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

В процессе разработки программного обеспечения неизбежно возникает вопрос, удовлетворяет ли написанный программистом код некоторым заранее заданным спецификациям, согласно которым должны были быть разработаны алгоритмы? Этот процесс проверки корректности кода, или, иначе, верификации, может быть построен как с использованием так называемых unit-тестов, проверяющих код 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 куда 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`, 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.

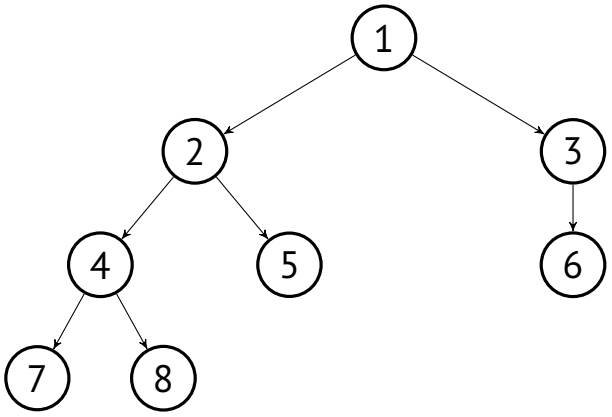
В данной работе ставится цель исследовать одну структуру данных с позиций подобного стиля верификации. Более точно, исследуется вариант структуры данных “Зиппер” для обобщенных деревьев, описывается сама структура данных и набор функций для работы с ней, формулируются желаемые свойства, которым должна подобная структура данных удовлетворять, составляются тесты и доказывается корректность подобных 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: `https://github.com/holmuk/coq-zipper`

1. Структура данных “Зиппер”

При реализации алгоритмов на чисто функциональных языках программирования часто возникают вопросы производительности, связанные с невозможностью модификации структур данных в таких языках. Эта проблема, может, однако, быть решена разработкой таких структур данных, которые не требуют полного воссоздания исходной структуры при модификации некоторой ее части, как это обычно бывает при работе с такими языками.

Можно легко продемонстрировать проблему на примере несложной структуры данных, как, 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 данной 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 опера-

цию с корнем дерева. Таким образом, модификация, произведенная в узле, требует в дальнейшем копирования родительского узла, 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, однако, 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 [1].

Строго говоря, “Зиппер” — не есть структура данных, но конструкция, позволяющая на базе некоторой сложной структуры данных построить новую структуру, позволяющую при работе с данными обращаться лишь к той части исходных данных, которые требуют модификации, оставляя оставшуюся часть нетронутой [1]. В данной работе рассматривается “Зиппер” для обобщенных деревьев.

1.1. “Зиппер” для обобщенного дерева

Любое дерево T можно представить как пару (называемой “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 порождаемый из типа исходного дерева.

Используя инструментарий из [2], если тип нашего дерева будет описан как (здесь и далее используется нотация из [2])

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

то тип контекста может быть легко выведен

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

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 $Z = (T, nil)$, $O(1)$

- **Modify Z f** для некоторого $Z = (Z_T, Z_C)$ возвращает $Z' = (fZ_T, Z_C)$. $O(m)$, где m – время выполнения функции f. Очевидно, что f определена над деревьями.

Как видно, 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 с использованием Coq

Важно корректно описать изначальные структуры на внутреннем языке Coq, чтобы в дальнейшем избежать проблем с формальным доказательством интересующих нас свойств. Функциональная реализация алгоритмов и структур данных в этой работе базируется в основном на идеях Окасаки [3].

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

Аналогичные проверяющие предикаты были введены и для всех остальных операций.

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: удаление n-ого элемента из списка l (`nth_remove n l`) и добавление элемента x в список l перед элементом с индексом n (нумерация идет с нуля) (`nth_insert n l x`). 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 неизменных 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, однако, 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 $n \leq$ `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 Coqкак

```
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” доказательствах не будут особо рассматриваться элементарные случаи.

Итак, по предположению индукции, лемма верна для l и n таких, что $n \leq \texttt{length}\ l$, требуется доказать это для $n' = n + 1$ и $l' = [a']^\frown l$. Несложно отсюда вывести, что $n \leq \texttt{length}\ l$ и применить к этому факту предположение индукции, затем с использованием полученного представления переписать и упростить лемму, и увидеть, что желаемое свойство действительно выполняется. Автоматика Coqпозволяет не проводить достаточно нудные 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 Coqэто 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. Доказательство корректности “Зиппер”

Модифицируя или обозревая элементы дерева с использованием структуры данных “Зиппер”, мы неявно предполагаем, что моди-

фикация 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 нарушаться при выполнении стандартных операций. Было бы неплохо 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 [1], либо позаимствованы у Окасаки [3]. Результаты содержатся в файле `TreeZipper.v`.

Инвариантность Очевидно, что **ZipperToTree (TreeToZipper T) = T**. Это достаточно простое и тривиальное свойство, доказываемое самой автоматикой Coqна основании наших формализаций

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 [3] [4]:

- 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 всяких деревьев $T1$, $T2$, $T3$ выполняется $(T1 \triangleleft T2) \wedge (T2 \triangleleft T3) \implies T1 \triangleleft T3$

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 полученых 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, так как у нас имеется лишь два варианта выбора ($sz = 0$ или $sz > 0$), то QuickChick будет генерировать пустой список в примерно 50% случаев, что не есть желаемый результат.

В связи с этим используется стандартный комбинатор `freq` [5], позволяющий снизить вероятность появления nil 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 доказательства свойства **ZipperToTree (TreeToZipper T) = T** надлежит в первую очередь определить булево отношение равенства деревьев (`eq_tree`), поскольку `QuickChick` не способен проверять утверждения в рамках 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 буле-

вую 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)
```

что значит, что тесты были пройдены успешно и свойство инвариантости 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-то используемый в прикладном программировании язык, такой как C# или Erlang. 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, очевидно, существенной проблемы, среда Coq предлагает возможности для автоматической генерации верифицированного кода на некоторых промышленных функциональных языках программирования, таких как OCaml и Haskell. Эти средства были разработаны и подробно описаны Pierre Letouzey [6], все дальнейшие сведения взяты из его работы.

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

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

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

но тогда 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 (напри-

мер, Z выражается через Int в Haskell коде, а не строится заново по индукции),

Во-вторых, для удобства работы с полученным кодом нужно определить имена для структур данных и фунцкий, которые мы намереваемся использовать, поскольку имена, которые были им даны в Coq коде, 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

Было продемонстрировано, что в наше время задача создания верифицированного ПО, в принципе, посильна не только научным организациям уровня INRIA, но и рядовым разработчикам. Основной проблемой является не создание и оправдание некоторой научной базы, как это было в начале 1970-х, но лишь формулировка и доказательство теорем, которые отражают интересующие нас свойства программ, в рамках существующих и хорошо развитых систем верификации теорем, таких как `Coq` либо `Isabelle`.

Но это все еще остается сложной задачей, требующей чего-то там.

Список литературы

1. *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.
2. *McBride C.* The Derivative of a Regular Type is its Type of One-Hole Contexts (Extended Abstract). — 2009. — Апр.
3. *Okasaki C.* Purely Functional Data Structures. — 1996. — URL: https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf (дата обр. 12.07.2018).
4. *Cormen T.* Introduction to Algorithms. — The MIT Press, 2002.
5. *Team Q.* QuickChick Reference Manual. — URL: https://github.com/QuickChick/QuickChick/blob/8.9/QuickChickInterface.v (дата обр. 12.03.2019).
6. *Letouzey P.* L'extraction de programmes dans l'assistant Coq : дис. ... канд. / Letouzey Pierre. — L'Université de Paris-Sud, 2004.