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.

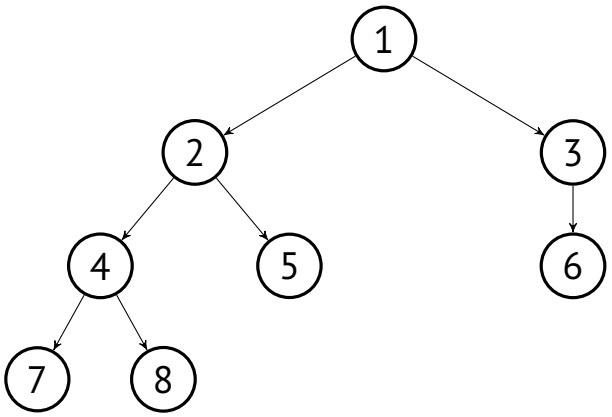
Одной из самых популярных систем автоматической верификации теорем является система `Coq` , разработанная в INRIA. В этой среде для тестирования написанных определений и теорем может использоваться плагин `QuickChick` , позволяющий сгенерировать набор случайных входных данных и проверить, является ли 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. 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.

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 заниала 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 Зиппером 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 исходных 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 Зиппера, тип, описывающий Зиппер, 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 модуль, который можно затем использовать повторно

```
Module TreeZipper (Import T: Typ).
Definition A := T.t.
...
Definition ZipperTree := prod Tree (list Context).
...
```

Как уже было сказано, введенные над Зиппером операции перед своим применением требуют проверки допустимости их применения. К примеру, операция **MoveTop Z** не может быть применена в том случае, если Z_C пуст. Это легко выразить в Coq :

```
Definition CorrectMoveTopConditions (Z: ZipperTree) :=
  (snd Z) <> [].
```

Аналогичные проверяющие предикаты были введены и для всех остальных операций.

2.1. Доказательство корректности Зиппер

Модифицируя или обозревая элементы дерева с использованием Зиппер, мы неявно предполагаем, что модификация будет затрагивать лишь ту часть дерева, которая не содержится в контексте (поскольку это вытекает из определения Зиппер), 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 вкратце те фундаментальные свойства, которые мы неявно от этой структуры данных ожидаем, и которые затем в дальнейшем в этой работе доказываются. Все эти свойства взяты из работы Gerard Huet [1], либо позаимствованы у Окасаки [3].

Инвариантность Очевидно, что **ZipperToTree (TreeToZipper T) = T**. Это достаточно простое и тривиальное свойство, доказываемое самой автоматикой Coq на 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 $P(\text{T\_nil})$, выполняется `PropertyOverTree` P 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 $Z' =$ `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 Зиппер.

3. Тестирование с использованием QuickChick

Как уже было сказано, есть возможность проводить тесты алгоритмов в самом `Coq` на случайных данных, используя `QuickChick` .

3.1. Общие замечания

Ограничения, которые мы накладываем на Зиппер (например, контекст не может быть вида $cntn, A, []$, 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, 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 $\frac{1}{sz+1}$.

3.2. Описание тестов

Опять же, в силу фундаментальных ограничений QuickChick , нельзя напрямую проверить сформулированные в Coq леммы на случайных данных. Требуется их переформулировать в терминах, доступных для проверки.

Описание пары тестов

4. Генерация верифицированного OCaml кода

Заключение

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).