Введение в Coq

Coq[1] – это система для работы с формальными доказательствами. Она предоставляет язык для записи математических определений, алгоритмов и теорем, а также среду для интерактивного построения доказательств.

Coq можно установить[2] на Windows, macOS и Linux. Для Visual Studio Code, Emacs и Vim существуют расширения[3], которые поддерживают работу с Coq.

В этом кратком введении мы рассмотрим, как формулировать и доказывать простые утверждения. Для более глубокого изучения на официальном сайте приведено множество источников[4].

1. Первые доказательства

Coq основан на одном из вариантов типизированного λ-исчисления, который называется исчислением индуктивных конструкций[5]. Любое выражение этого языка обладает типом. Узнать его можно с помощью команды `Check`[6]:

```
Check 7.
7 : nat
```

```
Check tt.
tt : unit
```

Мы видим, что выражение 7 имеет тип *nat*, а выражение *tt* – тип *unit*. Типы *nat* и *unit* определены в стандартной библиотеке. Посмотреть их определения можно с помощью команды `Print`:

```
Print unit.
Inductive unit : Set := tt : unit.
```

Мы видим, что тип *unit* имеет тип `Set`, и содержит единственный элемент *tt*. Используя команду `Inductive`, мы можем определять свои типы. Определим более распространенный тип, содержащий два элемента, – тип *bool*:

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

Этот и другие типы и функции, которые встретятся в этом введении, определены в стандартной библиотеке, но в учебных целях мы задаем их самостоятельно.

Упр. 1. Определите тип из трех элементов.

Имея тип *bool*, мы можем определять стандартные булевы функции, например:

```
Definition negb (b : bool) : bool :=
```

[1] https://coq.inria.fr

[2] https://coq.inria.fr/download

[3] https://coq.inria.fr/user-interfaces.html

[4] https://coq.inria.fr/documentation

[5] https://coq.inria.fr/doc/master/refman/language/cic.html

[6] https://coq.inria.fr/doc/master/refman/proof-engine/vernacular-commands.html – документация для команд

```
  match b with
  | true ⇒ false
  | false ⇒ true
  end.
Definition andb (b1 b2 : bool) : bool :=
  match b1 with
  | true ⇒ b2
  | false ⇒ false
  end.
```

Для удобства использования для функций можно задавать нотации:

```
Notation "b1 && b2":= (andb b1 b2).
```

Мы можем проверить, что функция возвращает нужные значения с помощью команды `Compute`:

```
Compute (negb false).
= true : bool
Compute (true && false).
= false : bool
```

Упр. 2. Какой тип у функций *negb* и *andb*?

Упр. 3. Определите функцию *orb*. Задайте для нее нотацию ||.

Чтобы убедиться в корректности определенной функции, мы можем доказывать утверждения про ее поведение. Например, от отрицания мы ожидаем, что его двойное применение оставляет исходное значение без изменений: $\forall\ b,\ negb\ (negb\ b) = b$.

Чтобы доказать такое утверждение нужно рассмотреть два случая:

– если b равно *true*, то $negb\ (negb\ true) = true$,

– если b равно *false*, то $negb\ (negb\ false) = false$.

Запишем это доказательство на Coq. Для этого воспользуемся тактиками[7]:

```
Lemma negb_involutive : ∀ b, negb (negb b) = b.
Proof.
  intros b. destruct b.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

Тактика `intros` вводит переменную под квантором всеобщности в контекст, преобразуя состояние доказательства к следующему виду:

$b : bool$

$\vdash$

$negb\ (negb\ b) = b$

Следующий шаг – проанализировать значения, которые может принимать b. Для этого используется тактика `destruct`, которая разбивает состояние доказательства на две ветви. В первой ветви b принимает значение *true* и нам нужно показать:

$\vdash$

$negb\ (negb\ true) = true$

С помощью тактики `simpl` мы упрощаем это утверждение, вычисляя значение выражения $negb\ (negb\ true)$. Состояние доказательства принимает вид:

$\vdash$

[7] https://coq.inria.fr/doc/master/refman/proof-engine/tactics.html – документация для тактик

$true = true$
Полученное утверждение доказывается с помощью тактики `reflexivity`.
Во второй ветви b принимает значение $false$ и нам нужно показать:
$\vdash$
$negb\ (negb\ false) = false$
Это утверждение доказывается аналогично.
Упр. 4. Докажите коммутативность $andb$: $\forall\ b1\ b2,\ b1\ \&\&\ b2 = b2\ \&\&\ b1$.

2. Индукция

Определим более сложный тип – натуральные числа:
– 0 является натуральным числом,
– если у нас есть натуральное число, то мы можем получить следующее натуральное число.

```
Inductive nat :=
| O : nat
| S : nat → nat.
```

Здесь S обозначает «следующее число» (successor). Таким образом, начиная с O и применяя S многократно, мы получаем все натуральные числа: $O = 0$, $S\ O = 1$, $S\ (S\ O) = 2$, ...

```
Notation "0":= O.
Notation "1":= (S O).
Notation "2":= (S (S O)).
Notation "3":= (S (S (S O))).
```

Для задания операции сложения на натуральных числах мы будем использовать рекурсивное определение. Такое определение вводится с помощью команды `Fixpoint`:

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

```
Notation "m + n":= (add m n).
```

Докажем некоторые свойства операции сложения.
Ноль — левый нейтральный элемент по сложению:

```
Lemma add_0_l : ∀ n, 0 + n = n.
Proof.
  intros n. simpl. reflexivity.
Qed.
```

Так как операция сложения определена с помощью рекурсии по первому аргументу, то утверждение $\forall\ n,\ 0 + n = n$ легко доказывается с помощью тактики `reflexivity`. Сложнее дело обстоит с утверждением $\forall\ n,\ n + 0 = n$ – здесь нам понадобится математическая индукция:

```
Lemma add_0_r : ∀ n, n + 0 = n.
Proof.
  intros n. induction n as [| n IH].
  - reflexivity.
```

```
  - simpl. rewrite IH. reflexivity.
Qed.
```

Тактика `induction` разбивает доказательство на два случая: база и шаг индукции. База индукции имеет вид:

$\vdash$

$0 + 0 = 0$

Это утверждение доказывается с помощью тактики `reflexivity`.

Шаг индукции имеет вид:

$n : nat$

$IH : n + 0 = n$

$\vdash$

$S\ n + 0 = S\ n$

Упростив выражение $S\ n + 0 = S\ n$ с помощью тактики `simpl`, приходим к $S\ (n + 0) = S\ n$. Теперь мы можем использовать индукционную гипотезу $IH : n + 0 = n$, чтобы заменить $n + 0$ на n с помощью тактики `rewrite`. Тактика `reflexivity` завершает доказательство.

С помощью тактики `apply` мы можем применять доказанные ранее утверждения. Например, при доказательстве шага индукции мы вместо тактики `rewrite` можем использовать утверждение из стандартной библиотеки `f_equal` : $\forall\ x\ y,\ x = y \to f\ x = f\ y$.

```
Lemma add_0_r' : ∀ n, n + 0 = n.
Proof.
  intros n. induction n as [| n IH].
  - reflexivity.
  - simpl. apply f_equal. apply IH.
Qed.
```

Команду `Search` можно использовать для поиска доступных утверждений. Например, утверждение `f_equal` можно найти с помощью такого запроса:

```
Search (?x = ?y → ?f ?x = ?f ?y).
```

Упр. 5. Докажите ассоциативность сложения.

```
Lemma add_assoc : ∀ m n o : nat, m + (n + o) = (m + n) + o.
Proof.
Admitted.
```

3. Классы типов

Доказанные свойства сложения говорят, что натуральные числа обладают структурой моноида. Моноидом называется набор элементов с ассоциативной операцией и нейтральным элементом. Эту алгебраическую структуру можно определить с помощью классов типов[8]:

```
Class Monoid {A : Type} (op : A → A → A) (e : A) : Prop := {
    assoc : ∀ x y z, op x (op y z) = op (op x y) z;
    idl : ∀ x, op e x = x;
    idr : ∀ x, op x e = x
  }.
```

[8] https://coq.inria.fr/doc/master/refman/addendum/type-classes.html

Определим экземпляр класса *Monoid*, т.е. покажем, что натуральные числа с операцией сложения обладают структурой моноида:

```
Instance NatAddMonoid : Monoid add 0 := {
    assoc := add_assoc;
    idl := add_0_l;
    idr := add_0_r
  }.
```

Упр. 6. Покажите, что *bool* с операцией *orb* тоже является моноидом.

Мы можем использовать классы типов при определении функций. Эти функции будут работать с любыми экземплярами этих классов типов. Например, определим свертку элементов списка с помощью моноидальной операции.

```
Generalizable Variables A op e.

Local Open Scope list_scope.

Fixpoint iterop `{Monoid A op e} (xs : list A) : A :=
  match xs with
  | nil ⇒ e
  | x :: xs ⇒ op x (iterop xs)
  end.
```

Для натуральных чисел такая свертка дает нам операцию сложения элементов списка.

```
Definition sum : list nat → nat := iterop.

Compute (sum (2 :: 0 :: 1 :: nil)).
= 3 : nat
```

Упр. 7. Что делает такая свертка для булева типа?

Мы можем использовать классы типов для доказательства утверждений. Например, докажем единственность нейтрального элемента в моноиде.

```
Section monoid_facts.
  Context `{Monoid A op e}.

  Lemma unique_id : ∀ a, (∀ x, op x a = x) → a = e.
  Proof.
    intros a Hr. rewrite ← (idl a). apply Hr.
  Qed.
End monoid_facts.
```

Упр. 8. Дайте определение частичного порядка и докажите, что натуральные числа обладают этой структурой.

4. Заключение

Мы научились определять типы и функции, доказывать простые утверждения, познакомились с классами типов. Один из лучших способов закрепить полученные знания и продолжить знакомство с Coq – книга Software Foundations, vol. 1: Logical Foundations, которая свободно доступна онлайн[9].

[9] https://softwarefoundations.cis.upenn.edu

5. Ответы к упражнениям

1.

```
Inductive T3 := x1 | x2 | x3.
```

2.

```
Check negb.
negb : bool -> bool

Check andb.
andb : bool -> bool -> bool
```

3.

```
Definition orb (b1 b2 : bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.

Notation "b1 || b2":= (orb b1 b2).
```

4.

```
Lemma orb_comm : ∀ b1 b2, b1 || b2 = b2 || b1.
Proof.
  intros b1 b2. destruct b1.
  - destruct b2.
    + reflexivity.
    + reflexivity.
  - destruct b2.
    + reflexivity.
    + reflexivity.
Qed.
```

5.

```
Lemma add_assoc' : ∀ m n o : nat, m + (n + o) = (m + n) + o.
Proof.
  intros m n o. induction m as [|| m IH].
  - reflexivity.
  - simpl. rewrite IH. reflexivity.
Qed.
```

6.

```
Lemma orb_assoc : ∀ a b c, orb a (orb b c) = orb (orb a b) c.
Proof.
  intros a b c. destruct a; reflexivity.
Qed.

Lemma orb_false_l : ∀ b, orb false b = b.
Proof.
  intros b. destruct b; reflexivity.
Qed.

Lemma orb_false_r : ∀ b, orb b false = b.
Proof.
  intros b. destruct b; reflexivity.
```

```
Qed.

Instance BoolOrbMonoid : Monoid orb false := {
    assoc := orb_assoc;
    idl := orb_false_l;
    idr := orb_false_r
  }.
```

7. Проверяет есть ли в списке *true*.

```
Definition any : list bool → bool := iterop.

Compute (any (false :: true :: false :: nil)).
= true : bool
```

8. Воспользуемся утверждениями из стандартной библиотекой.

```
Reset nat.
Require Import Arith.

Class PartialOrder {A : Type} (R : A → A → Prop) : Prop := {
    refl : ∀ a, R a a;
    trans : ∀ a b c, R a b → R b c → R a c;
    antisymm : ∀ a b, R a b → R b a → a = b
  }.

Instance NatLePO : PartialOrder le := {
    refl := Nat.le_refl;
    trans := Nat.le_trans;
    antisymm := Nat.le_antisymm
  }.
```