Интерактивное доказательство теорем с помощью Coq

Деркач Илья

23 декабря 2019 г.

1 Введение

Coq — это интерактивное программное средство доказательства теорем, включающее собственный язык функционального программирования с развитой системой типов. Данное средство позволяет формулировать теоремы, их доказательства и проверять рассуждения на корректность. Проверка типов в Coq гарантирует логическую корректность выполненных шагов в любой момент рассуждения. Вдобавок к этому среда предоставляет высокоуровневые средства для разработки доказательств, включая большую библиотеку общих определений и лемм, мощные тактики для полуавтоматического доказательства некоторых классов утверждений.

Моей задачей было разобраться с основами Coq, а также сформулировать и доказать корректность сортировки списка вставками. Ссылка на репозиторий, содержащий код с формулировками и доказательствами, приведена в последнем разделе. Настоящий отчет состоит из описания формулировок, доказательств, и некоторых тактик, использованных в процессе решения задачи.

2 Мотивация к использованию Coq

Создание надежного программного обеспечения весьма непросто. Масштаб и сложность современных систем, количество вовлеченных людей и диапазон требований, предъявляемых к ним, чрезвычайно затрудняют создание корректно работающего продукта. В то же время, растущее влияние обработки информации во всех сферах общественной жизни значительно увеличивает стоимость ошибок и ненадежности. Вероятно, наиболее сильную гарантию надежности программного обеспечения дает математическое доказательство его соответствия заданной спецификации. Несмотря на высокую стоимость такого подхода, применение Coq и ряда аналогичных систем во многих случаях делает эту задачу выполнимой на практике.

Также с помощью Coq было проверены доказательства ряда важных и трудных математических теорем. Например, в 2005 году доказательство проблемы четырех красок было верифицированно Жоржем Гонтье с использованием Coq. Сложность заключалась в том, что оригинальное рассуждение сводило задачу к перебору большого числа случаев, который выполнялся некоторой программой. Такой подход вызвал недоверие в математическом сообществе. Работа Гонтье свела вопрос о корректности алгоритма специального назначения к корректности ядра Coq, проверка которой практически возможна и достаточна для получения и других результатов того же рода.

3 Результаты

3.1 Формулировка задачи

Перед тем, как решать поставленную задачу, нужно разобраться, с какими объектами нам придется в дальнейшем работать. Сам по себе Coq крайне скуден на определение типов, так что математику приходится определять многое самостоятельно. Правда, как было сказано выше, система предоставляет множество библиотек, в которых уже определены некоторые типы и операции над ними. Мы будем пользоваться библиотеками `Nat` и `List`, которые описывают тип натуральных чисел и списки (в смысле функционального программирования) соответственно. Также нам придется самостоятельно определить ряд функций, описывающих процесс сортировки вставками. Именно их корректность будет доказана в дальнейшем.

После определения функции сортировки, нам потребуется проверить ее корректность. Однако для этого необходимо сформулировать условие верификации, а именно критерий отсортированности списка по неубыванию. Я

использовал для своего доказательства следующее утверждение.
Утверждение 1. *Список A' является сортировкой списка A тогда и только тогда, когда количество вхождений любого натурального числа n в A равно количеству вхождений n в A', и для любых двух соседних элементов A' верно, что стоящий из них левее небольше стоящего правее.*

Однако доказать сразу это утверждение не получится. Для этого придется сформулировать некоторые леммы, которые помогут в рассуждении. Для удобства читателя часть с доказательствами разбита на 4 части, каждая часть отделена соответсвующим комментарием. Первая часть включает в себя леммы о натуральных числах. Вторая - свойства порядка и сравнения натуральных чисел. В третьей части формулируются простейшие свойства списка. В последнем же разделе читатель может найти формулировку критерия в трех теоремах и их доказательство.

3.2 Определение объектов

Как выше было сказано, натуральные числа и список определены в библиотеке, однако приведем их определение здесь, так как это понадобится для понимания некоторых доказательств.

Так, тип `bool` задается в Gallina (функциональный язык, использующийся в Coq) следующим образом:

```
Inductive bool : Type :=
  | true
  | false.
```

Может показаться, что подобные определения могут задать только типы, принимающие конечное количество значений. Однако это не так, в Coq можно использовать индуктивные определения. Тогда натуральные числа можно задать как:

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

Грубо говоря, выше написано, что натуральное число - это ноль или иное натуральное число с прибавленной к нему единицей. Аналогично можно задать список, используя стандартное определение из курса матлогики и синтаксис Gallina.

Чтобы задать сортировку вставками мы определяем в начале функцию `insert`, которая соответствует одной итерации алгоритма сортировки.

```
Fixpoint insert (n : nat) (sorted : list nat) : list nat :=
  match sorted with
  | nil => n :: nil
  | m :: t => if n <? m then n :: sorted
                         else m :: (insert n t)
  end.
```

Прокомментируем определение выше. Функция `insert` также задается индуктивно. Она принимает на вход натуральное число `n` и список натуральных чисел `sorted`. Возвращает же она список натуральных чисел. После этого дано индуктивное определение самой функции, в зависимости от того, является ли список `nil` (пустым) или вида `m::t` (непустым, `m` - натуральное число, `t` - список натуральных чисел). Наблюдательный читатель может заметить, что функция `insert` перебирает все возможные значения списка типа `list nat`, описанные в индуктивном определении и сопоставляет им новый отсортированный список, согласно алгоритму сортировок вставками.

Через функцию `insert` задается функция `insert_sort`, которая сопоставляет списку натуральных чисел - отсортированый список. Синтаксис этой функции аналогичен разобранному выше. Забегая вперед, отметим, что корректность именно этой функции, заданной "на интуитивном уровне" придется строго доказывать (под корректностью имеется в виду, что выдаваемый список чисел функцией `insert_sort` будет удовлетворять *Утверждению 1*).

Кроме этого придется задать функцию `count` и `is_sorted`, с помощью которых мы сможем проверять *Утверждение 1.* Их индуктивное определение во многом схоже с раннее изложенными и совпадает с математическим индуктивным определением каждой из функций, поэтому останавливаться на них мы не будем.

3.3 Доказательство корректности сортировки

Критерий корректности сортировки списка весьма емок по количеству вложенных утверждений, поэтому для удобства доказательства мы разбили его на три следующие теоремы. Здесь и далее в скобках указано название утверждения в коде.

Теорема 1. ***(invariance_of_occurrences)*** $\forall l \in nat\ list\ , \forall n \in nat \rightarrow count\ n\ (insert\ n\ l) = count\ n\ l + 1.$

Теорема 2. ***(independence_of_occurrences)*** $\forall l \in nat\ , \forall n, m \in nat, n \neq m \rightarrow count\ n\ (insert\ m\ l) = count\ n\ l.$

Теорема 3. ***(sort_is_sort)*** $\forall l \in nat\ list \rightarrow is\_sorted\ (insert\_sort\ l) = true.$

Теоремы 1 и 2 в совокупности показывают, что количество элементов n на шаге алгоритма меняется на один тогда и только тогда, когда n был добавлен в список, в обратном случае 0. Отсюда количество элементов равных n в исходном списке и в отсортированном в конце равны.

Теорема 3 показывает, что любые два соседних числа отсортированного списка ориентированы в нужном порядке. Можно заметить, что *Теорема 3*, в отличие от *Теоремы 1* и *Теоремы 2* говорит о сортировке в целом, а не об отдельном ее шаге. Такой подход неудобен для доказательства, поэтому лучше сформулировать похожее утверждение, но для "промежуточного" шага. *Теорема 3* легко доказывается по индукции с помощью *Леммы 1*.

Лемма 1. ***(sorting_preservation)*** $\forall l \in nat\ list\ \forall n \in nat, is\_sorted\ l = true : is\_sorted\ (insert\ n\ l) = true.$

3.4 Доказательство вспомогательных лемм и описание некоторых тактик

Сформулированные утверждения в предыдущем пункте достаточно сложны и опираются на гораздо более простые факты и свойства натуральных чисел и списков. На примере этих утверждений хотелось бы продемонстрировать используемые тактики.

3.4.1 Лемма eq_ref

В процессе доказательства поребовалась следующая лемма:

Лемма 2. ***(eq_ref)*** $\forall n \in N \rightarrow n =?\ n = true.$

В начале может показаться, что эта теорема не имеет смысла, так как обе части равенства совпадают, а значит утверждение истинно. Однако это не так, потому что функция `eqb` (синонимом которой является `=?`) - индуктивно построена относительно натурального числа слева и справа от знака равенства. Она не сравнивает значения натуральных чисел (в привычном смысле натурального числа), поэтому приходится делать индукцию по построению. Ниже доказательство этой теоремы.

```
Lemma eq_ref :
  forall n : nat, n =? n = true.
Proof.
  induction n.
  + reflexivity.
  + simpl. rewrite -> IHn. reflexivity.
Qed.
```

В доказательстве используется тактика индукции по `n`. В первом случае рассматривается, когда `n = O` (первый случай из индуктивного определения натурального числа). В данном контексте теорема очевидна, потому что достаточно проверить, что `0 =? 0 = true`. Это верно в силу определения `=?`. Тактика `reflexivity` упрощает левую часть и завершает доказательство, сравнивая две одинаковые части равенства `true = true`.

Во втором же случае, когда `n = S n'`, тактика `simpl` упрощает равенство `S n' =? S n'` в `n' =? n'`. Тактика `rewrite` использует преположение индукции и переписывает утверждение `n' =? n' = true` в `true = true`. Применение `reflexivity` заканчивает доказательство.

Хотелось бы отметить, что возможная путаница, описанная в начале, могла быть связана с тем, что можно перепутать пропозициональный `=` и булевозначный предикат `=?`. Хоть они и эквивалентны на области определения натуральных чисел, но все равно являются объектами сущетсвенно различных типов с разными инструментами для работы с ними.

3.4.2 Лемма sublist_of_sorted_is_sorted

Если предыдущая лемма была связана со свойством натуральных чисел (рефлексивность сравнения), то следующая лемма связана со свойством отсортированного списка.

Лемма 3. ***(sublist_of_sorted_is_sorted)*** *Для любого отсортированного списка верно, что список, полученный из исходного путем удаления первого элемента, также отсортирован.*

Формальное утверждение и доказательство представлено ниже в синтаксисе Coq.

```
Lemma sublist_of_sorted_is_sorted :
  forall l n, is_sorted (n :: l) = true -> is_sorted l = true.
Proof.
  intros l n L.
  destruct (is_sorted l) as [| t h] eqn:E1.
  - reflexivity.
  - apply push_in_nonsorted with(m:=n) in E1.
    rewrite E1 in L. discriminate.
Qed.
```

Эта теорема возникает в доказательстве по индукции Теоремы 3.

В доказательстве *Леммы 3* встречаются новые тактики. Тактика `intros` перемещает переменные и гипотезы из цели в контекст. В данном случае имеется две переменные `l`, `n` и одна гипотеза `L : is_sorted (n :: l) = true`. Тактика `destruct` помогает «разложить» объект согласно индуктивному определению его типа. В данном случае доказательство разбивается на два случая: `is_sorted (l) = true` и `is_sorted (l) = false`. В первой части остается закончить доказательство, а во второй использовать лемму `push_in_nonsorted` в предположении подпункта с помощью тактики `apply`. Далее используем полученное в гипотезе L и получаем противоречие в предпосылке. Доказательство в этом случае завершаем с помощью тактики `discriminate`.

4 Ссылки

[1] Git repository: https://github.com/ilyaderkatch/coq_project.git