МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА
Механико-математический факультет

Кафедра математической логики и теории алгоритмов

Практикум по математической логике. COQ.

В.Н. Крупский, С.Л. Кузнецов

2 сентября 2013 г.

Предисловие

Система Coq является программным комплексом, предназначенный для формализации и проверки правильности математических рассуждений. Она представляет собой логическую среду, позволяющую описывать математические теории и в интерактивном полуавтоматическом режиме строить доказательства (формальные выводы). В основе системы лежат интуиционистская логика и теория типов CiC (Calculus of Inductive Constructions), что позволяет строить конструктивные доказательства и извлекать из них соответствующие алгоритмы в виде верифицированных программ (поддерживаются языки функционального программирования OCaml, Haskell и Scheme). При этом правильность построенных доказательств проверяется автоматически посредством сведения к задаче проверки правильности типизации термов в системе CiC.

Настоящее руководство служит введением в практику использования системы Coq. Оно рассчитано на читателя, раннее не знакомого с системой, но все же имеющего некоторое представление о математической логике и теории доказательств, а также интерес к практическому использованию этих дисциплин. При этом предполагается, что параллельно с чтением текста читатель будет экспериментировать с системой, разбирая все примеры и решая задачи с помощью CoqIde (графический интерфейс для системы Coq). В этом же предположении изложены решения задач в главе «Решения». Там приведены лишь инструкции, описывающие процесс построения доказательств, а чтобы проследить сами доказательства их надо построить средствами Coq.

Наше описание системы Coq не является полным и демонстрирует лишь основные ее инструменты, необходимые для содержательного использования системы. Поэтому рекомендуется всегда иметь под рукой стандартную документацию системы [1] [2]. В качестве дополнительных источников можно рекомендовать начальное руководство[3] и материалы курса “CoursDeCoq”[4], примеры которого мы интенсивно использовали.

[1] Reference Manual. `http://coq.inria.fr/distrib/current/refman/`

[2] The Coq Standard Library. `http://coq.inria.fr/distrib/current/stdlib/`

[3] G. Huet, G Kahn, C. Paulin-Morning. The Coq Proof Assistant. A Tuturial. `http://coq.inria.fr/distrib/current/files/Tutorial.pdf`

[4] F. Prost, G. Kahn. CoursDeCoq. `http://coq.inria.fr/pylons/pylons/contribs/view/CoursDeCoq/v8.4`

Следует также упомянуть книгу “Coq’Art”[5], английский вариант которой издан издательством Springer, а французский доступен для свободного скачивания с сайта книги.

[5]Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions.
`http://www.labri.fr/perso/casteran/CoqArt/index.html`

Содержание

Предисловие **2**

1 Введение, основная методология **6**
1.1 Парадигма “Propositions as Types” 6
1.2 Доказательства 7
1.3 Тактики 9

2 Логические связки **13**
2.1 Конъюнкция 13
2.2 Дизъюнкция 14
2.3 Константы True и False, отрицание 15
2.4 Квантор существования 16
2.5 Классическая логика 16
2.6 Сечение 17
2.7 Оформление теорем 18
2.8 Задачи 18
2.9 О тактике `auto` 19

3 Описание теорий посредством введения констант, секции **20**
3.1 Декларации 20
3.2 Определения 21
3.3 Индуктивные определения 21
3.4 Секции 23

4 Пример: теория отношений **25**
4.1 Relation_1 25
4.2 Задачи 27
4.3 Использование равенства 29
4.4 Задачи 30
4.5 Доказательства индукцией по построению 31
4.6 Задачи 37

5 Основные средства функционального программирования (Gallina) **38**
5.1 Аппликация и λ-абстракция 38
5.2 Разбор случаев (case analysis) 40
5.3 Булева сумма 41

5.4 Рекурсия 43
5.5 Типы данных 45
5.6 Задачи 50

6 Решения **53**
Задачи из 2.8 53
Задачи из 4.2 54
Задачи из 4.4 55
Задачи из 4.6 56
Задачи из 5.6 58

7 Зачетные задания **61**

Добавление. Примеры использования библиотек **63**
Библиотека Arith 63
Библиотека ZArith 67

Добавление. Извлечение программ **76**

1 Введение, основная методология

1.1 Парадигма “Propositions as Types”

Имеется параллель между интуиционистской импликацией и квантором всеобщности, с одной стороны, и функциональным типом и зависимым произведением параметрического семейства типов, с другой:

Тип	Высказывание
$A \to B$ населен функциями f, которые при при каждом $x : A$ определены и принимают значение $f(x) : B$.	$A \to B$ верно, если имеется конструкция f, которая каждое доказательство x высказывания A преобразует в доказательство $f(x)$ высказывания B.
$\Pi_{x:T} B(x)$ населен функциями f, которые при при каждом $x : T$ определены и принимают значение $f(x) : B(x)$.	$\forall x : T\,,\, B(x)$ верно, если имеется конструкция f, которая каждое значение x типа T преобразует в доказательство $f(x)$ высказывания $B(x)$.

Аналогично сопоставляются:

Тип	Высказывание
$A \times B$	$A \wedge B$
$A \oplus B$	$A \vee B$
$\sum_{x:T} B(x)$	$\exists x : T, B(x)$
$void$	$\bot$
$void \to void$	$\top$

Это соответствие приводит к интерпретации интуиционистской логики в теории зависимых типов: высказывания интерпретируются типами (всеми или некоторыми), а предикаты — семействами типов, зависящими от параметров. Верность высказывания означает населенность (непустоту) соответствующего типа. Классическая логика эмулируется дополнительным контекстом, который обеспечивает непустоту типа, соответствующего закону исключенного третьего.

В теории типов CiC (Calculus of inductive constructions, T. Coquand, C. Paulin, 1989), лежащей в основе системы Coq, в качестве основного конструктора типов выбрано зависимое произведение (`forall x:T, B(x)`) . Остальные из перечисленных конструкторов типов определяются с помощью поддерживаемых системой индуктивных определений через него и "большой" тип `Prop` (сорт, т.е. тип типов, населен всеми высказываниями).

Сам тип `Prop` является объектом типа $\texttt{Type}_0$, который, в свою очередь, населяет тип $\texttt{Type}_1$, и т.д. Все члены последовательности $\texttt{Type}_0$, $\texttt{Type}_1$, ... маскируются одним символом `Type`, который автоматически интерпретируется системой как $\texttt{Type}_i$ с подходящим индексом i. Еще одним объектом типа $\texttt{Type}_0$ является "большой" тип `Set`— тип всевозможных предметных областей (называемых также спецификациями функциональных программ).

1.2 Доказательства

В рамках указанной парадигмы обоснование справедливости (верности) высказывания A означает предъявление объекта типа A. В теории типов объектами служат типизованные λ-термы. Именно они и играют роль доказательств соответствующих высказываний. Их также можно рассматривать как строчные записи деревьев вывода, доказывающих эти высказывания в формализме натурального вывода. Задача проверки правильности типизации термов для практически применяемых вариантов теории типов разрешима за разумное время, что позволяет автоматически верифицировать соответствующие доказательства.

Непосредственное выписывание подходящих λ-термов оказывается практически невозможным ввиду их размера и сложности. Система Coq (и другие аналогичные интерактивные системы построения доказательств) упрощают эту задачу, позволяя строить соответствующие термы интерактивно, с привлечением ряда алгоритмов поиска выводов, частично автоматизирующих процесс построения.

Формализм теории типов оперирует с секвенциями вида

$$x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1}) \vdash t(x_1, \ldots, x_n) : B(x_1, \ldots, x_n),$$

формализующими суждения о типизации

"если x_1 имеет тип A_1, x_2 имеет тип $A_2(x_1)$, ..., то $t(x_1, \ldots, x_n)$ имеет тип $B(x_1, \ldots, x_n)$".

Другое возможное прочтение:

"если x_1 доказывает A_1, x_2 доказывает $A_2(x_1)$, ..., то $t(x_1, \ldots, x_n)$ доказывает $B(x_1, \ldots, x_n)$".

Возможно и смешанное прочтение, когда одни члены секвенции говорят о типизации объектов, а другие — о доказательствах высказываний. Отметим, что в обоих случаях допускается зависимость типов и высказываний от объектов и доказательств, описанных левее.

Работа пользователя в системе Coq направлена на построение соответствующего терма $t(x_1, \ldots, x_n)$. Доступная ему информация описывается секвенцией

$$x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1}) \vdash (???) : B(x_1, \ldots, x_n)$$

с метапеременной (???), значение которой надо определить. Это изображается таблицей

$$\frac{\begin{array}{l} x_1 : A_1 \\ x_2 : A_2(x_1) \\ \vdots \\ x_n : A_n(x_1, \ldots, x_{n-1}) \end{array}}{B(x_1, \ldots, x_n)}$$

без метапеременной (???), что позволяет (несколько упрощенно, затеняя суть дела) считать x_i не доказательствами соответствующих высказываний, а их сокращенными обозначениями.

Например, задача поиска такого терма $t = t(A, B, H, H0)$, для которого секвенция

$$A : Prop,\ B : Prop,\ H : (A \to B),\ H0 : A \vdash t : B$$

выводима (в CiC), изображается таблицей

$$\frac{\begin{array}{l} \texttt{A : Prop} \\ \texttt{B : Prop} \\ \texttt{H : A->\ B} \\ \texttt{H0 : A} \end{array}}{\texttt{B}},$$

в которой первые две строчки — декларации переменных `A` и `B`, а следующие две — посылки `(A -> B)` и `A`, снабженные метками `H` и `H0`. Упрощенное прочтение задачи: вывести `B` из гипотез `H` и `H0`, в которых `A,B` имеют тип `Prop`. Фактически будет решаться первый вариант задачи и результатом будет $t =$`H H0` (результат применения функции `H` к аргументу `H0`). В то же время пользователь может думать, что решает упрощенный вариант, но в этом случае смысл ответа для него останется загадкой.

Обычно все строки таблицы, кроме последней, называют посылками, а последнюю — заключением. Фактически посылки — это члены левой части соответствующей секвенции, а заключение — ее правая часть без неизвестного терма t.

1.3 Тактики

Один шаг интерактивного построения доказательства состоит в том, что пользователь выбирает одну из имеющихся тактик, а система ее применяет. Многие тактики для своего применения требуют дополнительные параметры, которые также указываются пользователем.

Тактика представляет собой сведение решаемой задачи построения терма t, для которого секвенция (цель) $\Gamma \vdash t : A$ выводима (в CiC), к аналогичным задачам для некоторых других секвенций (подцелей) $\Gamma_1 \vdash t_1 : A_1,\ \ldots, \Gamma_k \vdash t_k : A_k,\ k \geq 0$. Тактики соответствуют допустимым правилам исчисления CiC, а также содержат алгоритмы построения искомого терма t по термам $t_1, \ldots, t_k$. При применении тактики изображающая задачу таблица (без t)

$$\frac{\Gamma}{A}$$

заменяется на несколько таблиц, изображающих подцели:

$$\frac{\Gamma_1}{A_1} \quad \ldots \quad \frac{\Gamma_k}{A_k}\ .$$

Аналогичные шаги применяются к подцелям и т.д., пока количество подцелей не сократится до 0. Система запоминает последовательность примененных тактик и восстанавливает искомый терм t с помощью соответствующих алгоритмов. При этом система постоянно контролирует

правильность типизации всех термов, что исключает возможность получения ошибочных доказательств.

Пример. Тактика `assumption` (команда "`assumption.`") позволяет завершить доказательство:

$$\frac{\Gamma \quad \texttt{H : A}}{\texttt{A}} \quad \longmapsto \quad \texttt{Proof completed}$$

Пример. Рассмотрим более сложные тактики `intro` и `apply`, связанные с зависимым произведением. Они обеспечивают разбор квантора всеобщности и импликации в соответствии с секвенциальными правилами введения этих связок справа и слева от знака $\vdash$.

Команда "`intro.`" преобразует таблицы следующим образом:

$$\frac{\Gamma}{\texttt{forall x:T, A(x)}} \quad \longmapsto \quad \frac{\Gamma \quad \texttt{x : T}}{\texttt{A(x)}}$$

$$\frac{\Gamma}{\texttt{A -> B}} \quad \longmapsto \quad \frac{\Gamma \quad \texttt{H : A}}{\texttt{B}}$$

Искомый терм t получается из терма t_1 для подцели с помощью лямбда абстракции: $t = \lambda x : T.\, t_1$ и $t = \lambda H : A.\, t_1$ соответственно. В синтаксисе системы Coq они записываются так: $t = (\texttt{fun x : T =>}t_1)$ и $t = (\texttt{fun H : T =>}t_1)$.

Выбор идентификатора (`H`) осуществляет система, но пользователь может его изменить, указав желаемое в виде параметра: "`intro X.`", что приведет к замене `H` на `X`. Вариант "`intros.`" означает многократное применение тактики `intro` пока это возможно.

Тактика `apply` требует дополнительный параметр — терм, как правило, функционального типа. В простейших случаях "apply H." работает следующим образом:

$$\frac{\Gamma \quad \texttt{H : A-> B}}{\texttt{B}} \quad \longmapsto \quad \frac{\Gamma \quad \texttt{H : A-> B}}{\texttt{A}}$$

$$\frac{\Gamma}{\texttt{H : A}\text{--}\texttt{> B}\text{--}\texttt{> C}} \quad\longmapsto\quad \frac{\Gamma}{\texttt{H : A}\text{--}\texttt{> B}\text{--}\texttt{> C}} \quad , \quad \frac{\Gamma}{\texttt{H : A}\text{--}\texttt{> B}\text{--}\texttt{> C}}$$
$$\texttt{C} \qquad\qquad\qquad\qquad \texttt{A} \qquad\qquad\qquad\qquad \texttt{B}$$

$$\frac{\texttt{P : nat}\text{--}\texttt{> Prop}\quad \texttt{H : (forall n : nat, P n)}}{\texttt{P 35}} \quad\longmapsto\quad \texttt{Proof completed}$$

В первом случае для нахождения искомого терма t типа `B` тактике не хватает данных. Терм t имеет вид `H` t_1 (функция `H`, примененная к аргументу t_1), где t_1 — неизвестный пока терм типа `A`. Для его нахождения тактика формулирует подзадачу, которую еще предстоит решить. Во втором случае $t =$ `H` t_1 t_2, и тактика порождает две подзадачи нахождения неизвестных термов t_1, t_2. Третий случай не порождает подзадач. Здесь $t =$ `H` n. Тактика унифицирует термы `(P` n`)` и `(P 35)` и находит терм $n =$ `35`, в результате чего полностью определяется искомый терм $t =$ `H 35`.

В более сложных случаях тактика `apply` комбинирует эти три варианта поведения, однако иногда ей требуется подсказать значения тех параметров, которые она не в состоянии определит самостоятельно. Например, пусть требуется использовать транзитивность предиката $P(x, y)$ для того, чтобы вывести $P(1, 3)$ из $P(1, 2)$ и $P(2, 3)$:

```
P: nat -> nat -> Prop
H: forall x y z : nat, P x y -> P y z -> P x z
H0: P 1 2
H1: P 2 3

------------------------------------
P 1 3
```

Команда “`apply H.`” здесь вызовет сообщение об ошибке, объясняющее, что значение `y` определить не удалось. В самом деле, унификация `P x z` с `P 1 3` позволяет найти `x=1` и `z=3`, но значение `y` остается неизвестным. Правильно сработает команда “`apply H with 2.`”, подсказывающая `y=2`.

Имеются также близкие к `apply` тактики `eapply` и `lapply`.

Пример. Докажем простую теорему:

```
Theorem mp: forall A B :Prop, A->(A->B)->B.
Proof.
intros.
apply H0.
assumption.
Qed.
```

Процесс построения доказательства по существу является интерактивным определением терма `mp`, тип которого совпадает с формулировкой теоремы. Вот протокол этого процесса, сгенерированный командой "`Show Tree`":

```
  ============================
   forall A B : Prop, A -> (A -> B) -> B

 BY intros
    A : Prop
    B : Prop
    H : A
    H0 : A -> B
    ============================
     B

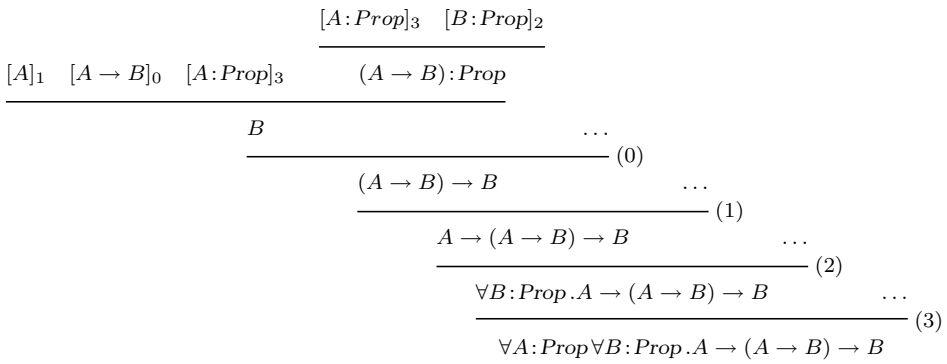
    BY apply H0
      ============================
       A
      BY assumption
```

Сам результирующий терм можно распечатать командой "`Print mp`" :

```
mp = fun (A B : Prop) (H : A) (H0 : A -> B) => H0 H
     : forall A B : Prop, A -> (A -> B) -> B
```

Замечание. В более привычной для математиков λ-нотации этот терм записывается так: $\lambda AB\colon Prop\ \lambda H\colon A\ \lambda H0\colon (A \to B)\,.\,(H0 \cdot H)$. Он обозначает функцию от четырех аргументов $A, B, H, H0$, возвращающую в качестве своего значения результат применения $H0$ к H. Первые два аргумента не входят в значение явно, но используются при его вычислении для проверки согласованности типов двух последних аргументов.

Для полноты приведем натуральный вывод[6], записью которого является терм `mp`.



2 Логические связки

В системе CiC импликация (функциональный тип) оказывается частным случаем интуиционистского квантора всеобщности (зависимого произведения), когда выражение под квантором не содержит вхождений связываемой квантором переменной:

$$(\texttt{A} \texttt{->} \texttt{B}) = (\texttt{forall x : A, B}), \qquad \texttt{x} \notin FV(\texttt{B}).$$

Остальные связки определяются посредством индуктивных определений.

2.1 Конъюнкция

Индуктивное определение конструктора типов `and` (набирается /\ ) выглядит так:

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

[6] Для соответствующего исчисления натурального вывода контроль за синтаксисом также включен в систему правил вывода. Этим объясняется наличие двух дополнительных посылок $A:Prop$ и $(A \to B):Prop$ в правиле modus ponens, обеспечивающих правильную построенность выражений A и $(A \to B)$. Аналогичные дополнительные посылки в других использованных правилах скрыты многоточием. В зависимости от деталей формулировки исчисления эти посылки в некоторых местах удается опускать.

Первая строка указывает тип константы `and` (аргументы `(A B : Prop)` и значение типа `Prop`). После "`:=`" задается конструктор (`conj`), который будет считаться правильным (и в данном случае единственно возможным) способом построения доказательства конъюнкции двух высказываний по доказательствам конъюнктивных членов. Тем самым, тип $\mathtt{A} \wedge \mathtt{B}$ населяется всевозможными термами вида `(conj u v)`, где `u:A`, `v:B`. При этом система автоматически добавляет в контекст ряд принципов индукции, утверждающих, что других объектов типа $\mathtt{A} \wedge \mathtt{B}$ нет (либо они неотличимы от указанных).

Основной тактикой для доказательства конъюнкции является `split` (команда "`split.`", которая на самом деле является сокращением для "`apply conj.`"). Она расщепляет цель на две подцели:

$$\frac{\Gamma}{\mathtt{A} \wedge \mathtt{B}} \quad \longmapsto \quad \frac{\Gamma}{\mathtt{A}} \quad , \quad \frac{\Gamma}{\mathtt{B}} \quad .$$

Для разбора любых индуктивных типов в посылке секвенции применяют тактику `elim` (команда "`elim H.`"). В случае конъюнкции она работает так:

$$\frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \wedge \mathtt{B}\end{array}}{\mathtt{C}} \quad \longmapsto \quad \frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \wedge \mathtt{B}\end{array}}{\mathtt{A} \mathrel{-\!\!>} \mathtt{B} \mathrel{-\!\!>} \mathtt{C}} \quad .$$

После нее разумно применить "`intros.`'' и получить

$$\frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \wedge \mathtt{B} \\ \mathtt{H0} : \mathtt{A} \\ \mathtt{H1} : \mathtt{B}\end{array}}{\mathtt{C}} \quad .$$

2.2 Дизъюнкция

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

Индуктивное определение связки `or` (набирается `\/`) содержит два конструктора доказательств дизъюнкции — `or_introl` и `or_intror`. Им

соответствуют две тактики: `left` и `right` (команды "`left .`" и "`right .`"), реализующие сведения:

$$\frac{\Gamma}{\mathtt{A} \vee \mathtt{B}} \quad \longmapsto \quad \frac{\Gamma}{\mathtt{A}} \qquad \text{и} \qquad \frac{\Gamma}{\mathtt{A} \vee \mathtt{B}} \quad \longmapsto \quad \frac{\Gamma}{\mathtt{B}}$$

Разбор дизъюнкции в посылке секвенции с помощью команды "`elim H.`" порождает две подцели:

$$\frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \vee \mathtt{B}\end{array}}{\mathtt{C}} \quad \longmapsto \quad \frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \vee \mathtt{B}\end{array}}{\mathtt{A} \to \mathtt{C}} \quad , \quad \frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \vee \mathtt{B}\end{array}}{\mathtt{B} \to \mathtt{C}}$$

К каждой из них разумно применить тактику `intro`. Это можно сделать сразу последовательностью команд "`elim H; intro.`", соединенных тактикалом ";" (управляющая конструкция языка тактик LTac). Тактикал ";" применяет вторую тактику ко всем подцелям, порожденным применением первой. Получится следующее сведение:

$$\frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \vee \mathtt{B}\end{array}}{\mathtt{C}} \quad \longmapsto \quad \frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \vee \mathtt{B} \\ \mathtt{H0} : \mathtt{A}\end{array}}{\mathtt{C}} \quad , \quad \frac{\begin{array}{l}\Gamma \\ \mathtt{H} : \mathtt{A} \vee \mathtt{B} \\ \mathtt{H0} : \mathtt{B}\end{array}}{\mathtt{C}}$$

2.3 Константы True и False, отрицание

```
Inductive False : Prop :=  .

Inductive True : Prop :=  I : True .

not = fun A : Prop => A -> False
     : Prop -> Prop
```

Как видно из приведенных определений, константы `False` и `True` являются индуктивными типами. Тип `False` пуст, а `True` содержит единственный элемент `I`. В то же время отрицание (набирается ~`A`) таковым не является, а определяется непосредственно через импликацию и `False`. Раскрыть это определение позволяет тактика `red` (команды "`red in |- *.`" и "`red in H.`").

Во многих случаях оказывается удобной тактика `contradict`. Команда "`contradict H.`" действует так:

$$\frac{\Gamma \quad \mathtt{H} : \sim \mathtt{A}}{\mathtt{B}} \longmapsto \frac{\Gamma}{\mathtt{A}} \qquad \frac{\Gamma \quad \mathtt{H} : \mathtt{A}}{\mathtt{B}} \longmapsto \frac{\Gamma}{\sim \mathtt{A}}$$

$$\frac{\Gamma \quad \mathtt{H} : \sim \mathtt{A}}{\sim \mathtt{B}} \longmapsto \frac{\Gamma \quad \mathtt{H} : \mathtt{B}}{\mathtt{A}} \qquad \frac{\Gamma \quad \mathtt{H} : \mathtt{A}}{\sim \mathtt{B}} \longmapsto \frac{\Gamma \quad \mathtt{H} : \mathtt{B}}{\sim \mathtt{A}}$$

2.4 Квантор существования

Интуиционистский квантор существования (набирается `exists x:A, P x`) также определяется индуктивно.

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

Единственный конструктор доказательств существования (`ex_intro`, строит объекты типа `(exists x:A, P x)`) требует в качестве параметров объект `a:A`, для которого верно `(P a)`, и доказательство того, что `(P a)` верно.

Основные тактики для работы с квантором существования следующие:

Команда "`exists a.`"

$$\frac{\Gamma}{\mathtt{exists\ x : A, P\ x}} \longmapsto \frac{\Gamma}{\mathtt{P\ a}}$$

Команда "`elim H.`"

$$\frac{\Gamma \quad \mathtt{H} : \ \mathtt{exists\ x : A, P\ x}}{\mathtt{B}} \longmapsto \frac{\Gamma \quad \mathtt{H} : \ \mathtt{exists\ x : A, P\ x}}{\mathtt{forall\ x : A,\ P\ x} \mathrel{-\!\!>} \mathtt{B}}$$

2.5 Классическая логика

Классическая логика получается добавлением дополнительного принципа — Закона исключенного третьего. Фактически мы переходим к рассуждениям внутри теории с дополнительной аксиомой

```
Axiom classic: forall P : Prop, P \/ ~P.
```

Тем самым к контексту добавляется новая константа `classic`, населяющая тип (`forall P : Prop, P` $\vee \sim$ `P`). Это можно сделать непосредственно, либо загрузить соответствующий стандартный модуль Classical командой

```
Require Import Classical.
```

Кроме декларации Закона исключенного третьего этот модуль содержит ряд его (доказанных) следствий, в частности,

```
Theorem NNPP: forall p : Prop, ~ ~ p -> p.
```

```
Theorem proof_irrelevance: forall (P : Prop)(p1 p2 : P), p1 = p2.
```

2.6 Сечение

Во многих случаях оказывается удобным использовать правило сечения:

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\,.$$

Оно реализовано в виде тактики `assert`, которой в качестве параметра надо передавать высекаемую формулу (команда "`assert A.`").

$$\frac{\Gamma}{\texttt{B}} \quad \longmapsto \quad \frac{\Gamma}{\texttt{A}} \quad , \quad \frac{\begin{array}{c}\Gamma \\ \texttt{H : A}\end{array}}{\texttt{B}}\,.$$

Близкий эффект достигается также командой "`cut A.`", реализующей правило Modus Ponens:

$$\frac{\Gamma}{\texttt{B}} \quad \longmapsto \quad \frac{\Gamma}{\texttt{A -> B}} \quad , \quad \frac{\Gamma}{\texttt{A}}\,.$$

2.7 Оформление теорем

Утверждения оформляются следующим образом (см. пример выше):

```
Theorem  <идентификатор>: < формулировка>.
Proof.
<доказательство>
Qed.
```

Вместо `Theorem` можно использовать `Lemma`, что никак не отразится на работе системы. Доказательство представляет собой последовательность команд, вызывающих подходящие тактики. Обычно в конце команды ставится точка. Знак “`;`”, который может появляться вместо точки между командами, обозначает тактикал, соединяющий две тактики в одну (вторая тактика применяется ко всем подцелям, порожденым первой).[7] Завершенное доказательство задает лямбда-терм t, тип которого совпадает с формулировкой теоремы. Команда “`Qed.`” определяет `<идентификатор>` как сокращение для t. В дальнейшем `<идентификатор>` можно использовать как любую другую константу указанного типа. Тем самым `Theorem` представляет собой вариант интерактивного определения констант[8]

2.8 Задачи

1. `Theorem ex1: forall A : Prop, A -> A.`
2. ```
   Theorem ex2:
   forall A B C : Prop, (A -> (B -> C)) -> ((A -> B) -> (A -> C)).
   ```
3. ```
   Theorem ex3:
   forall A B C D: Prop, (A -> C)/\(B -> D) -> A/\B -> C/\D.
   ```
4. `Theorem ex4: forall A : Prop, A -> ~~A.`
5. ```
   Require Import Classical.
   Theorem ex5: forall a : Prop, ~~ a -> a.
   ```

[7] Более сложная конструкция “τ`;[`τ_1 | ... | τ_n`]`” позволяет для каждой из порожденных подцелей задать свою тактику.

[8] Неинтерактивные определения также возможны. См. раздел 3.
   ```

6. `Theorem ex6:`
   `forall (p q : Type -> Prop) (a : Type),`
   `p a -> (forall x : Type, p x -> q x) -> q a.`

7. `Theorem ex7:`
   `forall (a b : Type) (p : Type -> Prop),`
   `p a \/ p b -> exists x : Type, p x.`

8. [9] `Theorem ex8:`
   `forall (A : Set) (R : A -> A -> Prop),`
   `(forall x y z : A, R x y /\ R y z -> R x z) ->`
   `(forall x y : A, R x y -> R y x) ->`
   `forall x : A, (exists y : A, R x y) -> R x x.`

2.9 О тактике `auto`

Тактика `auto` реализует автоматический поиск вывода (в стиле Пролога) посредством многократного применения тактик `assumption`, `intro` и `apply` , а также раскрытия известных ей определений. При этом если ей не удается построить вывод целиком, то она оставляет текущую цель неизменной. Тем самым, `auto` разумно применять лишь в качестве заключительной тактики. Не следует также рассчитывать, что она "догадается" разобрать одну из посылок: `auto` никогда не применяет тактику `elim`.

В качестве возможных аргументов для `apply` перебираются посылки текущей цели и леммы, хранящиеся в базе данных подсказок. По умолчанию подключена база подсказок `core`, но есть и другие стандартные базы (`arith`, `zarith`, `bool`, `datatypes`, `sets` и `v62`). Их можно подключить, указав явно (например, "`auto with arith, bool.`").[10]

Добавить свою подсказку в текущую базу данных можно так:

```
Theorem My: ... .
Proof. ... Qed.
Hint Resolve My.
```

[9] Для того, чтобы использовать транзитивность отношения `R`, тактике `apply` потребуется дополнительный аргумент. Значения `x` и `z` она найдет, унифицируя `R x z` с целью, а что подставлять вместо `y` — надо подсказать. Соответствующая команда "`apply H with t.`" подставляет терм `t` вместо `y`.

[10] База `v62` содержит наработки ранних версий Coq (до 6.2.4) и существует "для совместимости". Ее подключение заметно повышает результативность применения `auto`.

Формулировка подсказки (тип терма `My`) должна быть такой, чтобы команда "`apply My.`" была осмысленной (см. раздел 1.3). Запрос "`Print Hint *` " позволяет увидеть содержимое текущей базы данных подсказок.

3 Описание теорий посредством введения констант, секции

Описание прикладных теорий в системе Coq сводится к введению новых имен (констант). Это аппарат позволяет единообразно описывать как язык, так и аксиоматику теории. Для того, чтобы постулировать некоторое утверждение, достаточно формализовать его в виде типа `A:Prop`, после чего декларировать непустоту типа `A` введением новой (свободной) константы `c:A`. Механизм секций позволяет объединить основные определения и теоремы теории в единое целое.

3.1 Декларации

Ключевые слова `Variable` и `Hypothesis` (а также `Variables`, `Parameter`, `Parameters`, `Axiom`, `Conjecture`) являются почти синонимами. Они позволяют ввести в контекст новые константы (имена) и объявить их типы.

```
Variable M: Set.
Variable a b c: M.
Variable f: M -> M.
Variable P: M -> M -> Prop.

Hypothesis h: forall x: M , P x (f x).
```

Выбор варианта `Hypothesis` подсказывает пользователю, что введение имени (в нашем случае `h`) служит для того, чтобы объявить непустым его тип, содержательно понимаемый как высказывание $\forall x_{\in M}\, P(x, f(x))$, т.е. постулировать (конструктивную) истинность этого высказывания.

Некоторое различие между группами (`Variable`, `Variables`, `Parameter`, `Parameters`) и (`Hypothesis`, `Axiom`, `Conjecture`) проявляется лишь при использовании механизма секций (см. раздел 3.4).

3.2 Определения

Отметим, что введенные с помощью декларации имена являются свободными константами — им нельзя присвоить какие-нибудь значения, отличные от них самих. Для определения новых имен через уже имеющиеся служит конструкция `Definition`:

```
Definition UFun := M -> M.
Definition UPred := M -> Prop.
Definition Id (x : M) : M := x.
```

Последняя строка определяет `Id` как функцию аргумента `x:M` со значениями в типе `M`. Терм справа от "`:=`" (т.е. `x`) задает ее значение. С использованием лямбда-символики это определение можно переписать двумя способами[11]:

```
Definition Id' : UFun := fun x:M => x.     (* С указанием типа *)
Definition Id'' := fun x:M => x.            (* Без оного *)
```

Основная тактика для раскрытия определений — `unfold`. Пусть константа `C` введена посредством `Definition`. Команда "`unfold C.`" заменяет `C` на ее определение в заключении, а "`unfold C in H.`" — в посылке `H` (формально, в типе константы `H`). Добавление "`at i`" после `C` ограничивает действие тактики на i-тое вхождение `C` в соответствующий член секвенции.

В выражениях вида `forall ... , C T1 ... Tn` имеется возможность совместить раскрытие определения `C` с последующим упрощением (редукциями)[12]. Для этого служит тактика `red` (команды "`red.`" и "`red in H.`"). Она не требует указания имени `C`, которое извлекается из вида упрощаемого выражения автоматически.

После определения новой константы `C` разумно "научить" тактику `auto` пользоваться этим определением, добавив его к текущей базе данных подсказок командой "`Hint Unfold C.`".

3.3 Индуктивные определения

Более сложный вариант определений — индуктивное определение типа — вводится ключевым словом `Inductive`. В основе лежит семантика наи-

[11] Тактика `auto` умеет доказывать равенства `Id=Id'=Id''`.

[12] Непосредственно редуцировать выражение можно тактикой `compute`.

меньшей неподвижной точки. Задаются конструкторы объектов определяемого типа, а сам тип представляет собой наименьшее множество, замкнутое относительно применения конструкторов. Например:

```
Inductive nat : Set :=       (* Стандартные натуральные числа, *)
      O : nat                (* нуль - латинская буква O !!! *)
    | S : nat -> nat.

Inductive Li : Set :=        (* Списки натуральных чисел *)
     nu : Li
   | co: nat -> Li -> Li.

Inductive ListOf (A:Set):Set :=  (* Тип списков из элементов A,*)
     null : ListOf A                    (* тип A:Set - параметр *)
   | cons: A -> ListOf A -> ListOf A.
```

Стандартный для Coq'a тип натуральных чисел `nat` состоит из термов `O, S O, S (S O), ...` (обозначения 0,1,2,... также поддерживаются, но являются лишь сокращениями). Типичным примером терма типа `Li` является терм `co 1 (co 2 (co 3 nu))`, представляющий список [1;2;3]. Имя `ListOf` обозначает функцию, которая типу `A:Set` сопоставляет тип списков, состоящих из элементов `A`. Например, при `A=nat` имеем тип `ListOf nat`, аналогичный типу `Li`. Заметим, что `A` также является (неявным) параметром конструкторов `null` и `cons`, т.е. конструировать элементы типа `ListOf nat` надо с помощью `null nat` и `cons nat`. Например, список [1;2;3] будет выглядеть так:

```
cons nat 1 ( cons nat 2 ( cons nat 3 (null nat) ) )
```

См. также индуктивное определение дизъюнкции (раздел 2.2). Тип `or A B` (изображаемый как $A \vee B$) состоит из всевозможных термов `or_introl a` и `or_intror b`, где `a:A`, `b:B`.

Пусть T — индуктивный тип с параметрами $x_1 : T_1, \ldots, x_n : T_n$. Для доказательство цели с заключением вида `T t1...tn` требуется выбрать конструктор, с помощью которого будет построен терм типа $Tt_1...t_n$. Это достигается командой "`apply <конструктор>.`". Например, в случае дизъюнкции имеются два варианта — "`apply or_introl.`" или "`apply or_intror.`", сокращенной формой которых являются команды "`left.`"

и “right.”. Для автоматизации поиска вывода (тактика auto) рекомендуется добавлять все конструкторы к текущей базе данных подсказок с помощью команды “Hint Resolve <конструктор>.” .

Разбор посылок вида H: T t1...tn осуществляется командой “elim H.”, которая перебирает все способы построения термов типа $Tt_1...t_n$, порождая соответствующие подцели для каждого из конструкторов.[13] Научить тактику auto использовать elim не удается.

Замечание. Индуктивное определение типа T добавляет к формализму теории типов новый фрагмент, состоящий из правил типизации термов типом T. Конструкторы соответствуют правилам введения, что накладывает естественные ограничения на типы конструкторов. Они должны иметь вид $\forall \ldots \forall(\ldots \to T)$. Отметим также, что Coq автоматически добавляет в контекст ряд стандартных констант (T_rect, T_rec, T_ind), позволяющих индукцией по построению объекта определять другие типы, объекты и доказательства. Соответствующие им правила фактически утверждают, что каждый объект типа T неотличим от построенных с помощью конструкторов.

3.4 Секции

```
Section <название секции>.
...
End <название секции>.
```

Это окружение позволяет ограничить пределами секции область действия деклараций констант, введенных внутри секции с помощью ключевых слов Variable, Variables, Parameter, Parameters.[14] Вне секции эти константы недоступны. Там они преобразуются в дополнительные параметры всех определенных внутри секции имен. Рассмотрим следующий пример:

```
Section MySection.
Variable A B : Prop.
Definition C := A -> B.
```

[13] См. также близкие по сфере применения тактики distruct, case, induction.

[14] Это не относится к именам, введенным с помощью Hypothesis, Axiom, Conjecture. Они ведут себя так же, как введенные посредством Definition.

```
Theorem mp : A -> C -> B.
Proof. unfold C; intros H0 H1; apply H1; assumption. Qed.

End MySection.
```

Внутри секции константа `C : Prop` обозначает высказывание `A -> B`, определенное через `A, B`. Вне секции `C` становится именем функции, которая по аргументам `A, B : Prop` строит это же высказывание (т.е. именем импликации). Вот результат запроса "`Print C.`", сделанного вне секции:

```
Result for command Print C . :
C = fun A B : Prop => A -> B
     : Prop -> Prop -> Prop
```

Аналогичное происходит и с термом-доказательством `mp`:

```
Result for command Print mp . :
mp = fun (A B : Prop) (H0 : A) (H1 : A -> B) => H1 H0
      : forall A B : Prop, A -> C A B -> B
```

Появляются дополнительные аргументы `A, B`, а также соответственно изменяется формулировка теоремы (тип терма `mp`).

Таким образом, формулировки теорем внутри секции выглядят несколько проще, чем вне нее, за счет замены внешних кванторов всеобщности на декларации переменных, помещенные отдельно от теорем.

Прикладную теорию удобно представлять в виде секции, помещая в нее основные определения и теоремы. Затем ее можно будет сохранить в отдельном файле с расширением `.v` (например, `MyTheory.v`) и откомпилировать (из меню CoqIde). Получится библиотека (файл `MyTheory.vo`), которую можно использовать при разработке других теорий, загружая командами:[15]

```
Add LoadPath "<путь к каталогу, содержащему файл .vo>".
Require Import MyTheory.
```

После загрузки становятся доступными все имена, определенные в файле теории.

[15] Таблица LoadPath уже содержит некоторый набор путей, поэтому для загрузки стандартных библиотек первая строка не требуется. Посмотреть текущее ее состояние можно запросом `Print LoadPath`.

4 Пример: теория отношений

4.1 Relation_1

Рассмотрим следующий пример основных определений теории бинарных отношений на произвольном типе (Таблица 1). Он включен в дистрибутив Coq в виде библиотеки `Relations_1`[16] и может быть загружен командой

```
Require Import Relations_1.
```

Понятие бинарного отношения `Relation` определяется как тип двухместных функций, определенных на элементах произвольного типа `U:Type` со значениями в типе `Prop`. Тип `U` является параметром этого определения, поэтому вне секции (т.е. в состоянии после загрузки библиотеки) `Relation` будет обозначать функцию, сопоставляющую произвольному типу тип отношений на нем:

```
Relation = fun U : Type => U -> U -> Prop
     : Type -> Type
```

Тем самым тип отношений на натуральных числах (элементах типа `nat`) следует обозначать `Relation nat` . Примером элемента этого типа служит отношение равенства натуральных чисел:

```
Definition eqnat (x y :nat): Prop := x=y.
Theorem My : Relation nat.
Proof.
exact eqnat.
Qed.
```

То же самое относится к именам свойств отношений, определенным внутри секции. Они оказываются именами функций от дополнительных параметров `U:Type` и `R:Relation U`. Например, свойство симметричности отношений на натуральных числах представляется типом `Symmetric nat` , а утверждение о симметричности отношения равенства натуральных чисел запишется так:

[16] Разработчики: Frederic Prost и Gilles Kahn, INRIA Sophia-Antipolis, FRANCE

```
Section Relations_1.
 Variable U : Type.
 Definition Relation := U -> U -> Prop.
 Variable R : Relation.

 Definition Reflexive : Prop := forall x : U, R x x.
 Definition Transitive : Prop :=
   forall x y z : U, R x y -> R y z -> R x z.
 Definition Symmetric : Prop := forall x y : U, R x y -> R y x.
 Definition Antisymmetric : Prop :=
   forall x y : U, R x y -> R y x -> x = y :>U.
 Definition contains (R R' : Relation) : Prop :=
   forall x y : U, R' x y -> R x y.
 Definition same_relation (R R' : Relation) : Prop :=
   contains R R' /\ contains R' R.
 Inductive Preorder : Prop :=
     Definition_of_preorder :
       Reflexive -> Transitive -> Preorder.
 Inductive Order : Prop :=
     Definition_of_order :
       Reflexive -> Transitive -> Antisymmetric -> Order.
 Inductive Equivalence : Prop :=
     Definition_of_equivalence :
       Reflexive -> Transitive -> Symmetric -> Equivalence.
 Inductive PER : Prop :=
     Definition_of_PER : Symmetric -> Transitive -> PER.
End Relations_1.
Hint Unfold Reflexive.
Hint Unfold Transitive.
Hint Unfold Antisymmetric.
Hint Unfold Symmetric.
Hint Unfold contains.
Hint Unfold same_relation.
Hint Resolve Definition_of_preorder.
Hint Resolve Definition_of_order.
Hint Resolve Definition_of_equivalence.
Hint Resolve Definition_of_PER.
```

Таблица 1: Модуль Relations_1

```
Theorem eqnat_sym: Symmetric nat eqnat.
Proof.
unfold Symmetric, eqnat in |-*; auto.
Qed.
```

Выбор вариантов определения (`Definition` или `Inductive`) в библиотеке `Relations_1` не имеет однозначного объяснения. Например, можно дать следующее эквивалентное определение свойства “быть отношением предпорядка”:

```
Definition Preorder1 (U:Type) (R:Relation U):=
                   Reflexive U R /\ Transitive U R.

Theorem equiv_def: forall (U:Type) (R:Relation U),
         Preorder U R <-> Preorder1 U R.
Proof.
intros; red in |-*; unfold Preorder1.
split; intro h; elim h; auto with v62.
Qed.
```

4.2 Задачи

Рекомендуется добавлять доказанные утверждения в базу данных подсказок и пользоваться `auto with v62`.

1. Формализовать следующие утверждения и доказать.
   (a) Каждый предпорядок рефлексивен.
   (b) Каждая эквивалентность является предпорядком.
   (c) Каждая эквивалентность является частичной эквивалентностью (PER).
   (d) Отношение равенства натуральных чисел рефлексивно.
2. Дополнение симметричного отношения симметрично:

```
Definition Complement (U: Type) (R: Relation U) (x y: U) :
     Prop := ~ R x y.
Theorem Rsym_imp_notRsym: forall (U: Type) (R: Relation U),
     Symmetric U R -> Symmetric U (Complement U R).
```

3. Ограничение частичной эквивалентности на свою область определения рефлексивно:

```
Theorem PER_restricted: forall (U:Type) (R: Relation U),
    PER U R -> forall x:U, (exists y:U, R x y) -> R x x.
```

4. Предпорядок порождает эквивалентность:

```
Theorem Equiv_from_preorder :
 forall (U : Type) (R : Relation U),
 Preorder U R -> Equivalence U (fun x y:U => R x y /\ R y x).
```

5. Порядок (нестрогий) порождает эквивалентность.

```
Hint Resolve Equiv_from_preorder.
Theorem Equiv_from_order :
 forall (U : Type) (R : Relation U),
 Order U R -> Equivalence U (fun x y : U => R x y /\ R y x).
```

6. Отношение включения отношений является предпорядком (на типе `Relation U`):

```
Theorem contains_is_preorder :
 forall U : Type, Preorder (Relation U) (contains U).
```

7. Отношение равенства отношений является эквивалентностью (на типе `Relation U`).

```
Theorem same_relation_is_equivalence :
 forall U : Type, Equivalence (Relation U) (same_relation U).
```

8. Каждое симметричное и одновременно антисимметричное отношение содержится в отношении равенства:

```
Definition eqrel (U:Type) (x y : U): Prop := x=y.
Theorem sym_and_antisym: forall (U:Type) (R : Relation U),
 Symmetric U R -> Antisymmetric U R -> contains U (eqrel U) R.
```

4.3 Использование равенства

В теории типов равенство представляется частичным трехместным отношением "$x = y$ in T" (т.е. $x = y$ в типе T). Частичность его проявляется в том, что когда хотя бы один из аргументов x или y не принадлежит типу T, выражение $x = y$ in T считается не ложным, а бессмысленным. Соответствующая равенству константа `eq: (forall T: Type, T -> T -> Prop)` обозначает функцию, которая на таких аргументах не определена.

В синтаксисе Coq выражение `x=y` служит сокращением для `eq T x y`, где параметр `T` является неявным (implicit), т.е. автоматически извлекаемым в процессе типизации `x` и `y`. Система умеет использовать рефлексивность, симметричность и транзитивность равенства посредством тактик `reflexivity`, `symmetry` и `transitivity`.

Команда "`reflexivity.`" [17] : $\dfrac{\Gamma}{\texttt{t=t}} \quad \longmapsto \quad$ `Proof completed.`

Команда "`symmetry.`" : $\dfrac{\Gamma}{\texttt{u=v}} \quad \longmapsto \quad \dfrac{\Gamma}{\texttt{v=u}}$.

Команда "`transitivity t.`" : $\dfrac{\Gamma}{\texttt{u=v}} \quad \longmapsto \quad \dfrac{\Gamma}{\texttt{u=t}} \quad \dfrac{\Gamma}{\texttt{t=v}}$.

Каждое верное равенство $u = v$ позволяет в любом выражении заменить u на v или v на u. Это делает тактика `rewrite`.

Команда "`rewrite H.`" : $\dfrac{\Gamma \quad \texttt{H} : \texttt{u} = \texttt{v}}{\varphi(\texttt{u})} \quad \longmapsto \quad \dfrac{\Gamma \quad \texttt{H} : \texttt{u} = \texttt{v}}{\varphi(\texttt{v})}$.

Команда "`rewrite <- H.`" : $\dfrac{\Gamma \quad \texttt{H} : \texttt{u} = \texttt{v}}{\varphi(\texttt{v})} \quad \longmapsto \quad \dfrac{\Gamma \quad \texttt{H} : \texttt{u} = \texttt{v}}{\varphi(\texttt{u})}$.

Чтобы сделать аналогичную замену в посылке `H0`, надо к команде добавить суффикс "`in H0`". Суффикс "`in H0, H1 |-*`" позволяет сразу сделать замену в посылках `H0`, `H1` и заключении, а "`*|-*`" — всюду (кроме `H`).

Команда "`replace u with v.`" действует аналогично, но не требует, чтобы равенство `u=v` уже содержалось в контексте. Вместо этого она

[17] Эта тактика также умеет редуцировать термы в обоих частях равенства, поэтому ее можно применять и в более общем случае, когда обе части равенства становятся одинаковыми лишь после упрощения.

порождает дополнительную подцель, требующую доказать это равенство отдельно:

$$\frac{\Gamma}{\varphi(\mathtt{u})} \quad \longmapsto \quad \frac{\Gamma}{\varphi(\mathtt{v})} \quad \frac{\Gamma}{\mathtt{u} = \mathtt{v}}$$

4.4 Задачи

1. Доказать транзитивность равенства типов двумя способами (с помощью "`transitivity`" и без).

```
Theorem tr: forall (x y z:Type),
x=y -> y=z -> x=z.
```

2. Проверить возможности тактики `reflexivity`:

```
Theorem Ex_ar: (1+2)*2 = 6.
```

3. Про функцию f известно, что $f(0) = 0$ и $f(1) = f(0)$. Доказать следующие следствия:

```
Variable f: nat -> nat.
Hypothesis f00: f 0 = 0.
Hypothesis f10: f 1 = f 0.

Theorem  Ex_f1: forall k :nat,
k=0 \/ k=1 -> f k =0.

Theorem Ex_f2: f (f 1)= f 1.

Theorem Ex_compute: f (5-2*2) = f (3-3).
(* Use compute TACTIC *)
```

4. Доказать утверждение о транзитивности равенства, формализованное средствами теории `Relations_1`.

```
Require Import Relations_1.
Definition eqrel (U:Type) (x y : U): Prop := x=y.
Theorem Ex5_1:forall (U:Type), Transitive U (eqrel U).
```

5. Отношение равенства четных чисел является частичной эквивалентностью на типе `nat`:

```
Require Import Relations_1.
Definition eqeven (x y :nat):=(x=y)/\ exists z, x=2*z.
Theorem Ex5_2: PER nat eqeven.
```

6. Симметричное и одновременно антисимметричное отношение транзитивно:

```
Require Import Relations_1.
Theorem symm_antisym: forall (U:Type) (R : Relation U),
    Symmetric U R -> Antisymmetric U R -> Transitive U R.
```

7. Для коммутативной, ассоциативной операции "умножения", удовлетворяющей условию $x^2 = x$, доказать тождество $(xy)x = xy$:

```
Variable U:Set.
Variable m : U->U->U.

Hypothesis com: forall (x y:U), m x y = m y x.
Hypothesis ass: forall (x y z:U), m (m x y) z = m x (m y z).
Hypothesis idp: forall x:U, m x x = x.

Theorem XYX_XY: forall (x y:U), m (m x y) x = m x y.
```

4.5 Доказательства индукцией по построению

Рассмотрим расширение[18] `Relations_2` теории `Relations_1` (см. таблицу 2). Оно содержит индуктивные определения рефлексивного, транзитивного замыкания R^* (два эквивалентных варианта R^* и R^{*1}) и транзитивного замыкания R^+ отношения R. Система автоматически генерирует соответствующие им принципы индукции, предназначенные для доказательства утверждений индукцией по построению объектов индуктивного типа. Принципы индукции выражаются типами, принадлежащими большому[19] типу `Prop`. Система постулирует конструктивную истинность принципов индукции, населяя их соответствующими константами с суффиксом `_ind`.

[18] Оно также включено в дистрибутив Coq.

[19] Т.е. типу типов.

```
Require Import Relations_1.

Section Relations_2.
 Variable U : Type.
 Variable R : Relation U.

 Inductive Rstar : Relation U :=
  | Rstar_0 : forall x : U, Rstar x x
  | Rstar_n : forall x y z : U, R x y -> Rstar y z -> Rstar x z.

 Inductive Rstar1 : Relation U :=
  | Rstar1_0 : forall x : U, Rstar1 x x
  | Rstar1_1 : forall x y : U, R x y -> Rstar1 x y
  | Rstar1_n : forall x y z : U,
               Rstar1 x y -> Rstar1 y z -> Rstar1 x z.

 Inductive Rplus : Relation U :=
  | Rplus_0 : forall x y : U, R x y -> Rplus x y
  | Rplus_n : forall x y z : U, R x y -> Rplus y z -> Rplus x z.

 Definition Strongly_confluent : Prop := forall x a b : U,
   R x a -> R x b -> exists z : U, R a z /\ R b z.

End Relations_2.

Hint Resolve Rstar_0.
Hint Resolve Rstar1_0.
Hint Resolve Rstar1_1.
Hint Resolve Rplus_0.
```

Таблица 2: Модуль Relations_2

Например, индуктивное определение `Rstar` добавляет в контекст константу

```
Rstar_ind :
forall P : U -> U -> Prop,
(forall x : U, P x x) ->
(forall x y z : U, R x y -> Rstar y z -> P y z -> P x z) ->
forall u u0 : U, Rstar u u0 -> P u u0
```

Принцип индукции (тип константы `Rstar_ind`) для произвольного предиката $P(x, y)$ позволяет доказывать утверждения вида

$$\forall u, u_0 \in U (uR^* u_0 \to P(x, y))$$

разбором всех вариантов того, как может быть установлено uR^*u_0. Их два — по числу конструкторов индуктивного типа. В первом случае конструктор `Rstar_0` позволяет установить uR^*u_0 для $u = u_0 = x \in U$, поэтому надо доказывать $\forall x \in U\, P(x, x)$. Во втором случае конструктор `Rstar_n` позволяет установить uR^*u_0, когда $u = x$, $u_0 = z$ и для некоторого y известно, что xRy и yR^*z. По предположению индукции, [20] $P(y, z)$ также верно, поэтому достаточно доказать

$$\forall x, y, z \in U\, (xRy \to yR^*z \to P(y, z) \to P(x, z)).$$

Именно это сведение реализует команда "`elim H.`", когда посылка `H` имеет вид `H:Rstar` U R для некоторых U и R. То же самое относится ко всем индуктивным определениям — обращение к принципам индукции происходит неявно, посредством тактики `elim`.

Техника добавления доказанных утверждений в посылки

Для дальнейшего потребуется доказать следующие простые вспомогательные утверждения:

```
Require Import Relations_1.
Require Import Relations_2.
Lemma Rstar_reflexive :
 forall (U : Type) (R : Relation U), Reflexive U (Rstar U R).
```

[20] Индукция по длине цепочки $z_0Rz_1R \ldots z_{k-1}Rz_k$, где $y = z_0$, $z = z_k$.

```
 Proof. auto with v62. Qed.

Lemma Rplus_contains_R :
 forall (U : Type) (R : Relation U), contains U (Rplus U R) R.
 Proof. auto with v62. Qed.

Lemma Rstar_contains_R :
 forall (U : Type) (R : Relation U), contains U (Rstar U R) R.
 Proof.
 intros U R; red; intros x y H'; apply Rstar_n with y;
 auto with v62.
 Qed.
```

Как видно, почти со всей работой справляется тактика `auto with v62`, т.к. база данных `v62` содержит все необходимые подсказки (те, что указаны в таблицах 1, 2). Исключение составляет последняя лемма, т.к. в ее доказательстве тактике `apply` требуется подсказать значение ("`with y`"), которое она не может найти с помощью унификации.[21]

Отметим, что в формулировках лемм встречаются константы (`Reflexive`, `contains`), введенные ранее с помощью определений. Естественно ожидать, что в некоторых случаях применения этих лемм потребуется раскрыть соответствующие определения, что приведет к изменению типов других констант: `Rstar_reflexive`, `Rplus_contains_R`, `Rstar_contains_R`. Разумеется, система не позволит это сделать — типы констант, т.е. формулировки доказанных утверждений, меняться не должны.

В то же время тактика `unfold in H` позволяет раскрывать определения в посылках. Это возможно потому, что в посылке `(H:...)` само `H` является идентификатором переменной, а изменение ее типа фактически заменит эту переменную на другую, но с тем же идентификатором. Тем самым, чтобы перед применением доказанной леммы вида `c:F` иметь возможность раскрыть определения внутри ее формулировки `F`, следует добавить ее формулировку (в виде типа новой переменной, `H:F`) в качестве дополнительной посылки.

[21] Ситуация такова: `(Rstar_n u v w)` позволяет преобразовать доказательства uRv и vR^*w в доказательство uR^*w; дано xRy; требуется доказать xR^*y. Тактика `apply` сопоставляет uR^*w с xR^*y и находит $u = x$, $w = y$, но значение $v = y$ должен подсказать пользователь, руководствуясь рефлексивностью R^*.

Самый простой способ это сделать — с помощью тактики `generalize`. Команда "`generalize c.`" сводит задачу доказательства утверждения $\varphi(c)$ к задаче доказать более общее утверждение $\forall h : F\,\varphi(h)$, где F — тип терма c. Если φ не содержит терма c, то квантор всеобщности заменится на импликацию. Затем надо применить "`intro H.`", что приведет к появлению желаемой посылки `H:F`.[22]

$$\frac{\Gamma}{\varphi} \quad \overset{\texttt{generalize c}}{\longmapsto} \quad \frac{\Gamma}{\texttt{F}->\varphi} \quad \overset{\texttt{intro H}}{\longmapsto} \quad \frac{\begin{array}{l}\Gamma\\ \texttt{H : F}\end{array}}{\varphi}$$

Замечание. Тот же прием позволяет при введении `F` в посылки конкретизировать значения переменных, связанных внешними кванторами всеобщности. Например, пусть формулировка `F` леммы `c:F` имеет вид `forall x:U, G(x)`, а одна из посылок имеет вид `a:U`. Тогда терм `(c a)` будет иметь тип `G(a)`, поэтому последовательность команд

```
generalize (c a); intro H.
```

приведет к следующему:

$$\frac{\begin{array}{l}\Gamma\\ \texttt{a : U}\end{array}}{\varphi} \quad \longmapsto \quad \frac{\begin{array}{l}\Gamma\\ \texttt{a : U}\end{array}}{\texttt{G(a)}->\varphi} \quad \longmapsto \quad \frac{\begin{array}{l}\Gamma\\ \texttt{a : U}\\ \texttt{H : G(a)}\end{array}}{\varphi}$$

При этом в роли `c:F` можно использовать не только доказанное ранее утверждение, но и одну из посылок цели.

В качестве примера разберем доказательство включения $R^+ \subseteq R^*$:

```
Theorem Rstar_contains_Rplus: forall (U: Type) (R: Relation U),
                              contains U (Rstar U R) (Rplus U R).

Proof.
   intros U R; red in |- *.
   intros x y H; elim H.
```

[22] Заметим, что здесь неявно применено правило сечения с формулировкой доказанной леммы в качестве высекаемой формулы. Это можно сделать и явно: "`assert F. exact c.`", что дает другой способ. К сожалению, потребуется вставить полную формулировку `F` леммы в текст команды.

Первая строка избавляется от внешних кванторов и раскрывает определение `contains`. Затем в посылки помещаются допущения $x, y \in U$ и xR^+y; требуется установить xR^*y. Предлагается использовать индукцию по построению доказательства xR^+y (т.е. по построению термов, населяющих тип xR^+y). Тактика `elim` находит два варианта построения и формирует соответствующие подцели (с одинаковыми посылками):

```
U : Type
R : Relation U
x : U
y : U
H : Rplus U R x y
______________________________________(1/2)
forall x0 y0 : U, R x0 y0 -> Rstar U R x0 y0


______________________________________(2/2)
forall x0 y0 z : U,
R x0 y0 -> Rplus U R y0 z -> Rstar U R y0 z -> Rstar U R x0 z
```

Займемся первой. Ее заключение фактически совпадает с доказанной ранее леммой `Rstar_contains_R`, однако в формулировке леммы включение записано с помощью `contains`, а в заключении цели — через импликацию. Потребуется раскрыть определение `contains`, для чего перенесем формулировку леммы в гипотезы и там раскроем определение. Остальное доделает тактика `auto`.

```
   generalize Rstar_contains_R; intro h; red in h.
   auto.
```

Вторая подцель предлагает для произвольных $x_0, y_0, z \in U$ установить, что из x_0Ry_0, y_0R^+z, y_0R^*z следует x_0R^*z. Заметим, что второе предположение — лишнее, а из первого и третьего можно получить x_0R^*z применением конструктора `Rstar_n`. При этом тактике `apply` надо будет подсказать значение y_0. Она породит две тривиальные подцели, которые можно обработать тактикой `auto` (или `assumption`).

```
   intros x0 y0 z H0 H1 H2.
   apply Rstar_n with y0; auto.
Qed.
```

4.6 Задачи

Использовать `auto with v62`.

1. ```
   Theorem Rstar_transitive: forall (U: Type)(R: Relation U),
                                     Transitive U (Rstar U R).
   ```

   Указание: индукцией по построению доказательства xR^*y доказывать импликацию $yR^*z \to xR^*z$.

2. ```
   Theorem Rstar_cases:
    forall (U: Type) (R: Relation U) (x y: U),
    Rstar U R x y ->
    x = y  \/ (exists u : U, R x u /\ Rstar U R u y).
   ```

3. ```
   Theorem Rstar_equiv_Rstar1: forall (U: Type)(R: Relation U),
                       same_relation U (Rstar U R) (Rstar1 U R).
   ```

   Указание: использовать `Rstar_contains_Rplus` и `Rstar_transitive`.

4. ```
   Theorem Rsym_imp_Rstarsym: forall (U: Type) (R: Relation U),
                       Symmetric U R -> Symmetric U (Rstar U R).
   ```

5. $R \subseteq S^*$ влечет $R^* \subseteq S^*$:

   ```
   Theorem Sstar_contains_Rstar :
    forall (U : Type) (R S : Relation U),
    contains U (Rstar U S) R ->
    contains U (Rstar U S) (Rstar U R).
   ```

   Указание: использовать `Rstar_transitive`.

6. $R \subseteq S$ влечет $R^* \subseteq S^*$:

   ```
   Theorem star_monotone :
    forall (U : Type) (R S : Relation U),
    contains U S R -> contains U (Rstar U S) (Rstar U R).
   ```

   Указание: использовать доказанные факты `Sstar_contains_Rstar` и `Rstar_contains_R`. Применение первого из них сводит исходную задачу к доказательству включения $R \subseteq S^*$. Но $R \subseteq S$, поэтому надо установить включение $S \subseteq S^*$.

Для того, чтобы получить доказательство (т.е. терм типа) ($S \subseteq S^*$), надо `Rstar_contains_R` применить к `U` и затем к `S`. Тем самым, команды

"`generalize (Sstar_contains_R U S); intro h.`"

позволят добавить посылку `(h: contains U (Rstar U S) S)` к уже имеющейся `(H: contains U S R)`. С транзитивностью включения разберется `auto with v62`.

7.
```
Theorem RstarRplus_RRstar :
 forall (U : Type) (R : Relation U) (x y z : U),
 Rstar U R x y ->
 Rplus U R y z ->
 exists u : U, R x u /\ Rstar U R u z.
```
Указание: использовать `Rstar_contains_Rplus` и `Rstar_transitive`.

5 Основные средства функционального программирования (Gallina)

Система Coq имеет встроенный язык функционального программирования Gallina, предназначенный для спецификации объектов математических теорий. Математические объекты и их свойства представляются термами этого языка. Те же термы выступают в роли функциональных программ, задающих процесс синтаксического преобразования (упрощения) представлений посредством системы редукций, определяющей семантику используемых обозначений. Здесь мы приводим лишь основные конструкции языка, отсылая читателя к руководству "The Coq Proof Assistant Reference Manual" за более полным изложением.

5.1 Аппликация и λ-абстракция

Аппликация, т.е. применение функции `f: forall x: A,B(x)` к аргументу `a:A` записывается как произведение `(f a)` и имеет тип `B(a)`. Когда тип `B` не зависит от `x`, тип `f` сокращается до `A->B` и произведение `(f a)` имеет тип `B`. Скобки в выражении `(((f a1) a2)...an)` принято опускать и писать `f a1 a2...an`.

λ-абстракция предоставляет стандартный способ обозначения для функций. Соответствующее правило из теории типов выглядит так:

$$\frac{\Gamma, x : A \vdash t(x) : B(x)}{\Gamma \vdash (\lambda x{:}A.t(x)) : \forall x\, B(x)}$$

При этом для каждого $a : A$ предполагается редукция (β-редукция)

$$(\lambda x{:}A.t(x))\, a \to t(a),$$

которая означает, что объект $\lambda x : A.t(x)$ действует на a как функция, способ вычисления которой задан термом $t(x)$. Итерация λ-оператора $\lambda x_1{:}A_1.(\lambda x_2{:}A_2.(\ldots))$ обычно сокращается до $\lambda(x_1{:}A_1)(x_2{:}A_2).(\ldots)$.

В синтаксисе Coq эта конструкция записывается так:

```
fun (x1:A1) ... (xn:An) => t
```

Это терм типа `(forall (x1:A1)...(xn:An),B)`, где `B` — тип терма `t`, который (как и `t`) может зависеть от `x1`, ..., `xn`. Допускается также зависимость `A2` от `x1`, `A3` от `x1`, `x2`, и т.д.

В качестве примеров отметим следующие определение и теорему:

```
Definition f: nat -> nat -> nat := fun (m n:nat) => m+n.

Theorem  XX : forall (U:Type), Transitive U (fun (x y:U) => x=y).
```

Тактика `compute` (команда "`compute.`") позволяет проделать всевозможные редукции в цели. Команда "`compute in H.`" применяет редукции к гипотезе `H`, а запрос "`Eval compute in t.`" позволяет увидеть результат редуцирования произвольного терма `t`. Например:[23]

```
Theorem XXX: forall P:Prop, (fun x:nat => 2*x) 7 = 5 -> P.
Proof. intros P H; compute in H; contradict H; auto 10. Qed.
```

[23] Вариант команды "`auto 10.`" увеличивает глубину поиска вывода с 5 (по умолчанию) до 10. В процессе доказательства неравенства $14 \neq 5$ последовательно устанавливается, что $9 \neq 0$, $10 \neq 1$, ..., $14 \neq 5$, поэтому глубина поиска вывода должна быть не меньше 6.

5.2 Разбор случаев (case analysis)

Пусть задано индуктивное определение типа T с конструкторами $c_1, \ldots, c_k$, где c_i представляет функцию от n_i аргументов со значениями в типе T. Каждый терм t типа T имеет вид $c_i t_{i,1} \ldots t_{i,n_i}$ для некоторого i, причем это представление единственно. Разбор случаев позволяет через t определить выражение $h(t)$ разными способами в зависимости от вида терма t:

$$h(t) = \begin{cases} h_1(t_{1,1} \ldots t_{1,n_1}), & \text{если } t = c_1 t_{1,1} \ldots t_{1,n_1}, \\ \ldots \\ h_k(t_{k,1} \ldots t_{k,n_k}), & \text{если } t = c_k t_{k,1} \ldots t_{k,n_k}. \end{cases}$$

Чтобы выражению $h(t)$ удалось приписать тип, достаточно потребовать, чтобы все термы $h_i(t_{i,1} \ldots t_{i,n_i})$ имели общий тип T_1. Теория зависимых типов позволяет ослабить это условие: достаточно иметь семейство типов $T_1(z)$, индексированное элементами $z{:}T$, и требовать, чтобы

$$h_i(t_{i,1} \ldots t_{i,n_i}){:}T_1(t) \quad \text{при} \quad t = c_i t_{i,1} \ldots t_{i,n_i}.$$

Соответствующая редукция (ι-редукция) заменяет терм $h(t)$ на $h_i(t_{i,1} \ldots t_{i,n_i})$, сохраняя его тип $T_1(t)$.

В синтаксисе Coq разбор случаев реализует конструкция `match`. Вот простой примеры определения функции “предшественник” на натуральных числах:

```
Inductive nat : Set :=  O: nat | S: nat -> nat.
Definition pred (n:nat) := match n with
                             | O => n
                             | S u => u
                           end.
```

Более сложный случай демонстрирует определение преобразования булева значения `b:bool` в доказательство дизъюнкции `(b=true)\/(b=false)`. Подсказка “`return (b=true)\/(b=false)`” сообщает системе тип терма `(match ... end)`, который зависит от параметра `b`.

```
Inductive bool: Set :=  true: bool | false: bool.
Definition bool_case (b:bool) :=
      match b return (b=true)\/(b=false) with
       |true => or_introl (true=false)(refl_equal true)
       |false => or_intror (false=true) (refl_equal false)
```

```
      end.
Theorem bc : forall b:bool, (b=true)\/(b=false).
Proof. exact bool_case. Qed.
```

Следует отметить ограничение, накладываемое системой на использование `match` в случае, когда разбирается индуктивное определение типа $T : Prop$. Оно объясняется классическим принципом "Proof irrelevance", совместимым с теорией типов системы Coq. Согласно этому принципу, верность утверждения T не должна зависеть от того, как это утверждение доказано, поэтому все его доказательства (т.е. все объекты типа T) императивно объявляются равными между собой. Каждое корректное определение отображения из типа $T : Prop$ в некоторый другой тип T_1 должно быть согласовано с равенством, поэтому невозможно корректно сопоставить двум структурно различным доказательствам $t_1, t_2 : T$ два элемента $h_1, h_2 : T_1$ так, чтобы $h_1 \neq h_2$. В практической реализации этот запрет обеспечивается следующим (более сильным) ограничением:

> если `match` реализует разбор случаев индуктивного определения типа $T : Prop$, то тип T_1 терма `(match ... end)` также должен лежать в $Prop$.

Тем самым, с помощью `match` можно определить отображение, преобразующее доказательства дизъюнкции $A \vee B$ в доказательства $B \vee A$, но нельзя определить никакую числовую характеристику выводов $A \vee B$ (напр., если доказательство $A \vee B$ устанавливает A, то 0, если B, то 1; в системе нет необходимого для этого принципа — константы `or_rec`).

5.3 Булева сумма

Булева сумма высказываний `P Q :Prop` — это операция `sumbool`, аналогичная дизъюнкции, но возвращающая объект типа `Set` (обозначается `{P}+{Q}`).

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

Для нее нет ограничений на использование вместе с конструкцией `match`. Например, корректно следующее определение:

```
Definition f (n:nat) (h:{n=0}+{n<>0}):= match h with
                                         | left _  => 1
                                         | right _ => 0
                                       end.
```

Если индуктивное определение имеет два конструктора (в частности, таково определение `sumbool`), то вместо "`match ... end`" удобно пользоваться сокращенной записью "`if ... then ... else`". Приведенное выше определение перепишется так:

```
Definition f (n:nat) (h:{n=0}+{n<>0}):= if h then 1 else 0.
```

Функция f, получив в качестве аргументов натуральное число n и свидетельство справедливости одного из условий $n = 0$ или $n \neq 0$, возвращает в первом случае 1, а во втором — 0.

Булеву сумму высказываний можно "доказывать" в интерактивном режиме. Работают те же тактики, что и с дизъюнкцией. Например:

```
Lemma zero_dec: forall n:nat, {n=0}+{n<>0}.
Proof.
intro n. induction n;[left; reflexivity | right; discriminate].
Qed.

Lemma sumbool2or : forall (P Q :Prop), {P}+{Q} -> P\/Q.
Proof. intros P Q H; elim H; auto. Qed.
```

Теперь `zero_dec` можно использовать в качестве функции, преобразующей натуральные числа в вид, удобный для разбора случаев ($n = 0$ или $n \neq 0$), с последующим его использованием в других определениях:

```
Definition g (n:nat) := if (zero_dec n) then 1 else 0.
```

Основные факты про разрешимость равенства и неравенств в типе `nat` содержатся в стандартной библиотеке `Arith` (см. модули `Peano_dec` и `Compare_dec`). Регулярно используются две формы их представления — с помощью дизъюнкции и с помощью булевой суммы. Первая обычно используется при доказательстве теорем, а вторая — в определениях других объектов.[24] От второй формы легко перейти к первой (лемма `sumbool2or`); обратный переход в общем случае невозможен.

[24] Это разделение весьма условно, т.к. процесс доказательства являются интерактивным способом построения определения объекта типа `Prop`, а определения объектов с помощью `Definition` также допускают интерактивную форму `Proof. ... Defined.` (см. раздел 5.5).

5.4 Рекурсия

Разбор случаев может содержать рекурсивные самовызовы, что дает возможность определять функции рекурсией по построению одного из аргументов, являющегося объектом индуктивного типа. При этом допускаются лишь корректные определения, не порождающие бесконечных вычислений при любых значениях аргументов. Само определение вводится ключевым словом `Fixpoint`. Например, определение экспоненты с основанием 2 выглядит так:

```
Fixpoint exp2 (n:nat) := match n with
                          |O   => 1
                          |S x => 2*(exp2 x)
                         end.
```

Это определение раскрывается системой как вариант стандартного `Definition` с использованием "локальной" конструкции `fix`, позволяющей использовать определения рекурсивных функций анонимно (т.е. без введения идентификатора[25]) внутри других конструкций:

```
Definition exp2 :=
(fix f (n:nat) :nat := match n with
                        |0 =>1
                        |S x =>2*(f x)
                       end).
```

Определение рекурсией может содержать дополнительные параметры, но рекурсия допускается только по одному аргументу, выделенному подсказкой `{struct ...}`. Например, в определении операции возведения в степень рекурсия ведется по второму аргументу:

```
Fixpoint exp (m n:nat) {struct n} := match n with
                                      |O   => 1
                                      |S x => m*(exp m x)
                                     end.
```

Индексированные семейства типов или высказываний тоже можно определять рекурсией:

[25] Переменная `f` в выражении `(fix f ...)` локальна и не является именем рекурсивной функции вне него.

```
Fixpoint vect (n:nat):Set :=
                      match n with
                       |O   => nat
                       |S x => prod (vect x) nat
                      end.

Fixpoint F (n z:nat) {struct n} :Prop :=
                      match n with
                       |O   => z=0
                       |S m => (F m z) \/ z=n
                      end.
```

Тип `(vect 4)` представляет множество всех 5-мерных векторов,[26] координаты которых — натуральные числа, а тип `(F 3 1)` — высказывание

$$(1 = 0) \vee (1 = 1) \vee (1 = 2) \vee (1 = 3).$$

Для упрощения выражений вида `(f t)`, где функция `f` определена рекурсией, применяется тактика `simpl`. Она реализует разбор случаев, выбирая подходящий для терма `t`. Вызванная без параметров, она упрощает все подобные выражения. Ей можно указать конкретное выражение для упрощения (команда "`simpl (f t).`"), а также номера вхождений этого выражения в заключение (команда "`simpl (f t) at 1 3.`"). Эту тактику можно применять и к гипотезам, добавляя "`in H`".

В качестве примера установим, что определенная выше функция `exp2` удовлетворяет соответствующему функциональному уравнению.

```
Theorem exp2_is_exp2: forall n:nat, exp2 (S n)=2 * (exp2 n).
Proof. intro n. simpl. reflexivity. Qed.
```

Отметим, что незначительное изменение формулировки может затруднить применение `simpl`. Так, если терм `(S n)` заменить на `(1+n)`, то будет работать тот же скрипт доказательства, а если на `(n+1)`, то не будет. Дело в том, что используемая здесь функция сложения определяется рекурсией по первому аргументу. Когда этот аргумент 1, тактика `simpl` сначала реализует эту рекурсию, упрощая `(1+n)` до `(S n)`, после

[26] В определениии `vect` константа `prod: Type->Type->Type` обозначает (декартово) произведение типов. Для произведения типов есть и сокращенное обозначение `(A*B)%type`, но оно требует указания области применимости сокращения "`%type`".

чего работает как в базовом случае. Если же первый аргумент — переменная `n`, то предварительное упрощение с помощью рекурсии невозможно, поэтому разбор случаев из определения `exp2` также не удается проделать. Выход из положения — заменить `(n+1)` на `(S n)` командой "`replace (n+1) with (S n).`" и отдельно доказать равенство `n+1 = S n` тактикой `reflexivity`.

5.5 Типы данных

"Большой" тип `Set` (тип типов, сорт) предназначен для типизации пользовательских типов данных и спецификаций.[27] "Малые" типы `bool`, `nat`, `nat -> nat` и пр. являются объектами типа `Set`, который сам является объектом типа `Type`.

Сумма и произведение. Тип `Set` во многом аналогичен типу `Prop`. Он также замкнут относительно применения квантора `forall`, который в этом контексте называется зависимым произведением типов, и его частного случая — связки $\to$. Здесь также есть полные аналоги связок $\wedge$, $\vee$ — операции (декартова) произведения (`prod`) и дизъюнктного объединения, или суммы (`sum`) типов[28]:

```
Inductive prod (A B: Type): Type :=  pair: A -> B -> prod A B.

Inductive sum (A B: Type): Type := inl: A -> sum A B
                                 | inr: B -> sum A B.
```

В синтаксисе Coq также используются обозначения[29] `(A*B)%type` для произведения и `(A+B)%type` для суммы, а `(pair a b)` записывается как `(a,b)`.

Таким образом, тип `(A*B)%type` населен всевозможными парами `(a,b)`, где `a:A`, `b:B`. Функция `pair` также имеет два дополнительных неявных (implicit) параметра — типы `A` и `B`. Указывать их явно не следует, т.к.

[27] Под спецификацией здесь понимается конструкция типов, аналогичная теоретико-множественной конструкции $\{x \in A \mid P(x)\}$.

[28] Эти операции на самом деле определены на типе `Type`, но сохраняют тип `Set`.

[29] Суффикс `%type` подсказывает соответствующее пространство имен (scope), в данном случае, type_scope.

они определяются системой автоматически как типы основных аргументов a,b.

С типом суммы (A+B)%type ситуация несколько сложнее. Он населен всеми объектами видов (inl a) и (inr b), где a:A, b:B. Функции inl и inr также имеют оба типа A и B в качестве дополнительных параметров, но исходя из типа основного аргумента удается восстановить лишь один из них, а второй надо указывать явно: (inl B a) и (inr A b). Впрочем, можно отказаться от механизма автоматического восстановления неявных параметров с помощью @-нотации. Тогда надо писать (@inl A B a) и (@inr A B b).

Тип произведения оснащен функциями проекций fst и snd с неявными аргументами A B:Type:[30]

```
Definition fst (A B:Type) (p:(A*B)%type) : A :=
                                  match p with (x,_) => x end.

Definition snd (A B:Type) (p:(A*B)%type) : B :=
                                  match p with (_,y) => y end.
```

При обработке объектов типа суммы также удобно применять конструкцию match. Например, можно определить операцию "соединения" двух функций f:A->C и g:B->C в одну функцию типа (A+B)%type -> C:

```
Definition union (A B C: Type) (f: A->C) (g: B->C) :=
        fun p:(A+B)%type   =>   match p with
                                 | inl a => f a
                                 | inr b => g b
                                end.
```

Отметим некоторую специфику использования inl и inr в образцах внутри match — здесь не следует указывать их дополнительные типовые параметры A и B, т.к все они извлекаются из контекста (из типа p).

Зависимые суммы. Аналогом конструктивного квантора существования exists в типе Set служит зависимая сумма. Ее слабый вариант

[30] Объявление отдельных аргументов неявными, т.е. восстанавливаемыми автоматически, осуществляется отдельной декларацией, которую мы здесь не приводим.

`sig` определяется совершенно аналогично `ex` (см. 2.4 , абстрактный синтаксис для `exists`), с той лишь разницей, что возвращаемое значение теперь имеет тип `Type`:

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

Для типа `(sig A P)` используется обозначение `{x:A|P x}`. Если `A:Set`, то также `{x:A|P x}:Set`. Этот тип состоит из всех объектов вида `(exist x p)`, где `x:A` и `p:(P x)`.[31] С помощью конструкции `match` из объекта `h=(exist x p)` типа `{x:A|P x}` можно извлечь компоненты `x` и `p`. [32]

```
Definition proj1_sig (A:Type)(P:A -> Prop) :=
       fun h:{x:A|P x} => match h with (exist x p) => x end.

Definition proj2_sig (A:Type)(P:A -> Prop) :=
       fun h:{x:A|P x} => match h return P (proj1 A P h)
                         with (exist x p) => p end.
```

Построение объекта типа `{x:A|P x}` включает в себя предъявление `x` и доказательство того, что `x` обладает свойством `P`. В этом смысле тип `{x:A|P x}` играет роль спецификации объектов.

Спецификации можно доказывать как теоремы. Для этого `Definition` имеет интерактивный режим, аналогичный режиму `Theorem`. Единственное отличие состоит в том, что в конце доказательства вместо "`Qed.`" надо писать "`Defined.`".

Для доказательства зависимой суммы `{x:A|P x}` применяется та же тактика `exists`, что и для квантора существования. В качестве параметра ей надо указать значение `x`, которое удобно определить заранее. Например, докажем (полную) спецификацию функции $g(n) = \Sigma_{i=0}^{n-1} i$, т.е. что g принадлежит множеству $\{f \mid f(0) = 0 \wedge \forall n(f(n+1) = f(n) + n)\}$.

[31] Не следует путать конструктор `exist` с обозначением для квантора существования `exists`, а также с названием одноименной тактики .

[32] На самом деле при явном построении объектов типа `{x:A|P x}` с помощью `exist` надо также указывать `P`, т.е. писать `(exist P x p)`. Здесь `P` используется для вычисления типа построенного объекта. В то же время при использовании `exist` в образце для `match` писать `P` не следует (см. определение `proj`), т.к. вычислять тип образца не нужно.

```
Fixpoint g (n : nat) : nat := match n with
                               |O => O
                               |S p => (g p)+ p
                              end.

Definition spec:
   {f:nat->nat |(f 0 =0)/\(forall n:nat, f(S n)=f(n)+ n)}.
Proof.
exists g; split.
reflexivity.
intro; simpl g; reflexivity.
Defined.
```

Легко видеть, что доказательство спецификаций есть ничто иное, как некоторым образом стандартизованная форма верификации программ в функциональном программировании.

Другой вариант типа зависимой суммы — сильный — обозначается `{x:A & P x}`. В отличие от предыдущего, в нем допускаются `P:A->Type`. При этом если `A:Set` и `P:A->Set`, то также `{x:A & P x}:Set`. Тип населен объектами вида `(existS x p)`, где `x:A` и `p:(P x)`, и оснащен уже двумя проекциями `projS1` и `projS2`.

Эта конструкция применяется для построения новых предметных областей. Например, рассмотрим определенную в разделе 5.4 последовательность типов `vect 0`, `vect 1`, ..., где тип `vect n` состоит из всех $(n+1)$-мерных векторов с координатами типа `nat`. Сильную зависимую сумму этой последовательности удобно использовать для представления векторов произвольной положительной длины:

```
Definition Vector := {n:nat & vect n}.
Definition last_index (a: Vector) : nat := projS1 a.
Definition val (a: Vector) : vect (last_index a) := projS2 a.
```

Объекты типа `Vector` имеют вид `(existS n v)`, где `v:(vect n)` — сам вектор, а `n` на единицу меньше его размерности. Параметры `n`, `v` можно извлечь с помощью функций `last_index` и `val`. Для построения объектов типа `Vector` удобно определить следующие функции:

```
Definition init (n: nat):Vector :=
```

```
                              existS (fun x:nat => vect x) 0 n.
Definition add (a:Vector) (x:nat):Vector :=
   existS (fun x:nat => vect x) (S (last_index a)) ((Val a),x).
```

Тогда вектор (1,2,3) запишется как `add(add(init 1) 2) 3` .[33]

В качестве демонстрации возможностей функционального программирования оснастим тип `Vector` функцией `el: Vector -> nat -> nat`, реализующей доступ (read-only) к компонентам вектора по их индексам. Тем самым `Vector` становится типом массивов без возможности модификации элементов массива.

```
Definition last (a:Vector) : nat :=
           match a with
             | existS 0 x => x
             | existS (S m) (x,y) => y
           end.

Definition cut_last (a:Vector) : Vector :=
           match a with
             | existS   0 _       => a
             | existS (S m) (u,_) =>
                           existS (fun x:nat => vect x) m u
           end.

Fixpoint iterate (A:Type)(f:A->A)(x:A)(n:nat) : A :=
         match n with
          | 0 => x
          | S m => f (iterate A f x m)
         end.

Definition el (a:Vector)(i:nat) : nat :=
        last( iterate Vector cut_last a ((last_index a)-i) ).
```

Функция `last` извлекает последнюю компоненту вектора, функция `cut_last` — удаляет ее, а функция `iterate` реализует итерацию произвольной функции f заданное число раз (т.е. вычисляет $f^n(x)$). Для вы-

[33] Конструктор `existS` устроен аналогично `exist` (см. сноску 32), т.е. при конструировании объектов ему надо указывать дополнительный параметр `(fun x:nat => vect x)`, а внутри образцов для `match` — нет.

числения компоненты a_i вектора $(a_0, \ldots, a_n)$ достаточно удалить последнюю компоненту $(n - i)$ раз, после чего извлечь последнюю компоненту оставшегося вектора. Предлагаемая реализация удаления оставляет одномерный вектор неизменным, поэтому попытка вычислить компоненту с номером $i > n$ даст a_n.

5.6 Задачи

Общие указания. Обычная индукция по натуральному параметру n реализуется командой "`induction n.`" . Для решения некоторых задач возможно потребуется библиотека `Arith` (арифметика натуральных чисел), которую можно загрузить командой "Require Import Arith." [34] Тогда, в частности, будет доступна тактика `ring` (команда "`ring.`"), которая умеет доказывать все равенства, справедливые в силу аксиом кольца (для `nat` — полукольца, т.е. без аксиом для вычитания). Кроме того, станет доступной база подсказок `arith`, которая "научит" тактику `auto` пользоваться основными фактами этой теории, описывающей операции `+,-,*` и неравенства в типе `nat` (команда "`auto with arith.`").

1. Для определенной в разделе 5.4 функции `exp` доказать:

```
Theorem exp_is_exp:
forall (m n:nat), exp m 0 =1 /\ exp m (S n) = m*(exp m n).

Definition spec_exp: { f:nat->nat->nat |
forall (m n:nat), f m 0 = 1  /\  f m (S n) = m * (f m n)}.
```

2. В следующих задачах использовать определения из раздела 5.5.

```
Lemma last_index_of_init: forall (x:nat),
                   last_index (init x)=0.

Lemma last_of_init : forall (x:nat), last (init x)=x.
```

[34] Детальное описание библиотеки содержится в документации (файл Libraries.pdf и его html-версия на сайте `qoc.inria.fr`). Полезны также исходные файлы библиотеки, которые расположены в каталоге `theories/Arith/` дистрибутива Coq (файлы с расширением `.v`). Они содержат скрипты доказательств, которые рекомендуется использовать в качестве образцов для подражания.

```
Theorem init_is_init: forall (x:nat),
        last_index (init x)=0 /\ el (init x) 0 = x.
```

3. ```
   Lemma last_add1: forall (a:Vector) (x:nat),
               last_index (add a x) = 1 + (last_index a).

   Lemma last_add2: forall (a:Vector) (x:nat),
                             last (add a x)  = x.

   Lemma last_add3: forall (a:Vector) (x:nat),
                 el (add a x) (last_index (add a x)) = x.

   Lemma cut_last_add: forall (a:Vector) (x:nat),
                                  cut_last (add a x) = a.
   ```

4. ```
   Lemma iterate_0: forall (A:Type)(f:A->A)(x:A),
                                  iterate A f x 0 =x.

   Lemma iterate_S: forall (A:Type)(f:A->A)(x:A)(n:nat),
              iterate A f x (S n) = f (iterate A f x n).

   Theorem iterate_plus: forall (A:Type)(f:A->A)(x:A)(m n:nat),
    iterate A f x (m + n) = iterate A f (iterate A f x n) m.
   ```

5. ```
   * Lemma el_add: forall (a:Vector) (x i:nat),
          i <= last_index a  ->  el (add a x) i = el a i.
   ```

   Указание. Потребуется интенсивное использование команд "`rewrite H.`", а также "`replace u with w.`" и/или "`assert u=v.`". Арифметические факты про минус (например, $1 + (m - n) = (1 + m) - n$) нетривиальны для типа `nat` и требуют дополнительных условий ($n \leq m$). Их удобно доказывать в виде отдельных лемм. Для доказательства очевидных соотношений про `a:Vector` можно применить "`induction a.`".

6. Верифицировать определенное ниже преобразование `f2v` функций типа `nat -> nat` в последовательность векторов,

$$f \quad \mapsto \quad f(0),\ (f(0), f(1)),\ (f(0), f(1), f(2)),\ \ldots\ .$$
   ```

```
Fixpoint f2v (f:nat->nat) (n:nat){struct n} : Vector :=
match n with
 |0 => init (f 0)
 |S m => add (f2v f m) (f (S m))
end.

Lemma last_index_of_f2v: forall (f:nat->nat) (n:nat),
                         last_index (f2v f n) = n.

Lemma last_of_f2v: forall (f:nat->nat) (n:nat),
                   last (f2v f n) = f n.

* Theorem el_is_el: forall (f:nat->nat) (n i:nat),
                    i <= n -> el (f2v f n) i = f i.
```

6 Решения

Ниже предлагаются решения некоторых задач из предыдущих разделов. Решения представлены в виде скриптов для исполнения посредством CoqIde. Для детального анализа процесса построения выводов рекомендуется выборочно заменять ";" на ".", что позволяет сделать шаги доказательства более мелкими.

Задачи из 2.8

2.8 (1)
```
Proof. intros A H. exact H. Qed.
```

2.8 (2)
```
Proof.
intros A B C H H0 H1.
apply H;[assumption | apply H0; assumption].
Qed.
```

2.8 (3)
```
Proof.
intros A B C D H H0.
elim H; intros H1 H2; clear H.
elim H0; intros H3 H4; clear H0.
split;
[apply H1; assumption | apply H2;assumption].
Qed.
```

2.8 (4)
```
Proof.
intros A H; unfold not.
intro H0; apply H0; assumption.
Qed.
( Proof. intros A H; contradict H; assumption. Qed. )
```

2.8 (5)
```
Proof.
intros a H.
assert (a\/~a) as h. apply classic.
elim h; clear h.
intro; assumption.
intro. contradiction.
Qed.
```

2.8 (6)
```
Proof. intros p q a H H0; apply H0; assumption. Qed.
```

2.8 (7)
```
Proof.
intros a b p H.
elim H; [intro; exists a; assumption
| intro; exists b; assumption].
Qed.
```

2.8 (8)
```
Proof.
intros A R Trans Sym a h.
elim h; intros b H.
apply Trans  with b.
split. assumption.
apply Sym; assumption.
Qed.
```

Задачи из 4.2

В решениях задач этого раздела предполагается, что выполнена команда "`Require Import Relations_1.`".

4.2 (1)
```
Theorem Ex_a: forall (U:Type) (R: Relation U),
                              Preorder U R -> Reflexive U R.
Proof. intros U R H; elim H; auto. Qed.

Theorem Ex_b: forall (U:Type) (R: Relation U),
                            Equivalence U R -> Preorder U R.
Proof.
intros U R H.
apply Definition_of_preorder; elim H; auto.
Qed.

Theorem Ex_c: forall (U:Type) (R:Relation U),
                                 Equivalence U R -> PER U R.
Proof. intros; elim H;  auto with v62. Qed.

Definition eqrel (U:Type) (x y :U):Prop := x=y.
Theorem Ex_d: forall (U:Type), Reflexive U (eqrel U).
Proof. unfold Reflexive; red; auto. Qed.
```

4.2 (3)
```
Proof.
```

```
intros U R H a h; elim H; intros h1 h2.
elim h; intros b h3; apply h2 with b.
assumption.
apply h1; assumption.
Qed.
```

4.2 (4)
```
Proof.
intros U R h.
elim h; intros h0 h1; clear h.
apply Definition_of_equivalence; red in |-*; intros.
split; apply h0.          (* subgoal 1   *)
elim H; intros; clear H.  (* subgoal 2   *)
elim H0; intros; clear H0.
split; apply h1 with y; assumption.
elim H; auto.             (* subgoal 3   *)
Qed.
```

4.2 (6)
```
Proof. auto 10 with v62. Qed. (* proof_search_depth:=10 *)
```

4.2 (7) Доказывать это утверждение следует после завершения доказательства теоремы `Equiv_from_preorder` и добавления соответствующей подсказки командой "`Hint Resolve Equiv_from_preorder.`":

```
Proof. unfold same_relation. auto 10 with v62. Qed.
```

Задачи из 4.4

4.4 (1)
```
Proof. intros x y z H H0. transitivity y; assumption. Qed.
Proof. intros x y z H H0. rewrite H; assumption. Qed.
```

4.4 (3)
```
Proof.
intros k H; elim H; intro.
replace k with 0; apply f00.
replace k with 1; rewrite f10; apply f00.
Qed.
Proof. rewrite f10. rewrite f00. apply f00. Qed.
Proof. compute. apply f10. Qed.
```

4.4 (5)
```
Proof.
```

```
apply Definition_of_PER.
unfold Symmetric, eqeven. intros x y H. elim H. clear H.
intros H H0; split.
auto.
rewrite <- H;  assumption.
unfold Transitive, eqeven. intros x y z H H0. elim H.
intros H1 H2.
rewrite H1; exact H0.
Qed.
```

4.4 (6) Приведенное решение использует определение `eqrel` и предполагает доказанной теорему `sym_and_antisym`, позволяющую установить, что R содержится в отношении равенства. Последнее влечет транзитивность R.

```
Proof.
intros. red.
assert (contains U (eqrel U) R).
apply sym_and_antisym; auto.
intros. unfold contains, eqrel in H1.
assert (x=y). auto.
assert (y=z); auto.
rewrite H4; assumption.
Qed.
```

4.4 (7)
```
Proof.
intros.
rewrite com.
rewrite <- ass.
rewrite idp.
reflexivity.
Qed.
```

Задачи из 4.6

В решениях задач раздела 4.6 предполагаются выполненными команды:

```
Require Import Relations_1.
Require Import Relations_2.
```

4.6 (1)
```
Proof.
intros U R; red in |- *.
(* Оставить одну импликацию в заключении! *)
intros x y z H. elim H; auto with v62.
intros x0 y0 z0 H0 H1 H2 H3.
apply Rstar_n with y0; auto with v62.
Qed.
```

4.6 (2)
```
Proof.
intros U R x y H. elim H; auto with v62.
intros x0 y0 z H0 H1 H2. right; exists y0; auto with v62.
Qed.
```

4.6 (3) Предполагаются доказанными утверждения `Rstar_contains_R` и `Rstar_transitive`. Они добавляются в качестве дополнительных посылок с помощью тактики `generalize`, после чего удается раскрыть определения входящих в них констант.

```
Proof.
generalize Rstar_contains_R; intro T; red in T.
intros U R; unfold same_relation, contains in |- *.
split; intros x y H; elim H; auto with v62.
generalize Rstar_transitive; intro T1; red in T1.
intros x0 y0 z H0 H1 H2 H3.
apply T1 with y0; auto with v62.
intros x0 y0 z H0 H1 H2.
apply Rstar1_n with y0; auto with v62.
Qed.
```

4.6 (4) Предполагается доказанным утверждение `Rstar_transitive`.

```
Proof.
intros U R H; red in |- *.
intros x y H0; elim H0; auto with v62.
intros x0 y0 z H1 H2 H3.
generalize Rstar_transitive; intro T; red in T.
apply T with y0; auto with v62.
apply Rstar_n with x0; auto with v62.
Qed.
```

4.6 (5) Предполагается доказанным утверждение `Rstar_transitive`.

```
Proof.
unfold contains in |- *.
intros U R S H x y H0; elim H0; auto with v62.
generalize Rstar_transitive; intro T; red in T.
intros x0 y0 z H1 H2 H3; apply T with y0; auto with v62.
Qed.
```

4.6 (6) Предполагаются доказанными утверждения `Rstar_contains_R` и `Sstar_contains_Rstar`.

```
Proof.
intros U R S H.
apply Sstar_contains_Rstar.
generalize (Rstar_contains_R U S); auto with v62.
Qed.
```

4.6 (7) Предполагаются доказанными утверждения `Rstar_contains_Rplus` и `Rstar_transitive`.

```
Proof.
generalize Rstar_contains_Rplus; intro T; red in T.
generalize Rstar_transitive; intro T1; red in T1.
intros U R x y z H; elim H.
intros x0 H0; elim H0.
intros; exists y0; auto with v62.
intros; exists y0; auto with v62.
intros; exists y0; auto with v62.
split; [ try assumption | idtac ].
apply T1 with z0; auto with v62.
Qed.
```

Задачи из 5.6

В решениях задач раздела 5.6 предполагается выполненной команда: “`Require Import Arith.`”.

5.6 (1)
```
Proof. intros; simpl exp; auto. Qed.
```
```
Proof. exists exp. exact exp_is_exp. Defined.
```

5.6 (2) Со этими тремя утверждениями справляется тактика `auto`.

5.6 (3)
```
Proof. auto. Qed.
Proof. auto. Qed.
Proof.
intros a x; unfold el.
replace (last_index(add a x) - last_index(add a x)) with 0;
auto with arith.
Qed.
Proof. intros. elim a. auto. Qed.
```

5.6 (4)
```
Proof. auto. Qed.
Proof. auto. Qed.
Proof.
intros A f x m n. induction m.
rewrite iterate_0; auto with arith.
rewrite iterate_S. simpl. rewrite IHm; reflexivity.
Qed.
```

5.6 (5)
```
      (* Вспомогательное утверждение:                    *)
Lemma plus_minus_assoc: forall (m i:nat),
                                  i<=m -> 1+(m-i)=1+m-i.
Proof. intros. simpl plus. auto with arith. Qed.
      (* Основное утверждение:                           *)
Lemma el_add: forall (a:Vector) (x i:nat),
            i<= last_index a -> el (add a x) i = el a i.
Proof.
intros. unfold el. rewrite last_add1.
      (* Сократим на last:                               *)
assert
((iterate Vector cut_last (add a x) (1 + last_index a - i))
= (iterate Vector cut_last a (last_index a - i))).
2: rewrite H0; reflexivity.
      (* Перегруппируем слагаемые:                        *)
replace (1 + last_index a - i) with(1 + (last_index a - i)).
2: apply plus_minus_assoc; assumption.
```

```
      (* Выполним первую итерацию в л.ч. и упростим:      *)
rewrite plus_comm. rewrite iterate_plus. simpl.
      (* Восстановим вид a:Vector и обнаружим совпадение: *)
induction a. reflexivity.
Qed.
```

5.6 (6)
```
Proof. induction n;[auto | simpl; rewrite IHn; auto]. Qed.
Proof. induction n; auto. Qed.
Proof.
intros f n i H. elim H.
       (* Случай n = i.                                    *)
unfold el; rewrite last_index_of_f2v.
replace (i-i) with 0;
[ simpl iterate; apply last_of_f2v | auto with arith].
       (* Случай n = S m.                                  *)
intros. simpl. rewrite el_add. exact H1.
rewrite last_index_of_f2v. exact H0.
Qed.
```

7 Зачетные задания

(A)

1. `Theorem pr1: forall A B : Prop, A -> (B -> A).`

2. ```
   Theorem pr2:
   forall A B C : Prop, (A -> B) -> ((B -> C) -> (A -> C)).
   ```

3. `Theorem pr3: forall A B : Prop, A /\ B -> B /\ A.`

4. `Theorem pr4: forall A B : Prop, A \/ B -> B \/ A.`

5. `Theorem pr5: forall A B : Prop, (A\/ ~ B)/\B -> A.`

6. `Theorem pr6: forall A : Prop, ~~ (A \/ ~ A).`

7. ```
   Require Import Classical.
   Theorem pr7: forall a b : Prop, (~ a -> ~ b) -> b -> a.
   ```

8. ```
   Require Import Classical.
   Theorem pr8: forall a b : Prop, (a -> ~ b) -> b -> ~ a.
   ```

9. ```
   Require Import Classical.
   Theorem pr9: forall a b : Prop, (~ a -> b) -> ~ b -> a.
   ```

10. ```
    Theorem pr10:
    forall (A : Set) (a :A) (p : A -> Prop),
    (forall x : A, p x) -> (exists y : A, p y).
    ```

11. ```
    Theorem pr11: forall (Q: Type -> Type -> Prop),
    (exists x: Type, forall y: Type, Q x y) ->
    forall y: Type, exists x: Type, Q x y.
    ```

12. ```
    Theorem pr12: forall (Q: Type -> Type -> Prop),
    (exists x: Type, Q x x) -> exists (x y: Type), Q x y.
    ```

13. ```
    Theorem pr13: forall (Q: Type -> Type -> Prop),
    (exists (x y: Type), Q x y) -> exists (y x: Type), Q x y.
    ```

14. ```
    Theorem pr14: forall (P Q: Type -> Prop),
    (exists x: Type, P x -> Q x) ->
    (forall x: Type, P x) -> exists x: Type, Q x.
    ```
    ```

15.
```
Theorem pr15: forall (Q: Type -> Type -> Prop),
   (forall (x y: Type), Q x y) -> forall x: Type, Q x x.
```

(B)

(C)

Добавление. Некоторые примеры использования стандартных библиотек

Библиотека Arith

Она содержит много полезных фактов про операции `+,-,*` и отношения сравнения `<=,<,>` на типе `nat`, определенном индуктивно:

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

Сравнения. В качестве основного отношения сравнения выбрано `<=`, а отношения `<` и `>` определены через него.

```
Inductive le (n: nat) : nat -> Prop :=
    le_n: n <= n  |  le_S: forall m: nat, n <= m -> n <= S m.
Infix "<=" := le : nat_scope.
```

Если при доказательстве утверждения $\varphi(n, m)$ требуется разбор гипотезы $n \leq m$, то он происходит в соответствии с этим определением.[35] Надо установить $\varphi(n, n)$ и $\varphi(n, S\,m)$ при условии, что $\varphi(n, m)$ верно.

Факты про разрешимость условий, выраженных неравенствами, следует искать в модуле Arith.Compare_dec. Там они представлены как с помощью дизъюнкции, так и посредством булевой суммы. Например,

```
Definition decidable (P:Prop) := P \/ ~ P.
Theorem dec_le : forall n m, decidable (n <= m).

Definition zerop n : {n = 0} + {0 < n}.
```

Для реализации разбора случаев тактикой `elim` годятся обе формы. Так, команда "`elim (dec_le a b).`" разберет случаи $(a \leq b)$ и $\neg(a \leq b)$, а команда "`elim (zerop a).`" — случаи $(a = 0)$ и $(0 < a)$. Аналогичным образом можно использовать и "условные" факты, например,

```
Theorem not_eq : forall n m, n <> m -> n < m \/ m < n.
```

Команда "`elim (not_eq a b).`" произведет разбор случаев $(a < b)$, $(b < a)$, добавив при этом дополнительную подцель $(a \neq b)$.

[35] Команды "`elim H.`" или "`induction H.`", где `H: n<=m`.

Вычитание. Знак минус в типе `nat` означает усеченную разность, т.е. $m - n$ совпадает с обычной разностью при $m \geq n$ и равно 0 при $m < n$.

```
Fixpoint minus (n m:nat) : nat :=
  match n with
  | 0 => n
  | S k => match m with
           | 0 => n
           | S l => k - l
           end
  end
where "n - m" := (minus n m) : nat_scope.
```

Основные факты про усеченную разность собраны в модуле Arith.Minus, но их явно недостаточно для естественного использования. Например, непросто изменить порядок действий в выражении вида $a + b - c$ или перенести слагаемое из одной части равенства в другую. [36] Следующие леммы предназначены для упрощения таких преобразований.

```
(*
* Используем факты из библиотеки Arith:
* plus_minus (Coq.Arith.Minus)
* plus_comm (Coq.Arith.Plus)
* le_plus_minus (Coq.Arith.Minus)
* plus_assoc (Coq.Arith.Plus)
* minus_diag (Coq.Arith.Minus)
* not_le_minus_0 (Coq.Arith.Minus)
* lt_not_le (Coq.Arith.Lt)
* le_lt_or_eq (Coq.Arith.Lt)
*)

Require Import Arith.
```

[36] При подобных преобразованиях в типе `nat` приходится заботиться о том, чтобы для всех возникающих выражений вида $x - y$ выполнялось условие $x \geq y$. Иначе может нарушится корректность преобразования. Например, $(1 + 3) - 2 \neq (1 - 2) + 3$ в типе `nat`, т.к. левая часть равна 2, а правая — 3. Аналогично, из верного в типе `nat` равенства $1 - 2 = 0$ переносом в правую часть можно получить ложное равенство $1 = 2$.

```
Lemma to_rhp: forall (k m n:nat), m+k=n -> m = n-k.
Proof.
intros. apply plus_minus.
symmetry. rewrite plus_comm. assumption.
Qed.

Lemma to_lhp: forall (k m n:nat), n = m+k -> n-k = m.
Proof. symmetry. apply to_rhp; auto. Qed.

Lemma minus_to_rhp: forall (k m n:nat),
k <= n -> n-k = m -> n = m+k.

Proof.
intros. rewrite <- H0. clear H0.
rewrite (le_plus_minus k n) at 1;  auto with arith.
Qed.

Lemma minus_to_lhp: forall (k m n:nat),
k <= n -> m = n-k  -> m+k = n.
Proof. symmetry. apply minus_to_rhp; auto. Qed.

Lemma plus_minus_the_same: forall (k m n:nat), n+k-k = n.
Proof. intros. apply to_lhp; auto with arith. Qed.

Lemma minus_plus_the_same: forall (k m n:nat),
k<=n -> n-k+k = n.
Proof. intros. apply minus_to_lhp; auto with arith. Qed.

Lemma plus_minus2minus_plus: forall (k m n:nat),
k<=n -> n+m-k=n-k+m.
Proof.
intros. apply to_lhp. auto with arith.
rewrite <- plus_assoc.
rewrite plus_comm with m k.
rewrite plus_assoc.
rewrite minus_plus_the_same; auto with arith.
Qed.
```

Вот некоторые примеры применения развитой нами техники:

```
Lemma minus_positive: forall (k n:nat), (S k + n)-n<>0.
Proof.
intros; red. intro.
assert (S k +n = 0+n).
apply minus_to_rhp; auto with arith.
clear H; contradict H0. intro.
assert (S k = 0+n-n) as h.
apply to_rhp; auto with arith.
rewrite minus_diag in h; contradict h;
auto with arith.
Qed.

Lemma minus_0: forall (k n:nat), n - (k+n)=0.
intros. destruct k.
apply to_lhp;  auto with arith.
apply not_le_minus_0.
apply lt_not_le.
unfold lt; simpl. auto with arith.
Qed.
```

Перенос слагаемых из одной части неравенства в другую также вызывает затруднения. Здесь может помочь переход от неравенств к равенствам с помощью следующей леммы:

```
Lemma le_exists: forall (m n: nat),
m<=n -> exists k:nat, n = k + m.
Proof.
intros; exists (n-m). symmetry; apply minus_plus_the_same; auto.
Qed.
```

Вот пример ее применения:

```
Lemma le_to_rhp: forall (a b c:nat), a+b<=c -> a<=c-b.
Proof.
intros a b c H; generalize (le_exists (a+b) c H).
intro h; elim h; clear h; intros x H0.
rewrite H0. rewrite plus_assoc.
rewrite plus_minus_the_same; auto with arith.
Qed.
```

Стандартное определение нестрогого порядка `le` — индуктивное. В разделе 5.4 было предложено другое, рекурсивное определение:

$$x \leq n \Leftrightarrow (x = 0) \vee (x = 1) \vee \ldots \vee (x = n).$$

Установим их эквивалентность.

```
Fixpoint F (x n :nat) : Prop := match n with
                                  | 0 => ( x = 0)
                                  | S m => (F x m)\/ (x = S m)
                                end.

Theorem F_le: forall (x n :nat), F x n -> x <= n.
Proof.
intros x n H. induction n; simpl in H.
rewrite H; auto with arith.
elim H; auto with arith.
intro H0; rewrite H0; auto with arith.
Qed.

Theorem le_F: forall (x n :nat), x <= n -> F x n.
Proof.
intros x n H; induction n.
simpl; auto with arith.
simpl. elim (le_lt_or_eq x (S n) H); intro;
[ left; apply IHn; auto with arith |
  right; assumption ].
Qed.
```

Библиотека ZArith

Библиотека ZArith предназначена для работы с целыми числами в двоичном представлении (тип `Z`). В частности, в ней определено пространство имен `Z_scope`, которое используется для того, чтобы отличить обозначения арифметических операций в типе `Z` от аналогичных обозначений для типа `nat`. Подключение библиотеки и открытие пространства имен `Z_scope` осуществляется командами

```
Require Import ZArith.
Open Scope Z_scope.
```

После этого (1+2) будет обозначать 3:Z, а для обозначения аналогичного представления числа 3 в типе nat теперь надо использовать суффикс: (1+2)%nat.[37]

Тип Z. Он определяется следующим образом. Сначала в библиотеке PArith индуктивно определяется тип positive, состоящий из двоичных записей положительных натуральных чисел, с конструкторами xH (единица) и xO xI: positive -> positive (приписывание 0 и 1 справа). Индуктивное определение типа Z задает два вложения Zpos и Zneg типа positive в Z:

```
Inductive Z : Set :=
        Z0 : Z | Zpos : positive -> Z | Zneg : positive -> Z.
```

Несмотря на естественность определения, пользоваться непосредственно им не так удобно. В частности, соответствующий определению разбор случаев и стандартная индукция по n:Z приводят к необходимости рассуждать в весьма громоздкой смешанной теории ZArith + PArith. Проще использовать соответствующие стандартные заготовки, не выводящие за тип Z, например:[38]

```
(* См. Coq.ZArith.Zcomplements *)
Lemma Zcase_sign : forall (n : Z) (P : Prop),
       (n = 0 -> P) -> (n > 0 -> P) -> (n < 0 -> P) -> P.

(* См. Coq.ZArith.BinInt.Z: *)
Notation Zind := Z.peano_ind.
Theorem Z.peano_ind : forall P : Z -> Prop,
      P 0 ->
      (forall x : Z, P x -> P (Z.succ x)) ->
      (forall x : Z, P x -> P (Z.pred x)) -> forall z : Z, P z.
```

[37] По умолчанию открыто пространство имен nat_scope, поэтому если не открывать Z_scope специально, то (1+2) будет обозначать 3:nat, а 3:Z можно записать как 3%Z и как (1+2)%Z. Открытие нового именного пространства закрывает именное пространство, открытое ранее.

[38] К сожалению, принцип Zind имеет слишком сильные посылки, не учитывающие знак x, что ограничивает сферу его применения. Другие варианты заготовок принципа индукции для Z следует искать в Coq.ZArith.Wf_Z, Coq.ZArith.Zcomplements и Coq.ZArith.Zabs.

Сравнения. Отношения сравнения `<`, `<=`, `>`, `>=` в типе `Z` определяются через функцию сравнения `(x?=y)`, которая принимает одно из трех значений `Eq`, `Lt` или `Gt`, образующих трехэлементный тип `comparison`. Например,

```
Definition Z.lt (x y:Z) := (x ?= y) = Lt.
Notation Zlt := Z.lt.

Definition Z.le (x y:Z) := (x ?= y) <> Gt.
Notation Zle := Z.le.
```

Библиотека ZArith содержит много полезных лемм про сравнения, в основном сосредоточенных в `Coq.ZArith.BinInt`, `Coq.ZArith.Zorder` и `Coq.ZArith.auxiliary`, а также специальную тактику `elim_compare`, позволяющую разобрать случаи, соответствующие трем значениям функции сравнения `(x?=y)` (команда "`elim_compare x y.`").

Делимость, арифметика остатков, НОД, простота. Значительная часть заготовок, касающихся этих понятий, находится в дополнительной части библиотеки (`Coq.ZArith.Znumtheory`), которую надо загружать отдельно командой

```
Require Import Znumtheory.
```

Автоматизация. Библиотека ZArith имеет собственную базу данных подсказок `zarith`, которую следует использовать вместе с тактикой `auto` (команда "`auto with zarith.`").

Доступна тактика `ring` (команда "`ring.`"), которая умеет доказывать равенства, справедливые в силу аксиом кольца, и ее упрощающий вариант `ring_simplify` (умеет раскрывать скобки и приводить подобные члены).

Кроме того, имеется тактика `omega` (команда "`omega.`"), реализующая разрешающую процедуру для бесквантворных формул арифметики Пресбургера (пропозициональные комбинации равенств и неравенств термов, построенных с помощью сложения и вычитания).

О дублировании имен. В последних версиях Coq библиотека ZArith претерпела значительную реорганизацию. В версиях < 8.4 пользователи не имели прямого доступа к содержимому модулей, определенных

внутри файлов библиотеки (например, к содержимому модуля Z из библиотеки Coq.ZArith.BinInt, где введены основные определения и доказаны базисные факты теории). Для доступа пользователей к этим разработкам систематически использовались обозначения. Например, в файле BinInt.v введено множество обозначений такого рода:

```
Notation Zplus := Z.add.
Notation Zplus_comm := Z.add_comm.
```

В выборе обозначений (в основном) руководствовались следующим принципом: обозначение (Zplus_comm) получается из квалифицированного имени (Z.plus_comm) удалением точки. В версии 8.4 [39] открыт доступ к содержимому с использованием квалифицированных имен (<имя модуля> .<имя объекта>). При этом обозначения также сохранились в целях совместимости с предыдущими версиями.

Детальный разбор следующих примеров позволит освоиться и начать пользоваться библиотекой ZArith. Для удобства разбора примеры снабжены списками задействованных библиотечных фактов с указанием содержащих их разделов библиотеки. Рекомендуется держать соответствующие разделы документации перед глазами.

1. В качестве первого примера установим неравенство $x \leq x * x$ для всех целых чисел x. Достаточно с помощью Zcase_sign разобрать три случая: $x = 0$, $0 < x$ и $x < 0$.

Для первого хватает автоматического вычисления. В случае $0 < x$ устанавливаем неравенство $1 \leq x$ и помножаем обе его части на положительное x. Чтобы применить утверждение Zmult_gt_0_le_compat_r, обосновывающее корректность домножения, надо переписать целевое неравенство в виде $1 * x \leq x * x$. С помощью тактики pattern выделим первое вхождение x и заменим его на $1 * x$, а появившуюся подцель $1 * x = x$ впоследствии докажем автоматически.

Случай $x < 0$. Воспользуемся транзитивностью отношения $\leq$ и установим два неравенства: $x \leq 0$ (автоматически) и $0 \leq x * x$. Последнее неравенство фактически есть в библиотеке (sqr_pos), но сформулировано там с помощью $\geq$, поэтому цель следует переписать в том же виде с помощью Z.ge_le .

[39] Это последняя на момент написания этого текста версия.

```
Require Import ZArith.
Open Scope Z_scope.
(*
* Используем факты из библиотеки ZArith:
* Zcase_sign (Coq.ZArith.Zcomplements)
* Zmult_gt_0_le_compat_r (Coq.ZArith.Zorder)
* Zle_trans (Coq.ZArith.Zorder)
* Z.lt_le_incl (Coq.ZArith.BinInt.Z)
* Z.ge_le (Coq.ZArith.BinInt.Z)
* sqr_pos (Coq.ZArith.Zcomplements)
*)
Lemma le_x_xx: forall x:Z, x<= x*x.
Proof.
intro. apply (Zcase_sign x); intro H.
rewrite H; auto with zarith.
(*  case 0<x  *)
assert (1<=x); auto with zarith.
pattern x at 1; replace x with (1*x).
apply Zmult_gt_0_le_compat_r; auto with zarith.
auto with zarith.
(*  case x<0  *)
apply Zle_trans with 0. auto with zarith.
apply Z.ge_le. apply sqr_pos.
Qed.
```

2. Утверждение о четности произведения двух последовательных целых чисел может быть формализовано многими способами, использующими различные понятия, связанные с делимостью. Мы приводим четыре варианта формализации: с помощью предиката четности `Zeven`, из которой легко следует непосредственная формализация в языке арифметики, а также с использованием арифметики остатков и с помощью предиката делимости `(2 | n)`. Несмотря на то, что их формальные доказательства отражают одно и то же очевидное математическое рассуждение, процесс построения доказательства в каждом случае свой, весьма непохожий на остальные. Для использования предиката делимости в последнем случае требуется загрузить дополнительную библиотеку `Znumtheory`.

```
(*
* Zeven (Coq.ZArith.Zeven)
* Zeven_odd_dec (Coq.ZArith.Zeven)
* Zdiv2 (Coq.ZArith.Zeven)
* Zeven_div2 (Coq.ZArith.Zeven)
* Zmult_assoc (Coq.ZArith.BinInt)
* Zeven_Sn (Coq.ZArith.Zeven)
* Zmult_comm (Coq.ZArith.BinInt)
* Z_mod_plus (Coq.ZArith.Zdiv)
* Zeven_2p (Coq.ZArith.Zeven)
*)
Theorem even_n_Sn: forall n:Z, Zeven (n *(n+1)).
intros. elim (Zeven_odd_dec n); intro H.
(* case n - even *)
assert (exists k:Z, n=2*k).
exists (Zdiv2 n); apply Zeven_div2; assumption.
elim H0; intros k h; rewrite h.
rewrite <- Zmult_assoc; apply Zeven_2p.
(* case n - odd *)
assert (Zeven (n+1)). apply Zeven_Sn; assumption.
assert (exists k:Z, n+1=2*k).
exists (Zdiv2 (n+1)); apply Zeven_div2; assumption.
elim H1; intros k h; rewrite h.
rewrite Zmult_comm; rewrite <- Zmult_assoc. apply Zeven_2p.
Qed.
```

Здесь с помощью `Zeven_odd_dec` разбираются случаи четного и нечетного n. В первом из них n представляется в виде $2*k$, после чего множитель 2 выносится наружу и применяется лемма `Zeven_2`, утверждающая четность таких чисел. В случае нечетного n доказываем четность $n + 1$, после чего действуем аналогичным образом.

```
Corollary n_Sn_div2: forall n:Z, exists z, n*(n+1)=2*z.
Proof.
intros; exists (Zdiv2 (n*(n+1))).
apply (Zeven_div2); apply even_n_Sn.
Qed.
```

Используем арифметику остатков:

```
(*
* Z_mod_plus (Coq.ZArith.Zdiv)
* Z_modulo_2 (Coq.ZArith.Zeven)
* Zplus_assoc (Coq.ZArith.BinInt)
* Zplus_comm (Coq.ZArith.BinInt)
* Z_mod_mult (Coq.ZArith.Zdiv)
* Zmult_plus_distr_r (Coq.ZArith.BinInt)
* Zmult_permute (Coq.ZArith.BinInt)
*)
Theorem n_Sn_mod2_0: forall n:Z, (n*(n+1)) mod 2 = 0.
Proof.
intro n.
elim (Z_modulo_2 n); intro a.
(* case n=2*x *)
elim a. intros x H. rewrite H. rewrite <- Zmult_assoc.
generalize (x * (2 * x + 1)) as k; intro.
rewrite Zmult_comm. apply Z_mod_mult.
(* case n=2*x+1 *)
elim a. intros x H. rewrite H.
rewrite <- Zplus_assoc.
replace (1+1) with (2*1); auto with arith.
rewrite <- Zmult_plus_distr_r.
rewrite Zmult_permute.
generalize (n * (x + 1)) as k; intro.
rewrite Zmult_comm. apply Z_mod_mult.
Qed.
```

Лемма `Z_modulo_2` позволяет перебрать возможные варианты разложения числа n, т.е. $n = 2 * x$ или $n = 2 * x + 1$. В каждом из этих случаев разложение подставляется в целевое равенство, которое затем приводят к виду $(k * 2) \bmod 2 = 0$ и доказывают применением леммы `Z_mod_mult`.

```
Require Import Znumtheory.
(*
* Zmod_divide  (Coq.ZArith.Znumtheory)
*)
Corollary n_Sn_div2': forall n:Z, (2 | n*(n+1)).
Proof.
```

```
intros.
apply Zmod_divide; [auto with zarith | apply n_Sn_mod2_0].
Qed.
```

3. Следующие примеры демонстрируют предикаты простоты `prime` и взаимной простоты `rel_prime`. Предполагается загруженной библиотека `Znumtheory`.

```
(*
* prime (Coq.ZArith.Znumtheory)
* prime_intro (Coq.ZArith.Znumtheory)
* rel_prime  (Coq.ZArith.Znumtheory)
* Zis_gcd_intro (Coq.ZArith.Znumtheory)
*)
Lemma my_not_prime_1: ~ prime 1.
Proof.
intros H1. absurd (1 < 1). auto with zarith.
elim H1. auto.
Qed.

Lemma my_prime_2: prime 2.
Proof.
apply prime_intro; auto with zarith.
intros. unfold rel_prime.
apply  Zis_gcd_intro; auto with zarith.
intros.
assert (n=1). apply my_1; auto with zarith.
rewrite <- H2. assumption.
Qed.
```

4. Наконец, продемонстрируем использование тактики `omega`. В первой лемме она фактически делает всю работу, а во второй — выводит требуемую дизъюнкцию из оценки величины `(n mod 3)`.

```
Lemma with_omega_1: forall n:Z, 1<=n<2 -> n =1.
Proof. intros. omega. Qed.
Lemma with_omega_2: forall n :Z,
      (n mod 3 =0)\/(n mod 3 =1)\/(n mod 3 =2).
```

```
Proof.
intro. assert (0<= (n mod 3) <3);
[apply Z.mod_pos_bound; auto with zarith | omega].
Qed.
```

Добавление. Извлечение функциональных программ

Объекты типов `A:Set` несут в себе информацию смешанной природы — как вычислительного, так и логического характера. Наиболее отчетливо это видно в случае типов, имеющих вид зависимой суммы. Например, объект типа `{n:nat | exists x:nat, n=x+x}` содержит информацию о натуральном числе n вместе с доказательством p того, что n четно. Информация об n имеет вычислительный (алгоритмический) характер, — это алгоритм построения числа n из `O` с помощью приписывания четного числа символов `S`. Компонента p представляет логическую информацию, обосновывающую корректность этого алгоритма.

При обычном программировании логическая информация такого рода остается вне программы — в голове программиста или, в лучшем случае, в комментариях к программе. Важно, чтобы такая информация была и соответствовала действительности, но сам процесс исполнения программы к ней не обращается. Программирование в системе Coq позволяет пользователю (и заставляет его) манипулировать на формальном уровне не только с вычислительной, но и с логической информацией, а также обеспечивает автоматическую проверку корректности последней.

Доказательства свойств объекта в системе Coq обеспечивают наличие этих свойств у его вычислительной компоненты, поэтому сам объект может выступать в роли программы с доказательно верифицированными свойствами. Однако система Coq оказывается недостаточно эффективной в качестве вычислительной среды для исполнения программ. Это плата за необходимость манипулировать с логической составляющей, которую можно избежать, если выделить вычислительное содержание (computational content) объекта в виде отдельной программы на другом языке программирования, более приспособленном для вычислений.

Основным препятствием на этом пути служит несоответствие между богатой системой типов Coq и более бедными — у большинства распространенных языков программирования. В настоящее время это препятствие удалось преодолеть для трех языков функционального программирования: Ocaml, Haskell и Scheme. В системе Coq переключение языков осуществляется командой “`Extraction Language <язык>.`” (по умолчанию установлен Ocaml). Само извлечение осуществляется командой:

```
Extraction "<имя_файла>" <терм1> <терм2> ... <термN> .
```

В результате в текущем каталоге[40] будут созданы файл(ы) с кодом извлеченной программы, содержащей определения `<терм1> <терм2> ... <термN>` и все то, что необходимо для их вычисления. В случае Ocaml это два файла с заданным именем и расширениями `.mli` (интерфейс) и `.ml` (его реализация). Их можно компилировать обычным образом с помощью компилятора `ocamlc` или загрузить из интерактивной среды Ocaml (`toploop`) командой " `#use <имя_файла.ml>;;` ". [41]

В качестве примера рассмотрим извлечение программного кода для Ocaml из предложенной в разделе 5.5 реализации типа данных `Vector`. Извлечению подлежат функции "верхнего уровня" `init`, `add` и `el`, но вместе с ними автоматически будет извлечен код для всех функций и типов, участвующих в их построении. Это относится как к типам, определенным нами в 5.5 (напр., `Vector`), так и к стандартным типам системы Coq (`nat`, `prod`, `sigT`).

Предположим, что все утверждения про `Vector` из раздела 5.5 и задач 5.6(2), 5.6(3), 5.6 (4), 5.6 (5) уже доказаны.[42] Исполняем команду

```
Extraction "my_vector" init add el .
```

В текущем каталоге появляются файлы `my_vector.ml` и `my_vector.mli`.

С помощью запроса "`Pwd`" обнаруживаем, что текущим оказывается системный каталог Coq. Перемещаем файлы в другое место, более подходящее для экспериментов. Оттуда запускаем интерактивную среду Ocaml и загружаем файл `my_vector.ml`:

```
> ocaml
        Objective Caml version 3.09.2

# #use "my_vector.ml";;
type _ = Obj.t
type nat = O | S of nat
type ('a, 'b) prod = Pair of 'a * 'b
```

[40] Узнать текущий каталог можно с помощью запроса `Pwd` .

[41] Возможно также извлечение кода без создания файлов — в виде ответа на запрос "`Recursive Extraction <терм1> <терм2> ... <термN>`" .

[42] Если доказательство какого-нибудь факта из этого перечня еще не завершено, то следует вместо "`Proof. ... Qed.`" написать "`Admitted.`", что позволит отложить доказательство на будущее и временно считать недоказанное утверждение верным. Недоказанные утверждения влияют лишь на статус извлеченных программ, но не на сам процесс извлечения.

```
type ('a, 'b) sigT = ExistT of 'a * 'b
val projT1 : ('a, 'b) sigT -> 'a = <fun>
val projT2 : ('a, 'b) sigT -> 'b = <fun>
val minus : nat -> nat -> nat = <fun>
type vect = _
type vector = (nat, vect) sigT
val last_index : ('a, 'b) sigT -> 'a = <fun>
val val0 : ('a, 'b) sigT -> 'b = <fun>
val last : (nat, 'a) sigT -> 'b = <fun>
val cut_last : (nat, 'a) sigT -> (nat, 'a) sigT = <fun>
val iterate : ('a -> 'a) -> 'a -> nat -> 'a = <fun>
val init : 'a -> (nat, 'b) sigT = <fun>
val add : (nat, 'a) sigT -> 'b -> (nat, 'c) sigT = <fun>
val el : (nat, 'a) sigT -> nat -> 'b = <fun>
```

В процессе загрузки происходит компиляция и исполнение извлеченного кода. Среда реагирует перечислением определенных в нем объектов (последние 17 строк).

Обратим внимание на получившееся определение типа `nat`. Оно фактически повторяет соответствующее определение в системе Coq и никак не связано со стандартным для Ocaml типом целых чисел `int`. Для удобства дальнейшего использования запрограммируем два конвертора: из `nat` в `int` и обратно.

```
# let rec n2i = function O -> 0 | S m -> 1+(n2i m);;
val n2i : nat -> int = <fun>
# let rec i2n = function 0 -> O | n   -> S (i2n (n-1));;
val i2n : int -> nat = <fun>
```

Теперь можно создать тестовый вектор `test` = (3,5,7) с помощью `init` и `add`, а также проверить, что `el` правильно вычисляет его координаты:

```
# let test = add (add (init (i2n 3))(i2n 5)) (i2n 7);;
val test : (nat, 'a) sigT = ExistT (S (S O), <poly>)
# n2i (el test (i2n 1));;
- : int = 5
# n2i (el test (i2n 0));;
- : int = 3
# n2i (el test (i2n 2));;
- : int = 7
```

Чтобы все время не указывать конверторы явно, разумно ввести аналоги функций `init`, `add`, `el` для работы с типом `int`:

```
# let ml_init i = init (i2n i);;
val ml_init : int -> (nat, 'a) sigT = <fun>
# let ml_add v i = add v (i2n i);;
val ml_add : (nat, 'a) sigT -> int -> (nat, 'b) sigT = <fun>
# let ml_el v i = n2i (el v (i2n i));;
val ml_el : (nat, 'a) sigT -> int -> int = <fun>

# let test1 = ml_add (ml_add (ml_init 13) 15)17;;
val test1 : (nat, 'a) sigT = ExistT (S (S O), <poly>)
# ml_el test1 0;;
- : int = 13
# ml_el test1 1;;
- : int = 15
# ml_el test1 2;;
- : int = 17
```

Дальнейшие примеры извлечения программ и целых библиотек можно найти в “Reference Manual”.