Введение в Rocq

Содержание

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

В этом кратком введении мы рассмотрим, как формулировать и доказывать простые утверждения. Для более глубокого изучения на официальном сайте приведено множество источников[3].

Первые доказательства

Rocq основан на одном из вариантов типизированного λ-исчисления, который называется исчислением индуктивных конструкций[4]. Любое выражение этого языка обладает типом. Узнать его можно с помощью команды `Check`[5]:

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

```
Check tt.
```

[1] https://rocq-prover.org

[2] https://rocq-prover.org/install

[3] https://rocq-prover.org/docs

[4] https://rocq-prover.org/doc/master/refman/language/cic.html

[5] https://rocq-prover.org/doc/master/refman/proof-engine/vernacular-commands.html – документация для команд

```
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.
```

Тип *unit* имеет тип `Set`, и содержит единственный элемент *tt*. Используя команду `Inductive`, можно 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 :=
  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 Rocq. Для этого воспользуемся тактиками[6]:

```
Lemma negb_involutive : ∀ b, negb (negb b) = b.
Proof.
  intros b. destruct b.
```

[6] https://rocq-prover.org/doc/master/refman/proof-engine/tactics.html – документация для тактик

```
  - 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:

$\vdash$

$true = true$

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*: $\forall\ b1\ b2,\ b1\ \&\&\ b2 = b2\ \&\&\ b1$.

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
| S (n : 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 ∀ n, $0 + n = n$ легко доказывается с помощью тактики `reflexivity`. Сложнее обстоит дело с утверждением ∀ 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` : ∀ $x\ y$, $x = y \to f\ x = f\ y$.

```
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.
```

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[7]:

```
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
}.
```

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.
```

[7] https://rocq-prover.org/doc/master/refman/addendum/type-classes.html, https://softwarefoundations.cis.upenn.edu/qc-current/Typeclasses.html

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

Синтаксис и семантика

Rocq можно использовать не только для работы со стандартными математическими объектами, но и с объектами из теории языков программирования, что дает возможность доказывать утверждения о поведении программ. Для этого необходимо задать синтаксис и семантику языка программирования, на котором эти программы написаны. Рассмотрим, как это можно сделать, на примере простого императивного языка, программы на котором могут выглядеть следующим образом:

```
z := x;
y := 1;
while z > 0 do
  y := y * z;
  z := z - 1;
```

Как видно из примера наши программы работают с числами и содержат переменные, поэтому импортируем необходимые модули для работы с целыми числами и строками, которые используются для именования переменных:

```
Require Import String ZArith.
Local Open Scope string_scope.
Local Open Scope Z_scope.
```

Также видно, что наш простой язык должен поддерживать некоторый набор привычных операций на целых числах. Эти числа мы можем складывать, вычитать, умножать, сравнивать:

```
Inductive binop : Type := Oplus | Osub | Omul | Oeq | Olt.
```

Выражения в нашем языке могут быть переменными, константами или применением бинарной операции к двум выражениям. Абстрактный синтаксис выражений обычно задается в форме Бэкуса-Наура. Для наших выражений он имеет следующий вид:

```
exp := var
     | Z
     | exp op exp
```

Это определение легко переносится в Rocq с помощью индуктивных типов:

```
Inductive exp : Type :=
| Var (x : string)
| Const (n : Z)
| Binop (op : binop) (e1 e2 : exp).
```

Наш язык будет иметь:

- команду `skip`, которая ничего не делает;
- команду присваивания, которая записывает результат вычисления выражения в переменную;
- оператор последовательного выполнения двух команд;
- условный оператор;
- оператор цикла.

Аналогично, сначала зададим абстрактный синтаксис команд в форме Бэкуса-Наура:

```
com := 'skip'
     | var ':=' exp
     | com ';' com
     | 'if' exp 'then' com 'else' com
     | 'while' exp 'do' com
```

Затем перенесем это определение команд в Rocq с помощью индуктивных типов:

```
Inductive com : Type :=
| Skip
| Assign (x : string) (e : exp)
| Seq (c1 c2 : com)
| If (e : exp) (c1 c2 : com)
| While (e : exp) (c : com).

Notation "x ::= e":= (Assign x e) (at level 75).
Notation "c1 ;; c2":= (Seq c1 c2) (at level 80, right associativity).
```

Теперь можно записывать наши програмы в Rocq. В частности, программа из начала этого раздела в нашем абстрактном синтаксисе записывается так:

```
Definition factorial : com :=
  "z"::= Var "x";;
  "y"::= Const 1 ;;
  While (Binop Olt (Const 0) (Var "z"))
    ("y"::= Binop Omul (Var "y") (Var "z") ;;
     "z"::= Binop Osub (Var "y") (Const 1)).
```

Чтобы рассуждать о поведении программ нам необходимо определить еще и их семантику. Сначала определим семантику выражений. Для этого каждому обозначению бинарной операции из синтаксиса сопоставим функцию на целых числах, которая выполняет эту бинарную операцию:

```
Definition eval_binop (op : binop) : Z → Z → Z :=
  match op with
  | Oplus ⇒ Z.add
  | Osub ⇒ Z.sub
  | Omul ⇒ Z.mul
  | Oeq ⇒ fun m n ⇒ Z.b2z (m =? n)
  | Olt ⇒ fun m n ⇒ Z.b2z (m <? n)
  end.
```

Наши выражения содержат переменные, поэтому нам потребуется состояние, в котором эти переменные имеют значения. Наши состояния – это функции из строк в целые чила:

```
Definition state : Type := string → Z.
```

Теперь можно определить семантику выражения e в состоянии s как целое число:

```
Fixpoint eval (e : exp) (s : state) : Z :=
  match e with
  | Var x ⇒ s x
  | Const n ⇒ n
  | Binop op e1 e2 ⇒ eval_binop op (eval e1 s) (eval e2 s)
```

```
end.
```

Полученная семантика выражений используется для определения семантики команд. Для определения команды присваивания нам также пригодится функция, добавляющая переменные в состояние:

```
Definition update (x : string) (n : Z) (s : state) : state :=
  fun y ⇒ if string_dec x y then n else s x.
```

Семантику команд можно задавать разными способами. Приведем определение, которое называют операционной семантикой с большим шагом. Сначала запишем его с помощью правил вывода и нотации `s =[ c ]=> s'`, означающей, что выполнение команды `c` меняет состояние `s` на состояние `s'`:

```
                    ---------------                 (ESkip)
                    s =[ Skip ]=> s

                     eval e s = n
             -----------------------------          (EAsgn)
             s =[ x := e ]=> update x n s

                    s1 =[ c1 ]=> s2
                    s2 =[ c2 ]=> s3
                  -------------------               (ESeq)
                  s1 =[ c1; c2 ]=> s3

                     eval e s1 = n
        s1 =[ if n = 0 then c2 else c1 ]=> s2
      --------------------------------------        (EIf)
              s1 =[ If e c1 c2 ]=> s2

                     eval e s1 != 0
                     s1 =[ c ]=> s2
                 s2 =[ While e c ]=> s3
                 ----------------------             (EWhileTrue)
                 s1 =[ While e c ]=> s3

                     eval e s = 0
                  --------------------              (EWhileFalse)
                  s =[ While e c ]=> s
```

Перенесем это определение в Rocq с помощью индуктивных типов.

```
Inductive ceval : com → state → state → Prop :=
| ESkip : ∀ s, ceval Skip s s
| EAssign : ∀ s x e, ceval (x ::= e) s (update x (eval e s) s)
| ESeq : ∀ s1 s2 s3 c1 c2,
    ceval c1 s1 s2 → ceval c2 s2 s3 →
    ceval (c1 ;; c2) s1 s3
| EIf : ∀ s1 s2 c1 c2 e,
    ceval (if (Z.eq_dec (eval e s1) 0) then c2 else c1) s1 s2 →
    ceval (If e c1 c2) s1 s2
| EWhileTrue : ∀ s1 s2 s3 c e,
    eval e s1 ≠ 0 →
```

```
    ceval c s1 s2 → ceval (While e c) s2 s3 →
    ceval (While e c) s1 s3
| EWhileFalse : ∀ s c e,
    eval e s = 0 →
    ceval (While e c) s s.
```

Теперь мы можем доказывать утверждения про поведение простых программ. Рассмотрим следующую программу из одной команды:

```
x := x + 2
```

Определим ее в Rocq:

```
Definition plus2 : com :=
  "x"::= Binop Oplus (Var "x") (Const 2).
```

Докажем, что если она:

- начинает выполнение в исходном состоянии, где переменная x равна некоторому числу n,
- заканчивает свое выполнение,

то значение переменной x в конечном состоянии равно $n + 2$.

Кратко это обычно записывают с помощью троек Хоара:

```
{ x = n } x := x + 2 { x = n + 2 }
```

Для доказательства нам потребуется вспомогательное утверждение про функцию *update*:

```
Lemma update_same : ∀ x n s, (update x n s) x = n.
Proof.
  intros. unfold update. now destruct (string_dec x x).
Qed.

Lemma plus2_spec : ∀ (n : Z) (s s' : state),
  n = s "x"→ ceval plus2 s s' → n + 2 = s' "x".
Proof.
  intros n s s' Hn H.
  inversion H. subst. simpl. now rewrite update_same.
Qed.

Close Scope Z_scope.
```

Заключение

Мы научились определять типы и функции, доказывать простые утверждения, познакомились с классами типов, синтаксисом и семантикой языков программирования. Один из лучших способов закрепить полученные знания и продолжить знакомство с Rocq и теорией языков программирования – книга Software Foundations, которая свободно доступна онлайн[8].

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

Приложение А. Список некоторых тактик

С полным списком доступных тактик можно ознакомиться в официальном руководстве[9].

`intros`	перемещает гипотезы/переменные из цели в контекст
`reflexivity`	доказывает цель вида `x = x`
`apply ...`	доказывает цель с использованием гипотезы, леммы или конструктора
`apply ... in H`	применяет гипотезу, лемму или конструктор к гипотезе `H` в контексте (прямое рассуждение)
`apply ... with ...`	`with` позволяет явно указывать значения для переменных
`simpl`	упрощает выражения в цели
`simpl in H`	... в гипотезе `H`
`rewrite ...`	использует равенство для перезаписи цели
`rewrite ... in H`	... гипотезы `H`
`symmetry`	преобразует цель вида `a = b` в `b = a`
`unfold ...`	раскрывает определение в цели
`unfold ... in H`	... в гипотезе
`destruct ... as ...`	анализ случаев для значений индуктивно определенных типов
`destruct ... eqn:H`	указывает имя уравнения, которое будет добавлено в контекст, фиксируя результат анализа случаев
`induction ... as ...`	индукция по значениям индуктивно определённых типов
`injection ... as ...`	рассуждение на основе инъективности для равенств между значениями индуктивно определённых типов
`discriminate H`	рассуждение на основе неравенства конструкторов
`inversion H`	
`assert (H : e)`	вводит локальное утверждение с именем `H`
`generalize dependent x`	перемещает переменную `x` (и всё, что от неё зависит) из контекста обратно в явную гипотезу в цели
`assumption`	доказывает цель, если она совпадает с одним из утверждений в контексте

[9] https://rocq-prover.org/doc/master/refman/rocq-tacindex.html

Приложение Б. 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, a || (b || c) = (a || b) || c.
Proof.
  intros a b c. destruct a; reflexivity.
Qed.
```

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

```
Lemma orb_false_r : ∀ b, 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;
}.
```