САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Таран Кирилл Сергеевич

Метод применения теории типов Мартина-Лёфа для верификации программных систем

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

____
подпись

Научный руководитель:
к. ф.-м. н., доцент Булычев Д. Ю.

____
подпись

Рецензент:
Соломатов К. В.

____
подпись

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Taran Kirill

Method of Martin-Löf's type theory application for program system verification

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

signature

Scientific supervisor:
PhD Dmitri Boulytchev

signature

Reviewer:
Konstantin Solomatov

signature

Saint-Petersburg
2014

Оглавление

Введение **4**

1. Постановка задачи **6**

2. Обзор **7**

2.1. История развития теории типов 7

§ Ранние теории типов 7

§ Изоморфизм Карри-Говарда и зависимые типы 7

§ Теория типов Мартина-Лёфа 9

2.2. Инструменты для построения доказательств 11

§ Языки программирования с зависимыми типами 11

§ Особенности Coq и Agda 12

3. Предлагаемое решение **14**

3.1. Сценарий использования программных реализаций ТТМЛ 14

§ Легковесное верифицирование с помощью зависимых типов 14

§ Верифицирование сложных систем 15

3.2. Инструмент экстракции Java-кода по Coq-спецификации 18

§ Трансляция алгебраических типов данных 19

§ Трансляция функций как значений первого порядка 20

4. Апробация **25**

4.1. Метод операционных преобразований 25

4.2. Формализация алгоритма управления с помощью Coq 28

5. Результаты **32**

Заключение и возможность развития **33**

Введение

Важная проблема при разработке программного обеспечения — гарантия его корректности, то есть тщательная формулировка желаемых свойств продукта с последующей их проверкой. Обычно, эта задача решается неформально в процессе реализации программы; но с ростом объёма работы уследить за всевозможными нюансами реализации становится трудно, ошибки неизбежно происходят.

Подобные ошибки могут годами оставаться незамеченными, уже после выпуска продукта и на стадии поддержки. Особо дорогие ошибки могут стоить пользователю или заказчику до сотен миллионов долларов.

Проблема корректности ПО становится ещё более актуальной с учётом нескольких тенденций: количественного возрастания размера программных систем и их перемещения из "локального" пространства пользователя в "глобальное".

Под глобальным пространством, в основном, подразумеваются различные сети (интернет, частные вроде интрасетей компаний). Отличие его от локального пространства состоит в том, что хранение данных и их обработка происходят (полностью или частично) централизованно и, теоретически, могут быть доступны другим пользователям. Самым популярным примером являются интернет-сервисы, хранящие личную информацию пользователей и подверженные атакам с целью получить эту личную информацию.

Если абстрагироваться от конкретных примеров, то централизация проявляется:

- во времени — пользователи работают одновременно;
- логически — действия пользователей взаимосвязаны между собой;
- физически — вычисления производятся одним и тем же вычислителем, а данные часто хранятся на одном носителе информации.

Кроме повышенного параллелизма это влечёт за собой и бóльшую скорость и область распространения изменений. К примеру, в онлайн сервисах изменение кода может выпуститься за пару дней, десктопные приложения тоже могут автоматически обновиться через сеть; если обновление содержит ошибку, то она затрагивает сразу множество компьютеров, также и незамеченные уязвимости подвергают риску взлома сразу всех пользователей.

Таким образом, к программным системам, работающим с большим количеством пользователей в одном пространстве, предъявляются повышенные требования корректности и целесообразно применять формальные методы проверки. Одним из таких методов является формальная верификация — автоматизированная проверка продукта на соответствие спецификации, составленной с помощью формального языка. Этот метод позволяет минимизировать человеческий фактор при поиске ошибок в программе.

Для того, чтобы систему можно было верифицировать, нужно задать формальный язык для описания спецификации этой системы. Хорошим кандидатом на роль такого языка является интуиционистская теория типов [1], разработанная Пером Мартин-Лёфом. Эта теория является формальной системой, альтернативной наивной теории множеств; и она достаточно строгая и гибкая одновременно, чтобы обеспечить нас подходящим языком для верификации.

Интуиционистская теория типов имеет программные реализации в виде инструментов для построения доказательств (*proof assistants*) вроде Coq[1] и Agda[2]. Эти инструменты, по сути, являются языками программирования с *зависимыми типами* [2] и используются для формализации математики и автоматической проверки доказательств. Это означает, что с помощью теории типов можно кодировать различные математические структуры, функции и доказывать какие-либо свойства о них; а с помощью программных реализаций становится возможным автоматически проверять эти доказательства на корректность. Это именно те средства, которые нужны для верификации [3].

---

[1]http://coq.inria.fr

[2]http://wiki.portal.chalmers.se/agda

[3]Надо заметить, что инструменты реализующие какой-либо вариант теории типов Мартина-Лёфа могут иметь возможность автоматизации построения спецификации, но это не их основная задача. Предполагается, что спецификация составляется по большей части вручную и интерактивно, с использованием средств автоматизации.

1. Постановка задачи

Целью работы является создание промышленного подхода на основе теории типов Мартина-Лёфа [4] для разработки программных систем с возможностью формальных доказательств каких-либо свойств об этой системе или её компонентах. Под подходом понимается сценарий использования этой теории и набор паттернов использования существующих инструментов на основе этой теории. Чтобы подход можно было применять промышленно, необходима дополнительная инструментальная поддержка.

Для достижения этой цели выделены следующие задачи:

1. Изучить теорию типов Мартина-Лёфа, теоретические основы языков программирования с зависимыми типами, а также наиболее известные реализации (Coq, Agda), их возможности, функциональность и ограничения.

2. Разработать сценарий использования подобных инструментов, эффективные приёмы при работе с ними.

3. Дополнить наиболее зрелый proof assistant инструментом экстракции кода на промышленном языке программирования по спецификации программного продукта.

4. Выполнить апробацию подхода на промышленной программной системе или каком-либо алгоритме.

[4]Далее возможно сокращение ТТМЛ.

2. Обзор

2.1. История развития теории типов

Ранние теории типов

История теории типов начинается с Бертрана Рассела, предложившего в 1903 году *доктрину типов* [3]. Целью новой теории была замена наивной теории множеств, не подходящей для формализации математики из-за обнаруженных в ней противоречий, связанных с непредикативными определениями, таких как парадокс Рассела [3]. В теории типов Рассела вместо множеств рассматривались классы и типы, причём уже была выделена иерархия классов: термы, классы термов, классы классов и т.д.; а пропозициональные функции могли быть применены только к аргументам подходящего типа. Уже была видна связь между математической логикой и теорией типов, о чём свидетельствует статья Рассела [4], однако точного соответствия вроде интерпретации Брауэра-Гейтинга-Колмогорова [5] и изоморфизма Карри-Говарда ещё не установлено [6].

Одной из следующих версий теории типов была простая теория типов Чёрча [7]. Языком для описания функций в ней стало лямбда-исчисление — один из фундаментальных формализмов, с помощью которых возможно определение понятия вычислимости, эквивалентный по выразительности машине Тьюринга. Поэтому теория типов Чёрча имела больший уклон в сторону информатики и языков программирования.

Изоморфизм Карри-Говарда и зависимые типы

Примерно в это же время появилось соответствие Карри-Говарда, также известное как "доказательства как программы" — наблюдаемое структурное сходство между компьютерными программами и математическими доказательствами. Суть его состоит в том, что любой типизированной системе соответствует некоторая логика, и наоборот. Можно сказать, что высказывания есть типы, а их доказательства являются термами этих типов (программами).

Известная книга Б. Пирса по теории типов так объясняет зависимые типы: "В конструктивных логиках доказательство утверждения P состоит в демонстрации конкретного *свидетельства* в пользу P. Карри и Говард заметили, что это свидетельство во многом похоже на вычисление. Например, доказательство утверждения $P \Rightarrow Q$ можно рассматривать как механическую процедуру, которая, получая доказательство P, строит доказательство Q." [2]

В рамках соответствия Карри-Говарда следующие структурные элементы рассматриваются как аналогичные:

высказывание P	тип P
доказательство высказывания P	терм типа P
утверждение P доказуемо	тип P обитаем
конъюнкция $P \wedge Q$	произведение $P \times Q$
дизъюнкция $P \vee Q$	размеченное объединение $P + Q$
импликация $P \Rightarrow Q$	функциональный тип $P \to Q$
истинная формула	тип с единственным элементом
ложная формула	тип без элементов
квантор всеобщности $\forall$	зависимое произведение $\prod$
квантор существования $\exists$	зависимая сумма $\sum$

Как видно из таблицы, кванторам всеобщности и существования соответствуют *зависимые* произведение и сумма. Неформально мыслить зависимое произведение можно следующим образом: чтобы получить объект $\prod\limits_{i:I} X(i)$, нужно перемножить все типы $X(i)$ по типу I, т.е. поскольку произведением типов является тип из пар их элементов, необходимо составить кортеж, i-й элемент которого должен принадлежать типу $X(i)$. По сути, эта процедура напоминает поточечное задание некоторой функции, поэтому неудивительно, что один из частных случаев зависимого произведения — функциональный тип. Действительно, если мы фиксируем некоторые типы A и B, то $A \to B = \prod\limits_{a:A} B$. А если же мы возьмём тип из двух элементов (тип $bool$ в языках программирования), обозначаемый 2, то зависимое произведение с индексами из этого типа является другим частным случаем — простым произведением типов, т.к. $A \times B = \prod\limits_{i:2} \lambda\, i\,.\, if\ i = 0\ then\ A\ else\ B$. Аналогично и с зависимой суммой: $\sum\limits_{i:I} X(i)$ — дизъюнктное объединение всех $X(i)$ или тип, значениями которого являются всевозможные пары (i, x) при $x : X(i)$.

Зависимые типы в некотором смысле стирают различие между типами и термами: они позволяют составлять выражения, оперирующие термами и выдающие некоторые типы в зависимости от них. Более того, эти выражения становится возможным вычислять (редуцировать до нормальной формы). К примеру, одно из наиболее фундаментальных логических отношений — равенство двух объектов (как термов, так и типов), может быть определено как тип Id с единственным конструктором $refl\ x$, где x — некоторый параметр. Говорится, что есть доказательство утверждения $a = b$ для некоторых $a\ b : A$ (или существуют термы типа Id_A), если существует такой x, что нормальная форма как a, так и b равна x.

Про зависимые типы с точки зрения языков программирования ещё будет сказано в рамках обзора инструментов для построения доказательств.

Теория типов Мартина-Лёфа

По поводу теории типов Мартина-Лёфа прекрасно сказано в одной статье: “Теория типов Мартина-Лёфа может быть описана как интуиционистская теория итерированных индуктивных определений, разработанная в системе зависимых типов. Она изначально предполагалась стать полномасштабной системой для формализации конструктивной математики, но также подтвердила себя в качестве мощной системы для программирования. Теория соединяет выразительный язык спецификации (свою систему типов) и функциональный язык программирования (где все программы завершаются). Сейчас есть несколько инструментов для построения доказательства, основанных на этой теории, и много нетривиальных примеров из программирования, информатики, логики и математики, реализованных с помощью них.”[8]

Эта теория типов основывается на *индуктивных* типах — механизме определения нового типа с помощью объявления конструкторов: констант и функций, создающих термы этого типа. К примеру, можно задать тип натуральных чисел N в аксиоматике Пеано как индуктивный тип: он состоит из константы $O : N$ и функции следования $S : N \to N$. Это определение генерирует тип, термами которого являются O, SO, $S(SO)$ и т.д. С помощью механизма индуктивных типов можно задавать любые конечные типы (а для бесконечных есть *коиндуктивные*), т.е. любые интересные программисту структуры данных.

Надо заметить, что *алгебраические типы данных*, используемые в функциональных языках программирования, являются частным видом индуктивных типов в том смысле, что с помощью АТД не разрешено определять типы с индексами; расширение АТД, позволяющее это делать, называется *обобщёнными алгебраическими типами данных*. Индуктивные типы, с этой точки зрения, являются объединением обобщённых АТД и зависимых типов.

После добавления конечных (*финитных*) типов Мартин-Лёф добавляет в свою систему *трансфинитные* типы, вводя последовательность вложенных универсумов [1]. Универсум некоторого семейства типов определяется как наименьший тип, замкнутый относительно операций над этими типами. Следовательно, можно ввести универсум конечных типов, универсум универсумов конечных типов, следующий универсум и т.д.

В соответствии с изоморфизмом Карри-Говарда и интерпретацией Брауэра-Гейтинга-Колмогорова данная система типов позволяет интерпетировать *логику высшего порядка*, т.к. возможно использовать кванторы не только над типами первого универсума, но и над предикатами и типами высших универсумов. Это делает систему типов подходящим языком для составления каких-либо формальных утверждений, определения предикатов, формулирования теорем; а поскольку теория типов Мартина-Лёфа снабжена функциональным языком программирования, то и для спецификации необходимых свойств программных систем.

Больше про теоретические аспекты и про связь теории типов с программированием можно узнать из книг “Programming in Martin-Löf’s Type Theory”[9] и “Type Theory & Functional Programming”[10].

2.2. Инструменты для построения доказательств

Языки программирования с зависимыми типами

С точки зрения языков программирования, механизм зависимых типов можно считать расширением параметрического полиморфизма[2]. С помощью простого параметрического полиморфизма мы можем работать с типами, параметризованными другими типами; а благодаря зависимым типам мы можем параметризовать типы термами.

Классическим примером является сравнение типа список и типа вектор (код на языке Coq):

List.v

```
Inductive list (A : Type) : Type :=
 | cons : A -> list A -> list A
 | nil  : list A.
```

Vector.v

```
Inductive vector (A : Type) : nat -> Type :=
 | cons : forall n, A -> vector A n -> vector A (S n)
 | nil  : vector A 0.
```

Тип вектор аналогичен типу список за тем исключением, что вектор дополнительно хранит свою длину. Однако, это не просто пара `(n,l)` из списка `l` и числа `n`, являющегося его длиной — длина вектора зашифрована в *индексе* типа терма-вектора.

Обычно неформально отличают индексы индуктивного типа от его *параметров*. Параметром называют типовую переменную, значение которой фиксировано для всех конструкторов сразу и задаётся "извне", как переменная `A` здесь: `Inductive list (A : Type) : ...`. В то же время индексом называют переменную, значение которой может быть установлено в отдельном конструкторе; например, в вышеприведённом фрагменте тип вектор имеет индекс типа `nat`, который устанавливается в `(S n)` и `0` в конструкторах `cons` и `nil` соответственно. Визуально отличить параметры от индексов можно по тому, с какой стороны от знака принадлежности типу находится переменная в выражении `Inductive name x : forall y, ...` если переменная справа от `:`, как `y`, то это индекс, т.к. мы не зафиксировали его имя в сигнатуре, как сделали это с `x`. Подобный механизм индексов типа существует и в других функциональных языках (таких как OCaml и Haskell), и тип с индексами обычно называют *обобщённым алгебраическим типам*. Надо отметить, однако, что без зависимых типов мы не можем иметь в качестве значений индексов термы. Эти два концепта в теории типов являются основными характеристиками *индуктивных типов*.

Преимущество кодирования длины вектора n в его типе заключается в том, что мы можем использовать n в сигнатурах функций, работающих с векторами. Функция head, практически всегда идущая вместе с определением типа вектор, послужит хорошим примером:

List.v

```
Definition head {A : Type}
  (xs : list A) : option A :=
  match xs with
  | cons x xs' => Some x
  | _ => None
  end.
```

Vector.v

```
Definition head {A : Type} {n : nat}
  (xs : vector A (S n)) : A :=
  match xs with
  | cons n x xs' => x
  end.
```

Здесь мы определили функцию head, которая возвращает первый элемент списка или вектора. Вариант для списка проверяет наличие элементов в списке *динамически*, т.е. во время работы программы, и возвращает None в случае пустого списка; в то время как вариант для вектора производит *статическую* проверку непустоты вектора, т.к. функция определена для векторов длины (S n). Вариант с вектором намного лучше, т.к. во-первых, мы избавились от типа option, необходимого для кодирования не всюду определённых функций; а во-вторых, система программирования сообщит об ошибке на этапе компиляции, если мы не предоставим доказательства, что длина вектора-аргумента head больше нуля[5].

Особенности Coq и Agda

Одна из самых известных и зрелых систем для интерактивного доказательства теорем — Coq[6]. Как и все инструменты, рассмотренные в данной работе, Coq использует функциональный язык программирования с зависимыми типами — Gallina — для описания структур данных, определения функций и типов. Интерактивное доказательство теорем в Coqосуществляется с помощью предметно-ориентированного языка (*DSL*) для работы с контекстом доказательства — языком *тактик* Ltac. Контекст доказательства содержит различные переменные-гипотезы. Также есть встроенная библиотека тактик. Некоторые тактики способны полностью автоматически доказывать теоремы для алгоритмически разрешимых теорий (например, *omega* для арифметики Пресбургера).

Coq также позволяет разрабатывать и верифицировать программы на основе одного исходного кода, т.к. обладает механизмом извлечения кода на языках OCaml, Haskell и Scheme из исходных текстов Coq. При этом верификационная информация стирается для лучшей производительности.

[5]В определённых ситуациях программа проверки типов (*type checker*) сама способна предоставить это доказательство.

[6]http://coq.inria.fr

Другая система — Agda[7] — более молодая и имеет меньше средств для автоматизации доказательств. В отличие от Coq механизма тактик в Agda нет, вместо этого теоремы можно определить только с помощью функций (из соответствия Карри-Говарда следует, что функции и теоремы суть одно и то же). Однако отсутствие тактик в некоторой степени компенсируется *рефлексией* в последних версиях системы, т.е. возможностью внутри доказательства обратиться к доказываемому утверждению.

В Agda реализована подсистема Agsy для автоматического поиска *type inhabitant*, т.е. значения заданного типа. С помощью неё можно автоматически искать доказательства теорем, но этот инструмент недостаточно зрелый и имеет ряд ограничений.

Далее в тексте, кроме явно оговорённых случаев, будет использоваться язык Coq.

[7]http://wiki.portal.chalmers.se/agda

3. Предлагаемое решение

3.1. Сценарий использования программных реализаций ТТМЛ

Легковесное верифицирование с помощью зависимых типов

Я выделяю два аспекта теории типов Мартина-Лёфа, которые дают возможность верифицировать программное обеспечение.

Во-первых, можно использовать зависимые типы во время определения функций. К примеру, можно явно ограничить область определения функции подобно тому, как это было сделано в предыдущей главе с функцией взятия первого элемента списка. Это отсекает некоторое множество ошибок, связанных с вызовом функции с некорректными параметрами, и в некотором роде это "легковесный" способ верификации.

Во-вторых, можно описывать предикаты и строить для них доказательства, которые проверяются на этапе проверки типов. Эти предикаты можно оформлять в виде отдельных *теорем*, доказательство которых проверяется во время проверки типов.

На самом деле, оба варианта весьма близки из-за соответствия Карри-Говарда, описанного в предыдущих главах: любая функция является некоторой теоремой и, наоборот, любая теорема является некоторой функцией. К примеру, если теорема утверждает, что из A следует существование некоторого объекта $b : B$ такого, что выполняется $P(b)$, то мы можем интерпретировать это как вычисление, принимающее на вход объект типа `A`, а возвращающее объект `b` типа `B` вместе с термом некоторого типа `P` с параметром `b`.

Таким образом, возможность параметризации типов значениями намного мощнее, чем может казаться: мы можем не только ограничивать область видимости функции, но и производить вычисления, которые кроме результата возвращают доказательство корректности этого результата. Более того, мы можем определить функцию, одним из аргументов которой является некоторая информация о других её аргументах; и при этом, в отличие от других языков программирования, мы можем статически гарантировать, что данная информация действительно относится к аргументам, а не передана функции по ошибке.

В качестве примера можно вспомнить сигнатуру всё той же функции взятия первого элемента: `head {A : Type} {n : nat} (xs : vector A (S n)) : ...`. В некотором философском смысле можно воспринимать число `n` *доказательством* того, что вектор `xs` не пуст. Этот предикат *"вектор не пуст"* здесь закодирован неявно, в отличие от переусложнённого варианта сигнатуры с явным аргументом типа `NotEmpty`: `head A : Type n : nat (xs : vector A n) (NotEmpty xs) : ...`. Однако очевидно, что обе сигнатуры имеют одинаковый смысл для человека, потому что предикат *"вектор не пуст"*, соответствующий типу `NotEmpty`, эквивалентен предикату *"существует n такое, что длина вектора равна $n + 1$"*. Вторая сигнатура не

практична в применении к спискам, но хорошо иллюстрирует, как можно сообщать вычислению информацию о его параметрах.

Верифицирование сложных систем

Описанные выше приёмы подходят для гарантии отсутствия поверхностных ошибок, но использовать их для доказательства большого количества нетривиальных фактов об алгоритмах слишком трудоёмко, поскольку подразумевают задание настолько подробных сигнатур функций, что становится очень трудно запрограммировать эти функции. Ещё труднее запрограммировать эффективно вычисления, выдающие результат вместе с доказательством корректности.

Поэтому на практике многие свойства и их доказательства оформляются отдельно от реализации. Например, пусть определена некоторая функция $is_prime : \mathbb{N} \to bool$, определяющая, является ли некоторое число простым. Тогда мы можем объявить теорему $is_prime_correct : forall\ n : nat,\ is_prime\ n = true \Leftrightarrow IsPrime\ n$, где $IsPrime$ — некоторый предикат, эквивалентный неформальному понятию простоты числа, но для построения доказательства которого используется простой и, возможно, неэффективный алгоритм. В таком случае, теорема $is_prime_correct$ утверждает эквивалентность результатов эффективной и неэффективной, но понятной, реализации.

Подробнее с обобщёнными техниками верификации с помощью ТТМЛ можно познакомиться в книгах “Software Foundations”[11] и “Certified Programming with Dependent Types”[12].

Можно пойти ещё дальше и верифицировать с помощью ТТМЛ алгоритмы и системы, реализованные эффективно на промышленных языках программирования, а не с помощью ТТМЛ. Но для того, чтобы проводить какие-то рассуждения о такой системе, необходима её *модель*, описанная с помощью ТТМЛ, и некоторый метод, обеспечивающий переход от реальной системы к модели. Метод может быть трансляцией реальной программы в некоторый код, о котором можно рассуждать автоматизировано с помощью ТТМЛ; а может быть ручной установкой соответствия между реальной системой и абстрактной моделью.

Назовём только что описанный метод *аналитическим* — потому что он анализирует реальную систему. У его обоих вариантов есть общий недостаток, состоящий в том, что в случае необходимости изменить реальную систему необходимо вручную менять и абстрактную модель вместе с доказательствами, построенными для неё. В случае использования автоматической трансляции возможен некоторый автоматизированный анализ модели, который может нивелировать небольшие изменения, однако возможности автоматического доказательства теорем ограничены, а значит ручной корректировки модели не избежать.

В моей работе же я предлагаю *синтетический* метод. Он заключается в том, что модель алгоритма или системы с самого начала описывается в терминах ТТМЛ, а затем из неё *извлекается*[8] промышленное решение. При применении этого метода по-прежнему необходимо корректировать модель и доказательства при изменении системы, но преимущество в том, что описание системы производится только на языке ТТМЛ, поэтому код и теоремы сильнее связаны, чем при аналитическом подходе — раньше замечаются расхождения моделей и необходимые изменения производятся на одном уровне (уровне ТТМЛ), а не одновременно в реальной системе и в абстрактной модели. Такой подход больше напоминает метапрограммирование.

В аналитическом методе требовался инструмент, сопоставляющий реальной системе некоторую абстрактную модель; в синтетическом же методе требуется обратный инструмент — транслятор из абстрактной модели в реальную. Оба описанных метода требуют верификации этих дополнительных инструментов, однако верификация абстрактной модели и верификация дополнительного инструмента по отдельности может быть проще, чем верификация монолитной промышленной системы, за счёт разбиения задачи на ортогональные модули. Схожий принцип называется принципом или критерием де Брёйна[9].

Этот принцип можно применить к задаче верификации системы не только выделив раздельные этапы верификации, но и выделив критические фрагменты системы, корректность которых необходимо доказать. Таким образом снижается требуемое на верификацию количество работы, жертвуя надёжностью некоторых не столь важных компонент. Например, система может состоять из некоторого алгоритмического ядра и пользовательского интерфейса. Предположим, что от ядра зависит сохранность и приватность личных данных пользователей, а от интерфейса — только удобство использования. В таком случае, нецелесообразно принимать во внимание детали пользовательского интерфейса при верификации системы, а имеет смысл сосредоточить усилия на ядре.

[8] От английского *extraction* — названия процесса трансляции кода на Coq в код на других языках, совмещённого со *стиранием* верификационной информации.

[9] Этот принцип вкратце о том, что чтобы успешно верифицировать что-либо, мы должны выделить достаточно маленькое *ядро*, корректность которого достаточно просто доказать.[13]

Резюмируя вышеописанные рассуждения, приведу список этапов, который я предлагаю в качестве сценария разработки промышленной системы с доказанной корректностью:

1. Выделение из планируемой системы критического подмножества — главного алгоритма или ядра, которое стыкует различные модули системы.
2. Описание модели этого подмножества с помощью программной реализации ТТМЛ (например, Coq).
3. Формулирование ключевых свойств этого подмножества и их доказательство.
4. Построение и верификация транслятора абстрактной модели в реальную систему (в данной работе будет показано, как это можно сделать для системы, реализуемой на Java, но пока что без верификации трансляции).
5. Разработка остальной части системы на более практическом языке программирования, соединение этой части с транслированным из ТТМЛ ядром.

3.2. Инструмент экстракции Java-кода по Coq-спецификации

Одна из поставленных задач — снабдить выбранный *proof assistant* (Coq) возможностью генерации кода на промышленном языке программирования. Решение разработано на основе уже существующего в системе Coq модуля *экстракции.* Этот модуль, реализованный в виде отдельного плагина, позволяет генерировать код на языках OCaml, Scheme и Haskell. Однако эти языки практически не используются в индустрии. Поэтому было решено модифицировать данный плагин с целью добавить поддержку Java как целевого языка трансляции. Язык Java выбран из соображений как удобства генерации, поскольку он достаточно высокоуровневый, чтобы во время разработки не думать о деталях вроде управления памятью; так и востребованности — этот язык является одним из наиболее распространённых в промышленной разработке.

Языки OCaml, Scheme и Haskell относятся в семейству т.н. *функциональных* языков программирования, в отличие от Java. Это значит, что они (как и Coq) обладают рядом выразительных средств для работы с функциями в качестве *значений первого порядка.* К примеру, в них можно передавать функции в другие функции как параметры; объявлять функции, которые *замыкают* контекст внешней функции в месте объявления; *частично применять* функции или же конструировать новые функции-объекты с помощью *лямбда-нотации* [14].

В свою же очередь, язык Java большинство этих концептов напрямую не поддерживает, однако позволяет некоторые из них смоделировать. Например, создание функции в произвольном месте программы может быть представлено с помощью анонимного класса. Более того, существуют достаточно крупные библиотеки, моделирующие функциональные концепты в Java[10]. В версии Java8 поддержка функционального программирования значительно улучшена, однако генерация кода будет производиться для Java7: ради совместимости генерируемого кода со старыми версиями Java; а также ради выработки более общего подхода который можно будет с некоторой адаптацией применить к другим не-функциональным языкам программирования.

Таким образом, задача сводится к трансляции кода на OCaml, Scheme или Haskell в код на Java. Из этих трёх языков был выбран OCaml— его поддержка была добавлена изначально, поэтому плагин использует промежуточные структуры (синтаксические деревья, представления типов и т.п.), максимально приближенные к его структуре. Обсуждаемая модификация плагина использует эти промежуточные структуры и адаптирует их для языка без поддержки функциональной парадигмы. Таким образом, задача облегчается, т.к. не приходится иметь дело с достаточно сложным языком Coq, а достаточно воспользоваться уже разобранными промежуточными структурами. Далее следуют абстрактные описания методов, используемых во время трансля-

[10] http://functionaljava.org

ции для моделирования функциональных концептов в Java с примерами генерируемого кода.

Кратко всю проделанную модификацию можно разделить на две части:

1. трансляция *алгебраических типов данных* вместе с правилами для конструирования и *элиминирования* их значений, т.е. конструкторов и *сопоставления с образцом*;

2. трансляция функциональных типов данных вместе с правилами конструирования и элиминирования их значений; это включает в себя генерацию локальных и глобальных объявлений функций, применений функций (в т.ч. частичных), *инстанцирование* типовых переменных во время применения функций.

Трансляция алгебраических типов данных

Произвольный алгебраический тип данных моделируются с помощью абстрактного класса с некоторым *тэгом* — полем перечисляемого типа (*enum*), содержащего по одному значению на каждый конструктор данного АТД. Каждый конструктор АТД также транслируется в класс, реализующий этот абстрактный класс, с единственным конструктором, полями-аргументами конструктора и тэгом, соответствующим этому конструктору.

Пример трансляции определения функционального списка:

Coq

```
Inductive list (A : Type) : Type :=
 | cons : A -> list A -> list A
 | nil  : list A.
```

Java

```
public static abstract class list<A> {
  public Tag tag;
  public static enum Tag { nil, cons }
  public static final class nil<A> extends list<A> {
    public nil() {
      tag = Tag.nil;
    }
  }
  public static final class cons<A> extends list<A> {
    public final A field0;
    public final list<A> field1;
    public cons(final A arg0, final list<A> arg1) {
      field0 = arg0;
      field1 = arg1;
      tag = Tag.cons;
    }
  }
}
```

Такое представление позволяет легко воспроизводить *сопоставление с образцом*. По сути, в Java уже есть вариант сопоставления с образцом — switch-конструкция, но она ограничена на *перечислимые типы*, которые являются только частным случа-

ем алгебраических типов данных. Чтобы полностью транслировать сопоставление с образцом, необходимо также уметь сопоставлять значения *типов-произведений*; это реализуется с помощью обычного обращения к полю конструктора. Таким образом, мы имеем обе необходимые формы ветвления (по конструкторам и по полям) в Java и нужно только правильно их чередовать в соответствии с исходным кодом. Такой подход обрабатывает в том числе и вложенные образцы, т.к. они разворачиваются в последовательность сопоставлений во время получения абстрактного синтаксического дерева OCaml.

Пример трансляции простой функции, вычисляющей длину списка *len* [11]:

Coq

```
Fixpoint len {X} (xs : list X) : nat :=
  match xs with
  | cons x xs' => S (len xs')
  | nil => O
  end.
```

Java

```
...
nat var0 = null;
switch (((list<A>)arg1).tag) {
  case nil: {
    final list.nil<A> var1 = (list.nil<A>)arg1;
    final nat var2 = new nat.O();
    var0 = var2;
    break;
  }
  case cons: {
    final list.cons<A> var3 = (list.cons<A>)arg1;
    final nat var4 = len.apply(var3.field1);
    final nat var5 = new nat.S(var4);
    var0 = var5;
    break;
  }
}
return nat.var0;
...
```

В приведённом фрагменте видно, что код генерируется не самый оптимальный — к примеру, создаются промежуточные переменные, которые используются в следующей же инструкции. Это следствие того, что транслятор реализован как можно проще.

Трансляция функций как значений первого порядка

Java-код выше также не является самодостаточным определением функции `len`, его нужно дополнить подходящей сигнатурой функции (содержащей параметр `arg1` типа `list<A>`); однако сигнатуры вроде `<A> nat len(list<A> arg1) { ... }` недостаточно: в коде на функциональном языке программирования функция `len` может использоваться как *значение первого порядка* (к примеру, быть аргументом вызова другой функции).

[11] Здесь `len.apply(...)` — рекурсивный вызов, а `nat` — тип натуральных чисел в аксиоматике Пеано: `nat.S n = n + 1` и `nat.O = 0`

Для моделирования функций как значений первого порядка в JAVA обычно используется классы, реализующие некоторый *обобщённый* (*generic*) интерфейс `Function<From,To>` с типовыми переменными `From` и `To`, соответствующими типу аргумента и типу возвращаемых значений функции соответственно, и методом `apply`, имеющим сигнатуру `To apply(From arg) { ... }`. Функция от нескольких аргументов при этом моделируется с помощью функции, возвращаемое значение которой само является функцией.

Воспользовавшись таким подходом, мы бы закодировали нашу функцию `len` следующим образом:

Function.java

```
public interface Function<From,To> {
    To apply(final From arg);
}
```

Len.java

```
public class len<A> implements Function<list<A>,nat> {
  @Override
  public nat apply(final list<A> arg1) {
    ...
  }
}
```

В случае локального определения функции `len` класс будет анонимным, а полученный объект будет присвоен некоторой переменной:

Len.java

```
public <A> void test() {
  final Function<list<A>,nat> var = new Function<list<A>,nat>() {
    @Override
    public nat apply(final list<A> arg) {
      ...
    }
  };
}
```

После этого, можно несколько улучшить код, генерируемый для глобальных определений функций. В общем случае, в функциональных языках глобальное определение функции от n аргументов может быть термом, состоящим из $m \leq n$ *лямбда-абстракций* и тела, имеющего тип $(n-m)$-арной функции; а применение определённой функции может производиться к $k \leq n$ аргументам.

Также можно заметить, что одному и тому же глобальному определению n-арной функции на языке Coq можно сопоставить n представлений на языке Java, подходящих под описание данное выше, различающихся количеством i аргументов верхнего уровня, используемых в теле функции, которая возвращает локально определённую функцию $(n-m)$ аргументов. Дадим названия основным видам таких представлений:

$i = 0$	лямбда-представление
$i \in (0, m)$	промежуточное представление
$i = m$	основное представление
$i \in (m, n)$	расширенное представление
$i = n$	полное представление

Таким образом, основное представление — полностью соответствующее исходному, в нём нет лишних лямбда-абстракций (моделируемых анонимными классами) и нет неиспользуемых в теле определения аргументов. Если для функции f генерировать только такое представление, то во время применения f к $k < m$ аргументам будет появляться $(m - k)$ лишних вызовов метода `apply`; а при передаче f аргументом в функцию высшего порядка g, ожидающую k-арный аргумент $(k > m)$, необходимо дополнительно локально определять f', состоящую из $(k - m)$ лямбда-абстракций.

Поэтому, чтобы улучшить читаемость генерируемого кода и облегчить написание транслятора, предлагается заранее генерировать все n представлений: базовое, по описанному выше методу; расширенные представления, применяя метод `apply` нужное число раз; промежуточные представления, добавляя лямбда-абстракции. Крайние случаи расширенного и промежуточного представлений — полное, использующее n аргументов, и лямбда-представление, совсем не использующее аргументов.

Новый метод генерации кода можно продемонстрировать на функции *cnot*. Эта функция из теории квантовых вычислений очень хорошо подходит для демонстрации подхода: у неё 2 аргумента и в определении используется только одна лямбда-абстракция аргумента *control*, в зависимости от которого функция равна либо тождественному отображению, либо логическому отрицанию (в Coq — `id` и `negb` соответственно). Поэтому транслированный код должен содержать ровно по 1 виду представлений без промежуточных: полное, основное и лямбда-представление; что является довольно наглядным примером.

Генерируемый по новому методу код приведён на следующей странице. В этом фрагменте можно заметить, что использование функций `id` и `negb` транслировано как `id.apply()` и `negb.apply()` соответственно вместо более громоздкой записи с добавлением лишнего анонимного класса.

Пример подобной более громоздкой записи:

CNot.java

```
Function<bool,bool> var0 = null;
switch (((bool)arg1).tag) {
...
  final Function<bool,bool> var2 = new Function<bool,bool>() {
    @Override
    public bool apply(final bool arg) {
      negb.apply(arg);
    }
  }
  var0 = var2;
...
  final Function<bool,bool> var4 = new Function<bool,bool>() {
    @Override
    public bool apply(final bool arg) {
      id.apply(arg);
    }
  }
  var0 = var2;
...
}
return (Function<bool,bool>)var0;
```

Полный пример того, что генерируется в соответствии с новым методом:

Coq

```
Definition cnot (control : bool) :=
  if control then negb
             else id.
```

Java

```
public static class cnot {
  public static bool apply(final bool arg1, final bool arg2) {
    final Function<bool,bool> lambda = cnot.apply(arg1);
    return lambda.apply(arg2);
  }
  public static Function<bool,bool> apply(final bool arg1) {
    Function<bool,bool> var0 = null;
    switch (((bool)arg1).tag) {
      case true: {
        final bool.true var1 = (bool.true)arg1;
        final Function<bool,bool> var2 = negb.apply();
        var0 = var2;
        break;
      }
      case false: {
        final bool.false var3 = (bool.false)arg1;
        final Function<bool,bool> var4 = id.apply();
        var0 = var4;
        break;
      }
    }
    return (Function<bool,bool>)var0;
  }
  public static Function<bool,Function<bool,bool>> apply() {
    return new Function<bool,Function<bool,bool>>() {
      @Override
      public Function<bool,bool> apply(final bool arg1) {
        return cnot.apply(arg1);
      }
    };
  }
}
```

4. Апробация

4.1. Метод операционных преобразований

Предложенный мной подход был апробирован на методе операционных преобразований (*operational transformation*). Этот метод представляет собой семейство алгоритмов *оптимистичной синхронизации* — во время их работы синхронизируемым узлам разрешено иметь расходящиеся копии данных, но гарантируется, что изменения правильно распространяются на все узлы сети и что копии данных станут равными через какое-то время, если остановить пользовательский ввод.

Наиболее известен этот алгоритм тем, что используется в *коллаборативных редакторах реального времени*. Подобные редакторы — приложения, позволяющие нескольким людям редактировать одни данные, используя разные компьютеры *одновременно*. Самым известным примером является Google Docs.

Сам же метод операционных преобразований впервые был предложен Эллисом и Гиббсом в 1989 году[15]. Его преимущество состоит в том, что все изменения, производимые клиентом, применяются к локальной копии данных сразу же, не требуя блокировки; потому он даёт приемлемую производительность ввиду среды с высокими задержками, в которой работают пользователи. При этом, часть алгоритма описывает распространение локальных изменений между всеми пользователями. Поскольку доставка изменений занимает некоторое нефиксированное время, они могут быть обработаны в порядке, отличающемся от порядка их создания пользователями, и при этом на каждой локальной копии этот порядок может отличаться от порядка другой копии. Поэтому изменения, доставленные клиенту в некоторый момент времени могут иметь отличный от первоначального смысл или вообще не иметь его. Метод операционных преобразований как раз даёт способ преобразовать подобные изменения к таким, которые бы имели корректный смысл, и, как следствие, позволяет гарантировать равенство всех локальных копий (свойство *консистентности*) после обработки всех изменений. Метод состоит из двух частей: алгоритма *преобразования* операций, абстрагирующегося от сетевого взаимодействия и описанного в терминах *моделей* и *операций* над ними, и алгоритма *управления*, распространяющего изменения между узлами сети и вызывающего процедуру преобразования.

Своё название метод берёт от базового концепта — *операций*, таких как вставка символа в текстовый документ или удаление. Возможны и более сложные операции: например, работающие с древовидной моделью данных. *Применением* операции к *модели данных* называют преобразование данных узла в соответствии с некоторой заранее заданной функцией. К примеру, применение операции вставки символа к списку символов преобразовывает его в новый список символов, содержащий заданный символ на заданной позиции. Операции инициируются пользователем на некотором узле

сети и применяются к локальной копии данных, а затем доставляются до всех остальных узлов сети и обрабатываются некоторым *способом* на них. Этот способ должен быть чем-то сложнее *наивного* варианта, в котором изменения с других узлов просто применяются на данном узле. В качестве такого способа подход операционных преобразований предлагает перед применением *преобразовывать* операции *через* операции, представляющие собственные изменения.

Рассмотрим понятие *преобразования* на конкретном примере:

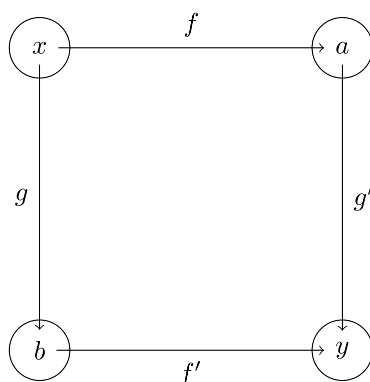

Рис. 1: Диаграмма двух операций f и g для одной модели x.

На данной диаграмме стрелки f и g — некоторые операции, которые переводят общую модель x в две в общем случае неравные модели a и b; а g' и f' — операции, применяемые для восстановления консистентности и полученные каким-то образом по операциям g и f соответственно. Говорят, что все эти операции *коммутируют*, т.е. составная операция $g' \circ f = f' \circ g$ переводит модель x в модель y:

$$g'(f(x)) = y$$
$$f'(g(x)) = y$$

В качестве конкретного примера таких f и g, что их преобразование не тривиально, можно привести модель "abd" вместе с операциями $insert(3,'c')$ вставки символа $'c'$ на третью позицию строки и $remove(2)$ удаления второго символа строки. Инстанцируем предыдущую диаграмму для этих операций.

В наивном варианте эта диаграмма выглядит так:

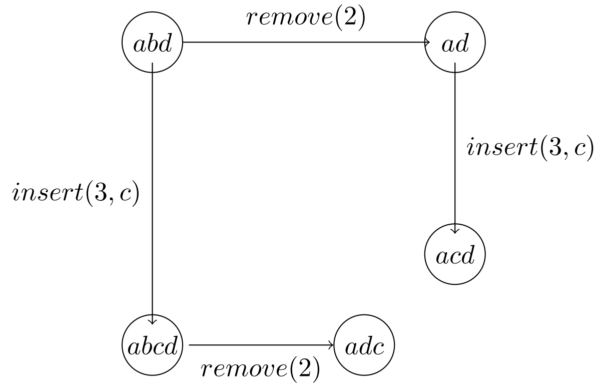

Рис. 2: Наивный вариант — вместо преобразования просто применям чужие изменения. Диаграмма не коммутирует, т.е. $g'(f(x)) \neq f'(g(x))$.

В данном случае нам бы подошли $f' = insert(2,'c')$ и $g' = remove(2)$:

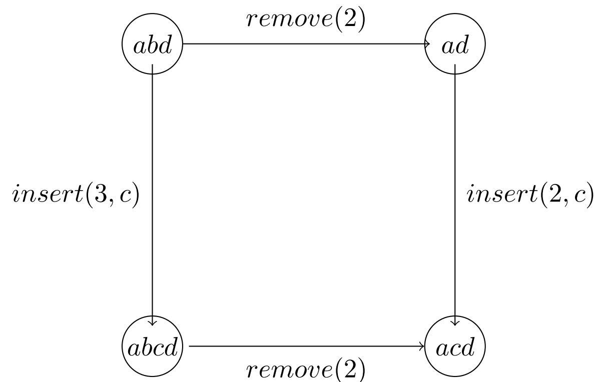

Рис. 3: Корректный вариант — операцию $insert(3,'c')$ преобразуем в $insert(2,'c')$. Диаграмма коммутирует, конечная модель — "acd".

Для вычисления функций f' и g' метод операционных преобразований предписывает задать некоторую функцию tr, которая преобразует одну операцию через другую таким образом, чтобы преобразованные операции *коммутировали* с исходными. Эта функция принимает пару операций и возвращает пару преобразованных друг через друга операций: $tr(f, g) = (g', f')$.

Важно заметить, что для корректного построения алгоритма операционных преобразований необходимо рассмотреть довольно большое число случаев, поскольку для n базовых операций естественным образом появляется n^2 случаев. Например, вполне

обоснованным выглядит базис из 4 операций: вставки, удаления, разделения и слияния, в таком базисе появляется 16 простейших случаев: вставка-вставка, вставка-удаление и т.д. Убедиться в корректности алгоритма без применения формальных методов уже для такого базиса становится трудно.

Нетривиальность этой задачи была продемонстрирована на реальных реализациях алгоритма операционных преобразований. Было обнаружено, что система Grove [15], разработанная Эллисом и Гиббсом, и система Joint Emacs не удовлетворяют некоторым важным свойствам, про что можно узнать подробнее из статей [16] и [17] соответственно. Обнаружение ошибок в известных реализациях метода операционных преобразований послужило дополнительной мотивацией для того, чтобы верифицировать алгоритм формально.

Формализация части подхода, касающейся собственно преобразования, проводилась Сергеем Синчуком и Антоном Милениным и будет подробно освещена в готовящейся статье “Certified Tree Operational Transformation”. В ней описан метод операционных преборазований в применении к древовидной модели данных.

4.2. Формализация алгоритма управления с помощью Coq

Моя работа проводилась в сотрудничестве с Сергеем Синчуком и Антоном Милениным. Ими была проделана внушительная работа по верификации базового алгоритма для нескольких моделей данных и для нескольких моделей операций (среди них в том числе и модель, подобная файловой системе, вместе со сложными операциями вроде разделения и слияния). Мой же вклад заключался в построении модели и алгоритма *управления*, описывающих сетевое взаимодействие между узлами, распространение изменений от одного клиента к другому и своевременный вызов процедуры преобразования.

В рамках формализации модели было решено разделить её на два случая в духе индуктивных доказательств: простейший случай для одного соединения (сети из двух элементов) и случай для n элементов (параллельная композия из n соединений с сервером, топология “звезда”). Предполагается, что доказать индуктивный шаг не представляет труда, однако пока что это не реализовано. База индукции же верифицирована.

Для начала определим базовые понятия. Алгоритм имеет два параметра: тип `model`, задающий возможные модели данных, и тип `cmd`, задающий возможные операции. Сетевой узел представляется структурой, содержащей модель данных, историю локально применённых операций (задаётся при помощи списка операций) и очередь входящих сообщений от других узлов. Сообщения бывают двух видов: уведомление (`Ack`) и команда выполнить некоторые изменения (`Do`), сообщения последнего вида содержат также список операций для выполнения.

Привожу сжатую версию модели сетевого взаимодействия на Coq вместе с генерируемым Java-кодом:

Coq

```
Inductive Message : Type :=
  | Do  : list cmd -> Message
  | Ack : Message.

Record Node := {
  data    : model;
  history : list cmd;
  input   : {
    size  : nat &
    queue Message size
  }
}.

Record Network := {
  first  : Node;
  second : Node
}.

Separate Extraction
  Network Node Message.
```

Java

```
public static abstract class Message {
    public Tag tag;
    public static enum Tag { Do, Ack }
    public static final class Do extends Message {
        public final list<cmd> field0;
        public Do(list<cmd> arg0) {
            field0 = arg0;
            tag = Tag.Do;
        }
    }
    public static final class Ack extends Message {
        public Ack() { tag = Tag.Ack; }
    }
}
public static abstract class Node {
    public Tag tag;
    public static enum Tag { RNode }
    public static final class RNode extends Node {
        public final model field0;
        public final list<cmd> field1;
        public final sigT<nat,queue<Message>> field2;
        public RNode(model arg0, list<cmd> arg1,
                  sigT<nat,queue<Message>> arg2) {
            field0 = arg0;
            field1 = arg1;
            field2 = arg2;
            tag = Tag.RNode;
        }
    }
}
public static abstract class Network {
    public Tag tag;
    public static enum Tag { RNetwork }
    public static final class RNetwork
                  extends Network {
        public final Node field0;
        public final Node field1;
        public RNetwork(Node arg0, Node arg1) {
            field0 = arg0;
            field1 = arg1;
            tag = Tag.RNetwork;
        }
    }
}
```

Рассматриваемый алгоритм посылки и обработки сообщений в общем случае (n узлов) состоит из следующих шагов:

1. Сообщение Do, соответствующее операциям пользователя передаётся серверу s_0.
2. После получения сообщения Do от узла s_i сервер:
   (a) изменяет локальную модель данных;
   (b) уведомляет отправителя о том, что его сообщение получено (сообщение Ack);
   (c) вычисляет преобразованную версию операций из сообщения (функция tr);
   (d) рассылает сообщения Do с преобразованными операциями по узлам $s_j, j \neq i$.
3. Клиенты получают сообщения Do от сервера, сегенерированные на шаге 2, проводят дальнейшие преобразования и обновляют локальные копии.

В вырожденном случае для сети из двух узлов шаг 3 не достигается, т.к. сервер не посылает никаких сообщений на шаге 2. Этот вырожденный случай можно закодировать следующим образом:

Псевдокод

```
handle (Do cmds) (current : Node) :=
  match (tr cmds (history current)) with
  | (new_history, new_cmds) =>
     match (ex new_cmds (data current)) with
     | Some new_data => current.history := new_history;
                        current.data := new_data
     | None => <error>
     end
  end.

receive (current other : Node) :=
  msg := pop current;
  match msg with
  | Do _ => handle msg;
            send Ack other
  | Ack => ()
  end.
```

При формализации алгоритмов бывает удобно работать с бинарными отношениями вместо функций. Поэтому введём понятие рефлексивно-транзитивного замыкания:

Coq

```
Inductive closure {X} (step : X -> X -> Prop) : (X -> X -> Prop) :=
  | id    : forall {x     : X}, transitive step x x
  | chain : forall {x y z : X}, step y z ->
      transitive step x y ->
      transitive step x z.
```

Функция `closure` принимает некоторое бинарное отношение и возвращает его рефлексивно-транзитивное замыкание. Используя это определение, обозначим различные бинарные отношения на множестве состояний сети:

в сети произошла передача сообщения	`receive_rel`
в сети произошёл пользовательский ввод	`user_rel`
любое событие (переход из состояния x в y)	`transition (x y : Network) :=` `user_rel x y   receive_rel x y`
произвольный отрезок событий в сети (рефлексивно-транзитивное замыкание `transition`)	`life := closure transition`
отрезок событий в сети, состоящий из только передачи сообщений (рефлексивно-транзитивное замыкание `receive_rel`)	`stabilization (x y : Network) :=` `closure reveive_rel`

Теперь произвольное развитие нашей сети может быть обозначено `life`, а развитие сети под воздействием только алгоритма управления — `stabilization`. В такой формулировке можно доказывать какие-либо свойства об алгоритме, например такое (доказательство строится на основании многочисленных лемм, которые здесь не приводятся):

Coq

```
Theorem stability : forall net, exists net',
  stabilization net net' /\ consistent net'.
Proof.
  intros net N. inversion_clear N as [d L].
  remember (stabilization_ready net) as SR.
  inversion SR; inversion_clear H as [S R].
  remember
    (apptransitive L (stabilization_life S))
    as L'.
  assert (N : natural x) by (
    unfold natural; exact (ex_intro _ d L')).
  exact (ex_intro _ x (conj S
    (consistency x (conj N R)))).
Qed.
```

В приведённом фрагменте формально доказано свойство, что существует некоторый вариант развития сети при остановленном пользовательском вводе, при котором сеть приходит в консистентное состояние. Следует, однако, заметить, что это свойство не гарантирует безопасности, т.к. доказывает *существование* благоприятного исхода, а не то, что любой исход — благоприятный. Данное свойство приведено исключительно из-за краткости формулировки и доказательства.

5. Результаты

В рамках данной работы поставленные задачи были решены, а именно:

1. Разработан сценарий использования инструментов для построения доказательств на основе ТТМЛ. Рассмотрены альтернативы, обоснован выбор применённого метода.
2. Стандартная функциональность экстракции Coq расширена поддержкой генерации Java-кода.
3. Метод успешно апробирован на алгоритме операционных преобразований: автором лично формализован один из случаев модуля сетевого взаимодействия. Есть свидетельства того, что метод успешно применялся для верификации других частей алгоритма.

Заключение и возможность развития

Главное направление дальнейшего развития данной работы — формальная верификация транслятора Coq в Java, поскольку это бы гарантировало корректность всей системы в целом, а не только её абстрактной модели.

Кроме того, важно расширить верификацию алгоритма управления на полноценный случай из n вершин; также не было бы лишним рассмотреть возможность конструирования и формализации подобного алгоритма для произвольной топологии, а не только топологии "звезда".

В целом, метод выглядит жизнеспособным и обоснованным; он способен породить массу полезных и интересных применений.

Список литературы

[1] P. Martin-Löf and G. Sambin. *Intuitionistic type theory.* Studies in proof theory. Bibliopolis, 1984.

[2] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[3] B. Russell. *The Principles of Mathematics.* Number v. 1 in The Principles of Mathematics. University Press, 1903.

[4] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908.

[5] A.S. Troelstra. History of constructivism in the 20th century.

[6] Subashis Chakraborty. Curry-howard-lambek correspondence.

[7] John L. Bell. Types, sets and categories.

[8] Thierry Coquand and Peter Dybjer. Inductive definitions and type theory: An introduction.

[9] Kent Petersson Bengt Nordström and Jan M. Smith. *Programming in Martin-Lof Type Theory.* 1990.

[10] Simon Thompson. *Type Theory and Functional Programming.* 1999.

[11] Benjamin C. Pierce et al. *Software Foundations.*

[12] Adam Chlipala. *Certified Programming with Dependent Types.*

[13] J.A. Robinson and A. Voronkov. *Handbook of Automated Reasoning.* Number v. 2 in Handbook of Automated Reasoning. Elsevier, 2001.

[14] A.J. Field and P.G. Harrison. *Functional programming.* International computer science series. Addison-Wesley, 1988.

[15] C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proc. 1989 ACM SIGMOD Int. Conf. on Management of data*, volume 18, pages 399–407, 1989.

[16] A. Imine, M. Rusinowitch, P. Molli, and G. Oster. Formal design and verification of operational transformation for copies convergence. *Theor. Comp. Sci.*, 351(2):167–183, 2006.

[17] G. Cormack. A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication. Technical report, Univ. Waterloo, 1995.