Бонусное домашнее задание
Математический анализ y2022, сем. 1, осень

December 31, 2022

Содержание

1 Введение

В данном домашнем задании студентам предлагается реализовать решения задач в интерактивном программном средстве доказательства теорем Coq. Домашнее задание разделено на три секции, каждая секция оценивается в

1 балл и заключается в решении определенных задач. Тем самым, на кону **3** бонусных балла.

Coq – это средство доказательства теорем, написанное на языке программирования OCaml. Данный инструмент представляет из себя среду, в которой задаются определения, аксиомы, конструкции, и доказываются теоремы. Широко используется в областях формализации математики и формальной верификации.

Бонусное домашнее задание затрагивает лишь экстремально мизерную часть всего функционала Coq'а.

1.1 Предупреждение

За любые шутки или намёки на шутки (эмоджи, эмотиконы, картинки, видео, текст, и так далее), связанные с названием инструмента Coq, которые, как показывает практика и уже состоявшееся взаимодействие со студентами, возникают при первой же возможности, **данное домашнее задание студента *не допускается* к проверке и все уже существующие бонусные баллы *аннулируются*.**

1.2 Где можно начать взаимодействие с Coq'ом?

Официальный сайт: `https://coq.inria.fr/`

Для решения задач вы можете кликнуть на "Try Coq in your browser" в параграфе "Running Coq". Откроется страница с уже имеющимся примером доказательства теоремы того, что функция, разворачивающая односвязный список, является инволюцией.

Кликните на синий значок листа бумаги с карандашом, расположенный слева от power symbol, чтобы открыть чистую страницу с запущенной интерактивной средой.

2 Задача 1

2.1 Базовейшее задание определений

Определения, аксиомы, теоремы, и так далее задаются в Coq последовательно, сверху вниз. В интерактивной среде вебсайта выше существует три главных сочетания клавиш для работы с ними в интерактивном режиме:

- Alt+Down: Обработать (т.е. проверить на корректность; например, с точки зрения правильного сопоставления типов, отсутствия синтаксических ошибок, и так далее) следующее выражение (т.е. определение, аксиома, теорема, и так далее).

- Alt+Up: Вернуться на выражении выше. Текущее обработанное выражение вернется в необработанное состояние.

- Alt+Enter: Обработать все выражения сверху вниз до позиции курсора. Уже обработанные выражения не будут обработаны снова.

2.1.1 Пример 1

Наипростейшие определения в Coq задаются следующим образом:

```
Definition x : nat := 0.
```

Это можно интерпретировать следующим образом: “Определим выражение `x`, имеющее тип `nat`, значением `0`”. В конце любых заданий (определений, аксиом, теорем, и так далее) необходимо ставить точку.

2.1.2 Пример 2

```
Definition const (A B : Type) : Type := A -> B -> A.
```

`const` принимает два типа, `A` и `B`, и возвращает тип, представляющий из себя функцию `A -> B -> A`, что читается следующим образом: “функция, принимающая аргумент типа `A` и аргумент типа `B` и возвращающая аргумент типа `A`”.

2.1.3 Пример 3

```
Definition refl {A : Type} (R : A -> A -> Prop) : Prop :=
  forall x : A, R x x.
```

`refl`

- принимает *неявный* тип `A` (иными словами, его можно не передавать явно и Coq попытается самостоятельно определить его исходя из типа `R`) (неявные аргументы передаются в фигурных скобках), затем
- принимает *явный* аргумент `R`, имеющий тип `A -> A -> Prop` (это интерпретируется как функция, принимающая два значения типа `A`, и возвращающая утверждение) (явные аргументы передаются в круглых скобках), и
- возвращает утверждение.

Тело `refl`'а – это утверждение о том, что `R` обладает свойством рефлексивности, т.е. для любого элемента `x` типа `A`, `R x x` выполняется.

`Prop` можно воспринимать как тип утверждений. Именно их можно либо доказывать, либо опровергать, в Coq.

Прошу также обратить внимание на то, как передаются аргументы `R`: без скобок и запятых. Это называется *аппликативный стиль*.

2.1.4 Пример 4

```
Theorem simple_theorem : 1 + 1 = 2.
Proof.
  simpl.
  reflexivity.
Qed.
```

Теорема о том, что $1+1 = 2$. Любые теоремы в Coq доказываются внутри тела `Proof. [...] Qed.`, где вместо `[...]` встаёт ваше доказательство. В данном выражении у значений `1` и `2` неявно выводятся типы: тип обоих значений – `nat` – тип натуральных чисел.

При определении любой теоремы интерактивная среда разделяется на две подсекции, разделяемые длинной чертой:

1. Снизу – список утверждений, которые нужно доказать. Они называются *целями*.

2. Сверху – список так называемых *гипотез* – имеющихся утверждений, которыми можно пользоваться для доказательства *текущей цели* (она идёт первой в списке целей). Список гипотез также называется *контекстом*.

Если гипотез нет (как в теореме `simple_theorem`), то выше разделяющей линии не будет гипотез. Как только все цели доказаны, валидно завершить доказательство с помощью `Qed.`.

1. `simpl` и `reflexivity` – это *тактики*, специфицирующие, как изменяется состояние текущего доказуемого утверждения (цели). Каждая тактика заканчивается точкой.

2. `Proof.` начинает доказательство. Текущая цель: $1 + 1 = 2$.

3. `simpl.` упрощает доказуемое выражение, вычисляя (редуцируя) те подвыражения, которые можно вычислить (средуцировать). Текущая цель становится $2 = 2$.

4. `reflexivity.` завершает доказательство текущей цели, если оно имеет вид `a = a`. На самом деле данная тактика тоже упрощает (прежде, чем завершить док-во или упасть с ошибкой) текущую цель, причем она сильнее `simpl`а.

5. `Qed.` завершает доказательство всей теоремы.

“`=`” является *типом данных*, определение которого мы разберем чуть позже.

2.2 Соответствие Карри-Ховарда I

Одним из ключевых инструментов является **соответствие (изоморфизм) Карри-Ховарда**, которое устанавливает взаимосвязи

- между математическими утверждениями и типами,
- между доказательствами математических утверждений и предоставлением значений соответствующих типов.

Именно благодаря нему существуют среды интерактивных доказательств теорем, представляющих из себя языки программирования, описанные соответствующей теории типов. В основном они реализованы на *исчислении конструкций* – полиморфной теории типов высшего порядка с зависимыми типами, созданной Тьерри Коканом.

2.3 Соответствие Карри-Ховарда II

Рассмотрим больше взаимосвязей.

- Ложной формуле соответствует тип данных, не имеющей ни одного конструктора. Этот тип называется False или bottom ($\perp$) и может в Coq определяться следующим образом:

  ```
  Inductive False : Prop := .
  ```

  Что может интерпретироваться как "задаем новый тип данных `False`, являющийся утверждением и не имеющем ни одного конструктора". Поскольку нет конструкторов, то ложная формула сама по себе не доказуема (ибо априори невозможно предоставить значение этого типа).

- Истинной формуле соответствует тип данных, имеющий ровно один конструктор. Этот тип называется True или top ($\top$) и может в Coq определяться следующим образом:

  ```
  Inductive True : Prop := I : True.
  ```

  У него есть ровно один конструктор `I`, следовательно утверждение типа `True` всегда доказуемо.

- Конъюнкции ($\wedge$) соответствует тип пропозиционального произведения:

  ```
  Inductive and (A B : Prop) : Prop := conj : A -> B -> A /\ B.
  ```

  Во-первых, здесь, `/\` – это так называемая "нотация" в Coq, которая в частности позволяет определять *инфиксные операторы*. Вот как она может задаваться для типа произведения:

  ```
  Notation "A /\ B" := (and A B).
  ```

Во-вторых, "тип произведения" – это, фактически, тип пары. Конструктор `conj` принимает аргумент типа `A` и аргумент типа `B` и конструирует аргумент типа `and A B` или `A /\ B`. В-третьих, "пропозиционального" – потому что `and` принимает два утверждения и возвращает утверждение, а не просто обычные типы (`Type`).

- Дизъюнкции ($\vee$) соответствует тип пропозициональной суммы:

  ```
  Inductive or (A B : Prop) : Prop :=
      or_introl : A -> A \/ B
    | or_intror : B -> A \/ B.
  ```

  "Тип суммы" означает, что `A \/ B` можно сконструировать двумя способами: либо через конструктор `or_introl`, который принимает аргумент типа `A`, либо через конструктор `or_intror`, который принимает аргумент типа `B`.

- Импликации ($\rightarrow$) соответствует тип функции `->`.

- Отрицанию ($\neg$) соответствует следующая функция:

  ```
  Definition not (A : Prop) := A -> False.
  ```

2.4 Пример 5

```
Theorem obviously_false : not (0 = 1).
Proof.
  unfold not.
  intros zero_eq_one.
  discriminate zero_eq_one.
Qed.
```

Докажем, что ноль не равен единице.

1. `unfold not.` раскрывает определение `not` в текущей цели. Эта тактика создана исключительно для удобства пользователей. Цель превращается в `0 = 1 -> False`.

2. `intros zero_eq_one.` вводит в контекст выражение `zero_eq_one` типа `0 = 1` (фактически, это похоже на лямбда-функцию или на замыкание в языках программирования, когда мы вводим аргумент). Контекст: `zero_eq_one : 0 = 1`. Цель: `False`. Нужно доказать ложь. Но... мы уже имеем ложь в контексте, не правда ли? Пока что нет! Для нас $0 = 1$ естественно не является истиной, так давайте же дадим сигнал Coq'у!

3. `discriminate zero_eq_one.` доказывает, что различные конструкторы слева и справа от "равенства" не могут давать равные значения ни при каких обстоятельствах. На данном этапе можно представлять

ноль и единицу как два значения типа `nat`, образованные разными конструкторами (скоро увидим, почему).

4. И вот теперь когда после применения тактики `discriminate` мы получили ложь в контексте, мы можем доказать **всё что угодно** (похоже на определение материальной импликации в классической логике, вспомните таблицу истинности)... В том числе, ложь.

2.5 УСЛОВИЕ

1. Определите свойство симметричности. **0.25 points.**
2. Определите свойство антисимметричности. **0.25 points.**
3. Определите свойство асимметричности. **0.25 points.**
4. Определите свойство транзитивности. **0.25 points.**

В качестве наглядной подсказки возьмите определение рефлексивности из примера 2.1.3.

3 Задача 2

В этой секции мы научимся конструировать кастомные типы данных и сопоставлять с образцом. Coq, помимо всего прочего – это функциональный язык программирования.

3.1 Натуральные числа

Обратимся к началу первого семестра матанализа и вспомним, как задавалось множество натуральных чисел. Теперь добавим туда ноль и превратим это множество в **тип**:

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

`O` – конструктор, описывающий ноль. `S` – конструктор, принимающий натуральное число, фактически описывающий операцию взятия следующего элемента. Тогда `1` можно определить как `S O`, `2` можно определить как `S (S O)`, и тому далее и так подобное. Так они и определены в Coq.

Новое слово: `Set`. Обычно используют вместо `Type` для предикативных индуктивных типов данных, над которыми хочется производить вычисления. В Coq существует целая иерархия таких типов (универсумов, сортов), например:
`Set : Type(1) : Type(2) : ... : Type(i) : Type(i + 1) : ...`
И вместо `Set` также можно подставить `Prop` и `SProp`. А на деле нюансов **катастрофическое** количество.

Определим функцию, возвращающую предыдущее натуральное число:

```
Definition pred : nat -> nat := fun n : nat =>
  match n with
  | O => O
  | S n' => n'
  end.
```

Подвыражение `match n with [...] end` означает, что мы деконструируем натуральное число n и рассматриваем все возможные способы того, как оно было сконструировано. Поскольку в `nat` присутствует ровно 2 конструктора, именно их мы и учитываем в match-выражении. match-выражение похоже на switch case или case of в императивных и функциональных языках.

Если `n` это ноль (т.е. `O`), то вернем его же (мы не можем вернуть "минус один", поскольку мы не можем его задать в рамках типа `nat`, посему приходится вернуть `O`).

Если `n` следует за каким-то `n'` (т.е. `n = S n'`), то вернем `n'`.

3.2 Равенство

Равенство в Coq это **утверждение**, более того – это индуктивный тип:

```
Inductive eq (A : Type) (x : A) : A -> Prop := eq_refl : eq x x.
Notation "x = y" := (eq x y).
```

БОЛЕЕ ТОГО, равенство – **зависимый тип**, т.е. тип, зависящий от терма (значения). `x` – значение типа `A`, и `eq` от него зависит.

У `eq` только один конструктор – `eq_refl` – который фактически декларирует факт того, что мы сможем создать значение типа `eq` тогда и только тогда, когда оба параметра `eq` равны. Эта деталь учитывается при указании типа конструктора (`eq x x`).

Докажем принцип конгруэнтности!

```
Theorem cong {A B : Type} (f : A -> B) (x1 x2 : A) :
  x1 = x2 -> f x1 = f x2.
Proof.
  intros h.
  rewrite h.
  reflexivity.
Qed.
```

1. Сперва, введём гипотезу о том, что `x1 = x2`, с помощью `intros`. Контекст: `h : x1 = x2`. Цель: `f x1 = f x2`.

2. Если `h` имеет тип `a = b`, то `rewrite h` заменяет в цели все вхождения `a` на `b`. Контекст не изменился, а цель: `f x2 = f x2`.

3. Обе части равенства одинаковы, завершаем доказательство с помощью `reflexivity`.

3.3 Существование

В Coq существование определено как тип данных, представляющий из себя своего рода *зависимое произведение*:

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  ex_intro : forall x : A, P x -> exists y, P y.
```

Его единственный конструктор – `ex_intro` – принимает значение `x` типа `A` и предикат `P`, зависящий от этого значения. `exists y, P y` – это удобная нотация для `ex`.

Докажем, что существует натуральное число, равное двум.

```
Example ya : exists n : nat, n = 2.
Proof. exists 2. reflexivity. Qed.
```

1. Если цель имеет тип `exists x : A, P x` и у нас есть доступ к `a : A`, то `exists a` преобразует цель: `P a`. После `exists 2` цель: `2 = 2`.

2. Refurekushibichi.

3.4 Полезные тактики

1. Если либо в контексте есть гипотеза `h : A`, либо она доступна откуда-то из вне, и нужно доказать `A`, то `exact h.` завершит доказательство.

2. Если в контексте есть гипотеза `h : A`, и нужно доказать `A`, то `assumption.` попробует доказать цель, перебрав гипотезы и найдя ту, которая подходит по типу или которая сводится к этому типу.

3. Если в контексте есть гипотеза `p : A /\ B`, то `destruct p as [a b].` разобьёт гипотезу на две: `a : A, b : B`.

4. Если в контексте есть гипотеза `p : A \/ B`, то `destruct p as [a | b].` разобьёт цель на две идентичные, при этом в первой цели гипотеза изменится на `a : A`, а во второй гипотеза изменится на `b : B`.

5. Если нужно доказать `A /\ B`, то `split.` Разобьёт цель на две: `A` и `B`, каждую из которых нужно будет доказать.

6. Если нужно доказать `A \/ B`, то `left.` изменит цель на `A`.

7. Если нужно доказать `A \/ B`, то `right.` изменит цель на `B`.

8. Если нужно доказать `B`, и существует гипотеза `h : A -> B`, то `apply h.` странсформирует цель в `A`.

9. `induction <iden>.` позволит доказать утверждение, скажем, `P`, используя структурную индукцию по индуктивному типу данных. Если `iden` – это натуральное число, то индукция будет в точности математическая (просто частный случай структурной). Рассмотрим случай,

когда `iden : nat`. После обработки вышеупомянутой тактики нужно будет доказать **две** цели:

(a) `P O`. Нужно доказать, что `P` от нуля выполняется. Новых гипотез в контекст не добавляется. Это есть база индукции.

(b) `P (S n')`, при условии, что в контексте есть доказательство `P n'`. Это есть индукционный переход, с добавленными в контекст как значением `n' : nat`, так и индукционной гипотезой `P n'`.

10. Если в контексте имеется гипотеза `f : A1 -> A2 -> ... -> An -> B`, а также если доступны `a1 : A1, ..., an : An`, то после обработки тактики `specialize (f a1 ... an).` контекст обновится: `f : B`. Это не что иное, как подстановка конкретных значений в функцию, доступную в контексте.

11. Если в контексте имеется гипотеза
`f : forall x1 : A1, ..., xn : An, B x1 ... xn`, а также если доступны `a1 : A1, ..., an : An`, то после обработки тактики
`specialize (f a1 ... an).` контекст обновится: `f : B a1 ... an`. Это не что иное, как подстановка конкретных значений в зависимую функцию, доступную в контексте.

3.5 УСЛОВИЕ

1. Докажите первые 9 аксиом пропозиционального исчисления (они действуют как в рамках классической логики, так и в рамках интуиционистской логики; обе отличаются своей десятой аксиомой):

   (a) $A \to B \to A$

   (b) $(A \to B) \to (A \to B \to C) \to (A \to C)$

   (c) $A \land B \to A$

   (d) $A \land B \to B$

   (e) $A \to B \to A \land B$

   (f) $A \to A \lor B$

   (g) $B \to A \lor B$

   (h) $(A \to C) \to (B \to C) \to (A \lor B \to C)$

   (i) $(A \to B) \to (A \to \neg B) \to \neg A$

   Each is worth 0.08 points.

   Каждый тип объявляем `Prop`'ом.

   В качестве референса используем полезные тактики и не забываем про `unfold` для, например, раскрытия определения отрицания, которое определялось выше.

2. Определим сложение натуральных чисел следующим образом:

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (add n' m)
  end.

Notation "x :+ y" := (add x y) (at level 61, left associativity).
```

`Fixpoint` позволяет задавать рекурсивные функции.

Докажите, что сложение ассоциативно. **0.28 points.**

Подсказка: принцип конгруэнтности и индукция, ну а также тактика `rewrite`.

4 Задача 3

4.1 УСЛОВИЕ

Определим композицию функций следующим образом:

```
Definition comp {A B C : Type} (g : B -> C) (f : A -> B) : A -> C :=
  fun x : A => g (f x).

Notation "g :.: f" := (comp g f) (at level 41, right associativity).
```

1. Определите свойство инъективности функции `f : A -> B`. **0.1 points.**
2. Докажите свойство инъективности функции `fun x : A => x`. **0.1 points.**
3. Определите свойство сюръективности функции `f : A -> B`. **0.1 points.**
4. Докажите свойство сюръективности функции `fun x : A => x`. **0.1 points.**
5. Определите свойство биективности функции `f : A -> B`. **0.1 points.**
6. Докажите, что композиция двух биективных функций биективна. (Подсказка: для этого нужно доказать, что композиция инъективных функций инъективна, аналогично с сюръекцией.) **0.5 points.**