МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
01.03.02 — Прикладная математика
и информатика

ВЕРИФИКАЦИЯ СТРУКТУРЫ ДАННЫХ «ЗИППЕР»

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
П. А. Грахова

Научный руководитель:
ст. п. В. Н. Брагилевский

Допущено к защите:
руководитель направления _____ В. С. Пилиди

Ростов-на-Дону
2019

Содержание

Введение 3

1. Структура данных «Зиппер» 5
   1.1. «Зиппер» для обобщенного дерева 6

2. Формальная верификация в системе Coq 8
   2.1. Теоретические основания 8
   2.2. Доказательство корректности 10
      2.2.1. Предварительные операции над списками 11
      2.2.2. Замечания по автоматизации 14
      2.2.3. Доказательство корректности «Зиппер» 17

3. Тестирование с использованием QuickChick 21
   3.1. Генерация данных 21
   3.2. Тестирование 22

4. Генерация верифицированного Haskell кода 23
   4.1. Теоретические основания 24
   4.2. Построение генератора кода 27

Заключение 29

Список литературы 30

Введение

В процессе разработки программного обеспечения важно, удовлетворяет ли написанный программистом код некоторым заранее заданным спецификациям, согласно которым должны были быть разработаны алгоритмы, выраженные в коде. Этот процесс проверки корректности кода, или, иначе, верификации, может быть построен как с использованием так называемых unit-тестов, проверяющих программу на большом количестве разнообразных входных данных, так и с помощью формального доказательства корректности интересующих нас участков кода. Несмотря на то, что второй вариант тестирования может формально гарантировать абсолютную корректность, в силу своей сложности он используется сравнительно редко, в промышленности гораздо более популярен первый способ. Однако в последнее время с развитием систем автоматической верификации теорем приобретает популярность использование комбинации этих двух методов, когда некоторые критически важные участки кода либо алгоритмы формулируются в виде теорем и верифицируются формально в подобных средах, в то время как корректность второстепенных частей «доказывается» посредством прогонки на тестах, возможно, в этих же средах, с помощью некоторых их расширений.

Одной из самых популярных систем автоматической верификации теорем является система Coq НУЖНА ССЫЛКА, разработанная в INRIA. В этой среде для тестирования написанных определений и теорем может использоваться плагин QuickChick, позволяющий сгенерировать набор случайных входных данных и проверить, является ли интересующая нас теорема либо свойство верными на этом наборе. Подобная проверка не гарантирует абсолютной правильности проверенных теорем (то есть то, что они будут верны для любых входных данных), но вероятность этого весьма высока, если был успешно пройден подобный unit-тест, что может вдохновить программиста либо исследователя на поиск настоящего доказательства.

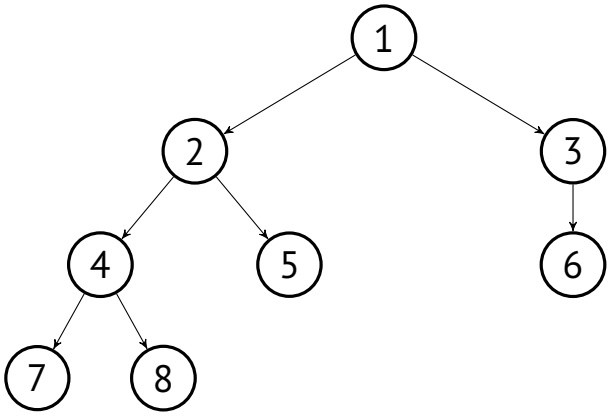
В данной работе ставится цель исследовать одну структуру данных с позиций подобного стиля верификации. Более точно, исследуется вариант структуры данных «Зиппер» для обобщенных деревьев, описывается сама структура данных и набор функций для работы с ней, формулируются желаемые свойства, которым должна подобная структура данных удовлетворять, составляются тесты и доказывается корректность некоторых свойств в нашей формулировке, затем на базе полученной формализации генерируется верифицированный код на языке Haskell. Похожий процесс при разработке ПО выполняется часто, цель данной работы — показать, что Coq может являться удобной средой для подобных манипуляций, когда важен высокий уровень доверия к полученным алгоритмам.

Код, сопровождающий данную работу, выложен на GitHub [1].

1. Структура данных «Зиппер»

При реализации алгоритмов на чисто функциональных языках программирования часто возникают проблемы с производительностью, порожденные невозможностью прямой модификации данных в таких языках. Эти проблемы можно решать разработкой структур данных, не требующих полного воссоздания исходной структуры при модификации некоторой ее части.

Можно продемонстрировать проблему на примере несложной структуры данных, например, бинарное дерево.




```
type A tree = T_nil
    | T_tr of A * A tree * A tree
```

Рисунок 1 — Некоторое бинарное дерево и код на языке OCaml, описывающий тип соответствующей структуры данных

Пусть, например, мы хотим изменить узел со значением 4. Поскольку данные нельзя модифицировать, мы создаем новый узел, отражающий желаемые модификации, и присваиваем ему в качестве левого и правого потомка узлы 7 и 8, соответственно. Однако 2 должно ссылаться на новую 4, что вынуждает нас копировать узел 2 и менять в нем ссылку на левого потомка. Затем мы проводим подобную операцию с корнем дерева. Таким образом, модификация, произведенная в узле, требует в дальнейшем копирования родительского уз-

ла, предка того узла и так далее до корня дерева, что занимает $O(n)$ времени (где n – расстояние от корня до узла) и плохо сказывается на производительности.

Мы могли бы представить исходное дерево как пару, состоящую из поддерева, порожденного узлом 4, и некоторой иной структуры, содержащей в компактном виде информацию, позволяющую восстановить исходное дерево. В таком случае, меняя что-то в 4, нам не требовалось бы ничего менять в предках этого узла, и модификация занимала бы константное время $O(1)$. Именно эта идея и лежит в основе структуры данных «Зиппер» (Zipper), описанной Gerard Huet [2].

Строго говоря, «Зиппер» — не структура данных, а конструкция, позволяющая на базе некоторой сложной структуры данных построить новую структуру, поддерживающую обращение лишь к той части исходных данных, которые требуют модификации, без изменения всего остального [2]. В данной работе рассматривается «Зиппер» для обобщенных деревьев.

1.1. «Зиппер» для обобщенного дерева

Любое дерево T можно представить как пару (называемую иногда в дальнейшем просто «Зиппером»), состоящую из его поддерева T' и связного списка с информацией, которая позволяет восстановить исходное дерево T. Корень поддерева T' назовем для удобства «курсором». Список с информацией состоит из элементов типа «контекст», однозначно порождаемого из типа исходного дерева.

Нужно коротко (на треть страницы) ввести и объяснить используемую далее нотацию как-то так: «Следуя [3], обозначим...»

Используя инструментарий из [3], если тип нашего дерева будет описан как (здесь и далее используется нотация из [3])

$$\textbf{tree} \mapsto \mu x.1 + x + x^2 + \ldots + x^n,\ n \in \mathbb{N},$$

то тип контекста может быть легко выведен

$$\textbf{context} \mapsto \partial_x \textbf{tree} = \partial_x \left(1 + x + x^2 + \ldots + x^n\right) = \\ = \partial_x 1 + \partial_x x + \ldots + \partial_x x^n = \\ = 1 + 2x + 3x^2 + \ldots + nx^{n-1},\ n \in \mathbb{N},$$

что может быть записано на языке Coq как

```
Inductive Context : Type :=
  | T_move: nat -> A -> list Tree -> Context.
```

Для удобства будем в дальнейшем обозначать «Зиппер» как Z. Дерево и массив с информацией, составляющие «Зиппер», будем обозначать как Z_T и Z_C, соответственно. Будем также писать $Z = (Z_T, Z_C)$.

Над структурой данных «Зиппер» определены следующие операции (если не указано иное, то ниже под T подразумевается некоторое дерево):

- **MoveTop Z** возвращает Z, полученное из исходного **Z** поднятием курсора на уровень выше, к родителю узла-курсора. Если это невозможно, **Z** возвращается неизменным. Выполняется за $O(1)$ времени.
- **MoveDown d Z** возвращает Z, полученное из исходного **Z** спуском курсора к d-ому ребенку узла-курсора. Если это невозможно, **Z** возвращается неизменным. Выполняется за $O(1)$ времени.
- **ZipperToTree Z** возвращает Z_T из Z, порожденного путем многократного применения **MoveTop** к **Z**, до тех пор, пока подобное применение не перестанет модифицировать **Z**. Выполняется за $O(n)$ времени, где n — расстояние от корня до текущего фокуса в исходном дереве.
- **TreeToZipper T** возвращает $Z = (T, nil)$, время — $O(1)$.
- **Modify Z f** для некоторого $Z = (Z_T, Z_C)$ возвращает новый «Зиппер» $Z' = (f Z_T, Z_C)$, функция f определена над деревьями. Вре-

мя выполнения этой операции — $O(m)$, где m — время выполнения функции f.

В случае возникновения ошибки вышеперечисленные операции оставляют исходные данные без изменений. Это было введено с целью упростить верификацию, однако может помешать при применении таких функций в разработке реальных приложений. С этой целью дополнительно вводятся функции, проверяющие допустимость входных данных, они будут описаны ниже.

2. Формальная верификация в системе Coq

2.1. Теоретические основания

Система интерактивного доказательства Coq предоставляет функциональный язык программирования Gallina, термы которого позволяют описывать программы, свойства программ и доказательства их корректности. В силу соответствия Карри–Ховарда объекты, описываемые в терминах некоторого функционального языка программирования (в данном случае Gallina), могут быть однозначно выражены в терминах некоторой логической системы, и наоборот. В Coq изоморфизмом, реализующим подобное соответствие, является так называемое исчисление индуктивных конструкций, или CoC. С точки зрения ядра Coq, все логические рассуждения есть также и рассуждения над типами. Таким образом, основной задачей ядра Coq является проверка корректности конструкций в рамках заданной системы типов, что и есть проверка корректности доказательства. Обычно такие доказательства пишутся с помощью конструкций языка Gallina и некоторого множества подпрограмм, называемых тактиками.

CoC CoC может рассматриваться как соответствие Карри–Ховарда между интуиционистской логикой и системой λC, находя-

щейся на вершине λ-куба Барендрегта Непонятно, у куба восемь вершин, о какой речь? Возможно, стоит описать точнее., что говорит о его исключительной выразительности и вычислительных возможностях. Ниже дано описание некоторых основных компонентов CoC, его полное описание можно найти в [4].

- **Термы и типы.** Все выражения в CoC называются термами либо типами, причем типы неотличимы от термов, за исключением того факта, что тип не может служить аргументом λ-абстракции. Если x имеет тип T, то пишут $x : T$. Термы определяются рекурсивно:

  1. **Prop** (тип логического утверждения), **Set** (тип конечного множества), **Type(i)**, $\forall i \in \mathbb{N}$ (все прочие типы, выводимые из двух базовых, коих бесконечное множество) — термы.
  2. Переменные (x, y, ...) и константы (a, b,...) — термы.
  3. Если x — переменная, и T, U — термы, то $\forall x : T, U$ и $\lambda x : T.U$ есть термы (унификация и λ-абстракция).
  4. Если T и U — термы, то применение $(T\ U)$ есть терм.
  5. Если x — переменная, и t, T, U — термы, то `let x := t:T in U` есть терм (так называемая «let-in» конструкция, свойственная функциональным языкам).

  Таким образом, в рамках CoC имеют смысл все понятия обычного типизированного λ-исчисления, такие как «свободные переменные» и «подстановка».

- **Логические операции.** CoC имеет лишь один базовый оператор для формирования логических утверждений: квантор всеобщности $\forall$. С помощью этого квантора выражаются все остальные

операции, например

$$A \implies B \equiv \forall x : A.B, \qquad (x \notin B)$$
$$A \wedge B \equiv \forall C : P.(A \implies B \implies C) \implies C$$

- **Стандартные типы данных.** Некоторые широко используемые типы данных выражены напрямую как термы CoC (иногда это выражение совпадает с выражением в кодировке Черча), например,

$$\textbf{Bool} \equiv \forall A : P.A \implies A \implies A$$
$$\textbf{Prod A B} \equiv A \wedge B$$

Пример изоморфизма Приведем ниже один из примеров изоморфизма Карри–Ховарда, реализуемый в терминах CoC.

Пусть имеется множество натуральных чисел $\mathbb{N}$ с нулевым элементом 0, которое можно рассматривать и как тип. Умножение чисел $x \cdot y$ можно рассматривать как функцию (терм) `mult` типа $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Аналогично можно рассматривать предикат сравнения $=$ как терм `eqnat` типа $\mathbb{N} \to \mathbb{N} \to$ `Prop`.

Тогда, к примеру, предикат $P(x) : x^2 = 0$ можно записать в терминах CoC как $\lambda x : \mathbb{N}.(\texttt{eqnat}\ (\texttt{mult}\ x\ x)\ 0)$. Этот терм будет иметь тип $\mathbb{N} \to$ `Prop`. К примеру, его применение к 1 будет $(\lambda x : \mathbb{N}.(\texttt{eqnat}\ (\texttt{mult}\ x\ x)\ 0))\ 1 \equiv (\texttt{eqnat}\ 1\ 0) \equiv \bot$. Откуда взялось $\bot$? Это обозначение нужно ввести и пояснить результат.

2.2. Доказательство корректности

Важно корректно описать изначальные структуры на внутреннем языке Coq, чтобы в дальнейшем избежать проблем с формальным доказательством интересующих нас свойств. Функциональная реали-

зация алгоритмов и структур данных в этой работе базируется в основном на идеях Окасаки [5].

Исходя из описания структуры данных «Зиппер», тип, описывающий его, может быть рассмотрен как произведение типов «дерево» и «список контекстов», где «контекст», в свою очередь, тоже тип, описанный ранее. Возможности стандартной библиотеки Coq позволяют описать «Зиппер» как независимый от базового типа модуль, который затем можно использовать повторно:

```
Module TreeZipper (Import T: Typ).
Definition A := T.t. (* Тип, индуцирующий дерево *)
Inductive Tree : Type :=
  | T_nil: Tree
  | T_tr:  A -> list Tree -> Tree
...
Definition ZipperTree := prod Tree (list Context).
...
```

Под базовым типом подразумевается тип, на котором построено исходное дерево. Как уже было сказано, введенные над «Зиппером» операции перед своим применением требуют проверки допустимости их применения. К примеру, операция **MoveDown D Z** не может быть применена в том случае, если D больше, чем количество потомков Z_T. Это легко выразить в Coq:

```
Definition CorrectMoveDownConditions D (Z: ZipperTree) :=
  D < length (NodesOf (fst Z)).
```

Аналогичные проверяющие предикаты были введены и для всех остальных операций.

2.2.1. Предварительные операции над списками

Поскольку операции над обобщенным деревом включают в себя операции над списками (в силу формального определения дерева),

то предварительно потребуется проделать некоторую работу с тем, чтобы определить минимальный набор таких операций и показать их корректность в том плане, который нам требуется. Все требуемые операции и доказательства верности нижеописанных их свойств вынесены в отдельный файл-модуль `Auxiliaries.v`.

Нам потребуются две основные операции над списками:

- `nth_remove n l` — удаление n-ого элемента из списка l;
- `nth_insert n l x` — добавление элемента x в список l перед элементом с индексом n (нумерация идет с нуля).

Если операцию провести невозможно, то список возвращается без изменений, в этом основное отличие новых операций от аналогичных функций в стандартной библиотеке Coq, возвращающих особое значение **None** в случае ошибки, что требует введения еще одного типа и что, как следствие, усложняет верификацию. В нашем случае, поскольку мы полагаемся на введенные выше проверяющие предикаты, такая дифференциация ошибок и неизменившихся исходных данных не требуется.

Представление списка после применения операции Когда мы вставляем элемент x в список в допустимую позицию, мы предполагаем, что после вставки элемента в список он будет находиться именно там, где мы хотели. Это вроде бы очевидное свойство тем не менее нуждается в доказательстве, когда речь идет не о некоторой абстрактной математической операции, но о вполне определенной функции, реализующей данный алгоритм. Иными словами, доказывая такое очевидное свойство, мы демонстрируем корректность введенной выше функции `nth_insert n l x`. Более того, подобные свойства, на которые можно положиться, упрощают в дальнейшем доказательства более сложных теорем.

Итак, если $n \leq \texttt{length}\ l$, то элемент x будет после вставки находиться внутри нового списка l' (либо их там будет на единицу больше),

и это будет единственное отличие l' от старого списка l, куда вставляется x (то есть найдутся такие списки a и b, что $a^\frown b = l \wedge a^\frown [x]^\frown b = l'$), и применение операции `nth_error` n из стандартной библиотеки к списку l' должна вернуть **Some** x.

Это свойство можно сформулировать в терминах Coq как

```
Lemma nth_insert_representation: forall n (l: list A) x,
  n <= length l ->
  Some x = nth_error (nth_insert n l x) n /\
  exists lh lt, nth_insert n l x = lh ++ x::lt /\ l = lh ++ lt
    /\ length lh = n /\ length lt = (length l) - n.
```

Доказывается оно индукцией по l, в случае пустого списка функция `nth_insert` имеет смысл лишь при $n = 0$, при таких условиях лемма верна, если $a = b = nil$. В ином случае, пусть $n = 0$. Тогда лемма верна при $a = nil, b = l$. В дальнейшем в подобных «неформальных» описаниях доказательств не будут особо рассматриваться элементарные случаи.

Итак, по предположению индукции, лемма верна для l и n таких, что $n \leq$ `length` l, требуется доказать это для $n' = n + 1$ и $l' = [a']^\frown l$. Отсюда можно вывести, что $n \leq$ `length` l, применить к этому факту предположение индукции, затем с использованием полученного представления переписать и упростить лемму, наконец, увидев, что желаемое свойство действительно выполняется. Автоматика Coq позволяет не проводить рутинные операции подстановки и арифметических вычислений вручную, но использовать для этого высокоуровневые тактики для работы с логикой первого порядка, такие как `firstorder` (вывод элементарных логических утверждений) и `omega` (доказательство утверждений в рамках арифметики Пресбургера).

Аналогично доказывается и подобная лемма для операции `nth_delete n l`

```
Lemma nth_remove_representation: forall n (l: list A),
  n < length l ->
  exists x lh lt, Some x = nth_error l n /\
```

```
    l = lh ++ x::lt /\ nth_remove n l = lh ++ lt
    /\ length lh = n /\ length lt = (length l) - n - 1.
```

Взаимосвязь функций nth_insert и nth_remove Если мы удалим из списка элемент, который только что вставили в него, то получим исходный список. В терминах Coq это утверждение может быть записано так:

```
Lemma nth_insert_remove: forall n l x,
  n < length l -> Some x = nth_error l n ->
  (nth_insert n (nth_remove n l) x) = l.
```

Для доказательства данного утверждения достаточно применить обе вышеописанные леммы.

2.2.2. Замечания по автоматизации

Как уже было сказано выше, использование высокоуровневых тактик и средств автоматизации упрощает доказательство. Здесь будет приведено сравнение доказательств одной и той же леммы, но первое из них проводится без какого-либо использования модулей автоматизации Coq, другое — использует их базовые возможности.

К примеру, требуется доказать несложную лемму

```
Lemma nth_remove_overflow: forall n l,
  n >= length l <-> nth_remove n l = l.
```

Первое доказательство является, по сути, прямой трансляцией строгого формального доказательства, проведенного человеком «на бумаге» без учета возможностей Coq.

```
Proof.
  induction n; destruct l; intros; simpl in *; split; intros.
  - reflexivity. (* 1 *)
  - apply le_n (* 2 *)
  - inversion H. (* 3 *)
  - apply not_lt. unfold not. intros. (* 4 *)
```

```
    apply f_equal with (f:=@length A) in H. simpl in H.
    induction (length l).
    + inversion H.
    + inversion H. apply IHn in H2.
      apply H2.
      rewrite <- H in H0. apply H0.
  - reflexivity. (* 5 *)
  - apply le_0_n. (* 6 *)
  - apply le_S_n in H. apply IHn in H. (* 7 *)
    rewrite H. reflexivity.
  - inversion H. apply IHn in H1. (* 8 *)
    apply f_equal with (B := list A) (f:=@tl A) in H. simpl in H.
    rewrite <- H in H1.
    apply le_n_S. apply H1.
Qed.
```

Вначале проведем индукцию по n, рассмотрим в каждом случае (необходимость и достаточность) варианты $l = nil$ и $l \neq nil$, всего получим 8 вариантов, которые нужно рассмотреть.

1. `nth_remove 0 [] = []` После исполнения левой части получим nil, обе части равны.

2. `0 >= length []` Очевидно, что $0 \geq 0$, но это утверждение здесь требует доказательства. Стандартная библиотека содержит этот факт под именем le_n.

3. Предположение `0 >= length (a::l)` лишено смысла, поскольку не выводится индуктивно из определения операции $\geq$ в Coq. Здесь требуется применить inversion.

4. Нужно доказать `0 >= length (a :: l)` в предположении `H: nth_remove 0 (a :: l) = a :: l`. То, что доказывается – абсурдно, поэтому работа сводится к нахождению противоречия (то есть к конструированию доказательства False, в терминах логики CoC). Таким образом, вводится предположение `H0: nth_remove 0 (a :: l) = a :: l`. Если применить к обе-

им частям H операцию length и упростить выражение, мы получим утверждение length l = S (length l). Оно ложно, мы показываем это индукцией по length l и применением inversion к невыводимым утверждениям (типам).

5. nth_remove (S n) [] = [], обе части равны, как в 1.
6. S n >= length []. В библиотеке есть лемма le_0_n, утверждающая, что $\forall n : \mathbb{N}, n + 1 \geq 0$, ее мы и применим.
7. nth_remove (S n) (a :: l) = a :: l с предположением индукции для n. Здесь применяем le_0_n к S n >= length (a :: l), затем применяем индукцию. Получим nth_remove n l = l, что при подстановке в доказываемое приводит к равенству.
8. S n >= length (a :: l) с предположением индукции для n. Применяя индуктивную гипотезу, получим гипотезу n >= length l. Применяя к обеим частям гипотезы H: nth_remove (S n) (a :: l) = a :: l операцию tl и упрощая, получим nth_remove n l = l. Затем используем подстановку и лемму le_n_S для завершения доказательства.

Как видно, рассуждения получились громоздкими и не всегда тривиальными. Более того, постоянно требуется помнить или искать леммы в стандартной библиотеке и применять тактики к гипотезам, указывая их имена.

Второе доказательство базируется на первом, но использует базовые автоматические тактики

```
Proof.
  induction n; destruct l; intros; simpl in *; intuition.
  - easy.
  - apply f_equal with (B := nat) (f:=@length A) in H.
     simpl in H; intuition.
  - assert (e: n >= length l) by omega; apply (IHn _) in e.
     rewrite e; auto.
```

```
  - inversion H as [H']; rewrite H'; apply IHn in H'. omega.
Qed.
```

Как видно, количество случаев для рассмотрения сократилось с 8 до 4, поскольку тривиальные случаи были решены самим Coq благодаря тактике `intuition`. Оставшиеся случаи соответствуют случаям 3, 4, 7 и 8 из первого доказательства. В первом случае Coq может сам найти противоречие в гипотезах, поэтому применяется тактика `easy`, одной из назначений которой является именно поиск очевидных противоречий. В втором случае длинные рассуждения с разбиениями были переложены на `intuition`, поскольку являются тривиальными индукциями. В остальных случаях манипуляции с арифметикой, требовавшие до этого поиска теорем из стандартной библиотеки, переложены на `omega`, которая уже была упомянута выше. В целом, мы перекладываем доказательства элементарных утверждений на вычислительное ядро Coq, концентрируясь на более общих шагах.

В качестве еще одного примера можно упомянуть использование тактик `autounfold` и `autorewrite` в файле `TreeZipper.v`, позволяющие избежать ручного переписывания условий леммы в терминах ранее введенных определений.

Coq предоставляет и более продвинутые средства автоматизации, помимо высокоуровневых тактик и команд управления процессом доказательства (к примеру, `Add Relation`, она встретится ниже), такие как язык тактик L_{tac} и альтернативный язык описания доказательств SSReflect, но в данной работе они не используются.

2.2.3. Доказательство корректности «Зиппер»

Модифицируя или обозревая элементы дерева с использованием структуры данных «Зиппер», мы неявно предполагаем, что модификация будет затрагивать лишь ту часть дерева, которая не содержится в контексте (поскольку это вытекает из определения «Зиппера»), и что некоторые свойства дерева, его «целостность», не будут на-

рушаться при выполнении стандартных операций. Следует убедиться в том, что наши ожидания действительно будут выполняться. С этой целью рассмотрим и проверим те фундаментальные свойства, которые мы неявно от этой структуры данных ожидаем. Все эти свойства взяты из работы Gerard Huet [2], либо позаимствованы у Окасаки [5]. Результаты содержатся в файле `TreeZipper.v`.

Инвариантность Очевидно, что **ZipperToTree (TreeToZipper T) = T**. Это достаточно простое и тривиальное свойство, доказываемое самой автоматикой Coq на основании нашей формализации.

Сохранение отношения «поддерево» Если Z представляет некоторое дерево T, то Z_T есть поддерево T. Для доказательства этого свойства надлежит в первую очередь описать, что есть «поддерево» в рамках нашей формализации. Будем считать, что отношение «T' есть поддерево T» (или $T' \triangleleft T$) удовлетворяет следующим аксиомам [5; 6]:

- для всякого дерева T выполняется $T \triangleleft T$;
- для всякого дерева T, массива деревьев l и элемента a выполняется $T \in l \implies T \triangleleft (\mathrm{T\_tr}\, a\, l)$;
- для любых трех деревьев $T1$, $T2$, $T3$ имеет место $(T1 \triangleleft T2) \wedge (T2 \triangleleft T3) \implies T1 \triangleleft T3$.

Подобное описание легко формализуется в Coq, более того, в целях автоматизации можно сообщить системе, что данное отношение является рефлексивным и транзитивным.

```
Add Relation (Tree) (@IsSubtreeOf)
  reflexivity proved by (@ISO_refl)
  transitivity proved by (@ISO_trans)
as ISO_relation.
```

В коде эта лемма описана как

```
Lemma ZipperSubtree: forall tr l,
  CorrectZipperToTreeConditions (tr, l) ->
  IsSubtreeOf tr (ZipperToTree (tr, l)).
```

Сохранение свойств дерева Если некоторое свойство верно для дерева, чьим представлением является Z, то оно будет верно и для Z_T.

Пусть у нас имеется дерево T и некоторый предикат P : **tree** $\mapsto$ **Bool** над деревом. Тогда будем говорить, что P применимо к T (или `PropertyOverTree` $P\ T$), если P верно для T и всех его поддеревьев. Подобное обобщение хорошо тем, что позволяет при необходимости определить сколь угодно сложную функцию проверки «правильности» дерева. Это определение «применимости» может быть формализовано следующими аксиомами:

- для всех P, для которых $P(\mathrm{T\_nil})$, выполняется `PropertyOverTree` $P\ \mathrm{T\_nil}$;
- для всех P, T, для которых верно $P(T)$ и $\forall T', T' \triangleleft T \implies P(T')$ выполняется `PropertyOverTree` $P\ T$.

Свойство, которое мы хотим доказать, можно переформулировать так: если для некоторого дерева T P применимо к T, то для (корректного) «Зиппера» Z, представляющего T, P применимо к Z_T. Эта лемма в коде записана как

```
Lemma ZipperPreserveProperty: forall P T,
  PropertyOverTree P T ->
  (forall Z, CorrectZipperToTreeConditions Z ->
    ZipperToTree Z = T -> PropertyOverTree P (fst Z)).
```

Она доказывается по индукции с последующим разбиением T на различные возможные варианты.

Корректность функции Modify Модифицируя данные в дереве с помощью `Modify`, мы неявно предполагаем, что

- применение функции $id_x : x \to x$ не поменяет состояния «Зиппера»;
- если $Z' = \texttt{Modify}\ Z\ f$, то $Z'_T = f Z_T$.

Оба этих свойства формулируются ниже и легко доказываются автоматикой Coq

```
Lemma ModifyIdentity: forall Z, Modify Z (fun t => t) = Z.
Lemma ModifyContext: forall Z f, f (fst Z) = fst (Modify Z f).
```

Корректность функций навигации Перемещая фокус по дереву с помощью «Зиппера», мы считаем, что подобные перемещения ничем не отличаются от навигации по обычному дереву, которое представляет структура данных «Зиппер».

Во-первых, нужно показать, что при перемещении фокуса внутри «Зиппера» дерево, которое он представляет, остается неизменным.

```
Lemma MoveDown_TreeInvariance: forall D Z,
  CorrectMoveDownConditions D Z ->
          ZipperToTree (MoveDown D Z) = ZipperToTree Z.
Lemma MoveTop_TreeInvariance: forall Z,
  ZipperToTree Z = ZipperToTree (MoveTop Z).
```

Во-вторых, когда мы спускаем фокус от корня к потомку внутри «Зиппера» Z и получаем новый «Зиппер» Z', мы предполагаем, что корень Z'_T будет потомком Z_T.

```
Lemma MoveDownCorrectness: forall D Z,
  CorrectMoveDownConditions D Z ->
  Some (fst (MoveDown D Z)) = RootSubtree D (fst Z).
```

В-третьих, если мы подняли фокус от потомка к корню в Z и получили Z', то тогда Z_T будет содержаться среди потомков Z'_T.

```
Lemma MoveTopCorrectness: forall a l Z,
  CorrectMoveTopConditions Z ->
  fst (MoveTop Z) = T_tr a l -> In (fst Z) l.
```

3. Тестирование с использованием QuickChick

Как уже было сказано, сформулированные в Coq определения могут быть проверены с помощью плагина QuickChick. В работе этот код содержится в файлах `QuickChickTest.v` и `QuickChickZipper.v`.

3.1. Генерация данных

Ограничения, которые мы накладываем на «Зиппер», не позволяют нам использовать стандартные механизмы QuickChick, автоматически выводящие генераторы для заданных типов. В связи с этим стоит детальнее рассмотреть то, как можно вручную описать генераторы для интересующих нас структур.

Самым важным (и сложным) является описание функции, возвращающей генератор случайного дерева высоты sz. Функция работает следующим образом: в случае $sz = 0$ возвращается пустой список (nil), иначе генерируется случайная величина типа A (тип данных, составляющих дерево), затем рекурсивно строится список деревьев высоты $sz - 1$ и из полученных данных составляется исходное дерево.

```
Fixpoint genTreeSized (sz: nat) (g: G A) : G (Tree) :=
  match sz with
  | 0 => returnGen T_nil
  | S sz' =>
    freq [ (1, returnGen T_nil) ;
      (sz, bindGen g (fun x =>
           bindGen (@listOf (Tree) (genTreeSized sz' g)) (fun l =>
         returnGen (T_tr x l)))) ]
  end.
```

Проблема в том, что, поскольку у нас имеется лишь два варианта выбора ($sz = 0$ или $sz > 0$), то QuickChick будет генерировать пустой список в примерно 50% случаев, что не будет удовлетворительным

результатом. В связи с этим используется стандартный комбинатор `freq` [7], позволяющий снизить вероятность появления пустого списка до $\frac{1}{sz+1}$.

Имея эту функцию, уже несложно описать генераторы для контекста и самой структуры данных «Зиппер», здесь приведем лишь код для генерации последней.

```
Definition genZipper (g: G nat) (gA : G A) : G (ZipperTree) :=
  bindGen (genTree g gA) (fun t =>
    bindGen (genContextList g (genContext g gA (genTree g gA)))
            (fun lC => returnGen (t, lC))).
```

Данный генератор принимает на вход генераторы натуральных чисел и данных типа A, затем, путем последовательного применения связки `bindGen`, комбинирует генератор для структуры данных «Зиппер» из ранее определенных генераторов `genTree` (генератор случайного дерева на базе `genTreeSized`), `genContextList` (генератор Z_C) и `genContext` (генератор элемента для Z_C).

3.2. Тестирование

Опять же, в силу фундаментальных ограничений QuickChick, нельзя напрямую проверить сформулированные ранее леммы на случайных данных. Требуется их переформулировать в терминах, доступных для проверки. В качестве примера рассмотрим этот процесс для одного несложного теста, причем мы принимаем, что $A = \mathbb{N}$.

Инвариантность Для проверки свойства **ZipperToTree (TreeToZipper T) = T** надлежит в первую очередь определить булево отношение равенства деревьев (`eq_tree`), поскольку QuickChick не способен проверять утверждения в рамках логики CoC. Будем считать, что деревья равны, если равны их представления в виде списков, полученных после прямого обхода дерева (этот обход выполняет функция `treeTraverse T`). Тогда мы можем использовать

стандартную булеву функцию `forallb`, чтобы проверить, что все элементы обеих списков равны в булевом смысле. Более того, длины этих списков тоже должны быть равны.

```
Definition eq_tree (T T': Tree) : bool :=
  let lT := (treeTraverse T) in let lT' := (treeTraverse T') in
  (beq_nat (length lT) (length lT')) &&
  forallb (fun p => beq_nat (fst p) (snd p)) (combine lT lT').
```

Осталось лишь сформулировать интересующее нас свойство и проверить его, используя введенные ранее генераторы.

```
Definition qc_ziptotree_id (T: Tree) :=
  eq_tree (ZipperToTree (TreeToZipper T)) T.

QuickChick (forAll
        (genTree (choose (0, 5)) (choose (0, 5)))
                qc_ziptotree_id).
```

Последняя операция выведет в консоль сообщение примерно следующего вида:

```
QuickChecking (forAll (genTree (choose (0, 5)) (choose (0, 5))) qc_ziptotree_id)
+++ Passed 10000 tests (0 discards)
```

Этот результат означает, что тесты были пройдены успешно и свойство инвариантности верно для большого количества данных, а значит, можно попытаться найти его строгое доказательство (что и было сделано ранее).

4. Генерация верифицированного Haskell кода

Как только было получено убедительное доказательство корректности интересующих нас алгоритмов, они должны быть переписаны вместе с определениями сопутствующих структур данных с внутреннего языка Coq на какой-либо используемый в прикладном программировании язык. Возникает, однако, проблема того, что

вручную переписанные с одного языка на другой формализации будут семантически отличаться друг от друга (в силу ошибок при переписывании, например), и тогда вся построенная ранее доказательная база к фактической реализации применима не будет.

Для решения этой существенной проблемы, среда Coq предлагает возможности для автоматической генерации верифицированного кода на таких промышленных функциональных языках программирования, как OCaml и Haskell. Эти средства были разработаны и подробно описаны Pierre Letouzey [8], все дальнейшие сведения взяты из его работы.

В рамках данной работы производится генерация кода на языке Haskell, описывающего структуру данных «Зиппер» и функции для работы с ней, на базе уже построенной формализации этой структуры на языке Coq (файл `Extraction.v`).

4.1. Теоретические основания

Наивным подходом к генерации кода на базе спецификации в Coq или иной другой среде является введение такой функции извлечения $\mathcal{E}$ (функция, переводящая объекты CoC в конструкции некоторого языка), которая перед началом своей работы полностью избавляется от логической части CoC (например, опускает предикаты), а затем переводит код из одного языка в другой в соответствии с несложными правилами, ведя себя как отображение.

Приведем пример из [8], демонстрирующий некорректность подобного подхода. К примеру, пусть мы имеем терм (функцию) типа $f\ x\ P = \forall x : A, (P\ x) \implies B$, где A, B — типы, P — некоторый логический предикат. С точки зрения CoC этот терм-функция принимает на вход некоторый аргумент x и доказательство p_x того, что $P\ x$. Если имеются еще термы $t : A$ и $p_t : (P\ t)$, то в рамках CoC корректными будут являться термы $(f\ t)$ и $(f\ t\ p_t)$. Первый из них вычисляться

напрямую не будет, поскольку не хватает аргумента p_x, второй же вычислится и вернет результат типа B.

Теперь применим «наивную» функцию извлечения $\mathcal{E}$ к двум полученным термам. Тогда $(f\ t)$ и $(f\ t\ p_t)$ породят один и тот же код $(\mathcal{E}(f)\,\mathcal{E}(t))$, поскольку логическая часть (то есть p_t) была удалена. Поведение извлеченного кода совпадет с поведением $(f\ t\ p_t)$, что не является ожидаемым поведением в случае с извлечением кода из первого терма и может привести к фатальным ошибкам.

CoC$_\square$ Решение вышеупомянутой проблемы заключается в введении нового бестипового языка CoC$_\square$, на базе CoC путем введения специальной нередуцируемой константы $\square$. Функция $\mathcal{E}$ переводит термы CoC в термы CoC$_\square$, затем к полученному результату применяются редукции и затем функция трансформации $[\![\cdot]\!]$. Стоит заметить, что полученный результат не будет являться семантически эквивалентным исходному Coq коду в силу отсутствия типизации в CoC$_\square$. О типах будет сказано ниже.

Функция извлечения $\mathcal{E}$ определяется рекурсивно по термам, типизируемым в рамках данного контекста Γ (здесь приведены некоторые основные правила):

1. $(\square)$ если t — схема типизации (то есть терм вида $\lambda X_1, \lambda X_2, \cdots \lambda X_n, s$, где s — тип) или признает Непонятно, что значит «признает тип» тип **Prop** в контексте Γ, то $\mathcal{E}(t) = \square$;
2. $\mathcal{E}_\Gamma(a) = a$, если a — переменная, константа или конструктор;
3. $\mathcal{E}_\Gamma(\lambda x : T, t) = \lambda x : \square, \mathcal{E}_{\Gamma;(x:T)}(t)$;
4. $\mathcal{E}_\Gamma(u\ v) = (\mathcal{E}_\Gamma(u)\ \mathcal{E}_\Gamma(v))$;
5. $\cdots$

В дальнейшем в CoC$_\square$ в качестве основной выполняется $\square$-редукция, заданная правилом $(\square\ u) \to_\square \square$ Тут точно нет опечатки?.

Извлечение типов Как уже было сказано, типы игнорируются функцией извлечения $\mathcal{E}$. Сделано это было потому, что строгая система типизации в языках OCaml и Haskell отличается от подобной в Coq, что вынуждает строить отдельную функцию извлечения типов $\hat{\mathcal{E}}$. Ниже приведем пример работы этой функции.

Пример работы функции $\hat{\mathcal{E}}$ Пусть имеется код на языке Coq, который требуется извлечь в код на языке Haskell:

```
Definition P (b: bool) : Set := if b then nat else bool.
Definition Sch2 (b: bool) : Set := P b.
Definition Sch3 : (bool -> Set) -> Set :=
        fun (X: bool -> Set) => X true -> X false.
```

Рассмотрим первые две строки: подобная зависимость типов от типов невозможна в Haskell, при таких случаях алгоритм $\hat{\mathcal{E}}$ генерирует специальный «пустой» тип $\mathbb{T}$, который по ходу выполнения алгоритма заменяется на какой-то определенный. Промежуточный результат работы алгоритма будет следующим:

```
data b p = T
data b sch2 = T p
```

что будет затем упрощено до

```
data p = T
data sch2 = p
```

При разборе определения `Sch3` возникает два возможных варианта разрешения проблемы типовой зависимости, порожденной недопустимым в Haskell типом `bool -> Set`:

1. В целях осторожности можно считать, что `X true` и `X false` есть неизвестные типы. Действительно, если кто-то в качестве `X` передаст `P`, определенную выше, то получим

   ```
   Sch P = nat -> bool
   ```

   Это ведет нас к первому возможному пути извлечения:

```
data x sch3 = T -> T
```

2. Если применить результат первого варианта к `(Sch3 (fun _ => nat))`, то вместо ожидаемого `nat` $\to$ `nat` получим $\mathbb{T} \to \mathbb{T}$. На практике подобное применение является практически единственно встречающимся, поэтому, если принять

   ```
   data x sch3 = x -> x
   ```

   то поведение извлечения будет ожидаемым в этом случае. Более того, и в случае `Sch P` поведение будет ожидаемым и извлечение вернет данные типа $\mathbb{T} \to \mathbb{T}$, поскольку мы уже до этого приняли $\mathtt{P} = \mathbb{T}$. Именно этот вариант и выбирает функция $\hat{\mathcal{E}}$.

Дальнейшее развитие теории дано в [8].

4.2. Построение генератора кода

В принципе, для генерации кода можно было бы ограничиться лишь следующими строками:

```
Require Extraction.
Extraction Language Haskell.
Extraction "Tree.hs" TreeZipper.
```

К сожалению, в этом случае результирующий файл будет содержать большое количество неэффективных с практической точки зрения определений для элементарных операций, более того, некоторые типы данных определены не будут (поскольку TreeZipper был ранее выведен вне зависимости от какого-то конкретного типа). С целью получить более практичный код, необходимо обратить внимание на возможности Coq по управлению процессом генерации кода.

Во-первых, стандартная библиотека предлагает определенные оптимизации для языка Haskell. В качестве примера в работе генерируется код для деревьев над целыми числами ($\mathbb{Z}$), поэтому в ней были

подключены модули `ExtrHaskellZInt` и `ExtrHaskellBasic`, которые заменяют некоторые элементарные определения из Coq на аналогичные определения из стандартной библиотеки Haskell (например, Z выражается через Int Точно Int, а не Integer? в коде на Haskell, а не строится заново по индукции).

Во-вторых, для удобства работы с полученным кодом нужно определить имена для структур данных и функций, которые мы намереваемся использовать, поскольку имена, которые были им даны в коде на Coq, не всегда удобны. К примеру, строки

```
Extract Inductive Tree => "tree" [ "t_nil" "t_tree" ].
Extract Inductive Context => "cntx" [ "move" ].
```

запишут в Haskell код те определения для конструкторов Tree и Context, которые мы хотим (или уже используем) в реальном коде.

В-третьих, такие простые функции, как `fst p` или `snd p`, можно сделать встраиваемыми, то есть всегда вставлять код, реализующий эти операции, на их место. В дальнейшем этот код будет однозначно оптимизирован компилятором Haskell.

```
Extraction Inline fst snd.
```

Заключение

В данной работе было продемонстрировано использование инструмента интерактивной верификации в рамках задачи построения верифицированных алгоритмов и структур данных на базе некоторых заранее заданных спецификаций, а также их тестирования и генерации программного кода. В качестве модели была взята структура данных «Зиппер» для обобщенного дерева и операции над нею.

В качестве инструмента автоматической верификации использовалась среда Coq, позволяющая формулировать и проверять доказательства в рамках исчисления индуктивных конструкций (CoC).

Для проверки промежуточных результатов использовался плагин QuickChick, позволяющий проверять утверждения в рамках булевой логики (как подмножества CoC) на большом множестве случайных данных.

В качестве языка программирования, верифицированный код на котором генерировался как заключительная стадия работы проекта, использовался функциональный язык программирования Haskell.

В рамках работы были формализованы основные операции и доказаны такие важные свойства структуры данных «Зиппер», как ее неизменность при работе немодифицирующих операций и сохранение взаимно-однозначного соответствия между структурой данных и деревом, которое она представляет, после любых допустимых операций, а также построен генератор кода на языке Haskell, отражающего исходные спецификации.

Список литературы

1. coq-zipper. — URL: https://github.com/holmuk/coq-zipper.
2. *Huet G.* The Zipper // J. Funct. Program. — New York, NY, USA, 1997. — Сент. — Т. 7, № 5. — С. 549—554. — ISSN 0956-7968. — DOI: 10.1017/S0956796897002864. — URL: http://dx.doi.org/10.1017/S0956796897002864.
3. *McBride C.* The Derivative of a Regular Type is its Type of One-Hole Contexts (Extended Abstract). — 2009. — Апр.
4. Coq Reference Manual. — URL: https://coq.inria.fr/distrib/current/refman/ (дата обр. 04.04.2019).
5. *Okasaki C.* Purely Functional Data Structures. — 1996. — URL: https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf (дата обр. 12.03.2019).
6. *Cormen T.* Introduction to Algorithms. — The MIT Press, 2002.
7. QuickChick Reference Manual. — URL: https://github.com/QuickChick/QuickChick/blob/8.9/QuickChickInterface.v (дата обр. 12.03.2019).
8. *Letouzey P.* Programmation fonctionnelle certifée: L'extraction de programmes dans l'assistant Coq : дис. ... канд. / Letouzey Pierre. — L'Université de Paris-Sud, 2004.