Coq

Евгений Кирпичев
SPbHUG
2008

Цель

Дать kick-start и облегчить самостоятельное изучение

Preconditions:

- Знакомство с идеями ФП
- Знакомство с основами логики

После:

- Интуитивное понимание самого важного в Coq
- Умение формулировать и доказывать простые вещи
- Уменьшенная вероятность застрять в книжке или серьезной задаче

Не цель

- Научить пользоваться Coq на серьезных задачах
- Строго и правильно рассказать о теории
- Рассказать обо всех особенностях синтаксиса
- Научить пользоваться тулами
- Рассказать о стандартной библиотеке

План

- Что такое Coq и что он может
- Теория
  - Curry-Howard
  - λ-куб
  - Интуиционистская логика
  - Индуктивные типы
- Kick-start, основные тактики
- Комбинаторы
- Разбор случаев и индукция
- Ltac
- Индуктивные предикаты и инверсия
- Сертифицированные функции
- And stuff

Что такое Coq

- Верификатор доказательств
- Язык для описания математики
- Язык программирования
- **Typechecker**

Что он может

- Задача о 4 красках (2500кб - тут)
- Trusted Logic: JavaCard (тут)
- Gemalto: JavaCard (тут)
- Компилятор си (тут)
- GMP sqrt (13000 строк - тут)
- Теорема о неполноте Гёделя (1300кб - тут)

- Стандартная библиотека + user contributions:
  - 275 библиотек, 8000 теорем,аксиом и определений
  - Логика, арифметика над натуральными и целыми, вещественные, множества, отношения, сетоиды, списки/потоки, IntMap, строки, сортировки
  - …

Структура Coq

Тулы	Front-end (coqtop, coqide); coqc
Либы	Initial; Basic; Standard; Contribs
Язык	Нотации; Секции; Implicit args; Records; …
Ядро	Ltac; Тактики; Модули; …; Coq core (typechecker)
Теория	Индуктивные типы; λ-куб; Curry-Howard

Теория: 1. Curry-Howard

- Теоремы ⇔ типы, доказательства ⇔ термы
- «Тип» доказательства = теорема, которую оно доказывает
- Терм доказывает населенность своего типа
- «Населенность» теоремы = ее доказуемость
- Проверка доказательства = проверка типа терма

```
is_human  :: person → Prop
thinks    :: person → Prop
is_mortal :: person → Prop

T         :: (x::person) → thinks x → is_human x
B         :: (x::person) → is_human x → is_mortal x
Socrates  :: person
ST        :: thinks Socrates

imp_trans :: (A::Prop) → (B::Prop) → (C::Prop) → (A → B) → (B → C) → (A → C)
imp_trans A B C AthenB BthenC = fun a => BthenC (AthenB a)

imp_trans (thinks Socrates) (is_human Socrates) (is_mortal Socrates)
          (T Socrates) (B Socrates) Socrates :: is_mortal Socrates
```

Теория: 2. Интуиционистская логика

<table>
<tr><td>Все доказательства конструктивны</td><td>Все функции вычислимы</td></tr>
<tr><td>A → B это «способ вычислить доказательство B, используя доказательство A»</td><td>A → B это «способ вычислить значение B, используя значение A»</td></tr>
<tr><td>False – теорема, не имеющая доказательства</td><td>False – тип, не населенный термами</td></tr>
<tr><td>~X = (X → False) – «X абсурдно, противоречит недоказуемости False»</td><td>~X = (X →False) – «X противоречит ненаселенности False»</td></tr>
<tr><td colspan="2">Нет аксиомы двойного отрицания, т.к. оно неконструктивно</td></tr>
<tr><td colspan="2">~~X = ((X → False) →False)
“~~X → X” - недоказуемо
неоткуда взяться терму типа X, т.к. нету →X, есть только →False !</td></tr>
<tr><td>Доказательства одной теоремы неразличимы
(proof irrelevance)</td><td>Термы одного типа различимы
Integer → Integer</td></tr>
</table>

Теория: 3. λ-куб

Сорты Set, Prop, Type

```
> Check 1.
        1 : nat

> Check nat.
        nat : Set

> Check Set.
        Set : Type(i)

> Check Type.
        Type(i) : Type(i+1)

> Check False.
        False : Prop

> Check (1<2).
        1 < 2 : Prop

> Check Prop.
        Prop : Type(i)

> Check (fun n:nat => n:n).
        (fun n:nat => n*n) : nat -> nat

> Check is_prime.
        is_prime : nat -> Prop

> Check list.
        list : Type -> Type

> Check and.
        and : Prop -> Prop -> Prop

> Check binary_word.
        binary_word : nat -> Type
```

Теория: 3. λ-куб

Квантор forall:∀

Для не-зависимых типов – синоним стрелки; не нужен

```
“sqr : nat -> nat” ~ “sqr : ∀ (_:nat), nat”
```

Для зависимых типов – нужен

```
compose : ∀(A B C:Prop), (A->B)->(B->C)->(A->C)

range : ∀(a b:nat), fixed_len_list (b-a+1)
range 5 10 : fixed_len_list (10-5+1)
```

Теория: 3. λ-куб

Квантор forall:∀

Для зависимых типов – нужен

```
binary_word_or : ∀ n:nat,
   binary_word n -> binary_word n -> binary_word n

symmetric : ∀ (A:Type) (R:A->A->Prop) (a b:A), R a b -> R b a

symmetric nat mutually_prime :
   ∀ a b:nat, mutually_prime a b -> mutually_prime b a

total_perm : ∀ (A:Type) (R:A->A->Prop),
   (∀a b:A, R a b) -> (∀a b:A, R b a)
```

Теория: 4. Редукции

- delta – раскрывает определения

```
sqr = fun a:Z => a*a
sqr 5 ➔ (fun a:Z => a*a) 5
```

- beta – лямбда-аппликация

```
(fun m n : Z => m + n) 5 6 ➔ 5 + 6
```

- zeta – let..in

```
let a:=5 in a+a ➔ 5+5
```

- iota – fixpoint структурной рекурсии

```
5 + 6 ➔(delta+iota)➔11
```

- Все функции завершаются (рекурсия только структурная)
- У всех термов есть нормальная форма
  - Одинаковая ➔ они «конвертируемы» (convertible)

Теория: 5. Индуктивные типы

- Похоже на Algebraic Datatypes
- Более похоже на GADT

- Тип задается набором конструкторов

```
Inductive the_type p1 p2 … pn : Prop/Set/Type =
  | constr1 : … -> … -> … -> the_type a1 a2 … an
  | constr2 : … -> … -> … -> the_type a1 a2 … an
  | …
```

- Чрезвычайно общо и мощно
  - Индуктивные (и рекурсивные) структуры данных
  - Индуктивные (и рекурсивные) предикаты
  - And more

Теория: 5. Индуктивные типы

- Теоремы про индуктивные типы обычно доказывают по индукции
- Функции над индуктивными типами обычно определяют по индукции
- Структура индукции совпадает со структурой типа

```
data Nat = Zero | S Nat
```

Доказательство по индукции на Nat:

`P(Zero)` ➔ `(∀n,P(n)` ➔ `P(S n))` ➔ `∀n, P(n)`

Типичная рекурсивная функция на Nat:

`f(Zero)` ➔ `(∀n,f(n)` ➔ `f(S n))` ➔ `∀n, f(n)`

Различие: `P:nat->Prop`
`f:nat->Set`

(есть еще `nat->Type`)

Coq <u>сам</u> выводит такие схемы при определении индуктивного типа

- Но это не аксиомы!

Теория: 5. Индуктивные типы

- Сорт Set или Type
  - Структурная рекурсия
    - `mytype_rect : ∀ … (P:mytype → Type), … → ∀ t:mytype, P t`
  - Индуктивные теоремы обо всех термах данного типа
    - `mytype_ind : ∀ … (P:mytype → Prop), … -> ∀ t:mytype, P t`
      - Частный случай mytype_rect
  - Структурно-рекурсивные функции
    - `mytype_rec`
      - Частный случай mytype_rect
- Сорт Prop
  - Теоремы обо всех предикатах
    - `mytype_ind` : Увидим.

Теория: 5. Индуктивные типы

Сорт Set: ADT

– Индукция дает теоремы обо всех термах такого типа

```
Inductive Color : Set :=
  | Red : Color | Green : Color | Blue : Color

Color_ind : ∀(P:Color->Prop),
   P Red -> P Green -> P Blue -> ∀c:Color, P c
```

Теория: 5. Индуктивные типы

Сорт Set: ADT

– Индукция дает теоремы обо всех термах такого типа

```
Inductive ZPoint : Set :=
  | mk_point : Z -> Z -> ZPoint

ZPoint_ind : ∀(P:ZPoint->Prop),
  ∀ (a b:Z), P (mk_point a b) -> ∀(p:ZPoint), P p
```

Теория: 5. Индуктивные типы

Сорт Set: ADT

- Индукция дает теоремы обо всех термах такого типа

```
Inductive btree (A:Type) : Type :=
  | bleaf : A -> btree A
  | bnode : A -> btree A -> btree A -> btree A
```

```
btree_ind : ∀(A:Type) (P:btree->Prop),
  (∀(a:A), P (bleaf a)) ->
  (∀(a:A) (t1 t2:btree A), P t1 -> P t2 -> P (bnode a t1 t2)) ->
  ∀(t:btree A), P t
```

Теория: 5. Индуктивные типы

Сорт Set: GADT

- Индукция дает теоремы обо всех термах такого типа

```
Inductive htree (A:Set) : nat -> Set :=
  | hleaf : A -> htree A 0
  | hnode : ∀n:nat, A -> htree A n -> htree A n -> htree A (S n).

htree_ind : ∀(A : Set) (P : ∀n : nat, htree A n -> Prop),
       (∀a : A, P 0 (hleaf A a)) ->
       (∀(n : nat) (a : A) (t1 t2: htree A n),
         P n t1 -> P n t2 -> P (S n) (hnode A n a t1 t2)) ->
       ∀ (n : nat) (t : htree A n), P n t
```

(Coq выдает немного другой терм, но по смыслу он такой же, с точностью до карринга)

Теория: 5. Индуктивные типы

Сорт Set: GADT

- Индукция дает теоремы обо всех термах такого типа

```
Inductive binary_word : nat -> Type :=
  | bw_empty : binary_word 0
  | bw_cons : ∀n:nat, bool -> binary_word n -> binary_word (S n).

binary_word_ind : ∀P : (∀n : nat, binary_word n -> Prop),
       P 0 bw_empty ->
       (∀(n : nat) (b : bool) (word' : binary_word n),
          P n word' -> P (S n) (bw_cons n b word')) ->
       ∀ (n : nat) (word : binary_word n), P n word
```

Теория: 5. Индуктивные типы

Сорт Set: GADT

– Индукция дает теоремы обо всех термах такого типа

```
Inductive htree (A:Set) : nat -> Set :=
  | hleaf : A -> htree A 0
  | hnode : ∀n:nat, A -> htree A n -> htree A n -> htree A (S n).
```

Почему не `htree (A:Set) (n:nat)` ?

– Все конструкторы должны конструировать один и тот же тип
– В данном случае конструируют `htree A`
– А с `n:nat` было бы `htree A 0` vs. `htree A (S n)`.

Почему не `htree : Set -> nat -> Set` ?

– Можно (правда, только с Type, а с Set почему-то нельзя), но:
– усложняется *принцип индукции*, *принцип инверсии* итп.

Теория: 5. Индуктивные типы

Сорт Prop: Индуктивные предикаты

– Индукция дает индуктивные доказательства других предикатов при условии данного

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : (le n) n
  | le_S : ∀ (m:nat), (le n) m -> (le n) (S m)

le_ind
     : ∀ (n : nat) (P : nat -> Prop),
       P n ->
       (∀ m : nat, (le n) m -> P m -> P (S m)) ->
       ∀ q : nat, (le n) q -> P q
```

Подстановка (le n) => P

Теория: 5. Индуктивные типы

Сорт Prop: Индуктивные предикаты

– Индукция дает индуктивные доказательства других предикатов при условии данного

```
Inductive even : nat -> Prop :=
  | O_even : even O
  | plus_2_even : ∀ n:nat, even n -> even (S (S n)).
```

Подстановка even => P

```
even_ind
     : ∀ P : nat -> Prop,
      P 0 ->
       (∀ n : nat, even n -> P n -> P (S (S n))) ->

       ∀ n : nat, even n -> P n
```

Теория: 5. Индуктивные типы

Сорт Prop: Индуктивные предикаты

- Индукция дает индуктивные доказательства других предикатов при условии данного

```
Inductive sorted (A:Set) (R:A->A->Prop): list A -> Prop :=
  | sorted0 : sorted A R nil
  | sorted1 : ∀ a:A, sorted A R (a::nil)
  | sorted2 : ∀(x y:A) (s:list A),
               R x y -> sorted A R (y::s) -> sorted A R (x::y::s).
```

Подстановка (sorted A R) => P

```
sorted_ind
     : ∀ (A:Set) (R:A->A->Prop) (P:list A->Prop),
       P nil ->
       (∀ a : A, P (a::nil)) ->
       (∀ (x y:A) (s:list A),
        R x y -> sorted A R (y::s) -> P (y::s) -> P (x::y::s)) ->

       ∀ s:list A, sorted A R s -> P s
```

Теория: 5. Индуктивные типы

Сорт Prop: Индуктивные предикаты

- Индукция дает индуктивные доказательства других предикатов при условии данного

```
Inductive clos_trans (A:Type) (R:A->A->Prop) : A -> A -> Prop :=
  | t_step : ∀ x y:A, R x y -> (clos_trans A R) x y
  | t_trans : ∀ x y z:A,
              (clos_trans A R) x y -> (clos_trans A R) x z -> (clos_trans A R) y z.
```

Подстановка (clos_trans A R) => P

```
clos_trans_ind
     : ∀ (A : Type) (R P : A -> A -> Prop),
       (∀ x y : A, R x y -> P x y) ->
       (∀ x y z : A,
        (clos_trans A R) x y -> (clos_trans A R) x z -> P x y -> P x z -> P y z) ->

       ∀ x y : A, (clos_trans A R) x y -> P x y
```

(слегка отличается от вывода Coq – карринг/порядок аргументов во втором клозе)

Теория: 5. Индуктивные типы

Сорт Prop: Индуктивные предикаты

Что пихать в параметры, а что в «тип типа»?

```
Inductive divided_by_3_or_5 (n:nat) : Prop :=
  | by_3 : ∀ (m:nat), 3*m = n -> divided_by_3_or_5 n
  | by_5 : ∀ (m:nat), 5*m = n -> divided_by_3_or_5 n.

divided_by_3_or_5_ind
     : ∀ (n : nat) (P : Prop),
       (∀ m : nat, 3 * m = n -> P) ->
       (∀ m : nat, 5 * m = n -> P) -> divided_by_3_or_5 n -> P
```

Предикаты, которые верны, если доказуемо (divided_by_3_or_5 n)

```
Inductive divided_by_3_or_5' : nat->Prop :=
  | by_3 : ∀ (n m:nat), 3*m = n -> divided_by_3_or_5' n
  | by_5 : ∀ (n m:nat), 5*m = n -> divided_by_3_or_5' n.

divided_by_3_or_5'_ind
     : ∀ P : nat -> Prop,
       (∀ n m : nat, 3 * m = n -> P n) ->
       (∀ n m : nat, 5 * m = n -> P n) ->
       ∀ n : nat, divided_by_3_or_5' n -> P n
```

Предикаты, которые верны для n, если доказуемо (divided_by_3_or_5') n

Результат квантифицирован так же, как и индуктивный тип.

Теория: 5. Индуктивные типы

Сорт Prop: Фундаментальная логика

- Индукция дает индуктивные доказательства других предикатов при условии данного

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

True_ind :
          ∀ P : Prop, P -> True -> P



Inductive False : Prop :=
  пусто.

False_ind :
          ∀ P : Prop, False -> P
```

Теория: 5. Индуктивные типы

Сорт Prop: Фундаментальная логика

- Индукция дает индуктивные доказательства других предикатов при условии данного

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

(доказательство `and A B` **формируется из доказательства A и доказательства B)**

```
and_ind
     : ∀ A B P : Prop, (A -> B -> P) ->
       (A /\ B -> P)
```

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

(доказательство `or A B` **формируется из доказательства A или из доказательства B)**

```
or_ind
      : ∀ A B P : Prop, (A -> P) -> (B -> P) ->
        (A \/ B -> P)
```

Теория: 5. Индуктивные типы

Сорт Prop: Фундаментальная логика

– Индукция дает индуктивные доказательства других предикатов при условии данного

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : ∀ x : A, P x -> ex A P
```

(нотация: `ex A P ≡ exists a:A, P a`**)**
(доказательство `ex A P` **формируется из *любой* пары** `(a:A, доказательство P a)` **)**

```
ex_ind
      : ∀ (A : Type) (P : A -> Prop) (Q : Prop),
       (∀ x : A, P x -> Q) ->
        (ex A P -> Q)
```

```
Inductive eq (A : Type) (x:A) : A -> Prop :=
  | refl_equal : (eq A x) x
```

```
eq_ind
      : ∀ (A : Type) (x : A) (P : A -> Prop),
       P x ->
        ∀ y : A, (eq A x) y -> P y
```

(нотация: `(eq A x y) ≡ (x=y)`, **аргумент A – *неявный (implicit argument)*)**

Теория: 5. Индуктивные типы

Сорт Prop: Фундаментальная логика

- Индукция дает индуктивные доказательства других предикатов при условии данного

```
Inductive eq (A : Type) (x:A) : A -> Prop :=
  | refl_equal : (eq A x) x

eq_ind
     : ∀ (A : Type) (x : A) (P : A -> Prop),
       P x ->
       ∀ y : A, (eq A x) y -> P y
```

Сравним:

```
Inductive eq' (A:Type) : A -> A -> Prop :=
  | refl_equal' : ∀ (a:A), (eq' A) a a

eq'_ind : ∀ (A : Type) (P : A -> A -> Prop),
        (∀ a : A, P a a) ->
         ∀ y z : A, eq' A y z -> P y z
```

Теория: 5. Индуктивные типы

Спецификации.
Функция + доказательство корректности

```
Inductive pred_spec (n:nat) : Set :=
  | pred_data : ∀ (p:nat), (S p = n) -> pred_spec n.

Inductive sqrt_spec (n:nat) : Set :=
  | sqrt_data : ∀ x:nat, x*x <= n -> n < (S x)*(S x) -> sqrt_data n.

Inductive prime_decomp_spec : nat -> Set :=
  | already_prime : ∀ (x:nat), is_prime x -> prime_decomp_spec x
  | decomp : ∀ (x:nat) (a b:nat),
      a*b = x -> is_prime a -> a < b < x -> prime_decomp_spec x.

Definition sqrt : ∀(n:nat), sqrt_spec n :=
   … (Придется вычислить не только корень, но и
     доказательство, что это действительно корень!) …
```

Из терма такого типа можно достать и ответ, и доказательство правильности ответа.

Индукция по ним, кажется, не очень интересна.

Теория: 5. Индуктивные типы

- Задаются несколькими конструкторами
  - Возможно, рекурсивными
  - Конструкторы разделимы (C … <> D …)
  - Конструкторы инъективны (C a = C b ➔ a = b)
  - Это не аксиомы

- Имеют тип и сорт сами по себе

- С ними ассоциирован *принцип индукции*
  - Set → структурная рекурсия
  - Prop → индуктивное д-во верности других предикатов при условии данного
    - На самом деле, отличается от Set только с точностью до Proof Irrelevance (принципу не передается *значение* индуктивного типа)
  - Type – их обобщение (?)

- С ними ассоциирован *принцип инверсии*

Ugh!

Kick-start

Запросы

- `Check (терм)` **= вывести тип терма**

```
> Check (5+5).
   5+5 : nat
> Check nat.
   nat : Set
> Check Set.
   Set : Type
> Check Type.
   Type : Type
> Check fun (f g : nat->nat) (n:nat) => g (f n).
   fun (f g : nat -> nat) (n : nat) => g (f n)
     : (nat -> nat) -> (nat -> nat) -> nat -> nat
```

- `Print` **= показать значение идентификатора**

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

Kick-start

- `Search` – все факты с участием идентификатора

```
> Search le.
    gt_S_le: ∀ n m : nat, S m > n -> n <= m
    gt_le_S: ∀ n m : nat, m > n -> S n <= m
    le_refl: ∀ n : nat, n <= n
    le_trans: ∀ n m p : nat, n <= m -> m <= p -> n <= p
    le_O_n: ∀ n : nat, 0 <= n
    ...
```

- `SearchPattern` **– поиск по шаблону**

```
> SearchPattern (?X1 * _ <= ?X1 * _)%Z.
    Zmult_le_compat_l:
      ∀ n m p : Z, (n <= m)%Z -> (0 <= p)%Z -> (p * n <= p * m)%Z
```

Увы, не поддерживает стрелки – `SearchPattern (_<_ -> _<_)` **не работает.**

- …

Kick-start

Вычисление

- `Eval [compute | cbv | lazy | hnf | simpl | fold | unfold] {beta,delta,zeta,iota} [sym1 sym2 .. ] in expr` – произвести редукции

```
> Eval cbv delta [sum_mul_dif] in (sum_mul_dif 5 2).
    = (fun a b : nat => let s := a + b in let d := a - b in s * d) 5 2
    : nat
> Eval cbv beta delta [sum_mul_dif] in (sum_mul_dif 5 2).
    = let s := 5 + 2 in let d := 5 - 2 in s * d
    : nat
> Eval cbv beta zeta delta [sum_mul_dif] in (sum_mul_dif 5 2).
    = (5 + 2) * (5 - 2) : nat
> Eval cbv beta iota delta [plus sum_mul_dif] in (sum_mul_dif 5 2).
    = let s := 7 in let d := 5 - 2 in s * d : nat
> Eval cbv beta iota zeta delta [plus sum_mul_dif] in (sum_mul_dif 5 2).
    = 7 * (5 - 2) : nat
> Eval cbv beta iota zeta delta in (sum_mul_dif 5 2).
    = 14 : nat
> Eval compute in (sum_mul_dif 5 2).
    = 14 : nat
```

lazy дает те же результаты, но иногда быстрее работает.

Kick-start

Объявления

- **`Definition f a1 a2 .. an := …`**

```
Definition sqr (x:nat) : nat := x*x.
Definition sqr : nat->nat := fun (x:nat) => x*x.
Definition compose :
     ∀ (A:Set) (B:Set) (C:Set), (B->C) -> (A->B) -> (A->C) :=
   fun _ _ _ g f a => g (f a).
```

- **`Theorem, Lemma`** – синоним Definition, но полученный идентификатор *скрыт* (opaque)
  - Не поддается дельта-редукции (подстановке)

```
Theorem le_35_36 : 35 <= 36 := le_S 35 (le_n 35).
```

- **`Axiom, Parameter`** – параметр вообще без определения

```
Axiom double_neg : ∀ (P:Prop), ~~P -> P.
```

Kick-start

Выражения

- λ-аппликация

```
f x y z ≡ (f x y) z ≡ ((f x) y) z ≡ (f x) y z
```

- λ-абстракция

```
fun (x:A) (y:B) (…) => res
```

- let..in

```
let sum := m + n in
let dif := m - n in
(sum,dif)
```

- Pattern matching

```
Definition greater_than_3 (n:nat) : Prop :=
   match n with
   | S (S (S (S _))) => True
   | _ => False
   end.

Definition sq_dist (p:ZPoint) : Z :=
  match p with (mk_point x y) => x*x + y*y end.
```

С зависимыми типами – гораздо сложнее.

Kick-start

Рекурсивные объявления

- Fixpoint

```
Fixpoint sum (m n:nat) {struct m} :=
  match m with
  | O => n
  | S m' => S (sum m' n)
  end.
```

- Через *_rec

```
> Check nat_rec.
    nat_rec : ∀ P : nat -> Set,
       P 0 -> (∀ n : nat, P n -> P (S n)) -> ∀ n : nat, P n

> Definition sum (m:nat) := nat_rec
     (fun _ => nat->nat)
     (fun n => n)
     (fun m' s_m' => fun n => S (s_m' n))
    m.
    sum is defined

> Eval compute in (sum 5 8).
     = 13 : nat
```

Процесс доказательства

- Доказательство = предъявление терма нужного типа
- Можно все теоремы доказывать явными термами
- Тактики позволяют строить терм *постепенно*
  - Доказать несколько *целей в контексте*
  - Цель – тип, контекст – набор гипотез вида (имя : тип)
  - Тактики меняют, добавляют, удаляют цели и гипотезы
    - Но behind the scenes при этом просто постепенно строится терм
- Тактики можно комбинировать
  - Увидим

Основные тактики

Тактика intros

«Доказать X →Y→Z» ➔«Доказать Z в предположении X, Y»

```
Theorem imp_trans : ∀ (A B C:Prop), (A -> B) -> (B -> C) -> (A -> C).
Proof.
    1 subgoal
    ______________________________________(1/1)
    ∀ A B C : Prop, (A -> B) -> (B -> C) -> A -> C
   intros A B C ab bc a.
    1 subgoal
    A : Prop
    B : Prop
    C : Prop
    ab : A -> B
    bc : B -> C
    a : A
    ______________________________________(1/1)
    C
```

Основные тактики

Тактика apply

«Свести доказательство X к доказательству Y при помощи доказательства X→Y»

```
Theorem imp_trans : ∀ (A B C:Prop), (A -> B) -> (B -> C) -> (A -> C).
Proof.
  intros A B C ab bc a.
…
ab : A -> B
bc : B -> C
a : A
______________________________________(1/1)
C
    apply bc
…
______________________________________(1/1)
B
    apply ab
…
______________________________________(1/1)
A
```

Основные тактики

Тактика assumption

«X уже доказано»

```
…
____________________________________(1/1)
A
    assumption.
Proof completed.
Qed.
```

Тактика exact

«Вот доказательство»

```
ab : A -> B
bc : B -> C
a : A
____________________________________(1/1)
C
    exact (bc (ab a)).
Proof completed.
Qed.
```

Основные тактики

Тактики intros, apply, assumption соответствуют трем правилам вывода типов

- **intros** – лямбда-абстракция
- **apply** – лямбда-аппликация
  - В позиции функции может быть любой терм
- **assumption** – поиск в контексте
- **exact** T ~ apply T, если это конец доказательства

- С их помощью можно доказать любую теорему
  - т.к. дерево вывода типов доказательства можно превратить в последовательность intros/apply/assumption
  - Но не стоит.

Основные тактики

- Diamond lemma

```
Variables P Q R T:Prop.
Lemma diamond : (P -> Q) -> (P -> R) -> (Q -> R -> T) -> P -> T.
Proof.
  intros pq pr qrt p.
pq : P -> Q
pr : P -> R
qrt : Q -> R -> T
p : P
______________________________________(1/1)
T
  apply qrt.
______________________________________(1/2)
Q

______________________________________(2/2)
R
  apply pq.
______________________________________(1/2)
P
  assumption.
______________________________________(1/1)
R
  apply pr.
______________________________________(1/1)
P
  assumption.
Proof completed.
 Qed.
```

Основные тактики

- auto
  - Поиск в глубину (`auto 12`) по `apply` (только контекст),`intros`,`assumption`
  - И указанным гипотезам (`auto with arith`…)

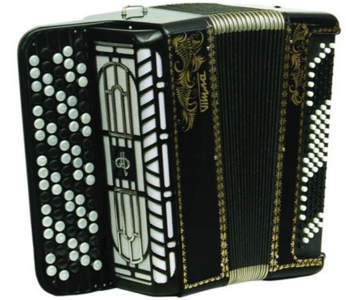
```
Variables P Q R T:Prop.
Lemma diamond : (P -> Q) -> (P -> R) -> (Q -> R -> T) -> P -> T.
Proof.
  auto.
Qed.
```

Комбинаторы

- Комбинатор `;`

```
Theorem then_example : P -> Q -> (P -> Q -> R) -> R.
 Proof.
   intros p q pqr.
   apply pqr; assumption. (* 2 цели, обе можно решить одним
  assumption*)
 Qed.
```

Комбинаторы



- Комбинатор [| | | |]
  - Пусть tac порождает несколько целей
  - `tac; [ on_first_goal | on_second_goal | …]`

```
Theorem compose_example : (P -> Q -> R) -> (P -> Q) -> (P -> R).
  Proof.
    intros pqr pq p.
    apply pqr; (* Две цели: P и Q *)
      [assumption | apply pq; assumption].
  Qed.
```

Комбинаторы

- Комбинатор ||
  - tac || if_tac_fails_try_this_one

```
Theorem orelse_example : (P -> Q) -> R -> ((P -> Q) -> R -> (Q -> R) -> T)
    -> T.
Proof.
  intros pq r pqrqrt.
  apply pqrqrt; (* 3 Цели: P->Q, R, Q->R *)
    (assumption || intro h1; assumption).
Qed.
```

Комбинаторы

- Комбинатор repeat
  - `repeat tac`
  - Повторять, пока возможно

```
Check plus_assoc.
    plus_assoc : ∀ n m p : nat, n + (m + p) = n + m + p

Theorem plus_stuff : ∀ (a b c d:nat), (a+b)+(c+(a+d)) = a+((b+c)+(a+d)).
  intros; repeat rewrite plus_assoc; apply refl_equal.
Qed.
```

Комбинаторы

- Тактика idtac
  - Оставить как есть

```
Lemma L3 : (P -> Q) -> (P -> R) -> (P -> Q -> R -> T) -> P -> T.
  intros pq pr pqrt p.
  apply pqrt; (* 3 цели: P, Q, R *)
    [idtac | apply pq | apply pr]; assumption.
Qed.
```

Комбинаторы

- Тактика fail
  - «Если уж досюда добрались, то дело плохо»
  - `some_tac ; fail` – «попробовать завершить доказательство с помощью some_tac»
  - Prolog: `!, fail`.

Вспомогательные тактики

- `cut T`
  - «Докажем в предположении T, а там уж и T докажем»
- `assert T`
  - «T – верно! Сейчас докажу, а там и до исходной цели недалеко»
- `refine` *`Term`*
  - «Доказательство примерно такое, надо только понять, что *вот тут* и *вон там*»
- `change` *`Term`*
  - «По сути-то, это то же самое, что и вот *это*»
  - Термы должны быть конвертируемы (иметь одинаковую нормальную форму – а она у обоих, слава богу, всегда есть
- `pattern` *`Term`* `at` *`[+/-]occurenceNo`*
  - change,pattern - полезность увидим попозже

Вспомогательные тактики

- cut

```
Theorem cut_example : ∀ (P Q R T:Prop),
  (P->Q) -> (Q->R) -> ((P->R) -> T -> Q) -> ((P->R) -> T) -> Q.
Proof.
  intros P Q R T pq qr prtq prt.
  cut (P -> R). (* 2 цели: (P->R)->Q и P->R *)
  intros pr; apply prtq; (* P->R , T *) [exact pr | apply prt; exact pr]. (* (P->R)->Q *)
  intros p; apply qr; apply pq; assumption.                                 (* P->R      *)
Qed.
```

Вспомогательные тактики

- assert

```
Theorem cut_example : ∀ (P Q R T:Prop),
  (P->Q) -> (Q->R) -> ((P->R) -> T -> Q) -> ((P->R) -> T) -> Q.
Proof.
  intros P Q R T pq qr prtq prt.
  cut (P -> R). (* 2 цели: P->R и (P->R)->Q *)
  intros p; apply qr; apply pq; assumption.                        (* P->R     *)
  intros pr; apply prtq; (* P->R , T *) [exact pr | apply prt; exact pr]. (* (P->R)->Q *)
Qed.
```

Вспомогательные тактики

- refine

```
Require Import Arith.

Check le_trans.
    le_trans : ∀ n m p : nat, n <= m -> m <= p -> n <= p

Check mult_le_compat_l.
    mult_le_compat_l : ∀ n m p : nat, n <= m -> p * n <= p * m

Check mult_le_compat_r.
    mult_le_compat_r : ∀ n m p : nat, n <= m -> n * p <= m * p

Theorem le_mult_mult : ∀ a b c d:nat, a <= c -> b <= d -> a*b <= c*d.
Proof.
  intros a b c d ac bd.
  refine (le_trans _ _ _ (_:a*b<=a*d) (_:a*d<=c*d));
    [apply mult_le_compat_l | apply mult_le_compat_r]; assumption.
         (* b <= d *)                (* a <= c *)
Qed.
```

Очень полезно для разработки *функций*, особенно в случае зависимых типов

Пропозициональная логика

Операторы and, or, not

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

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

> Print not.
    not = fun A : Prop => A -> False
         : Prop -> Prop

Theorem conj3 : ∀ P Q R:Prop, P -> Q -> R -> P/\Q/\R.
Proof (fun _ _ _ p q r => conj p (conj q r)).

Theorem disj4 : ∀ P Q R S:Prop, R -> P\/Q\/R\/S.
Proof (fun P Q R S r => or_intror _ (or_intror _ (or_introl _ r))).
```

Пропозициональная логика

«Индукция» по and, or

```
Check and_ind.
    and_ind : ∀ A B P : Prop, (A -> B -> P) -> A /\ B -> P
Theorem proj1 : ∀ A B:Prop, A/\B -> A.
Proof (fun A B ab => and_ind (fun a b => a) ab).
```

```
Check or_ind.
    or_ind  : ∀ A B P : Prop, (A -> P) -> (B -> P) -> A \/ B -> P

Theorem diamond_or : ∀ (P Q R S:Prop), (P->R) -> (Q->R) -> (S -> P\/Q) -> S -> R.
Proof.
  intros P Q R S pr qr spq s.
  refine (or_ind _ _ (spq s)); assumption.
Qed.
```

Пропозициональная логика

Тактики для and, or

- split
  - Не только and – любой индуктивный тип с 1 конструктором

```
Theorem conj3 : ∀ P Q R:Prop, P -> Q -> R -> P/\Q/\R.
Proof.
  intros; split; [assumption | split; assumption].
Qed.
```

- left, right
  - Не только or – любой индуктивный тип с 2 конструкторами

```
Theorem disj4 : ∀ P Q R S:Prop, R -> P\/Q\/R\/S.
Proof.
  intros; right; right; left; assumption.
Qed.
```

Автоматические тактики

- trivial = auto 0
  - `a+b = a+b / assumption / …` (очень простые вещи)
- auto
- tauto – Пропозициональные тавтологии

```
Theorem tauto_ex :
    ∀ A B C D:Prop, (A->B)\/(A->C)->A->(B->D)->(C->D)->D.
  tauto.
Qed.
```

Автоматические тактики

- intuition tac
  - Использует дерево решений от tauto
  - В листьях применяет tac
  - Мощная штука
  - Еще увидим

```
Theorem intuition_ex : ∀ n p q:nat, n<=p \/ n<=q -> n<=p \/ n<=S q.
  intros n p q; intuition auto with arith.
Qed.
```

Экзистенциальные тактики

- eapply, eassumption, eauto
- Используют унификацию вместо matching

```
Theorem le_mult_mult' : ∀ a b c d:nat, a <= c -> b <= d -> a*b <= c*d.
Proof.
  intros a b c d ac bd.
    (* le_trans : n <= m -> m <= p -> n <= p *)
    (* "n <= p" ~= "a*b <= c*d" ➔
    {n := a*b, p := c*d, m := ?1} *)
  eapply le_trans.  (* a * b <= ?1 | ?1 <= c * d      *)

    (* "P*X <= Q*X" ~= "a*b <= ?1" ➔
        {P := a, X := b, Q := ?2, ?1 := ?2*b} *)
  eapply mult_le_compat_r.    (* Сводим к P <= Q : a <= ?2      *)
    (* a <= ?2       | ?2*b <= c*d      *)

    (* ??? ~= "a <= ?2"
        a <= c подходит ➔ {?2 := c} *)
  eassumption.      (* c*b <= c*d                  *)

  apply mult_le_compat_l.
  assumption.
Qed.
```

Сводим к a*b <= c*b <= c*d

унификация

подстановка

результат

Равенства

```
eq_ind
     : ∀ (A : Type) (x : A) (P : A -> Prop),
      P x -> ∀ y : A, x = y -> P y
```

“Если двое равны, то любой предикат на них верен одновременно”

```
Theorem rewrite_ex1 : ∀ (a b:nat), a=b -> a+1=b+1.
  intros a b H.
```

Возьмем предикат «P(y) = {a+1 = y+1}».

Он верен для a, а значит, раз a=b, то верен и для b.

```
  apply (eq_ind a (fun y => a+1=y+1) (refl_equal (a+1)) b H).
Qed.
```

Равенства

- То же самое делает тактика rewrite

```
Theorem rewrite_ex1 : ∀ (a b:nat), a=b -> a+1=b+1.
  intros a b H; rewrite H; apply refl_equal.
Qed.


Check plus_assoc.
     plus_assoc : ∀ n m p : nat, n + (m + p) = (n + m) + p

Theorem plus_stuff : ∀ (a b c d:nat), (a+b)+(c+(a+d)) = a+((b+c)+(a+d)).
  intros; repeat rewrite plus_assoc; apply refl_equal.
Qed.

Theorem plus_permute_23 : ∀ n m p:nat, n+m+p = n+p+m.
Proof.
  intros n m p.
  repeat rewrite <- plus_assoc. (* n + (m + p) = n + (p + m) *)
  rewrite (plus_comm m p).      (* plus_comm m p : m+p = p+m *)
  trivial.
Qed.
```

rewrite в обратную сторону

аргумент rewrite – любой терм, кончающийся на _=_

Противоречия и отрицание

- `not P = ~P = (P → False)`
- **Ложь доказывает все, что угодно.**
- `apply False_ind,` **then prove** `False.`

```
Check False_ind.
False_ind
     : ∀ P : Prop, False -> P

Theorem absurd1 : False -> 1 = 2.
  intros; apply False_ind; assumption.
Qed.

Theorem absurd2 : ∀ (P Q:Prop), P -> ~P -> Q.
  intros P Q p np.  (* Q *)
  apply False_ind.  (* False *)
  apply np.         (* P *)
  assumption.
Qed.

Theorem double_neg : ∀ P:Prop, P -> ~~P. (* P -> (P->False) -> False *)
Proof.
  intros P p np.
  apply np.
  assumption.
Qed.
```

Не путать с ~~P → P! Двойного отрицания нет!

Противоречия и отрицание

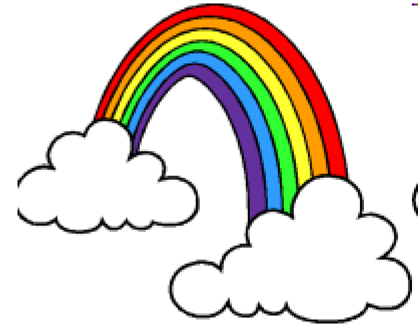
- Доказательство «от противного» все-таки есть

```
Theorem contrap : ∀ A B:Prop, (A->B) -> ~B -> ~A.
Proof.
  intros A B.
  unfold not.
  apply imp_trans.
Qed.
```

- Типичная ошибка в рассуждениях: коммутативность импликации

```
Theorem dyslexic_imp_false : (∀ P Q:Prop, (P->Q) -> (Q->P)) -> False.
Proof.
  intro dyslexic_imp.
  apply (dyslexic_imp False True);     (* False -> True | True *)
    [apply False_ind | trivial].
Qed.
```

Разбор случаев



- case, destruct
  - Пусть Q – индуктивного типа T
  - Надо доказать …Q… → (что-то)
  - Q мог быть создан любым из конструкторов T (из чего-то)
  - Доказать надо для всех случаев, т.е. конструкторов

```
Inductive rainbow : Set :=
  | red | orange | yellow | green | lightblue | blue |
   violet.

Definition next_color (c:rainbow) : rainbow :=
  match c with | … end.
```

Разбор случаев

Кстати, у нас есть rainbow_rec!

```
rainbow_rec : ∀ P : rainbow -> Set,
        P red -> P orange -> … -> P violet ->
        ∀ r : rainbow, P r

Definition next_color : rainbow->rainbow :=
  rainbow_rec (fun _ => rainbow) orange yellow … red.
```

Кстати, у нас есть и rainbow_ind!

```
rainbow_ind : ∀ P : rainbow -> Prop,
        P red -> P orange -> … -> P violet ->
        ∀ r : rainbow, P r
```

Действительно, доказательство по индукции ничем не отличается от вычисления по индукции – просто вычисляется *высказывание*.

Разбор случаев

```
Theorem next_green_then_red : ∀ (c:rainbow), next_color c = green -> c = yellow.
  intros c H.
  case c.
      7 subgoals
      c : rainbow
      H : next_color c = green
      ______________________________________(1/7)
      red = yellow


      ______________________________________(2/7)
      orange = yellow

      ______________________________________(3/7)
      yellow = yellow

      ______________________________________(4/7)
      green = yellow

      ______________________________________(5/7)
      lightblue = yellow

      ______________________________________(6/7)
      blue = yellow

      ______________________________________(7/7)
      violet = yellow
```

Whoops. case стирает информацию.

Разбор случаев

```
Theorem next_green_then_red : ∀ (c:rainbow), next_color c = green -> c = yellow.
  intros c. H.
  case c.
      7 subgoals
      c : rainbow
      ______________________________________(1/7)
      next_color red = green -> red = yellow

      ______________________________________(2/7)
      next_color orange = green -> orange = yellow
      ______________________________________(3/7)
      next_color yellow = green -> yellow = yellow
      ______________________________________(4/7)
      next_color green = green -> green = yellow
      ______________________________________(5/7)
      next_color lightblue = green -> lightblue = yellow
      ______________________________________(6/7)
      next_color blue = green -> blue = yellow
      ______________________________________(7/7)
      next_color violet = green -> violet = yellow
```

Этот - trivial

Эти – абсурдны, но это придется доказать

Различимость конструкторов

- Если у термов разные конструкторы – они не равны

```
Theorem next_red_green_then_whatever :
   next_color red = green -> red = yellow.
  simpl. (* orange = green -> red = yellow *)
  intros H; apply False_ind. (* Seriously, wtf? H:orange=green |- False
   *)
```

Заменим на эквивалентную цель (т.е. *конвертируемую* с False),
в которой будет проще убедиться с помощью “orange = green”

```
  change ((fun c:rainbow => match c with | orange => True | _ => False
   end)
          green).
```

Ну, если уж orange и green – одно и то же…

```
  rewrite <- H. (* green превращается в orange, а весь терм в True *)
  trivial.
Qed.
```

Разбор случаев

Booleans

```
Print bool.
    Inductive bool : Set :=  true : bool | false : bool
Check bool_rec.
    bool_rec
         : ∀ P : bool -> Set, P true -> P false -> ∀ b : bool, P b
```

- *Совсем* не то же самое, что Prop
- Просто тип с двумя конструкторами
- Есть двойное «отрицание» и т.п.

Разбор случаев

Booleans

```
Print bool.
    Inductive bool : Set :=  true : bool | false : bool
Check bool_rec.
    bool_rec
          : ∀ P : bool -> Set, P true -> P false -> ∀ b : bool, P b
Definition bool_not := bool_rec (fun _ => bool) false true.
Definition bool_xor (a:bool) (b:bool) : bool :=
  bool_rec (fun _ => bool) (bool_not b) b a.
Definition bool_or (a:bool) (b:bool) : bool :=
  bool_rec (fun _ => bool) true b a.
Definition bool_and (a:bool) (b:bool) : bool :=
  bool_rec (fun _ => bool) b false a.
Definition bool_eq (a:bool) (b:bool) : bool :=
  bool_rec (fun _ => bool) b (bool_not b) a.
```

Разбор случаев

Booleans

```
Theorem not_not : ∀ b:bool, bool_not (bool_not b) = b.
Proof.
  intro b; case b; simpl; trivial.
Qed.

Theorem not_or : ∀ (b1 b2:bool), bool_not (bool_or b1 b2) = bool_and (bool_not b1) (bool_not b2).
Proof.
  intros b1 b2.
  case b1; case b2; simpl; trivial.
Qed.

Theorem not_and : ∀ (b1 b2:bool), bool_not (bool_and b1 b2) = bool_or (bool_not b1) (bool_not b2).
Proof.
  intros b1 b2.
  case b1; case b2; simpl; trivial.
Qed.
```

Скучно.

Позже с помощью Ltac построим мощную тактику.

Теоремы о существовании

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : ∀ x : A, P x -> ex A P
```

Нотация: `exists x:A, P x.`

Чтобы *конструктивно* доказать существование, надо предъявить терм, удовлетворяющий P.

```
Inductive le (n : nat) : nat -> Prop :=
    | le_n : n <= n
    | le_S : ∀ m : nat, n <= m -> n <= S m

Theorem pred_ex : ∀ n:nat, 1 <= n -> exists m:nat, S m = n.
Proof.
  intros n n_big.
  case n_big.
  (* exists m : nat, S m = 1 *)
  (* ∀ m : nat, 1 <= m -> exists m0 : nat, S m0 = S m *)
  refine (ex_intro _ 0 _); (* 1=1 *)
     trivial.
  intros m m_big; refine (ex_intro _ m _); (* S m = S m *)
     trivial.
Qed.
```

Теоремы о существовании

- То же самое делает тактика exists

```
Theorem pred_ex : ∀ n:nat, 1 <= n -> exists m:nat, S m = n.
Proof.
  intros n n_big.
  case n_big.
  exists 0; trivial.
  intros m m_big; exists m; trivial.
Qed.
```

Теоремы о существовании

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : ∀ x : A, P x -> ex A P
```

Если существование конструктивно доказано, значит, терм, удовлетворяющий `P x` (аргумент `ex_intro`) – *существует*, его можно достать из конструктора.

```
Definition extract (A:Type) (P:A->Prop) (h:exists x:A, P x) :=
  match h with | ex_intro x _ => x end.
```

```
Error:
Incorrect elimination of "h" in the inductive type "ex":
the return type has sort "Type" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Type
because proofs can be eliminated only to build proofs.
```

Pattern matching по терму сорта Prop нельзя делать при построении терма сорта Type/Set – иначе бы **нарушалось Proof Irrelevance** (результат бы зависел от способа доказательства).

Теоремы о существовании

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : ∀ x : A, P x -> ex A P
```

Pattern matching по терму сорта Prop нельзя делать при построении терма сорта Type/Set – иначе бы нарушалось Proof Irrelevance (результат бы зависел от способа доказательства).

А при построении терма сорта Prop – можно:

```
Theorem ex_imp_ex : ∀ (A:Set) (P Q:A -> Prop),
    (exists a:A, P a) -> (∀ a:A, P a->Q a) -> (exists a:A, Q a).
Proof.
  intros A P Q exa pq.
  destruct exa. (* Добавляются гипотезы x:A, H:P x *)
  exists x.     (* Q x *)
  apply pq; exact H.
Qed.

Print ex_imp_ex.
     ex_imp_ex = fun (A : Set) (P Q : A -> Prop) (exa : exists a : A, P a) (pq : ∀ a : A, P a -> Q a)
        =>
       match exa with
       | ex_intro x H => ex_intro (fun a : A => Q a) x (pq x H)
       end
```

Доказательства по индукции

- То же, что и просто рекурсивная функция
- Только вычисляется терм сорта Prop

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A -> list A -> list A

Fixpoint len (A:Type) (s:list A) {struct s} :=
  match s with
  | nil => 0
  | cons _ s' => S (len A s')
  end.

Fixpoint append (A:Type) (s r:list A) {struct s} :=
  match s with
  | nil => r
  | cons h s' => cons h (append A s' r)
  end.
```

Доказательства по индукции

Но уже есть готовые *рекурсоры*:

```
list_rec
     : ∀ (A : Type) (P : list A -> Set),
       P nil ->
       (∀ (a : A) (l : list A), P l -> P (a :: l)) ->
       ∀ l : list A, P l

Definition len' (A:Type) : list A -> nat :=
  list_rec (fun _ => nat) 0 (fun a s' len' => S len').

Definition append' (A:Type) : list A -> list A -> list A :=
  list_rect (fun _ => list A->list A)
           (fun r => r)
           (fun a _ append_s' => fun r => cons a (append_s' r)).


Implicit Arguments len [A].
Implicit Arguments append [A].
```

Но есть и готовый *индуктор*:

```
list_ind
     : ∀ (A : Type) (P : list A -> Prop),
       P nil ->
       (∀ (a : A) (l : list A), P l -> P (a :: l)) ->
       ∀ l : list A, P l
```

С его помощью можно сочинять рекурсивные доказательства.

Доказательства по индукции

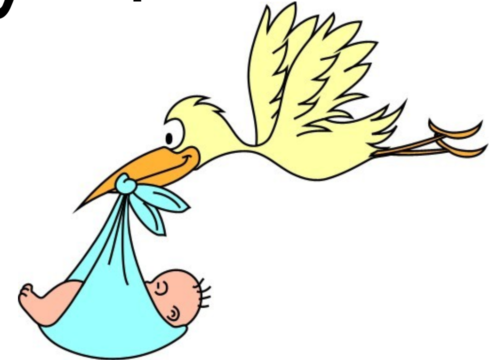
```
list_ind
     : ∀ (A : Type) (P : list A -> Prop),
       P nil ->
       (∀ (a : A) (l : list A), P l -> P (a :: l)) ->
       ∀ l : list A, P l

Definition append_len : ∀ (A:Type) (s r:list A),
        len (append s r) = len s + len r.
  intros A s r.
  refine (list_ind (fun s => len (append s r) = len s + len r)
                _ _ s).
  (* len (append nil r) = len nil + len r *)
  (* ∀ (a : A) (l : list A),
     len (append l r) = len l + len r ->
     len (append (a :: l) r) = len (a :: l) + len r*)
  simpl; trivial.
  intros a q IHs; simpl; rewrite IHs; trivial.
Qed.
```

Доказательства по индукции

- То же самое делает induction или elim
  - Отличаются не очень сильно

```
Theorem append_len' : ∀ (A:Type) (s r:list A),
   len (append s r) = len s + len r.
Proof.
  intros A s r; induction s.
  simpl; trivial.
  rewrite IHs; trivial.
Qed.
```

Откуда берется индукция?



```
list_ind
     : ∀ (A : Type) (P : list A -> Prop),
       P nil ->
       (∀ (a : A) (l : list A), P l -> P (a :: l)) ->
       ∀ l : list A, P l
```

Ничего особенного, обычная структурная рекурсия (катаморфизм), т.е. Fixpoint.

Сейчас сами напишем.

```
Fixpoint list_ind' (A:Type) (P:list A->Prop)
                   (HNil:P nil)
                   (HCons:∀ (a:A) (s:list A), P s -> P (cons a s))
                   (s:list A) : P s :=
  match s return (P s) with
  | nil => HNil
  | cons a s' => HCons a s' (list_ind' A P HNil HCons s')
  end.
```

Типы веток match должны быть одинаковы.
Тут это не так: `P nil` vs. `P (cons a s')`.
И то и другое – `P s`, но самостоятельно Coq до этого догадаться не может: это задача унификации 2го порядка, а она неразрешима.

Ltac

- Макро-тактики

```
Theorem le_5_25 : 5 <= 25.
   repeat (apply le_n || apply le_S). (* Муторно *)
Qed.

Ltac le_star := repeat (apply le_n || apply le_S).
```

Или так: (рекурсивная тактика)

```
Ltac le_star := apply le_n || (apply le_S; le_S_star).

Theorem le_5_25 : 5 <= 25.
   le_star.
Qed.
```

Ltac

- Макро-тактики на стероидах
- Pattern matching по контексту и по цели – Prolog-style (в контексте *ищется* подходящая гипотеза)
- Сочиним тактику для контрапозиции

  (∀ A B:Prop, (A->B) -> ~B -> ~A.)

```
Ltac contrapose H :=
  match goal with
  | [ th:(~_) |- (~_)] => intro H; apply th
  end.

Theorem contrapose_ex : ∀ x y:nat, ~x=y -> x <= y -> ~y<=x.
  intros x y x_neq_y x_le_y. (* ~ y<=x *)
  contrapose H.              (* ~(x=y) |- ~(y<=x) ➔ y<=x |- x=y *)
  auto with arith.           (* x<=y, y<=x |- x=y *)
Qed.
```

Ltac

- Макро-тактики на стероидах
- Тактика «Упростить все что можно»

```
Ltac simpl_all :=
  match goal with
  | [ g:_ |- _ ] => progress simpl in g; simpl_all
  | [ |- _ ] => progress simpl
  end.
```

Без `progress` зациклится, т.е. `simpl` всегда успешен.

Ltac

Booleans

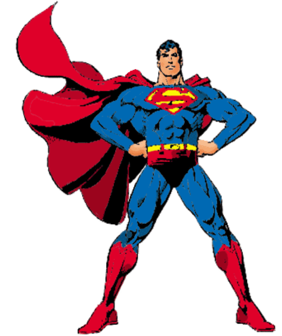
- Таблица истинности

```
Ltac truth_table :=
  intros;
  try simpl_all;
  match goal with
  | [ b:bool |- _ ] => destruct b; truth_table
  | [ |- _ ] => progress intuition; intuition truth_table
  end.
```

case не подходит, т.к. он оставит гипотезу в контексте, и мы опять найдем ее

Без этого тоже зациклимся, т.к. intuition всегда успешен

Ltac



Booleans

- Таблица истинности
- Я же говорил, что intuition - мощная штука

```
Theorem when_xor : ∀ a b:bool, bool_not a = b -> bool_xor a b = true.
  truth_table.
Qed.

Theorem when_xor_3 : ∀ a b c:bool, bool_xor a (bool_xor b c)=true -> a=false \/ b=c.
  truth_table.
Qed.
```

Индуктивные предикаты

- Бывают и посложнее, чем `le`

```
Inductive sorted (A:Set) (R:A->A->Prop): list A -> Prop :=
  | sorted0 : sorted A R nil
  | sorted1 : ∀ a:A, sorted A R (cons a nil)
  | sorted2 : ∀ (x y:A) (s:list A),
                R x y -> sorted A R (cons y s) -> sorted A R (cons x (cons y s)).

Inductive clos_trans (A:Type) (R:relation A) : A -> A -> Prop :=
  | t_step : ∀ x y:A, R x y -> clos_trans A R x y
  | t_trans : ∀ x y z:A, clos_trans A R x y -> clos_trans A R x z -> clos_trans A R y z.
```

Индуктивные предикаты

- Бывают и посложнее, чем `le`

```
Inductive sorted (A:Set) (R:A->A->Prop): list A -> Prop :=
  | sorted0 : sorted A R nil
  | sorted1 : ∀ a:A, sorted A R (cons a nil)
  | sorted2 : ∀ (x y:A) (s:list A),
                R x y -> sorted A R (cons y s) -> sorted A R (cons x (cons y s)).

Theorem le_O_n : forall n:nat, 0 <= n. intros; auto with arith. Qed.
Hint Resolve sorted0 sorted1 sorted2 : sorted_base.
Hint Resolve le_n le_S le_O_n : nat_le.

Theorem sorted_nat_123 : sorted le (1::(2::(3::nil))).
  auto with sorted_base nat_le.
Qed.

Theorem xy_ord : forall (x y:nat), le x y -> sorted le (x::(y::nil)).
  auto with sorted_base.
Qed.
```

Hint databases для auto

Это все легко.

Инверсия

```
Inductive sorted (A:Set) (R:A->A->Prop): list A -> Prop :=
  | sorted0 : sorted A R nil
  | sorted1 : ∀ a:A, sorted A R (cons a nil)
  | sorted2 : ∀ (x y:A) (s:list A),
                R x y -> sorted A R (cons y s) -> sorted A R (cons x (cons y s)).

Theorem sorted1_inv : ∀ (A:Set) (le:A->A->Prop) (x:A) (s:list A),
  sorted le (cons x s) -> sorted le s.
Proof.
  intros A le x s H.
     1 subgoal
     A : Set
     le : A -> A -> Prop
     x : A
     s : list A
     H : sorted le (x :: s)
     ________________________________________(1/1)
     sorted le s
```

H был образован одним из трех конструкторов. Должно сработать case/elim/destruct.

Инверсия

```
Theorem sorted1_inv : ∀ (A:Set) (le:A->A->Prop) (x:A) (s:list A),
  sorted le (cons x s) -> sorted le s.
Proof.
  intros A le x s H.
     1 subgoal
     A : Set
     le : A -> A -> Prop
     x : A
     s : list A
     H : sorted le (x :: s)
     ______________________________________(1/1)
     sorted le s
```

H был образован одним из трех конструкторов. Должно сработать case/elim/destruct.

```
  elim H.
     H : sorted le (x :: s)
     ______________________________________(1/3)
     sorted le s
```

Whoops. Ничего не изменилось.
case, induction, destruct – не лучше.

Инверсия

```
Theorem sorted1_inv : ∀ (A:Set) (le:A->A->Prop) (x:A) (s:list A),
  sorted le (cons x s) -> sorted le s.
Proof.
  intros A le x s H.
      1 subgoal
      A : Set
      le : A -> A -> Prop
      x : A
      s : list A
      H : sorted le (x :: s)
      ______________________________________(1/1)
      sorted le s
 inversion H.
```

```
| sorted0 : sorted A R nil
| sorted1 : ∀ a:A, sorted A R (a::nil)
| sorted2 : ∀ (x y:A) (s:list A),
    R x y ->
    sorted A R (y::s) ->
    sorted A R (x::(y::s)).
```

le ~= R, x::s ~= a::nil

```
      a : A
      H1 : a = x
      H2 : nil = s
      ______________________________________(1/2)
      sorted le nil

      auto with sorted_base.
```

le ~= R, x::s ~= x::y::s

```
      x0 : A
      y : A
      s0 : list A
      H2 : x <= y
      H3 : sorted le (y :: s0)
      H0 : x0 = x
      H1 : y :: s0 = s
      ______________________________________(2/2)
      sorted le (y :: s0)

      auto with sorted_base.
Qed.
```

Сертифицированные функции

```
Inductive pred_spec (n:nat) : Set :=
  | pred_data : ∀ (p:nat), (S p = n) -> pred_spec n.
```

Такое встречается очень часто. Есть специальный тип и нотация.

```
Inductive sig (A:Set) (P:A->Prop) : Set :=
  | exist : forall x:A, P x -> sig A P.
```

```
Inductive ex (A:Type) (P:A->Prop) : Prop :=
  | exist : forall x:A, P x -> sig A P.
```

Отличается от ex тем, что из sig *можно и нужно* достать аргумент.

Нотация: `{x:A | P x}`

Сертифицированные функции

```
Definition pred (n:nat) : (1 <= n) -> {p:nat | S p = n}.
  intros n n_big. (* {p:nat | S p = n} *)
  case_eq n.
  (* n = 0 -> {p:nat | S p = 0} *)
  (* forall n0 : nat, n = S n0 -> {p : nat | S p = S n0} *)
  intros H; apply False_rec. (* 1<=n и n=0 ?! *)
  rewrite H in n_big; inversion n_big. (* inversion видит, что 1<=0 не может быть
                                        результатом никакого конструктора le *)
  intros n0 H; exists n0; trivial.
Qed.
```

Сертифицированные функции

Decidability,
Или еще раз о разнице между Prop и bool

```
Inductive cmp : Set := less | equal | greater.

Definition compare_nats (a b:nat) : cmp :=
  match (a < b) with
   | True -> less
   | False -> match (a > b) with
              | True -> greater
              | False -> equal
              end
  end.
```

Ха!

- Нельзя делать pattern matching на доказательствах
- Даже если бы было можно: a<b верно – еще не значит, что a>b неверно
- Это отдельное понятие: decidability
- ∀ a b:nat, {a<b} + {a=b} + {a>b}
- А для разработки сертифицированных функций это очень важно

Сертифицированные функции

```
Inductive sumbool (A B:Prop) : Set :=
  | left : A -> sumbool A B | right : B -> sumbool A B
```

Нотация: `{A}+{B}`

```
test_even : ∀ n:nat, {even n} + {even (pred n)}

Z_le_gt_dec : ∀ x y:Z, {x <= y} + {x > y}

eq_dec (A:Type) := ∀ x y:A, {x=y} + {~(x=y)}
```

Сертифицированные функции

```
Definition nat_le_gt_dec : forall (a b:nat), {a<=b} + {a>b}.
  intros a.
  elim a.
  intros b; left; auto with arith.
  intros n Frec b'.
  elim b'.
  right; auto with arith.
  intros n0 Hrec.
    a : nat
    n : nat
    Frec : forall b : nat, {n <= b} + {n > b}
    b' : nat
    n0 : nat
    Hrec : {S n <= n0} + {S n > n0}
    ________________________________________(1/1)
    {S n <= S n0} + {S n > S n0}
  case (Frec n0); [left | right]; auto with arith.
Qed.
```

Сертифицированные функции

Частичные функции

- {Ответ} /\ {доказательство некорректности аргументов}

```
Inductive sumor (A:Set) (B:Prop) : Set :=
  | inleft : A -> sumor A B | inright : B -> sumor A B
```

Нотация: `A+{B}`

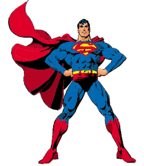
```
Definition pred' (n:nat) : {p:nat | S p=n} + {n=0}.
  intros n; case_eq n.
  (* n = 0 -> {p : nat | S p = 0} + {0 = 0} *)
  (* forall n0 : nat, n = S n0 -> {p : nat | S p = S n0} + {S n0 = 0} *)
  intros; right; trivial.
  intros n0 H; left; exists n0; auto with arith.
Qed.
```

Program extraction

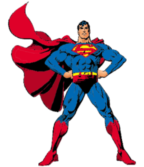
- Из функции можно сгенерить код на Haskell / ML / Scheme
- При этом термы типа Prop *стираются* (proof irrelevance)
- И получается просто гарантированно корректная функция
- См. Reference Manual

Арифметика на стероидах

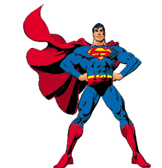
- ring, field, omega, fourier



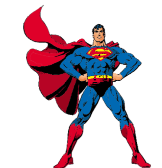
```
Theorem ring_ex : ∀ x y:Z, (x+y)*(x+y) = x*x + 2*x*y + y*y.
  intros; ring.
Qed.
```



```
Require Import Reals.
Theorem field_ex : forall x y:R, y <> 0 -> (x+y)/y = 1+(x/y).
  intros x y H; field.
  trivial.
Qed.
```



```
Require Import Omega.
Theorem omega_ex : ∀ x y z t:Z, x<=y<=z /\ z <= t <= x -> x = t.
  intros; omega.
Qed.
```



```
Require Import Fourier.
Theorem fourier_ex : forall x y:R, x-y>1 -> x-2*y<0 -> x>1.
  intros x y H1 H2; fourier.
Qed.
```

Описание семантики языков

- Индуктивный тип для синтаксиса языка

```
Inductive program : Set :=
  | Skip : program
  | Assign : lvalue -> rvalue -> program
  | Sequence : program -> program -> program
  | While : expr -> program -> program.
```

- Индуктивный предикат для семантики “s1 + program = s2”

```
Inductive exec : state -> program -> state -> Prop :=
| execSkip : ∀ (s:state), exec s Skip s.
| execSequence : ∀ (s1 s’ s2:state) (p1 p2:program),
   exec s1 p1 s’ -> exec s’ p2 s2 -> exec s1 (Sequence p1 p2) s2.
| …
```

Общая рекурсия

- Рекурсия по вспомогательному аргументу, который все время уменьшается
- Обобщение: «Вполне фундированные отношения» (well-founded relations)
  - Не существует бесконечной цепочки `x1 R x2 R x3 R` …
  - Тогда если шаг итерации соответствует R, то функция завершается
  - well_founded_induction
  - Отношение «x = C y», где C – конструктор индуктивного типа – вполне фундировано.
- Определение индуктивного предиката, описывающего область определения функции, и похожего на структуру ее тела
- …

Ссылки

- Официальный сайт
- Reference manual
- Тут и На amazon.com - книжка Coq’Art
- Официальный туториал
- Тут - большой туториал в форме FAQ
- Тут - перевод официального туториала на русский язык
- Тут - верификация программ, пример – двоичные деревья
- Тут - чей-то курс «Interactive theorem proving»
- Тут - Coq in a hurry, от самого Yves Bertot
- Тут - свежий несложный туториал
- Тут - теория в основе Coq
- Тут - про то, как верифицировали компилятор си
- Тут - реализация Haskell Prelude на Coq

Спасибо!

Вопросы?