (элементы сорта $*$) зависят от термов; типы (элементы сорта $\square$) в зависимостях не участвуют.

Три оси, выходящие из базовой вершины, порождают следующие системы:

– термы, зависящие от типов: система $\lambda 2$ (лямбда-исчисление с полиморфными типами, система F);
– типы, зависящие от типов: система $\lambda\underline{\omega}$ (лямбда-исчисление с операторами над типами);
– типы, зависящие от термов: система λP (лямбда-исчисление с зависимыми типами).

Остальные системы представляют собой различные комбинации перечисленных зависимостей. Система $\lambda P\omega$ (полиморфное λ-исчисление высшего порядка с зависимыми типами) фактически представляет собой исчисление конструкций. Далее рассмотрим подробнее системы $\lambda 2$, $\lambda\underline{\omega}$ и λP которые в совокупности представляют собой чистое исчисление конструкций.

Полиморфные типы.

Система F — полиморфное λ-исчисление — система типизированного лямбда-исчисления, отличающаяся от просто типизированной системы наличием механизма универсальной квантификации над типами[3].

Систему F иногда так-же называют лямбда-исчислением второго порядка (λ 2), поскольку по соответствию Карри-Говарда она аналогична интуиционистской логике второго порядка, в которой разрешена квантификация не только по отдельным объектам (термам), но и по предикатам (типам)[3].

Определение Системы F является естественным расширением $\lambda_{\to}$, простого типизированного лямбда-исчисления. В $\lambda_{\to}$ λ-абстракция служит для абстрагирования типов из термов, а с помощью применения вместо абстрагированных частей подставляются значения.

Мотивацией для подобного рода расширения простого типизированного λ- исчисления послужила следующая ситуация — определенное поведение применимо к аргументам различного типа, например, удваивающая функция, которая может быть применена к типу Nat, Bool, Fun и другим. Такия ситуация в простом типизированом λ-исчислении решается с помощью написания функции для каждого типа. Наличие нескольких функций, каждая из которых применима к своему типу аргумента (типы аргуметов различны для разных функций), но эти функции ведут себя одинаково нарушает один из принципов разработки программ - принцип абстракции[3]: каждая существенная область функциональности в программе должна быть реализована всего в одном месте программного кода. Если различные фрагменты кода реализуют аналогичную функциональность, то, как правило, имеет смысл слить их в один фрагмент, абстрагируя различающиеся части.

Таким образом, нужен способ абстрагировать тип терма, а затем конкретизировать абстрактный терм аннотациями нужного типа. Для абстрагирования типов из термов, а также для последующего заполнения абстракций вводится новая форма абстракции $\lambda X.t$, параметром который служит тип, и новую форму применения, $t\ [T]$, в которой аргументом служит выражение типа[4].

Синтаксис полиморфного λ-исчисления:

Термы: $t\ ::=\ x \mid \lambda x : T.t \mid t\ t \mid \lambda X.t \mid t\ [T]$

Значения: $v\ ::=\ \lambda X : T.t \mid \lambda X.t$

Типы: $T\ ::=\ X \mid T \to T \mid \forall X.T$

Контексты: $\Gamma\ ::=\ \emptyset \mid \Gamma,\ x : T \mid \Gamma, X$

Вычисление:

$$\frac{t_1\ \to\ t_2}{t_1\ t\ \to\ t_2\ t} \qquad \frac{t_1\ \to\ t_2}{v\ t_1\ \to\ v\ t_2} \qquad \frac{}{(\lambda x : T_{11}.t_{12})\ v\ \to\ [x \mapsto v]t_{12}} \qquad \frac{t_1\ \to\ t_2}{t_1\ [T]\ \to\ t_2\ [T]}$$

$$\frac{}{\lambda X.t\ [T]\ \to\ [X \mapsto T]\ t}$$

Типизация:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma,\ x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 \ : \ T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 \ : \ T_{12}}$$

$$\frac{\Gamma,\ X \vdash t : T}{\Gamma \vdash \lambda X.t \ : \ \forall X.T} \qquad \frac{\Gamma \vdash t \ : \ \forall X.T}{\Gamma \vdash t[T_1] \ : \ [X \ \to T_1]T}$$

Система $\lambda\underline{\omega}$ Пирс 465 операции над типами

Зависимые типы

Зависимые типы, впервые, были использованы в различных системах, направленных на построение и проверку автоматических доказательств. С практической точки зрения, зависимый тип — тип, который зависит от значения объекта. В качестве простейшего примера можно рассмотреть тип $String_n$ — бинарные строки длинны n. Данный тип записит от выбора n, который имеет тип натурального числа (n: int). Согласно изоморфизму Карри-Говарда оператор создания такого типа из натурального числа соответствует предикат по int. Такой предикат называют конструктором типа[5].

Классификация конструкторов в соответствие с их доменоми вводит понятие сорта (kind): конструктор $String_n$ имеет сорт $int \Rightarrow *$, где $*$ — сорт всех типов.

Определение объекта типа $String_n$ может оказаться однородным по n, то есть может существовать общая процедура, которая ведет себя обинаково со строками любой длинны, например, превращает их в строку из нулей длинны n. Типом такой процедуры является $(\forall x : \ int)\ String_x$[5, 6].

В общем случае, тип вида $(\forall x : \ T)\ \sigma$ является типом функции, применимой к объектам типа T и возвращающий объект типа $\sigma\ [x \ := \ a]$ для каждого аргумента $a \ : \ T$. Эта идея является более общей, чем идея типа функции ($\to$), если x не является свободной переменной в σ, то тип $(\forall x : \ T)\ \sigma$ является типом $T \to \sigma$ [5, 7].

Теоретико-множественным аналогом зависимого типа является продукция (произведение). Если $\{A_t\}\ t \in T$ является индексированым семейством множеств, то продукция этого семейства множество:

$$\prod_{t \in T} A_t = \{f \in (\bigcup_{t \in T} A_t)^T : f(t) \in A_t, \text{ for all } t \in T\}.$$

Для $f \in \Pi_{t \in T} A_t$ значение $f(t)$ принадлежат множеству A_t, предположительно рахличном для каждого значения агрумента. Если все A_t равны фиксированному множеству A, то получится равенство: $\Pi_{t \in T} A_t = A^T$, что еще раз подтверждает соотношение $\to$ с $\forall$.Таким образом, импликация является частным случаем квантором всеобщности.

Как в случае и с другими теориям типов, системы с зависимыми типами базируются на теоретическом λ-исчислении с абстрактным синтаксисом, правилами типизациии и оценки [6, 7]. λ-исчисление с зависимыми типами является системой λP. Система λP является большим расширением простого типизированного λ-исчисления, даже без введения квантора существования. Рассмотрим данную систему не вводя квантор существования[5].

Имеются три сорта выражений: объекты-выражения, конструкторы и сорта. Тип рассматривается как частный случай конструктора. Контексты в системе λP определяются как последовательность предположений, не всякая последовательность объявлений может рассматриваться как валидный контекст, зависит от выводимости определенных суждений.

Вводится константа для сорта — $*$.

Синтаксис λP [5, 8]:

Термы: $t \ ::= \ x \ | \ \lambda x : T.t \ | \ t \ t$

Типы: $T \ ::= \ X \ | \ \Pi x : T.T \ | \ T \ t$

Сорта: $K \ ::= \ * \ | \ \Pi x : T.K$

Контексты: $\Gamma \ ::= \ \oslash \ | \ \Gamma, \ x : T \ | \ \Gamma, \ X :: K$

Валидные сорта:

$$\Gamma \vdash *$$

$$\frac{\Gamma \ \vdash \ T :: * \qquad \Gamma, \ x : T \ \vdash K}{\Gamma \ \vdash \Pi x : \ T.K}$$

Правила приписывания сорта:

$$\frac{\Gamma \ \vdash \ T \ :: \ * \qquad \Gamma, \vdash \ K}{\Gamma \ \vdash \ x :: K}$$

$$\frac{\Gamma \ \vdash \ T_1 :: * \qquad \Gamma, \ x : T_1 \ \vdash T_2 :: *}{\Gamma \ \vdash \Pi x : \ T_1.T_2 :: *}$$

$$\frac{\Gamma \ \vdash \ S :: \Pi x : \ T.K \qquad \Gamma \ \vdash t :: T}{\Gamma \ St : [x \mapsto t]K}$$

$$\frac{\Gamma \ \vdash \ T :: K \qquad \Gamma \vdash K \equiv K`}{\Gamma \ \vdash \ T :: K`}$$

Правила типизации:

В приведенных выше правилах типизации и приписывания сорта используется понятие эквивалентности сорта и типа. Правила, согласно которым два сорта или типа можно считать эквивалентными приведены в приложении.

$$\frac{x : T \ \in \Gamma \qquad \Gamma \vdash T :: *}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash S :: * \qquad \Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.t \ : \ \Pi x : \ S.T}$$

$$\frac{\Gamma \vdash t_1 :: \Pi x : \ S.T \qquad \Gamma \vdash t_2 :: S}{\Gamma \ t_1 \ t_2 : [x \mapsto t_2]T} \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \equiv T^{'} :: *}{\Gamma \vdash t : T^{'}}$$

Согласно изоморфизму Карри-Говарда зависимым типам соответсвует логика предикатов первого порядка[5, 7]. Предикат B над типом A рассматривается как функция значения типа над A, и, следовательно, универсальное квантификация совпадает с зависимым продуктом: $\forall \ x \ : \ A.B(x)$ эквивалентно $\Pi \ x : \ A.B(x)$

Зависимые типы позволяют перенести некоторые вычисления на фазу проверки типов, предостовляя при этом возможность создавать более выразительные доказательства используя изоморфизм Карри-Говарда.

Исчисление конструкций

Наиболее развитой в λ-кубе системой является — это λP_ω, также известная как исчисление конструкций (Calculus of Constructions). Основное преимущество чистого исчисления конструкций — «замкнутость системы »,где математические и вычислительные нотации могут быть представлены с использованием квантификации высшего порядка[9]. Однако это представление является не удовлетворительным как с вычислительной, так и с логической точек зрения. Это привело к расширению данной теории индуктивными определениями как объектами первого класса [10, 11], которое называется исчисление индуктивных конструкций (Calculus of Inductive Constructions). В качестве индуктивного определения можно рассмотреть пример следующего определения натураьных чисел: натуральное число n — это либо z (zero, ноль), либо $S \ m$ (следующее за натуральным числом m).

Исчисление конструкций расширяет λ-исчисление с зависимыми типами следующим образом[8]: вводится новый валидный терм: $all \ x : T.t$; вводятся типы Prop — утверждения (propositions) и Prf — семейство доказательств. Элементы типа Prop представляют собой утверждения и типы данных (не типы доказальства утвержений), напрмер, натуральные числа. Семейство типов Prf присваивает каждому утверждению или типу данных $p \ : \ Prop$ тип $Prf \ p$ этого доказательства или в случае типов данных его членов. Вводятся следующие правила приписывания сорта и типизации:

– приписывание сорта:

$$\Gamma \vdash \ Prop \ :: \ * \qquad \Gamma \vdash \ Prf \ :: \ \Pi x : Prop.*$$

– типизация:

$$\frac{\Gamma \vdash T :: * \qquad \Gamma, x : T \vdash t : Prop}{\Gamma \vdash all\ x : T.t\ :\ Prop}$$

1.1.2 Индуктивные типы

2. Разработка моделей и алгоритмов

Булева функция — функция вида $B^n \to B$, где $B = 0, 1$. Систему булевых функций называют функционально полной, если из ее набора функций можно представить любую булеву функцию. Теорема (критерий) Поста позволяет проверить является ли система функций полной. Идея теоремы состоит в том, чтобы рассматривать множество всех булевых функций РА как алгебру относительно операции суперпозиции. Данная алгебра называется алгеброй Поста. Эта алгебра содержит в качестве своих подалгебр множества функций, замкнутых относительно суперпозиции. Их называют ещё замкнутыми классами. Их называют ещё замкнутыми классами. Пусть R — некоторое подмножество РА. Замыканием [R] множества R называется минимальная подалгебра РА, содержащая R. Иными словами, замыкание состоит из всех функций, которые являются суперпозициями R. Очевидно, что R будет функционально полно тогда и только тогда, когда [R]=PA. Таким образом, вопрос, будет ли данный класс функционально полон, сводится к проверке того, совпадает ли его замыкание с РА.

Оператор [_] является оператором замыкания. Он обладает следующими свойствами:

- $R \subseteq [R]$
- $R_1 \subseteq R_2 \Rightarrow [R_1] \subseteq [R_2]$
- $[[R]] = [R]$

Пост сформулировал необходимое и достаточное условие полноты системы булевых функций. Для этого он ввел в рассмотрение следующие замкнутые классы булевых функций: функции, сохраняющие константу $T_0 T_1, S, M, L$..

Класс функций сохраняющих ноль T_0 : , ,f(0, 0, ..., 0) = 0.

Класс функций сохраняющих единицу T_1 : , ,f(1, 1, ..., 1) = 1.

Класс самодвойственных функций S: говорят, что функция самодвойственна , если $f(\overline{x_1}, \ldots, \overline{x_n}) = \overline{f(x_1, \ldots, x_n)}$. *Иными словами, функция называется самодвойственной, если на противоположных наборах она принимает противоположные значения.*

Класс монотонных функций M: говорят, что функция монотонна, если $\forall i (a_i \leqslant b_i) \Rightarrow f(a_1, \ldots, a_n) \leqslant f(b_1, \ldots, b_n)$.

Класс линейных функций L: говорят, что функция линейна , если существуют такие $a_0, a_1, a_2, \ldots, a_n$ *где* $a_i \in \{0, 1\}, \forall i = \overline{1, n}$, *что для любых* $x_1, x_2, \ldots, x_n$ *имеет место равенство:* $f(x_1, x_2, \ldots, x_n) = a_0 \oplus a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \ldots \oplus a_n \cdot x_n$.

3. Результаты проектирования

4. Реализация и экспериментальная проверка

Coq — интерактивное программное средство доказательства теорем, использующее собственный язык функционального программирования с зависимыми типами. Позволяет записывать математические теоремы и их доказательства, удобно модифицировать их, проверяет их на правильность. Пользователь интерактивно создаёт доказательство сверху вниз, начиная с цели (то есть от гипотезы, которую необходимо доказать). Coq может автоматически находить доказательства в некоторых ограниченных теориях с помощью так называемых тактик. Coq применяется для верификации программ[12].

Формальная спецификация в Gallina состоит из последовательности объявлений (declarations) и определений (definitions). Также можно вводить комманды, но они не являются частью формальной спецификации, а выполняют информационные запросы или сервисные функции[12].

4.1 Реализация CIC в системе автоматического доказательства Coq

Формальной основой Coq, как уже было отмечено ранее является индуктивное исчисление конструкций (CIC). Выражения в CIC это термы, а все термы имеют типы, существуют типы для функций (или программ), атомик-типы — типы данных, типы для доказательств, а так же типы для типов. В частности, любой объект, обрабатываемый формализмом, должен принадлежать типу. Например, квантор всеобщности относится к типу. Типы для типов называются сортами. Типы и сорта сами по себе являются термами.

У всех сортов есть тип и существует бесконечная валидная иерархия типизации, чьи базовые сорта - Prop и Set. Сорт Prop намеревается быть типом логических предложений. Если M - логическое предложение, то он обозначает класс членов, представляющих доказательства M. Объект m, принадлежащий M, свидетельствует о том, что M доказуемо. Объектом типа Prop называется предложение. Сорт Set представляет собой тип небольших наборов. Это включает в себя типы данных, такие как логические, числовые, а также продукции, подмножества и типы функций по этим типам данных. Сортами Prop и Set можно манипулировать как обычными термами, а это значит что они сами имеют тип. Поскольку предположение что сорт Set имеет тип Set приводит к инконсистентности теории [13], язык CIC имеет бесконечно много сортов. В дополнение к Set and Prop имеется иерархия юниверсов Type(i) для любого целого i.

Как и Set Type(i) содержат маленькие множества, такие как логический тип, натуральные числа, а также подмножества и типы функций . Но, в отличие от Set, они также содержат

большие множества — сорта Set и Type(j) для j <i, а также все продукции, подмножества и типы функций над этими сортами.

Формально множество сортов S определяется следующим образом: $S \equiv Prop,\ Set,\ Type(i) \mid i \in N$

$Prop : Type(1),\ Set : Type(1),\ andType(i) : Type(i+1)$

Пользователь не должен явно указывать индекс i при обращении к типу юниверса (i). Он только пишет Type. Сама система генерирует для каждого экземпляра Type новый индекс для юниверса и проверяет, что ограничения между этими индексами могут быть решены. С точки зрения пользователя имеется Type: Type.

4.2 Спецификация языка Gallina

Теории строятся из аксиом, гипотез, параметров, лемм, теорем и определений констант, функций, предикатов и множеств. Язык Gallina позволяет разрабатывать математические теории и доказательства спецификаций программ.

Квалифицированные идентификаторы (qualid) обозначают глобальные константы (определения, леммы, теоремы, замечания или факты), глобальные переменные (параметры или аксиомы), индуктивные типы или конструкторы индуктивных типов. Простые идентификаторы (или короткий идентификатор) являются синтаксическим подмножеством квалифицированных идентификаторов. Идентификаторы также могут обозначать локальные переменные, какие нет у квалифицированных идентификаторов.

Числа не имеют определенной семантики в исчислении. Это простые обозначения, которые могут быть привязаны к объектам через механизм обозначения. Первоначально цифры связаны с представлением Пеано натуральных чисел.

Различные конструкции, такие как fun, forall, fix и cofix связывают переменные. Связывание представляется при помощи идентификатора. Если переменная привязки не используется в выражении, идентификатор может быть заменен символом $\_$*, (ident: type).,* $:(ident_1\ \ldots\ ident_n : type)$*. Некоторые конструкции позволяют привязывать переменную к значению. Это называется «let-binder». В let-binder одновременно может быть введена только одна переменная. Также можно указать тип переменной следующим образом:* $(ident : term := term)$*. Возможны списки binder-ов. В случае fun и forall предполагаетя что хотя бы одно из связываний является предположением, иначе они становятся идентичными.*

Выражение $fun\ ident : type => term$ *определяет абстракцию переменной ident, типа type, над термом term. Выражение* $forall\ ident : type, term$ *обозначает произведение переменной ident типа type, над термом term. Выражение* $term_0\ term_1$ *обозначает применение* $term_0$ *к* $term_1$*. Выражение* $term : type$ *является выражением приведения типа. Выражение*

$let\ ident\ :=\ term_1\ in\ term_2$ *локально связывает* $term_1$ *с переменной* $ident$ *в терме* $term_2$.

4.3 Механизм вывода

Имеется параллель между интуиционистской импликацией и квантором всеобщности, с одной стороны, и функциональным типом и зависимым произведением параметрического семейства типов, с другой. Это соответствие приводит к интерпретации интуиционистской логики в теории зависимых типов: высказывания интерпретируются типами (все- ми или некоторыми), а предикаты — семействами типов, зависящими от параметров. Верность высказывания означает населенность (непустоту) соответствующего типа. Классическая логика эмулируется дополнитель- ным контекстом, который обеспечивает непустоту типа, соответствующего закону исключенного третьего[2].

В рамках указанной парадигмы обоснование справедливости (верности) высказывания A означает предъявление объекта типа A. В теории типов объектами служат типизованные λ-термы. Именно они и играют роль доказательств соответствующих высказываний. Их также можно рассматривать как строчные записи деревьев вывода, доказывающих эти высказывания в формализме натурального вывода. Задача проверки правильности типизации термов для практически применяемых вариантов теории типов разрешима за разумное время, что позволяет автоматически верифицировать соответствующие доказательства. Непосредственное выписывание подходящих λ-термов оказывается практически невозможным ввиду их размера и сложности. Система Coq (и другие аналогичные интерактивные системы построения доказательств) упрощают эту задачу, позволяя строить соответствующие термы интерактивно, с привлечением ряда алгоритмов поиска выводов, частично автоматизирующих процесс построения [2].

Формализм теории типов оперирует с секвенциями вида

$x_1 : A_1,\ x_2 : A_2(x_1),\ \ldots\ ,\ x_n : A_n(x_1,\ \ldots\ ,\ x_{n-1})\ \vdash\ t(x_1,\ \ldots\ ,\ x_n) : B(x_1,\ \ldots\ ,\ x_n)$, *формализующими суждения о типизации «если* x_1 *имеет тип* A_1, x_2 *имеет тип* $A_2(x_1)$,. . . , *то* $t(x_1, \ldots, x_n)$ *имеет тип* $B(x_1, \ldots, x_n)$*».*

Работа пользователя в системе Coq направлена на построение соответствующего терма $t(x_1, \ldots, x_n)$. *Доступная ему информация описывается секвенцией* $x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1}) \vdash (???) : B(x_1, \ldots, x_n)$ *с метапеременной (???), значение которой надо определить.*

Один шаг интерактивного построения доказательства состоит в том, что пользователь выбирает одну из имеющихся тактик, а система ее применяет. Многие тактики для своего применения требуют дополнительные параметры, которые также указываются пользователем. Тактика представляет собой сведение решаемой задачи построения терма t, *для которого секвенция*

$$\frac{\begin{array}{l} x_1 : A_1 \\ x_2 : A_2(x_1) \\ \vdots \\ x_n : A_n(x_1, \ldots, x_{n-1}) \end{array}}{B(x_1, \ldots, x_n)}$$

Рис. 4.1 – l

(цель) $\Gamma \vdash t : A$ *выводима (в CIC), к аналогичным задачам для некоторых других секвенций (подцелей)* $\Gamma_1 \vdash t_1 : A_1, \ldots, \Gamma_k \vdash t_k : A_k, k \geq 0$. *Тактики соответствуют допустимым правилам исчисления CIC, а также содержат алгоритмы построения искомого терма* t *по термам* $t_1, \ldots, t_k$. *При применении тактики изображающая задачу таблица (без t)* $\frac{\Gamma}{A}$ *превращается в* $\frac{\Gamma_1}{A_1} \ldots \frac{\Gamma_k}{A_k}$. *Аналогичные шаги применяются к подцелям и т.д., пока количество подцелей не сократится до 0. Система запоминает последовательность примененных тактик и восстанавливает искомый терм* t *с помощью соответствующих алгоритмов. При этом система постоянно контролирует правильность типизации всех термов, что исключает возможность получения ошибочных доказательств[2]. В системе автоматического доказательства Coq существует возможность самостоятельно писать тактики используя специально разработанный для этого язык разработки тактик.*

$\bot$			$\frac{\Gamma \vdash ? : \texttt{False}}{\Gamma \vdash ? : C}$	`exfalso`
$\neg$	$\frac{\Gamma, h : A \vdash \texttt{False}}{\Gamma \vdash ? : \neg A}$	`intro` h	$\frac{\Gamma \vdash h : \neg A \quad \Gamma \vdash ? : A}{\Gamma \vdash ? : C}$	`destruct` h
$\rightarrow$	$\frac{\Gamma, h : A \vdash ? : B}{\Gamma \vdash ? : A \rightarrow B}$	`intro` h	$\frac{\Gamma \vdash h : A \rightarrow B \quad \Gamma \vdash ? : A}{\Gamma \vdash ? : B}$	`apply` h
$\forall$	$\frac{\Gamma, y : A \vdash ? : B[x \leftarrow y]}{\Gamma \vdash ? : \forall x : A, B}$	`intro` y	$\frac{\Gamma \vdash h : \forall x : A, B \quad \Gamma \vdash t : A}{\Gamma \vdash ? : B[x \leftarrow t]}$	`apply` y `with` $(x := t)$
$\wedge$	$\frac{\Gamma \vdash ? : A \quad \Gamma \vdash ? : B}{\Gamma \vdash ? : A \wedge B}$	`split`	$\frac{\Gamma \vdash h : A \wedge B \quad \Gamma, l : A, m : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` (l, m)
$\vee$	$\frac{\Gamma \vdash ? : A}{\Gamma \vdash ? : A \vee B}$ $\frac{\Gamma \vdash ? : B}{\Gamma \vdash ? : A \vee B}$	`left` `right`	$\frac{\Gamma \vdash h : A \vee B \quad \Gamma, l : A \vdash ? : C \quad \Gamma, l : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` $[l \mid l]$
$\exists$	$\frac{\Gamma \vdash t : A \quad \Gamma \vdash ? : B[x \leftarrow t]}{\Gamma \vdash ? : \exists x : A, B}$	`exists` t	$\frac{\Gamma \vdash h : \exists x : A, B \quad \Gamma, x : A, l : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` (x, l)
$=$	$\frac{t \equiv u}{\Gamma \vdash ? : t = u}$	`reflexivity`	$\frac{\Gamma \vdash h : t = u \quad \Gamma \vdash ? : C[x \leftarrow u]}{\Gamma \vdash ? : C[x \leftarrow t]}$	`rewrite` h

Рис. 4.2 – Логические правила и соответствующие им тактики

В большинстве случаев, для построения доказательства в системе Coq используется такой

подход как «целенапрвленное доказательство» (goal directed proof) который использует следующий тип сценария[14]:

1. *Пользователь вводит утверждение, которое он хочет доказать, используя команду Theorem или Lemma, при этом именуя ее.*
2. *Система Coq отображает формулу в качестве формулы, которая должна быть доказана, возможно, предоставляя контекст локальных фактов, которые могут быть использованы для этого доказательства (контекст отображается над горизонтальной линией, написанной =====, цель отображается под горизонтальной линия).*
3. *Пользователь вводит команду(тактики) для декомпозиции цели на более простые подцели.*
4. *Система Coq отображает список формул, которые еще нужно доказать.*
5. *Повторяется 3 шаг до тех пор, пока не останется ни одной формулы, которую нужно доказать.*

Когда нет больше целей, доказательство завершено, оно сохраняется при помощи команды Qed с именем заданным на 1 шаге. Оформляется теорема следующим образом:

```
Theorem <идентификатор>: < формулировка>.
Proof.
<доказательство>
Qed.
```

В дальнейшем <идентификатор> можно использовать как любую другую константу указанного типа. Тем самым Theorem представляет собой вариант интерактивного определения констант[14].

4.4 Введение констант

Описание прикладных теорий в системе Coq сводится к введению новых имен (констант). Это аппарат позволяет единообразно описывать как язык, так и аксиоматику теории. Для того, чтобы постулировать некоторое утверждение, достаточно формализовать его в виде типа $A : Prop$*, после чего декларировать непустоту типа A введением новой (свободной) константы c:A. Механизм секций позволяет объединить основные определения и теоремы теории в единое целое[2].*

Ключевые слова $Variable$ *и* $Hypothesis$ *(а также* $Variables$, $Parameter$, $Parameters$, $Axiom$, $Conjecture$*) являются почти синонимами. Они позволяют ввести в контекст новые константы (имена) и объявить их типы[14].*

Некоторое различие между группами (Variable, Variables, Parameter, Parameters) и (Hypothesis, Axiom, Conjecture) проявляется лишь при использовании механизма секций.

```
Section <название секции>.
...
End <название секции>.
```

Это окружение позволяет ограничить пределами секции область действия деклараций констант, введенных внутри секции с помощью ключевых слов Variable, Variables, Parameter, Parameters. Вне секции эти константы недоступны. Там они преобразуются в дополнительные параметры всех определенных внутри секции имен.

Введенные с помощью декларации имена являются свободными константами — им нельзя присвоить какие-нибудь значения, отличные от них самих. Для определения новых имен через уже имеющиеся служит конструкция Definition[14].

Более сложный вариант определений — индуктивное определение типа — вводится ключевым словом Inductive. В основе лежит семантика наименьшей неподвижной точки. Задаются конструкторы объектов определяемого типа, а сам тип представляет собой наименьшее множество, замкнутое относительно применения конструкторов[14].

4.5 Разбор случаев

Пусть задано индуктивное определение типа T с конструкторами $c_1, \ldots, c_k$, где c_i представляет функцию от n_i аргументов со значениями в типе T. Каждый терм t типа T имеет вид $c_i t_{i,1} \ldots t_{i,n_i}$ для некоторого i, причем это представление единственно. Разбор случаев позволяет через t определить выражение $h(t)$ разными способами в зависимости от вида терма t[2]:

$$h(t) = \begin{cases} h_1(t_{1,1} \ldots t_{1,n_1}), & t = c_1 t_{1,1} \ldots t_{1,n_1} \\ \ldots & \\ h_k(t_{k,1} \ldots t_{k,n_k}), & t = c_k t_{k,1} \ldots t_{k,n_k} \end{cases}$$

Чтобы выражению $h(t)$ удалось приписать тип, достаточно потребовать, чтобы все термы $h_i(t_{i,1} \ldots t_{i,n_i})$ имели общий тип T_1. : $T_1(z)$, индексированное элементами $z : T$, и требовать, чтобы $h_i(t_{i,1} \ldots t_{i,n_i}) : T_1(t)$ при $t = c_i t_{i,1} \ldots t_{i,n_i}$. Соответствующая редукция заменяет терм $h(t)$ на $h_i(t_{i,1} \ldots t_{i,n_i})$, сохраняя его тип $T_1(t)$.

В синтаксисе языка Gallina системы Coq разбор случаев реализует конструкция match:

```
match term [dep_ret_type] with
| C1 _ … _ => term1
```

```
.......
| Cn _ … _ => termn
end.
```

Заключение

В заключении в тезисной форме необходимо отразить результаты работы:

– *аналитические (что изучено/проанализировано);*

– *теоретические;*

– *инженерные (что спроектировано);*

– *практические (что реализовано/внедрено).*

Примерная формула такая: по каждому указанному пункту приводится по 3-5 результатов, каждый результат излагается в объеме до 5 фраз или предложений.

Также есть смысл привести предполагаемые направления для будущей работы.

Общий объем заключения не должен превышать 1,5 страниц (1 страницы для УИРов).

Список литературы

1. *Chlipala Adam. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. — MIT Press, 2013.*
2. *Крупский ВН, Кузнецов СЛ. Практикум по математической логике. COQ. — Кафедра математической логики и теории алгоритмов, 2013.*
3. *Пирс Бенджамин. Типы в языках программирования // М.: Лямбда Пресс Добросвет. — 2012. — P. 656.*
4. *Girard Jean-Yves. The system F of variable types, fifteen years later // Theoretical computer science. — 1986. — Vol. 45. — P. 159–192.*
5. *Sørensen Morten Heine, Urzyczyn Pawel. Lectures on the Curry-Howard isomorphism. — Elsevier, 2006. — Vol. 149.*
6. *Baxter Samuel. An ML Implementation of the Dependently Typed Lambda Calculus. — 2014.*
7. *Hofmann Martin. Syntax and semantics of dependent types // Extensional Constructs in Intensional Type Theory. — Springer, 1997. — P. 13–54.*
8. *Pierce Benjamin C. Advanced topics in types and programming languages. — MIT press, 2005.*
9. *Paulin-Mohring Christine. Inductive definitions in the system coq rules and properties // International Conference on Typed Lambda Calculi and Applications / Springer. — 1993. — P. 328–345.*
10. *Coquand Thierry, Paulin Christine. Inductively defined types // COLOG-88 / Springer. — 1990. — P. 50–66.*
11. *Pfenning Frank, Paulin-Mohring Christine. Inductively defined types in the Calculus of Constructions // International Conference on Mathematical Foundations of Programming Semantics / Springer. — 1989. — P. 209–228.*
12. *Huet Gérard, Kahn Gilles, Paulin-Mohring Christine. The Coq proof assistant: a tutorial: version 8.7.1 : Ph. D. thesis / Gérard Huet, Gilles Kahn, Christine Paulin-Mohring ; Inria. — 2017.*
13. *Coquand Thierry. An analysis of Girard's paradox : Ph. D. thesis / Thierry Coquand ; INRIA. — 1986.*

14. Bertot Yves. Coq in a Hurry. — 2006.

A. Правила эквивалентности типов, видов и термов

Правила эквивалентности для видов:

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: * \qquad \Gamma,\, x : T_1 \vdash K_1 \equiv K_2}{\Gamma \vdash \Pi\, x : T_1.K_1 \equiv \Pi\, x : T_2.K_2} \qquad \frac{\Gamma \vdash K}{\Gamma \vdash K \equiv K}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2}{\Gamma \vdash K_2 \equiv K_1} \qquad \frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma \vdash K_2 \equiv K_3}{\Gamma \vdash K_1 \equiv K_3}$$

Правила эквивалентности для типов:

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: * \qquad \Gamma,\, x : T_1 \vdash S_2 \equiv T_2 :: *}{\Gamma \vdash \Pi\, x : S_1.S_2 \equiv \Pi\, x : T_1.T_2 :: *} \qquad \frac{\Gamma \vdash T :: K}{\Gamma \vdash T \equiv T :: K}$$

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: \Pi x : T.K \qquad \Gamma,\, t_1 \equiv t_2 : T}{\Gamma \vdash S_1\, t_1 \equiv S_2\, t_2 : [x \mapsto t_1]K} \qquad \frac{\Gamma \vdash T \equiv S :: K}{\Gamma \vdash S \equiv T :: K}$$

$$\frac{\Gamma \vdash S \equiv U :: K \qquad \Gamma \vdash U \equiv T :: K}{\Gamma \vdash S \equiv T :: K}$$

Правила эквивалентности для термов:

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: * \qquad \Gamma,\, x : S_1 \vdash t_1 \equiv t_2 : T}{\Gamma \vdash \lambda x : S_2.t_2 : \Pi x : S_1.T} \qquad \frac{\Gamma \vdash t_1 \equiv S_1 : \Pi x : S.T \qquad \Gamma \vdash t_2 \equiv s_2 : S}{\Gamma \vdash t_1\, t_2 \equiv s_1\, s_2 : [x \mapsto t_2]T}$$

$$\frac{\Gamma,\, x : S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x : S.t)\, s \equiv [x \mapsto s]\, t : [x \mapsto s]\, T} \qquad \frac{\Gamma \vdash t : \Pi\, x : S.T \qquad x \notin FV(y)}{\Gamma \vdash \lambda x : T.t\, x \equiv t : \Pi\, x : S.T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \equiv t : T} \qquad \frac{\Gamma \vdash t \equiv s : T}{\Gamma \vdash s \equiv t : T} \qquad \frac{\Gamma \vdash s \equiv u : T \qquad \Gamma \vdash u \equiv t : T}{\Gamma \vdash s \equiv t : T}$$