МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет»

Институт математики, механики и компьютерных наук им. И.И.Воровича
Кафедра информатики и вычислительного эксперимента

Насека Андрей Игоревич

РЕАЛИЗЦИЯ ОБЪЕКТОВ ОБЩЕЙ АЛГЕБРЫ С ПОМОЩЬЮ СИСТЕМЫ COQ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению 02.04.02 – Фундаментальная информатика и информационные технологии, направленность программы «Компьютерные науки»

Научный руководитель –
проф., д.ф.-м.н. Пилиди Владимир Ставрович

Рецензент –
ст.преп., к.ф.-м.н. Абрамян Михаил Эдуардович

Ростов-на-Дону – 2019

СОДЕРЖАНИЕ

Введение

Данная работа является учебным пособием по мягкому введению в автоматическое доказательство теорем в Coq. Наибольшее внимание в ней уделено технологии создания тактик, которые позволяют проследить стратегию доказательства и автоматизировать этот процесс.

Для реализации данной задачи были выбраны объекты общей алгебры, так как они уже известны студентам младших курсов. В частности, основной структурой, с помощью которой показаны подходы к созданию алгоритмов автоматизации, является множество. Поэтому материал, представленный в этой работе, будет наиболее понятен читателю.

Множество является ключевой структурой в общей алгебре, так как все другие объекты реализовываются на его основе. Поэтому на их примере объяснены основные принципы формализации и автоматизации теорем, которые в последующем можно будет использовать при реализации более сложных структур общей алгебры, что также представлено в данной работе.

Ключевая цель диссертации - показать, как различные подходы к автоматизации могут влиять на процесс доказательства, и разобрать способы, с помощью которых можно эффективно формализовывать и доказывать теоремы с минимальной вовлеченностью человека в процесс доказательства.

Работа состоит из четырех частей. В первой представлена реализация базового множества. Показаны основные принципы формализации индуктивных определений и функций для математических объектов.

Во второй части рассматриваются способы доказательства теорем на примере факта о том, что операция вложенности множеств является отношением частичного порядка. Для этого было доказано,

что данное отношение является рефлексивным, транзитивным и антисимметричным.

Также в этом разделе наглядно показано, как реализованные теоремы могут быть использованы для доказательства других утверждений на примере канонического сравнения для множеств.

В третьем разделе представлены способы автоматического доказаетльства теорем. Цель - максимально упростить этот процесс и свести к минимому участие человека в нем.

В последнем четвертом разделе представлены несколько способов формализации более сложных объектов общей алгебры, таких как группы, основанные на базовом множестве. Показано, что автоматические тактики, определенные ранее, работают и с ними тоже.

Конечная цель работы выглядит следующим образом: человек разработал стратегию доказательства, а компьютер реализовал её, не нагружая программиста излишними деталями. Такой баланс между действиями человека и автоматизацией системы Coq очень важен и является ключевым фактором.

В работе была использована система автоматического доказательства теорем Coq, а также вспомогательная библиотека $ssreflect$, содержащая полезные принципы и стратегии доказательства теорем, которые продемонстрированы в процессе работы.

1. Реализация базового множества

В данном разделе будет представлена формализация множества. Множество - это интуитивно понятный объект, который не имеет четко сформулированного определения. В упрощенном техническом смысле - это набор элементов, необязательно похожих между собой.

Для простоты исследований в работе рассматриваются множества натуральных чисел. Хотя определение, которое здесь представлено, является обобщенным и может быть использовано для любого типа, т.е. мы можем создать множество с объектами любой природы.

В процессе работы показано, как правильно формализовывать объекты с помощью индуктивных определений. Для более лучшего восприятия использован механизм нотаций - введения специальных обозначений, которые пользователь хочет видеть в своих определениях.

Для множеств введен базовый набор функций, который нужен для написания теорем и их доказательств в системе Coq. В реализации представлен механизм использования рекурсии и стандартных средств разработки, необходимых для этого.

Основная цель раздела - подготовить фундамент, на основе которго будет происходить обучение основным приемам доказательства, выявлению общей стратегии и принципов, присущих утверждениям, которые связанны с объектами общей алгебры.

На пути к этой цели вводится синтаксис языка Coq и библиотеки $ssreflect$, а также тактики, с помощью которых необходимо вести доказательство. Для большей наглядности работа основных команд и тактик показана на конкретных примерах.

1.1. Индуктивное определение множества

В системе Coq индуктивное определение множества будет содержать следующую идею: необходимо определить вспомогательную структуру - **список** и наложить на нее ряд **ограничений**. Это можно реализовать разными способами, но для удобства использования был выбран следующий варинт:

```
Variable T : Type.

  Inductive BSet : Type :=
    | Empty : BSet
    | Elem : T -> BSet -> BSet
    .
```

Данная структура является классическим списком - он либо может быть пустым (конструктор $Empty$) или состоять из элемента типа T и списка, который в свою очередь также может быть либо пустым, либо содержать элемент и список (конструктор $Elem$). Данное определение является рекурсивным.

Теперь для того, чтобы наш список являлся множеством нам необходимо наложить на него два ограничения (они тоже могут быть сформулированы по-разному). Вот реализация данных ограничений в Coq:

- Первое ограничение говорит о том, что, если во множество добавить два одинаковых элемента типа T, то оно не должно измениться. Таким образом, данная реализация множества является **множеством без повторов**. Данное ограничение формализовано в виде аксиомы (использование аксиом в Coq является более удобным, чем учитывание этих ограничений сразу в определении, но позже при реализации более сложных структур будет ислользован альтернативный вариант определений):

```
Axiom Ident1 : forall (s : BSet) (x : T),
                 Elem x (Elem x s) = Elem x s.
```

- Второе ограничение говорит о том, что порядок добавления элементов во множество не играет никакой роли. Т.е. будем считать множество набором элементов, которые хранятся в произвольном порядке. Данное определение также представлено в виде аксиомы:

```
Axiom Ident2 : forall (s : BSet) (x y : T),
               Elem x (Elem y s) = Elem y (Elem x s).
```

Введем обозначения для базового множества с помощью нотации в Coq:

```
Notation "{ # }" := Empty.
Notation "{ # x }" := (Elem x { # }).
Notation "{ # x ; y ; .. ; z }" :=
          (Elem x (Elem y  .. (Elem z { # }) ..)).
```

Данная нотация работает рекурсивно для множества любой длины. После ее введения можно убедиться, что нотация работает корректно, с помощью команды для проверки типов в Coq - $Check$:

```
Check ({ # }).          (* Корректно *)
Check ({# 3}).          (* Корректно *)
Check ({# 3; 4}).       (* Корректно *)
Check ({# 3; 4; 5}).    (* Корректно *)
```

1.2. Базовые операции для множества

Теперь надо ввести основные операции, которые будут необходимы для формализации и доказательства теорем.

Одной из основных является вложенность одного множества в другое. Она понадобится, когда будет формализовываться теорема об отношении частичного порядка.

Помимо вложенности, реализованы дополнительные операции, например, вставка/удаление элемента (они будут необходимы при доказательстве теоремы об антисимметричности).

1.2.1. Принадлежность элемента множеству

Одной из главных операций для объектов общей алгебры является принадлежность элемента данному объекту. Данную операцию для множества мы определим с помощью рекурсивной функции:

```
Fixpoint In (x : T) (s : BSet T) : bool :=
  match s with
    | Empty => false
    | Elem y sub => (x == y) || (In x sub)
  end.
```

Алгоритм данной функции следующий: если множество является пустым, то элемент в нем содержаться не может, а если во множестве есть хотя бы один элемент, то происходит проверка на равенство целевого элемента с текущим из множества и, либо они должны быть равны, либо во множестве должен найтись другой равный данному.

Симметричной операций, основанной на данной, будет являться проверка на то, что данного элемента во множестве нет. Она определяется обычным отрицанием результата применения предыдущей функции:

```
Definition notIn (x : T) (s : BSet T) : bool :=
  (In x s) == false.
```

Для данной операции также введем обозначение с помощью нотации:

```
Notation "x /in s" := (In x s)
Notation "x /~in s" := (notIn x s)
```

Проверить корректность работы данных функций можно с помощью операции $Eval\ compute$ в Coq:

```
Eval compute in (7 /in { # }).          (* => False *)
Eval compute in (7 /in {# 4; 7; 6}).    (* => True  *)
Eval compute in (7 /in {# 4; 6; 9}).    (* => False *)
Eval compute in (7 /~in {# 4; 7; 6}).   (* => False *)
Eval compute in (7 /~in {# 4; 6; 9}).   (* => True  *)
```

1.2.2. Добавление и удаление элемента

Для эффективной работы со множеством необходимо реализовать операции добавления и удаления элемента из данного множества.

Добавление элемента в множество не должно наше множество превращать во множество с повторами, т.е. операция вставки должна быть корректной и сохранять структуру множества правильной.

```
Definition AddBSet (x : T) (s : BSet T) :=
  match (In x s) with
    | true => s
    | false => Elem x s
  end.
```

Для обеспечения этого условия используется проверка принадлежности данного элемента множеству, и если он не содержится в нем, то только тогда происходит добавление (с помощью конструктора $Elem$), в противном случае множество не меняется и остается прежним.

Удаление элемента из множества на Coq выглядит следующим образом:

```
Definition AddBSet (x : T) (s : BSet T) :=
  match (In x s) with
    | true => s
    | false => Elem x s
  end.
```

Удаление элемента происходит в два этапа. Первый этап: проверка множества на равенство нулю, если оно пустое, то и удалять из него нечего, а если в нем содержится хотя бы один элемент, то происходит сравнение этого объекта с удаляемым, и, в случае верного равенства, данный объект не включается во множество при сборке множества (множество собирается на рекурсивном возврате).

Для данных операций введем следующие обозначения с помощью нотации:

```
Notation "x =>> s" := (AddBSet x s)
Notation "s \ x" := (SubBSet s x)
```

Проверка показывает корректную работу данных функций:

```
Eval compute in (3 =>> { # }).       (* => { # 3 } *)
Eval compute in (3 =>> { # 1; 3 }).  (* => { # 1; 3 } *)
Eval compute in (3 =>> { # 1; 2 }).  (* => { # 1; 2; 3 } *)
-----------------------------------------------------
Eval compute in ({ # } \ 3).         (* => { # } *)
Eval compute in ({ # 1; 3 } \ 3).    (* => { # 1 } *)
Eval compute in ({ # 1; 2 } \ 3).    (* => {# 1; 2 } *)
```

1.2.3. Операция вложенности множеств

Одна из важных операций в теории множеств и общей алгебры - это вложенность одного объекта в другой. В Coq данная операция будет реализована рекурсивно с помощью Fixpoint:

```
Fixpoint subseteq (s1 s2 : BSet T) :=
    match s1 with
    | Empty => true
    | Elem x sub => (In x s2) && (subseteq sub s2)
    end.
```

Проверка вложенности одного множества в другое происходит по струтуре первого множества, которое вкладывается. Если данное множество является пустым, то, так как пустое множество вложенно в любое другое множество, в том числе и в пустое тоже, то в данном случае будем считать вложенность корректной операцией.

Если первое множество не является пустым, то мы перебираем каждый элемент из данного множество и проверяем, принадлежит ли этот элемент второму множеству. Т.е. здесь мы использовали уже определенную ранее функцию принадлежности элемента множеству.

Для данной операции также введем обозначение с помощью $Notation$:

```
Notation "s1 [= s2" := (subseteq s1 s2).
```

Проверка показывает, что данное определение функции является корректным:

```
Eval compute in ({ # } [= { # }).          (* => true *)
Eval compute in ({ # } [= { # 3 }).        (* => true *)
Eval compute in ({ # 3 } [= { # 4; 3}).    (* => true *)
Eval compute in ({ # 3 } [= { # 4; 5}).    (* => false *)
```

1.2.4. Мощность множества

Мощность множества - это проверка количества элементов в данном объекте. Реализация данной опреации в Coq ничем не отличается от реализации на любом другом языке программирования. Мы используем рекурсию, чтобы пройтись по структуре множества и на рекурсивном возврате считаем количество рекурсивных вызовов (т.е. количество элементов в множестве).

```
Fixpoint power (s : BSet) : nat :=
  match s with
    | Empty => 0
    | Elem _ sub => 1 + (power sub)
  end.
```

Обозначение для множества с помощью нотации:

```
Notation "| s |" := (power s).
```

И проверка корректности работы:

```
Eval compute in (|{ # }|).             (* => 0 *)
Eval compute in (|{# 3}|).             (* => 1 *)
Eval compute in (|{# 3; 4}|).          (* => 2 *)
Eval compute in (|{# 3; 4; 5}|).       (* => 3 *)
Eval compute in (|{# 3; 4; 1; 7}|).    (* => 4 *)
```

2. Формализация и доказательство теорем

В данном разделе будет показана формализация нескольких теорем для множества без автоматического доказательства. Основной теоремой данного раздела будет доказательство факта того, что операция вложенности множеств является отношением частичного порядка.

А также введение операции сравнения для множеств и доказательсво теоремы $reflect$ о каноничном сравнение для них (т.е. показать, что определенная нами функция сравнения работает правильно, и мы можем ее использовать в качестве базовой функции при доказательстве теоремы и переходить от пропозиционного сравнения к вычислительному).

Задача этого раздела показать, как формализуются теоремы в Coq и происходит их строгое доказательство. Несмотря на кажущуюся простоту некоторых теорем, строгое доказательство некоторых, даже простых фактов, может быть довольно сложным. Поэтому был выбран один факт, на примере которого будут разобраны ключевые моменты доказательства в Coq.

Для того, чтобы доказать основные теоремы об отношении частичного порядка и сравнении множеств, необходимо доказать ряд вспомогательных фактов-лемм, которые будут помогать в доказательстве.

Другой основной задачей данного раздела является выявление общих этапов доказательства, которые повторяются во многих похожих теоремах, что позволит после автоматизировать доказательство посредством введения новых автоматических тактик.

2.1. Отношение частичного порядка для вложенности множеств

Бинарное отношение R на множестве S называется отношением частичного порядка, если выполняются следующие условия:

- Отношение R является **рефлексивным**:

  $\forall s \in S,\ s\ R\ s.$

- Отношение R является **транзитивным**:

  $\forall s1,\ s2,\ s3 \in S,\ s1\ R\ s2\ and\ s2\ R\ s3\ \Rightarrow\ s1\ R\ s3.$

- Отношение R является **антисимметричным**:

  $\forall s1,\ s2 \in S,\ s1\ R\ s2\ and\ s2\ R\ s1\ \Rightarrow\ s1\ =\ s2.$

Так как целью доказательства является установление того факта, что отношение вложенности множеств является отношением частичного порядка, то необходимо показать, что оно является рефлексивным, транзитивным и антисимметричным. В терминах нашей операции в Coq эти теоремы будут формулироваться следующим образом:

```
1) Рефлексивность:
    forall (s : BSet T),
            s [= s.
2) Транзитивность:
    forall (s1 s2 s3 : BSet T),
            s1 [= s2 -> s2 [= s3 -> s1 [= s3.
3) Антисимметричность:
    forall (s1 s2 s3 : BSet T),
            s1 [= s2 -> s2 [= s1 -> s1 = s2.
```

2.1.1. Лемма о вложенности множеств

Для доказательства теоремы о рефлексивности необходимо доказать один вспомогательный факт:

```
Lemma SaveSubSet : forall (s1 s2 : BSet T) (x : T),
                   s1 [= s2 -> s1 [= Elem x s2.
```

Данная лемма говорит о том, что если мы имеем два множества, и одно из них вложено в другое, то при добавлении элемента во второе множество вложенность должна сохраниться. Т.е. фактически

мы доказываем независимость операции вложенности от добавления элемента во второе множество.

Напротив, если бы мы добавляли элемент в первое множество, то операция множества не обязательно должна была сохраниться, т.к. мы могли добавить элемент, которого не было во втором множестве, и тогда вложенность бы нарушилась.

Доказательство данного утверждение будет проводиться по индукции по структуре первого множества. Индукция в Coq выполняется тактикой $elim$:

```
elim => /= // x s1 H1 s2 y H2.
```

База идукции докажется автоматически, т.к. она будет тривиальной. А индукционное предположение придется доказывать вручную. После применения данной операции мы будем иметь две гипотезы и одну цель:

```
H1  : forall (s2 : BSet T) (x : T),
             s1 [=s2 -> s1 [=Elem x s2
H2  : (x /in s2) && (s1 [=s2)
Goal: ((x == y) || x /in s2) && (s1 [=Elem y s2)
```

Как видно, в цели мы имеем вычислительные логические операции типа $bool$: && - логическое И, || - логическое ИЛИ, == - логическое сравнение. С типом $bool$ доказывать теоремы достаточно трудно, большинсво теорем доказываются посредсвом типа $Prop$ - пропозиционный логический тип.

Для того, чтобы перейти от типа $bool$ к типу $Prop$ необходимо применить теорему $reflect$, которая доказана для каждой отдельной логической операции типа $bool$ в библиотеке $ssreflect$.

Например, для && применение теоремы $andP$ для цели $Goal$ выглядит следующим образом:

```
case: andP => /= //;
unfold not => V1; apply: falseP;
apply: V1.
```

Здесь происходит добавление теоремы $andP$ в секцию известных гипотез. После этого с помощью тактики $case$ происходит разложение теоремы $andP$ на два случая:

```
 Goal (&& => /\) -> true
~Goal (&& => /\) -> false
```

Если цель с заменной пропозиционной логической операцией И истинна, то должна следовать логическая истинна типа $bool$. И наоборот, если ложна, то следует ложь типа $bool$.

Первое утверждение Coq в состоянии сам доказать автоматически, а вот второе необходимо преобразовать вручную. Так как в правой части второго утверждения имеется логичесий not, то его нужно раскрыть (точнее посмотреть на внутреннее устройство функции not. После этого мы получаем гипотезу, ту, которую мы хотели, из которой следует $False$ и в цели $false$. Таким образом, мы имеем следующую конструкцию:

```
V1 : G -> False
Goal : false
```

так как $false$ в цели здесь пропозиционный, мы также применяем теорему $reflect$ по переходу от вычислительного $fasle$ типа $bool$ к пропозиционному $False$. В итоге мы имеем следующую конструкцию:

```
V1 : Goal (&& => /\) -> False
Goal : False
```

Данная конструкция подчиняется правилу математической логики высказываний Гильберта - основное правило вывода: если мы имеем выводимую формулу A и выводимую формулу $A- > B$, то формула B также выводима.

С помощью этого правила мы сможем сделать следующую вывод: так как мы имеем выводимую формулу $A- > B$ и нам нужно доказать формулу B, то вместо этого нам достаточно доказать, что

выводима формула A и тогда, по основному правилу вывода логики Гильберта, будет доказана и выводимость формулы B.

Записывать условно данную тактику мы будем следующим образом, где $|-$ - знак секвенции (слева от знака располагаются гипотезы, а справа цели):

```
A -> B |- B => |- A
```

Поэтому мы можем применить это правило для данной ситуации с помощью тактики $apply$ в Coq:

```
apply: V1.
```

Точно также мы применили до этого теорему $falseP$ для перехода к пропозиционному $False$ (данная теорема выглядит достаточно просто $False -> false$. Видно, что она также подчиняется основному правилу вывода).

После применнения данной последовательности тактик и переходов мы добились того, что теперь вместо логического И типа $bool$ в цели появилось пропозиционное И:

```
Старая Цель: (x == y) || x /in s2 && s1 [=Elem y s2
Новая  Цель: (x == y) || x /in s2 /\ s1 [=Elem y s2
```

Теперь цель представляет собой набор из двух целей, связанных логической операцией И (конъюнкцией). В данном случае стратегия всегда одинаковая - для того чтобы конъюнкция двух формул была истинна необходимо и достаточно, чтобы каждая из формул была истинна. Поэтому необходимо разбить цель на две подцели, и доказать истинность каждой по отдельности. Данная операция в Coq выполняется с помощью тактики $split$: После ее применения мы получим две цели:

```
Goal 1: (x == y) || x /in s2
Goal 2: s1 [=Elem y s2
```

В первой цели есть логическая операция ИЛИ типа $bool$. Аналогично операции И необходимо применить теорему $reflect$ для перехода к пропозиционному ИЛИ. Переходы осуществляются таким же набором тактик за исключением применения базовой теоремы: вместо теоремы $andP$ используется orP. После этого цель преобразуется к следующему виду:

```
Goal 1: x == y \/ x /in s2
```

Далее, мы имеем две формулы связанных логической операцией ИЛИ (дизъюнкцией). Для того, чтобы данная формула была истинно необходимо и достаточно, чтобы одна из подформул была истинна. Поэтому необходимость в доказательстве двух формул отпадает. В данном случае в секции гипотез мы имеем следующий факт:

```
H2 : (x /in s2) && (s1 [=s2)
```

Гипотеза $H2$ содержит две формулы, связанные конъюнкцией и левая из формул как раз совпадает с правой подформулой в цели. Поэтому нам необходимо как-то извлечь из гипотезы $H2$ левую подформулу и применить ее к правой подцели.

Для этого также необходисо применить теорему $reflect$ для логического И. Но последовательность тактик для применения данной теоремы в гипотезе измениться:

```
case: andP H2 => // H2 _. destruct H2 as [H2 H3].
```

Первым делом необходимо опустить гипотезу в секцию вывода формул (т.е. в цель), а также теорему $andP$. После этого необходимо разложить теорему $andP$ на два случая, как и в случае ранее. Один из случаев докажется автоматически с помощью Coq, а второй случай преобразуется к следующему виду:

```
Goal 1 : x /in s2 /\ s1 [=s2 -> true -> x /in s2
```

Как видно цель теперь состоит из трех последовательных формул: $A- > B- > C$. Формула B является тривиальной, поэтому ее можно опустить из рассмотрения. А формула A как раз преобразовалась к нужному виду. Ее необходимо поднять в секцию гипотез с помощью операции $=>$. Данную операцию можно применить после каждой тактики точно так же как и операцию $:$ - она наоборот, опускает гипотезу в секцию доказательства (целей). В итоге гипотеза преобразуется к следующему виду:

```
H2 : x /in s2 /\ s1 [=s2
```

Теперь гипоза состоит из двух подформул , связанных пропозиционной конъюнкцией. Так как гипотеза истинна, значит каждая из подформул является истинной. Поэтому мы можем разбить данную гипотезу на две с помощью операции $destruct$ на гипотезы $H2$ и $H3$.

Теперь необходимо в цели взять только правую часть дизъюнкции. Это делается с помощью тактики $right$ в Coq (аналогично тактика $left$ - оставляет только левую часть дизъюнкции). После этого мы имеем:

```
H2 : x /in s2
Goal 1: x /in s2
```

Таким образом, необходимо доказать цель, которая уже формально доказана в гипотезе. Данное доказательство можно выполнить как с помощью тактики $apply$, так и просто автоматически.

В результате чего первая цель будет доказана. Переходим ко второй цели. Мы имеем следующие факты:

```
H1 : forall (s2 : BSet T) (x : T),
     s1 [=s2 -> s1 [=Elem x s2
H2 : x /in s2 && s1 [=s2
Goal 2: s1 [=Elem y s2
```

Как видно первая гипотеза $H1$ и цель имеют следующую структуру:

```
H1: A -> B |- Goal 2: B
```

Видно, что данная структура подчиняется основному правилу вывода. Поэтому мы можем заменить нашу цель B на формулу A и доказывать ее. Получаем:

```
H2 : x /in s2 && s1 [=s2
Goal 2: s1 [= s2
```

Теперь мы получили ту же самую ситуацию, что и с первой подцелью: мы имеем гипотезу, состоящую из двух формул, связанных конъюнкцией и правая из них совпадает с целью. Применив те же самые рассуждения, мы докажем данную цель и всю теорему. В конце мы увидим сообщения о том, что целей больше нет и все доказано, после чего мы можем пользоваться этой леммой при доказательстве других фактов.

```
No more subgoals. (* секция доказательства *)
```

Данная лемма показала общие принципы доказательства по индукции, а также использование переходов между пропозиционными и логическими операциями типа $bool$ с использованием теорем $reflect$. Как будет видно дальше, в других теоремах принципы доказательства будут очень похожи.

Отличия будут, но основная направляемость и последовательность действий будут схожи. И это будет использовано при автоматизации доказательств в третьем разделе.

2.1.2. Теорема о рефлексивности

Теперь необходимо доказать теорему о рефлексивности операции вложенности множеств. В Coq данная теорема будет выгляеть следующим образом:

```
Theorem Reflexivity : forall (s : BSet T), s [= s.
```

Доказательство следующей теоремы будет проходить по следующему алгоритму: необходимо рассмотреть случаи, когда s является пустым и когда не пустым множеством. Т.е. нам необходимо рассмотреть варианты, узнать, с помощью какого конструктора было образовано данное множество - $Empty$ или $Elem$. Для этого мы применим тактику $case$, которая в сущности и разбивает на варианты.

```
case => // /= x sub.
```

По сути $case$ в данном случае будет выполнять ту же функцию, что и тактика $elim$, за исключением того, что при использовании индукции мы получаем дополнительную гипотезу, а при разбивке на случаи мы лишаемся такой возможности. Поэтому чисто с технической точки зрения все равно какую тактику мы будем использовать в данном случае (грубо говоря $elim$ сильнее $case$), если мы используем индукцию и будем знать больше информации, чем нам необходимо, от этого доказательство хуже не станет.

Поэтому при автоматизации доказательства мы воспользуемся этим фактом, и в тех случаях, где нужно разбивать на случаи и применять тактику $case$, мы будем использовать $elim$, что позволит нам не вдаваться в анализ и проводить доказательства по единому шаблону.

После применения тактики $case$ первый случай, когда множество является пустым, доказался автоматически. Для этого мы применяли специальные тактики:

- /= - вычислительная тактика, которая позволяет максимально все упростить и вычислить, что возможно, если мы имеем в доказательстве обычные функции реализованные с помощью $Fixpoint$.
- // - тактика, которая пытается автоматически доказать теоремы, если это возможно с помощью стандартных средств в Coq.

Вторая основная цель, имеет следующий вид:

```
Goal : ((x == x) || x /in sub) && (sub [=Elem x sub)
```

Как видно, в данном случаи мы не имеем ни одной гипотезы. Поэтому доказательство сводиться к единственному случаю - применить теорему $reflect$ к операции конъюнкции и разбить цель на два случая с помощью тактики $split$. Аналогичные действия мы проводили при доказательстве предыдущей леммы. После применения данной последовательности тактик мы будем иметь две цели:

```
Goal 1 : (x == x) || x /in sub
Goal 2 :  sub [=Elem x sub
```

Как видно в первой цели, мы имеем дизъюнкцию типа $bool$, необходимо опять перейти к пропозиционной дизъюнкции и после этого можно заметить, что левая подформула является тривиальной $x == x$. Это истинное утверждение, поэтому доказательство данной цели мы будем проводить только по левой части, а правую откинем, так как мы уже выяснили ранее, что для доказательства формулы с дизъюнкцией необходимо и достаточно, чтобы всего лишь одна из подформул была истинна. В итоге первая цель будет полностью доказана.

Вторую цель с первого взгляда непонятно как доказывать, но если присмотреться повнимательнее, то станет ясно, что данная структура очень похожа на лемму, которая была доказана в предыдущем разделе. Только здесь множества являются одинаковыми. Т.е. необходимо доказать, что список будет вложен в себя же, и если добавить элемент, то вложенность сохранится.

Но для применения леммы нам необходим факт того, что данный список sub вложен в себя же. Данный факт мы можем получить, применив индукцию. Ее можно применить сейчас, либо, чтобы упростить программу, можно вначале вместо $case$ использовать индукцию

$elim$. Таким образом, процесс доказательства останеться таким же, но мы получим в свое распоряжение дополнительную гипотезу:

```
H : sub [=sub
```

Ее мы и используем для доказательства данной теоремы. Это можно сделать двумя способами. Посмотрим формально на логическую схему имеющихся фактов:

```
A := sub [= sub
B := sub [= Elem x sub
SaveSubSet : A -> B
H : A
Goal 2 : B
```

Так как лемма $SaveSubSet$ доказана для любых двух множеств $s1$ и $s2$, а также для любого элемента x, то вместо $s1$ и $s2$ мы можем выбрать одно и то же множество sub, так как по условию леммы она работает для любых множеств, в том числе и для одинаковых, а в качестве элемента x будет тот же самый элемент x.

Первый путь доказательства - строго по основному правилу логики Гильберта: мы имеем выводимую формулу $A- > B$ и A, поэтому мы можем вывести формулу B:

```
apply SaveSubSet in H.
```

Таким образом, гипотеза и цель будут одинаковы, поэтому цель будет доказана автоматически (из соображений рефлексивности доказываем себя через себя же).

Второй способ - использование основного правила вывода в обратную сторону, как мы делали это при доказательстве леммы, т.е. если выводима формула $A- > B$ и необходимо вывести B, то мы можем вместо B вывести A:

```
apply: SaveSubSet.
```

После этого теорема о рефлексивности будет полностью доказана. Мы увидели, что в предыдущей лемме и в данной теореме есть много общих тактик в доказательстве, и стратегия практически ничем не отличается. Отличия только в использовании леммы для доказательства теоремы, но это уже конкретное точечное действие, которое отражает специфику данной теоремы.

В перспективе необходимо получить автоматизированную систему, которая будет автоматически производиить одинаковые действия, повторяющиеся во многих теоремах, а человек при доказательстве теоремы сосредоточится именно на главных аспектах - общей логике доказательства.

2.1.3. Теорема о транзитивности

Теорема о транзитивности в Coq формализуется следующим образом:

```
Theorem Transitivity : forall (s1 s2 s3 : BSet T),
                       s1 [= s2 -> s2 [= s3 -> s1 [= s3.
```

Данная теорема не будет исключением, она также доказывается по индукции. После применения тактики $elim$ и переноса всех гипотез в свою секцию мы получаем следующую схему доказательства:

```
iH   : forall s2 s3 : BSet T,
       s1 [=s2 -> s2 [=s3 -> s1 [=s3
H1   : (x /in s2) && (s1 [=s2)
H3   : s2 [=s3
Goal : (x /in s3) && (s1 [=s3)
```

Как видно в цели и гипотезе можно применить теорему $reflect$ о переходе к пропозиционным конъюнкциям, после чего можно разбить цель на две подцели с помощью тактики $split$, а гипотезу разбить на две с помощью тактики $destruct$:

```
iH     : forall s2 s3 : BSet T,
         s1 [=s2 -> s2 [=s3 -> s1 [=s3
H1     : x /in s2
H2     : s1 [=s2
H3     : s2 [=s3
Goal 1 : x /in s3
Goal 2 : s1 [=s3
```

В данной схеме мы не имеем гипотезы, которая могла бы строго доказать первую цель. Но если внимательно проанализировать имеющиеся данные, то можно сделать вывод: видно что мы имеем гипотезу $H2$, в которой известно, что элемент $x \in s2$, а в гипотезе $H3$ мы знаем, что $s2$ вложено в $s3$. Исходя из логики, очевидно, что элемент x также должен принадлежать и множеству $s3$, в противном случае нарушилась бы вложенность $s2$ в $s3$, что противоречило бы гипотезе $H3$.

Поэтому необходимо доказать дополнительную лемму, которая обобщает и формализует данный факт:

```
Lemma SubInTwo : forall (s1 s2 : BSet T) (x : T),
            x /in s1 -> s1 [= s2 -> x /in s2.
```

Доказательство данного факта мы опустим, так как в нем нет ничего нового - стандартная индукция с последующим применением теоремы $reflect$ и уже известных тактик.

Теперь, когда мы знаем следующий факт: если элемент принадлежит вкладываемуму множеству, то он принадлежит и тому, в которое множество вкладывается - мы можем доказать первую цель теоремы о транзитивности:

```
by apply SubInTwo with (x := x) in H3.
```

Указание $with$ в тактике $apply$ позволяет уточнить системе, что подставляется вместо параметров леммы, это бывает полезно, когда Coq не может определить это автоматически ил есть несколько вариантов применения данной леммы.

Указание by - это аналог $//$ - попытка автоматически доказать теорему в случае тривиальных фактов и целей.

После этого остается доказать вторую цель. Проанализируем имеющуюся схему доказательства:

```
iH     : A -> B -> C
H2     : A
H3     : B
Goal 2 : C
```

Видно, что мы имеем выводимые формулы $A -> B -> C$, A и B, поэтому по основному правилу вывода логики Гильберта выводима формула C. Здесь можно применить тактику $apply$ как к гипотезе $H3$, так и к цели (как при доказательстве теоремы о рефлексивности). После этого теорема о транзитивности полностью доказана.

2.1.4. Лемма о разбивке множества

Теорема об антисимметричности очень сложна в доказательстве, так как для этого необходимо было доказать несколько вспомогательных фактов и пришлось ввести дополнительные операции, такие как вставка и удаление элементов.

Доказательство всех теорем описано не будет, с ним можно ознакомиться в репозитории данной работы. Опишем только сами теоремы и кратко поясним основные моменты доказательства.

Одной из ключевых является лемма о разбивке множества. Ее формализация на Coq выглядит следующим образом:

```
Lemma SplitIntoParts : forall (s1 s2 : BSet T) (x : T),
                       s1 [= s2 -> s2 [= Elem x s1
                       -> or (s2 = s1) (s2 = Elem x s1).
```

Данная лемма является довольно интересной, так как ее суть состоит в ручной разбивке множества на несколько независимых вариантов (своего рода ручной $destruct$, который позволит разбить гипотезу при доказательстве теоремы об антисимметричности).

Мы имеем цепочку из двух последовательных вложений:

- s1 [= s2
- s2 [= Elem x s1

Так как в первом вложении и во втором есть множество $s2$, то по свойству транзитивности, которое было доказано ранее

s1 [= s2 [= Elem x s1

мы можем сделать вывод, что множество $s2$, являясь посредником между $s1$ и $Elem\ x\ s1$, может либо совпадать со множеством $s1$ - это самое минимальное множество, либо быть $Elem\ x\ s1$ - это самое максимальное множество, других вариантов для множества $s2$ нет.

Доказательство данной теоремы происходит также по индукции, но после ее применения, необходимо применить ряд дополнительных лемм для полного доказательства. Приведем последовательность необходимых для этого фактов. После применения индукции и стандартных преобразований с теоремой $reflect$ мы имеем следующую схему доказательства:

```
H0     : s2 [={ # x}
Goal 1 : s2 = { # } \/ s2 = { # x}
Goal 2 : s2 = Elem y s1 \/ s2 = Elem x (Elem y s1)
```

Для доказательства первой цели необходимо формализовать и доказать следующий факт:

```
forall (s : BSet T) (x : T), s [= { # x } ->
                 or (s = { # }) (s = { # x }).
```

Формально это вспомогательный факт, который учитывает имеющуюся гипотезу $H0$ и показывает, что лемма о разбивке множества выполняется, если множество является пустым.

Для доказательства второй цели необходим следующий факт:

```
forall (s : BSet T) (x : T), or (x /in s) (x /~in s).
```

Фактически - это указание по разбивке на два несовместных варианта - ручной $case$. Мы показываем, что существует всего два варианта: либо элемент принадлежит множеству, либо нет. Третьего не дано.

После того, как мы разобьем наше доказательство на два варианта - мы будем иметь две одинковые цели, но разные гипотезы: в первом случае элемент y будет принадлежать множеству $s2$, а во втором не будет.

Применить данную разбивку можно с помощью следующих тактик:

```
move: EL7 => H4. pose (H5 := H4 s1 y). move: H5 => H5.
destruct H5 as [H5 | H5].
```

Тактика $pose$ позволяет подставлять в теоремы, содержащие $forall$ вместо параметров конкретные переменные, делая таким образом копии теорем на кокретных аргументах.

```
move: EL7 => H4. pose (H5 := H4 s1 y). move: H5 => H5.
destruct H5 as [H5 | H5].
```

Теперь при доказательстве второй цели мы имеем следующую схему доказательства:

```
H : forall (s2 : BSet T) (x : T),
    s1 [=s2 -> s2 [=Elem x s1 -> s2 = s1 \/ s2 = Elem x s1
H1 : s2 [=Elem x (Elem y s1)
H2 : y /in s2
H3 : s1 [=s2
H4 : y /in s1
Goal 2_1 : s2 = Elem y s1 \/ s2 = Elem x (Elem y s1)
```

Мы хотим применить предположение индукции, но видно, что оно определено для любого множества $s2$, но множество $s1$ там конкретное, и в предположении видно, что $s2 \; [= \; Elem \; x \; s1$, а в гипотезе известно, что $s2 \; [= \; Elem \; x \; (Elem \; t \; s1)$.

Поэтому необходимо доказать следующее вспомогательное утверждение:

```
forall (s : BSet T) (x : T), x /in s -> s = Elem x s.
```

Если элемент принадлежит множеству, то тогда он при добавлении этого элемента в данное множество не измениться, так как у нас множество определены без повторов.

В гипотезе $H2$ известно, что элемент y принадлежит множеству $s2$, поэтому мы сможем применить формализованнный ранне факт, заменив $s2 \; [= \; Elem \; x \; (Elem \; y \; s1)$ на $s2 \; [= \; Elem \; x \; s1$. После чего можно применить предположение индукции, получая тем самым новую гипотезу:

```
H3 : s2 = s1 \/ s2 = Elem x s1
```

Как видно, новая гипотеза содержит пропозиционную дизъюнкцию, поэтому рабиваем ее на случаи и каждый рассматриваем в отдельности. Каждый из этих случаев доказывается уже известными методами, поэтому подробности доказательства опустим.

Переходим ко второй цели, для нее потребуется довольно хитрый прием.

```
iH : forall (s2 : BSet T) (x : T),
     s1 [=s2 -> s2 [=Elem x s1 ->
     s2 = s1 \/ s2 = Elem x s1
H2 : s1 [=s2
H3 : s2 [=Elem y (Elem x s1)
```

Нам необходимо исключить элемент из множества, чтобы наша гипотеза удовлетворяла условию индукции. Если исключить элемент из обоих множеств, то их вложенность сохраниться.

Второй факт говорит нам о том, что если элемент не принадлежит первому множеству, и мы знаем факт того, что это множество вложено в другое, то мы можем смело исключить данный элемент из второго множества, при этом вложенность полностью сохранится.

```
forall (s1 s2 : BSet T) (x : T),
        s1 [= Elem x s2 -> s1 \ x [= s2.

forall (s1 s2 : BSet T) (x : T),
        x /~in s1 -> s1 [= s2
        -> s1 [= (s2 \ x).
```

После применения данных лемм к гипотезам $H2$ и $H3$ получаем:

```
H2 : s1 [=(s2 \ y)
H3 : s2 \ y [=Elem x s1
```

Далее, они обе подпадают под индукционное предположение, если в качестве параметра $s2$ мы возьмем множество $s2 \setminus y$. После этого индукционное предположение также разбивается на случаи, в каждом из которых применяется одна и та же стратегия. Покажем ее для первого случая:

```
H3   : s2 \ y = s1
Goal : s2 = Elem y s1 \/ s2 = Elem x (Elem y s1)
```

Нам необходимы две леммы. Первая говорит нам о том, что если известно, что два множества равны между собой, то в таком случае при добавлении в каждое по одинаковому элементу равенство сохраниться.

```
forall (s1 s2 : BSet T) (x : T),
        s1 = s2 -> Elem x s1 = Elem x s2.
```

А вторая лемма говорит о том, что если элемент принадлежит множеству, то если его удалить из него, а потом снова добавить, то ничего не изменится. Это работает только тогда, когда элемент изначально был во множестве, иначе удаление сработает в холостую, ничего не удалив, и при последующем добавлении элемента множество измениться, так изначально там данного элемента не было, а теперь появился.

```
forall (s : BSet T) (x : T),
        x /in s -> Elem x (s \ x) = s.
```

После применения данных лемм к гипотезе $H3$ мы получим, что данная гипотеза совпадает с левой подформулой цели. После чего данная цель будет доказана, вторая доказывается симметрично, за исключением того, что гипотеза $H3$ совпадет с правой подформулой цели и доказательство будет вестись по правой части из соображений рефлексивности доказательства.

После того как данная лемма будет полностью доказана, ее можно применить в доказательстве теоремы об антисимметричности.

2.1.5. Теорема об антисимметричности

Последняя теорема, которая необходима для доказательства того, что операция вложенности множеств является отношением частичного порядка - это теорема об антисимметричности. В Coq она формализуется следующим образом:

```
Theorem Antisymmetry : forall (s1 s2 : BSet T),
                       s1 [= s2 -> s2 [= s1 -> s1 = s2.
```

Доказательство данной теоремы будем вести с помощью разбивки на случаи с помощью тактики $case$. Причем, в первый раз мы ее применим для разбивки множества $s1$ на пустое и непустое, а после, в случае, когда $s1$ является пустым, мы применим тактику повторно, чтобы разложить множество $s2$ на такие же случаи, и, т.к. в данной теореме используется вложенность списка в другой - эти случаи докажутся автоматически. А второй случай, когда $s1$ не является пустым, т.е. образован с помощью конструктора $Elim$, докажем вручную.

После применения стандартных преобразований и тактик схема доказательства выглядит слудующим образом:

```
H1   : y /in s2
H2   : s2 [=Elem y s1
H3   : s1 [=s2
Goal : Elem y s1 = s2
```

Как видно из схемы, мы имеем две гипотезы $H3$ и $H2$, которые подподают под доказанную ранее лемму о разбиении множества, поэтому мы можем ее применить тактикой $apply$ либо к гипотезе $H2$, либо к $H3$ (так как они оба используются в качестве предволожения в лемме):

```
H1   : y /in s2
H2   : s2 = s1 \/ s2 = Elem y s1
Goal : Elem y s1 = s2
```

После этого, с помощью тактики $destruct$ мы разбиваем данную гипотезу на две, каждую из которых можно доказать уже с помощью доказанных фактов ранее (полное доказательство можно посмотреть в репозитории).

После доказательства данной теоремы мы подтвердили, что операция вложенности множеств является отношением частичного порядка. В процессе доказательства мы выявили несколько ключевых фактов и общих черт доказательства разных теорем, и теперь мы можем приступить к автоматизации процесса доказательства теорем (будет описано в 3 разделе).

В заключении данного подраздела об отношении частичного порядка добавим, что также был доказан ряд более мелких фактов, которые понадобились в доказательстве, а также была доказана обратная теорема об антисимметричности, т.е. если мы знаем, что два множества равны между собой, то мы можем утверждать, что одно из них вложено в другое. Данный факт будет использован в следующем подразделе для доказательсвта канонического сравнения для множеств.

2.2. Каноническое сравнение для множеств

В данном подразделе будет формализовано сравнение для множеств с помощью обычной функции-предиката. А также доказана теорема $reflect$ об эквивалентности нашего определения со стандартным пропозиционным, что придаст гибкость и позволит в любой момент доказательства перейти от одного определения к другому. Теоремы $reflect$ являются одним из основных инструментов библиотеки $ssreflect$ для Coq, и умение доказывать их и использовать будет очень полезным навыком в будущем.

2.2.1. Функция сранения для множеств

Определим функцию сравнения с помощью Fixpoint:

```
Fixpoint eqBSet (s1 s2 : BSet T) :=
        (subseteq s1 s2) && (subseteq s2 s1).
```

Данная функция определяется с помощью опреации вложенности множеств Два множества мы будем считать равными, если одно множество соответственно вложено в другое и наоборот. Данное определение очень схоже с обратной теоремой об антисимметричности множества, только там мы использовали пропозиционное сравнение, а теперь данная функция основана на типе $bool$ и является вычислительной, т.е. в процессе доказательства теорем, используя данное определение сравнения можно добиться автоматического доказательства некоторых теорем, использующих сравнение в своем определении.

Нотацию для данного сравнения мы будем использовать стандартную, но для того, чтобы ее использовать необходимо доказать теорему $reflect$. Она о том, что наше определение является корректным и эквивалентным пропозиционному, только после этого мы сможем использовать стандартное сравнение типа $bool$, которое будет основано на определенной нами функции $eqBSet$.

2.2.2. Теорема $reflect$

Теорема $reflect$ для функции $eqBSet$ в Coq будет формализована в следующем виде:

```
Theorem eqBSetP : Equality.axiom eqBSet.
```

Это стандартный вид теорем $reflect$. Доказательсвто данного факта необходимо также проводить с использованием тактики $case$, т.е. разбивать на случаи. После ее применения без дополнительных манипуляций мы получаем следующие цели:

```
Goal 1: forall y : BSet T, ssrbool.reflect
               ({ # } = y) (eqBSet { # } y)

Goal 2: forall (t : T) (b y : BSet T), ssrbool.reflect
               (Elem t b = y) (eqBSet (Elem t b) y)
```

Первую подцель можно также доказать с помощью тактики $case$, разбив на случаи множество y и использовав тактику $constructor$, которая позволяет, использовать структуру определения множества в процессе доказательства, т.е. если данный факт можно доказать используя определение множества.

Описанный процесс можно объеденить в одну большую тактику, которая представляет сабой комбинацию из более простых и очевидных:

```
case => /= // [| x s1 s2];
    do [by case => /= //; constructor |].
```

Вторую цель также можно разложить на случаи с помощью тактики $case$ и применить тактику $constructor$. Также можно упростить цели с помощью стандартных тактик $reflect$, после чего схема доказательства примет следующий вид:

```
H1     : x /in s2
H2     : s1 [=s2
H3     : s2 [=Elem x s1
Goal 1 : Elem x s1 = s2
Goal 2 : Elem x s1 <> s2
```

Данные цели можно доказывать очень долго, раскладывая все на случаи и применяя индукцию. Но заметим, что данные гипотезы и цели очень похожи на теорему об антисимметричности операции вложенности множеств. Поэтому нет необходимости раскладывать этот факт на случаи. Необходимо его подогнать под определние теоремы об антисимметричности, и данные цели будут доказаны.

Для этого необходимо доказать следующий факт:

```
forall (s1 s2 : BSet T) (x : T),
        x /in s2 -> s1 [= s2 -> Elem x s1 [= s2.
```

Если мы знаем, что между двумя множествами есть вложенность одного в другое, а также факт того, что некоторый элемент принадлежит другому (тот, в котором происходит вложение), то, если мы добавим данный элементв первое множество, вложенность должна сохраниться. Доказательство данной леммы мы опустим, оно достаточно стандартное.

```
forall (s1 s2 : BSet T) (x : T),
        x /in s2 -> s1 [= s2 -> Elem x s1 [= s2.
```

После применения данной теоремы к гипотезе $H2$ мы получаем:

```
H2     : Elem x s1 [=s2
H3     : s2 [=Elem x s1
Goal 1 : Elem x s1 = s2
```

И теперь по теореме об антисимметричности данная цель становиться тривиальной. Аналогично доказывается вторая подцель, только там будет использована обратная теорема об антисимметричности.

Таким образом, теорема $reflect$ доказана. И теперь можно использовать функцию $eqBSet$ в качестве стандартного сравнения и использовать переходы между пропозиционным и вычислительным сравнением.

2.2.3. Использование стандартного сравнения

Для того, чтобы использовать в качестве стандартного сравнения определенную нами функцию $eqBSet$ необходимо предоставить Coq, доказанную теорему $reflect$ и указать, что теперь функция $eqBSet$ является стандартной:

```
Canonical BSet_eqMixin := EqMixin eqBSetP.
Canonical BSet_eqType  := Eval hnf in
                          EqType (BSet T) BSet_eqMixin.
```

После этого мы можем проверить, что стандартное сравнение работает корректно:

```
Eval compute in
    ({ # } == { # }).                          (* => true  *)
Eval compute in
    ({ # } == { # 1 }).                        (* => false *)
Eval compute in
    ({ # 1 } == { # 2 }).                      (* => false *)
Eval compute in
    ({ # 1; 1; 1} == { # 1}).                  (* => true  *)
Eval compute in
    ({ # 1; 2; 3; 1; 2} == { # 2; 1; 1; 3}). (* => true  *)
Eval compute in
    ({ # 1; 2; 3; 1; 2} == { # 2; 1; 1 }).   (* => false *)
```

3. Автоматическое доказательство теорем

В данном разделе будут представлены способы автоматизации, которые помогут доказывать теоремы. Вариантов автоматизации существует много, поэтому представленные здесь тактики являются одним из примеров.

В предыдущем разделе был доказан ряд теорем и изучены основные принципы доказательства в Coq на примере отношения частичного порядка для множеств и канонического сравнения как инструмента для более гибкого доказательства теорем.

Цель автоматизации - максимально упростить процесс доказательства теорем и свести к минимому вовлеченность человека в процесс доказательства. Для более очевидных теорем процесс может быть вообще сведен к одной автоматической тактике.

В более сложных фактах автоматизировать процесс будет труднее, так как каждая теорема является специфической и выделить общий алгоритм вообще для всех случаев является проблематичным.

Поэтому ключевая задача заключалась в том, чтобы выделить общие принципы доказательства во всех теоремах, которые были использованы в процессе работы и попытаться автоматизировать повторяющиеся вещи, чтобы человек доказывал только ключевые моменты теоремы и не вдавался во все тонкости доказательств.

Таким образом, данный процесс можно сравнить с высокоуровневым доказательством теорем. Т.е. мы не вдаемся в детали по перемещению гипотез, применению индукций, применению теорем $reflect$ и всех прочих технических вещей, которые связаны с системой Coq и библиотекой $ssreflect$, а ограничиваемся отслеживанием логики доказательства - следим за последовательностью применяемых нами основных фактов и лемм.

Таким образом, человек будет разрабатывать стратегию доказательства, а компьютер будет ее реализовывать, не нагружая программиста излишними деталями. Такой баланс между вовлеченностью че-

ловека и автоматизацией системы Coq мы и попытаемся соблюсти, ну или хотя бы приблизимся к нему.

3.1. Механизм создания тактик

Для автоматического доказательства мы определим внутри системы Coq механизм $Ltac$ - инструмент для создания тактик, с помощью которого мы попытаемся создать универсальную тактику упрощения и с помощью неё максимально доказать теорему автоматически.

Определим базовую тактику преобразования, которой мы будем пользоваться и постепенно расширять:

```
Ltac matchT :=
  idtac.
```

Данная тактика пока ничего не делает, так как в ней просто используется тактика $idtac$, которая никак не изменяет доказательство. Это, своего рода,заглушка, необходимая для определения нашей тактики.

Данная тактика будет служить основным преобразованием. Она будет расширяться в процессе рассмотрения разных случаев доказательства.

Чтобы рассматривать разные случаи и ситуации, мы будем использовать механизм сопоставления секвенций. Т.е. алгоритм будет проверять, какая складывается ситуация в данный момент доказательства, и в соответствии с этим будет искать такую же определенную в тактике $matchT$. Если механизм такую ситуацию найдет, то предпримет соответствующие действия, которые прописаны в данной ситуации. Если нет, то тактика ничего не сделает, и доказательство дальше не продвинется.

Для описания данной ситуации в доказательстве мы будем использовать секвенции:

$$H_1 : h_1,\ H_2 : h_2,\ ...,\ H_n : h_n \mid -\ Goal => actions$$

- $\mid -$ - знак секвенции, слева от знака гипотезы, справа - цель
- $=>$ - знак логического следования, слева от знака секвенция, справа действие, которое необходимо выполнить
- H_i - переменные, используются в качестве имен для гипотез
- h_i - структура гипотезы, с помощью которой будет искаться похожая гипотеза в процессе доказательства
- $Goal$ - структура цели
- $actions$ - действие, которое необходимо выполнить, если данная секвенция описывает текущую ситуацию в доказательстве

Для начала попробуем добавить вариант разбора данной секвенции, для произвольного случая, используя заглушку. Плюс ко всему организуем циклический перебор тактик с помощью специальной тактики $repeat$, которая будет позволять после нахождения подходящего варианта и применения соответствующих действий, повторять этот процесс до тех пор, пока у нас не закончаться все варианты, которые мы можем применить в сложившейся ситуации доказательства.

```
Ltac matchT :=
    repeat (multimatch  goal with
      | _ => idtac
    end) => /= //; try intros.
```

Помимо заглушки в данной тактике используется $multimatch$. Данный вариант позволяет искать совпадения по нескольким вариантам и, если подходят несколько, то применять каждый последовательно, а не только первый найденный вариант, таким образом позволяя максимально применять автоматизированные действия.

Также после каждого применения $idtac$ происходит попытка автоматического упрощения и доказательства целей с помощью стандартных тактик в Coq.

Кроме того, после каждого применения $multimatch$ происходит попытка переноса гипотез в секцию гипотез с помощью тактики $intros$. Это аналог тактики $move$ из библиотеки $ssreflect$. Так как гипотезы могут образоваться не всегда, мы применяем именно попытку воспользоваться данной тактикой. Для этого мы указываем ключевое слово try, и если после применения тактики $intros$ не произойдет никаких ошибок, то она применится, в противном случае выполнится тактика $idtac$, т.е. ничего не измениться.

Далее задача по автоматизации доказательства теорем состоит в добавлении разнообразных случаев и этапов доказательства, и указаний, что необходимо сделать в том или ином случае.

3.2. Автоматическая индукция

Из процесса доказательства теорем видно, что большинство из них доказываются методом индукции. Поэтому логично вынести данное преобразование в отдельную тактику. Так как не все теоремы необходимо доказывать индуктивным методом, а при попытке это сделать может произойти ошибка, и данная тактика просто не сработает - необходимо попытаться применить индукцию с помощью тактики try.

После применения индукции появляется возможность автоматически доказать базу индукции или упростить предположение - для этого мы тоже будем пытаться применить тактики $/ =$ и $//$. Применение индукции вынесим в отдельную тактику $baseT$, которая будет использовать определенную нами ранее тактику $matchT$.

```
Ltac baseT :=
    try elim => /= //; intros => /= //;
    repeat matchT.
```

После применения тактики $elim$ обычно возникают новые гипотезы, которые необходимо переместить в секцию гипотез. Для этого после применения тактики $elim$ и попытки упрощения целей, мы применяем тактику $intros$, которая как раз и перемещает выводимые формулы в секцию гипотез.

Далее вызывается тактика $matchT$, с помощью которой определяются все правила упрощения доказательства. Теперь индукция будет выполняться автоматически, достаточно просто применить тактику $baseT$.

```
Goal : forall (s1 s2 : BSet T) (x : T),
                s1 [=s2 -> s1 [=Elem x s2
_____________________________________________ baseT
H    : forall (s2 : BSet T) (x : T),
            b [=s2 -> b [=Elem x s2
H0   : (t /in s2) && (b [=s2)
Goal : ((t == x) || t /in s2) && (b [=Elem x s2)
```

Как видно из доказательства, после применения тактики $baseT$ база индукции доказалась автоматически, появилось индукционное предположение, и все выводимые формулы были перемещены в секцию гипотез. Кроме того, было произведено упрощение целей и гипотез,что позволило увидеть внутреннее устройство целей, а также все используемые в них логические операции.

Если в доказательстве необходимо применять последовательно несколько раз тактику $case$ можно использовать другую тактику:

```
Ltac caseT :=
   try repeat case => /= //;
   repeat matchT.
```

Тактика пытается раскладывать цель на случаи, пока это возможно, благодаря чему за одну операцию происходит максимальное упрощение и разбивка цели на случаи, а также максимальное упрощение и попытка автоматического доказательства некоторых случаев.

3.3. Тактики $reflect$

Наиболее часто применяемыми тактиками во всех теоремах были тактики перехода между вычислительными логическими операциями типа $bool$ и пропозиционными типа $Prop$. Поэтому для данных переходов введем специальные тактики:

```
Ltac andT :=
    let V := fresh "V" in
      case: andP => /= //;
      unfold not => V1; apply: falseP;
      apply: V1.
```

Данная тактика применяется для перехода между вычислительной конъюнкцией и пропозиционной в цели. Это ровно те действия, которые мы выполняли вручную при доказательстве теорем: применение тактики $case$ к теореме $andP$ из библиотеки $ssreflect$ и последующее упрощение.

В данной тактике используется указание $let...in$ - данный механизм используется для создания новых переменных в программе. Мы указываем, что хотим получить новую переменную V и говорим, что она новая, т.е. до этого ее не было в программе. Это мы указываем с помощью ключего слова $fresh$ и говорим, что данная переменная будет отображаться для пользователя как $"V"$ (строго говоря, это может быть и не так, переменная будет иметь в программе одно имя, а для пользователя оно может быть совершенно другим).

Если такое имя уже использовалось в программе, то из-за того, что мы указали ключевое слово $fresh$, имя переменной измениться, к нему будет добавлен индекс, начиная с 0, т.е. переменная будет иметь имя $V0$, и если такое имя будет в программе, то тогда $V1$ и так далее, пока не окажется свободного имени.

После введения данной тактики переход между вычислительной и пропозиционной конъюнкцией будет происходить за одну тактику:

```
Goal : ((t == x) || t /in s2) && (b [=Elem x s2)
_____________________________________________ andT
Goal : ((t == x) || t /in s2) /\ (b [=Elem x s2)
```

Аналогично определяется и тактика для вычислительной дизъюнкции. Отличие только в использовании теоремы orP вместо $andP$.

Теперь введем тактики для таких же переходов, только для гипотез. Причем, помимо простой замены вычислительной логической операции на пропозиционную мы будем также разбивать гипотезы на две независимые.

```
Ltac andH  H :=
    let H' := fresh "H'" in
    let H'' := fresh "H''" in
    case: andP H => /= // H H'';
          destruct H as [H H'] => /= //.
```

В данной тактике мы избавляемся от вычислительной конъюнкции. Для этого нам необходимы две новые перменные H' и H''. Так как в результате применения тактики $case$ мы получаем две новые гипотезы (чаще всего вторая гипотеза является истинной, но иногда она бывает нетривиальной в некоторых теоремах, поэтому терять ее нельзя).

```
H0 : (t /in s2) && (b [=s2)
_____________________________________________ andH H0
H0 : t /in s2
H' : b [=s2
H'' : true
```

Как видно, данная гипотеза разбилась на три. Третья является тривиальной и нужно это учесть, чтобы тривиальные гипотезы автоматически уничтожались, а оставались только информативные (так как человеку нет никакой необходимости созерцать бесчисленное количество тривиальных фактов, которые просто будут захламлять доказательство, сбивая фокус программиста).

Для дизъюнкции тактика будет выглять немного иначе:

```
Ltac orH H :=
  let H' := fresh "H'" in
  case: orP H => /= // H H';
        destruct H as [H | H] => /= //.
```

Здесь используется всего одна новая переменная. Так как $destruct$ в тактики $andH$ разбивал гипотезу на две выводимые формулы, поэтому они имели разные имена. При использование же логического ИЛИ в гипотезе мы знаем, что либо левая подформула является выводимой, либо правая. Т.е. при разбивке гипотезы на две части мы получаем две ветки альтернативного доказательства: в первой - истинной является левая подформула, и необходимо предоставить доказательство как для данной ситуации, так и для второй части доказательства.

Помимо конъюнкции и дизъюнкции в гипотезе может встретиться вычислительное сравнение. Поэтому необходимо также ввести тактику, которая использует теорему $reflect$ и переходит к пропозиционному определению.

```
Ltac eqT H :=
  let H' := fresh "H'" in
  move: H; case: eqP => // H H'.
```

Здесь используется стандартное разбиение на случаи теоремы eqP из библиотеки $ssreflect$. Использовать тактику $destruct$ как в предыдущих случаях необходимости нет, так как в гипотезе просто заменяется одно равенство на другое.

Также можно заметить, что все три тактики, которые работают с гипотезами в качестве аргумента принимают гипотезу H. Поэтому применение осуществляется вызовом ($eqT\ H$). Ниже приведен пример использования тактики eqT:

```
H0 : Elem s1 == s2
_____________________________________________ eqT H0
H0 : Elem s1 = s2
H' : true
```

После того, как мы определили вспомогательные тактики их необходимо объеденить в одну (добавить в тактику $matchT$), чтобы они все работали совместно.

```
Ltac matchT :=
repeat (multimatch  goal with
  | |-  context [ _ || _ ] => orT              (* + *)
  | |-  context [ _ && _ ] => andT             (* + *)
  | H : context [ _ || _ ] |- _ => orH H       (* + *)
  | H : context [ _ && _ ] |- _ => andH H      (* + *)
  | H : context [ _ == _ ] |- _ => eqT H       (* + *)
  | H : context[ true ] |- _ => clear H        (* + *)
end) => /= //; try intros.
```

Тактики orT и $andT$ применяются только к цели, поэтому секвенция, описывающая данный случай не учитывает гипотезу (слева от знака секвенции просто ничего не указывается). Для поиска вычислительных символов в цели и гипотезах используется поиск в контексте, т.е. если в цели есть где-либо вычислительная операция, то данная секвенция подподает под данную ситуацию доказательства и будут применяться соответствующие действия (тактики orT или $andT$).

При использовании тактик для гипотез, мы, наоборот, не учитываем цель, т.е. справа от знака секвенции ничего нет. Гипотеза также подбирается, исходя из наличия вычислительной операции в контексте.

Помимо основных тактик, также была добавлена тактика $clear\ \ H$, которая удаляет тривиальные гипотезы, заведомо являющиеся истинными, так как в результате применения тактик $orH,\ andH,\ eqT$ образуются тривиальные факты, от которых необходимо избавиться.

Аналогичным способом будет происходить расширение тактики $matchT$ с помощью других случаев, которые будут описаны далее. Тактику $matchT$ можно применять как отдельно (просто для упрощения доказательства), так и внутри тактик $baseT$ и $caseT$, как один из

этапов доказательства после применения индукции и разбиения на случаи.

Приведем пример использование тактики $baseT$ на лемме о вложенности множеств:

```
forall (s1 s2 : BSet T) (x : T),
        s1 [=s2 -> s1 [=Elem x s2
_____________________________________________ baseT
H    : forall (s2 : BSet T) (x : T),
                b [=s2 -> b [=Elem x s2
H0   : t /in s2
H'   : b [=s2
Goal : (t == x) || t /in s2 /\ b [=Elem x s2
```

Как видно из примера, выполнилось доказательство по индукции, также разбилась гипотеза на $H0$ с помощью тактики $andH$ на три (одна из них - тривиальная, удалилась). Также применилась тактика $andT$, и произошла замена вычислительной конъюнкции на пропозиционную. Заметим, что вычислительная дизъюнкция в цели не заменилась, так как она является частью левой подформулы цели, т.е. она не является внешней операцией, тактика $baseT$ заменяет внешние операции, а не внутренние.

Также видно, что мы не стали определять тактику для замены вычислительного сравнения в цели. В этом нет необходимости, так как вычислительное сравнение позволяет автоматически вычислять результат в случаи равенства двух оперантов. Поэтому в большинстве случаев лучше оставить вычислительное сравнение, а если это будет необходимо в конкретной теореме, то это уже лучше сделать вручную.

3.4. Булевы логические операции

Теперь перейдем непоредственно к математической логике. Так как мы в предыдущем разделе гарантировали переход от вычислительных логических операций к пропозиционным, то теперь мы мо-

жем применять законы логики для упрощения доказательства. Основными логическими операциями для упрощения выступят конъюнкция, дизъюнкция, эквиваленция и отрицание.

Наиболее сложной для упрощения и анализа операцией является дизъюнкция, так как для ее доказательства необходимо доказать одну из подформул, а какую именно, сказать не всегда получается, пока доказательство не станет достаточно тривиальным.

3.4.1. Тактика $split$ для эквиваленции и конъюнкции

Первый случай: если нам необходимо доказать эквивалентность двух определений, то она заменяется на две импликации с помощью тактики $split$ в Coq. Добавим данную ситуацию в тактику $matchT$:

```
| |- context [ _ <-> _ ]  => split            (* + *)
```

После добавления данной ситуации наша цель будет разбиваться на две, и в каждом из случаев необходимо будет привести доказательство:

```
Goal : A <-> B
_________________ split
Goal 1 : A -> B
Goal 2 : B -> A
```

Аналогичную тактику необходимо использовать, если мы имеем цель с внешней операцией пропозиционной дизъюнкции:

```
| |- _ /\ _ => split                          (* + *)
```

В данном случае необходимость поиска по контексту отпадает, так как Coq для пропозиционных операций сам определяет, что они внешнее (исключение составляет только эквивалентность, в данном случае система не может применить тактику без контекста).

После применения данной тактики, наше цель также разбивается на две подцели, каждую из которых необходимо доказать, имея в распоряжении одинаковые гипотезы.

```
Goal : A /\ B
________________ split
Goal 1 : A
Goal 2 : B
```

3.4.2. Разбиение дизъюнкции

Теперь перейдем к дизъюнкции, которая находится в цели. Так как для доказательства дизъюнкции необходимо, чтобы всего лишь одна из подформул была истинна, то неочевидно какую из подформул стоит откинуть, а какую оставить. Поэтому разберем несколько случаев.

Самый очевидный вариант, когда мы знаем как разбивать дизъюнкцию - если одна из подформул является тривиальной, т.е. если подформула является истинной, или можно привести ее к таковой с помощью упрощения.

Для этого мы введем следующую тактику:

```
| |- _ \/ _ => do [by left | by right]            (* + *)
```

В данной тактике происходит попытка доказательства теоремы либо с помощью правой, либо с помощью левой части. В данном случае нет необходимости использовать тактику try, т.к. если попытка не увенчается успехом, то данная тактика применяться не будет и возникнет необходимость поиска других вариантов.

```
Goal : A == true \/ B
____________________ left
Goal : A == true
____________________ done
Цель доказана
```

```
Goal : A \/ B == true
____________________ right
Goal : B == true
____________________ done
Цель доказана
```

В данной схеме тактика $done$ - синоним тактики by. Можно записать либо $left;\ done$, либо $by\ left$. Т.е. мы используем данный вариант

только в том случае, когда можем доказать теорему за один шаг, заведомо зная, что одна из подформул доказуема сама по себе.

Также мы можем разбить дизъюнкцию, если знаем, что одна из подформул находится в гипотезе, таким образом цель также станет тривиальной.

```
| H : ?P |- _ \/ ?P => right                    (* + *)
| H : ?P |- ?P \/ _ => left                     (* + *)
```

Здесь для определения одинаковых подформул используются переменные с символом $?$. Т.е. вместо данных переменных в процессе поиска будут подставляться разнообразные значения, вплоть до больших формул. Данный случай будет применяться по следующей схеме:

```
H    : A                        H    : B
Goal : A \/ B                   Goal : A \/ B
_____________ left              _____________ right
H    : A                        H    : B
Goal : A                        Goal : B
```

Еще один вариант разбиения дизъюнкции - когда подформулы являются противоположными друг другу. При этом правая подформула отрицает левую и наоборот.

```
| |- context [ is_true(?P) \/ is_true(?P == false) ] =>
     destruct P (* + *)
```

Т.е. если мы имеем в цели утверждение о том, что подформулы дизъюнкции являются противоположными, то должны разбивать это утверждение с помощью тактики $destruct$. Схема применения будет выглядеть следующим образом:

```
Goal : A \/ A == false
_____________________________________________ destruct A
H : A == true              H : A == false
G : A \/ A == false        G : A \/ A == false
```

```
____________________ left    ____________________ right
H : A == true                H : A == false
G : A                        G : A == false
____________________ -> A    ____________________ -> A
G : true                     G : false == false
```

Данная стратегия может применяться и в следующем контексте: когда требуется разбить утверждение где-нибудь внутри формулы, то произойдет разбиение на две цели. Например, эта тактика будет очень хорошо применима при использовании условного оператора выбора if.

3.4.3. Тактика $unfold$ и операция отрицания

Отрицание само по себе не влечет особо больших проблем при автоматизации, так как её стратегия состоит в раскрытии отрицания с помощью импликации:

```
Goal : ~A
___________________ unfold not
Goal : A -> False
```

В данном случае тактика $unfold$ просто раскрывает оператор и заменяет на его определение. В Coq оператор отрицания определяется именно так. Тактика будет описана следующим способом:

```
| |- context [ not ] => unfold not
```

Помимо прямого раскрытия отрицания тактика $unfold$ может быть полезна при раскрытие некоторых определений. Например, в наших доказательствах встречалось раскрытие с помощью тактик $unfold$ функций, $notIn$ - это функция, которая показывает, что данный элемент не вложен во множество, и раскрытие функции $AddBSet$ - добавление элемента во множество.

Поэтому эти случаи можно добавить в тактику $matchT$:

```
| |- context [ AddBSet ] => unfold AddBSet        (* + *)
| |- context [ notIn ]   => unfold notIn          (* + *)
```

Тактику $unfold$ трудно применять ко всем случаям, так как количество функций может быть большим и не все стоит раскрывать в том или ином доказательстве. Так как наша цель написать тактику, которая будет автоматизировать процесс разных теорем, то ограничемся раскрытием только данных функций, при необходимости раскрытие какой-либо другой в конкретной теореме можно сделать вручную.

3.4.4. Преобразование гипотез с логическими операциями

С преобразованием целей мы разобрались, но теорема не будет доказываться автоматически, если мы не преобразуем гипотезы. Преобразование конъюнкции и дизъюнкции в гипотезах будут происходить с помощью одной и той же тактики $destruct$.

Только в случае конъюнкции гипотеза разбивается на две, и они обе используются для доказательства одной цели, а в случаи дизъюнкиции гипотеза разбивается на два случая, когда истина левая подформула и когда правая, и каждый из случаев доказывается по отдельности.

Тактики для данных случаев описываются следующим образом:

```
| H : _ \/ _ |- _ => destruct H
| H : _ /\ _ |- _ => destruct H
```

Здесь цель нам совершенно не важна, поэтому мы ее опускаем. Подбор случаев происходит исключительно по гипотезе, есть в ней дизъюнкция или конъюнкция или же нет. Схема доказательства данной тактики следующая:

```
H    : A /\ B                  H    : A \/ B
Goal : C                       Goal : C
_____________ destruct H       _____________________ destruct H
H    : A                       H    : A     H    : B
```

```
H'   : B                    Goal : C     Goal : C
Goal : C
```

Также помимо дизъюнкции и конъюнкции необходимо избаляться от отрицания в гипотезе с помощью тактики $unfold$, а также раскрывать отрицание, которое находится внутри функции $notIn$:

```
| H : context [ not ]   |- _ => unfold not in H      (* + *)
| H : context [ notIn ] |- _ => unfold notIn         (* + *)
```

Кроме того, в гипотезах необходимо удалять дубликаты, так как из-за этого могут возникнуть проблемы, например, зацикливание тактики. Для этого добавим следующее правило в тактику $matchT$:

```
| H : ?P, H' : ?P |- _ => clear H'          (* + *)
```

Тактика $clear$ удаляет дубликат гипотезы H, если таковой имеется. Сравнение проводится чисто по гипотезам, цели в данном варианте опускаются.

После упрощения и введения тактик для логических операций доказательство уже очень сильно упростилось. Приведем пример на все той же лемме о вложенности множества.

```
H0 : t /in s2
Цель в пункте 3.3 : (t == x) || t /in s2 /\ b [=Elem x s2
_______________________________________________
H0 : t /in s2
Цель сейчас : b [=Elem x s2
```

Как мы видим, по сравнению с доказательством в пункте 3.3 теперь цель заметно упростилась. Правая часть автоматически доказалась, так как у нас есть известная гипотеза, которая совпадает с одной из подформул левых частей.

3.5. Анализ условного оператора выбора if

В теоремах, где используется операция удаления из множества, внутри доказательства возникает оператор if, так как в определении данной функции происходит сравнение элементов. И поэтому возникает такая неопределенность и необходимоть данной тактики.

Анализ условного оператора происходит по следующему алгоритму:

```
| |- context [ if ?P then _ else _ ] =>
     match P with
     | context[ _ == _ ]  => case: eqP
     | _ => destruct P
    end
```

Для начала в контексте цели ищется оператор if, при его обнаружении производится анализ условия - в нашей тактики оно обозначено $?P$. Нас интересуют два случая, которые попадались нам при доказательстве теорем.

В первом случае условие представляет собой равенство переменных, причем равенство используется вычислительное, а не пропозиционное. Поэтому в данном случае необходимо перейти к пропозиционному сравнению с помощью теоремы eqP из библиотеки $ssreflect$.

Во втором же случае мы просто имеем некоторе условие, и его необходимо разбить на случаи равенства истине или нет. Для этого мы используем тактику $destruct$. В отличии от тактики $case$, тактика $destruct$ разбивает именно на случаи равенства истине или лжи, т.е. на две подцели. А $case$ используется для разбиения по структуре конструктора.

```
Goal : if C then A else B
_____________________________________ baseT
H    : C == true    H    : C == false
Goal : A            Goal : B
```

3.6. Тактики $rewrite$ и $apply$ как основной инструмент доказательства теорем

Тактики $rewrite$ - наиболее часто используемый инструмент в доказательствах, также как и тактика $apply$. Но, так как они часто используются, их автоматизация бывает достаточно затруднительна. Поэтому количество теорем, которые могут быть применены и переписаны, является очень большим, и скорее всего более точечным для каждой теоремы, чем глобальным и общим.

В связи с этим полностью автоматизировать данные тактики не получится. Мы разберем те случаи, в которых данные тактики все же можно автоматизировать, и покажем, как можно улучшить автоматизацию для конкретных групп теорем, заранее зная стратегию доказательства.

3.6.1. Автоматизация тактики $rewrite$

Самый очевидный случай применения переписывания - когда в гипотезе мы имеем переменную, которая чему-то равна и данная переменная присутствует в цели. Для этого случая применим следующую стратегию:

```
| H : ?x = _ |- context[?x] => rewrite H    (* + *)
```

Т.е. здесь просто происходит замена одного терма другим. Приведем пример использования данной тактики:

```
H    : Elem x s1 = s2
Goal : Elem y (Elem x s1) = Elem y s2
_____________________________________ baseT
H    : Elem x s1 = s2
Goal : Elem y s2 = Elem y s2
```

Как видно, в данном примере терм $Elem\ x\ s1$ в цели заменился на $s2$, таким образом произошла перезапись. Также данная перезапись

будет работать и в случае, когда в цели есть несколько одинаковых переменных, которые требуют замены.

Проблема здесь заключается в том, что не в каждом доказательстве необходимо делать перезапись. Здесь и начинаются трудности, но они скрыты в структуре доказательства конкрекной теоремы, и уже программист должен решать, как проводить стратегию доказательства Автоматические инструменты лишь упрощают жизнь - фокусируют взгляд, экономят время и избавляют от ненужной рутины, а не заменяют программиста как единицу работы.

Также тактику $rewrite$ можно применить для перезаписи термов в гипотезах. Работа данной тактики происходит аналогичным способом как и при перезаписи термов в цели:

```
| H : ?x = _, H' : context[?x] |- _ =>
                   rewrite H in H'            (* + *)
```

Также при перезаписи можно встретиться с бесконечной перезаписью, если мы имеем две симметричные гипотезы, которые поочереди перезаписывают друг друга. Это можно исправить, удалив соответствующий дубликат гипотезы:

```
| H : ?P = ?Q, H' : ?Q = ?P |- _ => clear H'  (* + *)
```

Помимо переписывания одинаковых термов можно перезаписывать конкретные теоремы. Это видно на примере перезаписи аксиом $Ident1$ и $Ident2$. Аналогично можно использовать перезапись и для любых других теорем, перезапись которых необходимо будет автоматизировать.

```
| |- context [ Elem ?x (Elem ?x _) ] => by rewrite Ident1
| |- context [ Elem ?x (Elem ?y _) ] => by rewrite Ident2
```

Перезапись аксиомы $Ident1$ произойдет, если в цели будет ситуация, при которой во множество будут добавляться несколько одинаковых элементов. Но перезапись произойдет только в том случае, если цель после этого докажется, не испортив процесс доказательства.

Как уже говорилось ранее, тактика $rewrite$ сложна в автоматизации, и если злоупотребить упрощением, теорему можно не доказать вообще.

Аналогично происходит и перезапись аксиомы $Ident2$. В случае, когда два элемента добавляются во множество, мы можем их переставить, если цель будет автоматически доказана.

3.6.2. Автоматизация тактики $apply$

Применение тактики $apply$ также, как и тактики $rewrite$ затрудняется тем, что в каждой теореме необходимо применять разные факты, причем еще и в определенной последовательности. Поэтому, если не знать заранее стратегию доказательства теоремы, обобщить применение тактики нельзя.

В некоторых случаях можно воспользоваться данной тактикой, например, если у нас заведомо есть гипотеза, которая подпадает под основное правило вывода Гильберта, то можно попробовать применить эту тактику, если за конечное число шагов данная теорема не зайдет в тупик.

```
| H : _ -> ?P |- ?P => apply H                (* + *)
```

Помимо такого локального применения можно написать обобщенную тактику и применять ее в тех случаях, когда она не будет заводить доказательство в тупик. Данная тактика не будет включена в обобщенный алгоритм,ведь в общем случае она приведет доказательство в хаотичный беспорядок и может сломать его:

```
Ltac applyA :=
match goal with
  | H  : _, H2 : _ |- _ => apply H in H2; applyA
  | _ => idtac
end.
```

Данная тактика очень сильна в том плане, что основана фактически на полном переборе и везении- повезет доказать хорошо, нет - применять не будем. В лучшем случае тактика ничего не изменит, в худшем - зациклится. Поэтому применять такие глобальные обобщения необходимо с осторожностью.

3.7. Пример автоматического доказательства

После проделанных манипуляций необходимо проверить, как работает данная автоматизация и работает ли вообще. Приведем пример доказательства леммы о вложенности множеств:

```
Goal : forall (s1 s2 : BSet T) (x : T),
                s1 [=s2 -> s1 [=Elem x s2
_______________________________________ baseT
No more subgoals.
```

И так мы добились полного автоматического доказательства для данной леммы. Это уже успех. Другие простые факты также доказываются автоматически с помощью применения одной тактики $baseT$.

Приведем доказательтсво теоремы о рефлексивности:

```
Goal : forall (s : BSet T), s [= s.
_______________________________________ baseT
H    : b [=b
Goal : b [=Elem t b
_______________________________________ by apply: L6.
No more subgoals.
```

Доказательство данной теоремы прошло неполностью автоматически, но в целом так и должно быть. Наше участие в доказательстве свелось к тому, что мы просто выяснили, что данная теорема доказывается с помощью леммы о вложенности множеств.Но это логично, ведь когда мы проводим доказательство на бумаге, то тоже говорим, какой факт лежит в основе того или иного доказательства.

При необходимости можно автоматизировать и данную теорему, просто ввести дополнительное условие:

```
| |- _ => by apply L6
```

И после добавление данной ситуации, доказательство этой теормы будет происходить автоматически. Но делать это уже нужно опционально для конкретных групп теорем по необходимости. В иных случаях написать доказательство в таком виде будет гораздо быстрее, чем пытаться все максимально автоматизировать. Во всем необходима золотая середина, и данный случай как раз из таких.

4. Способы реализации математических объектов

В данном разделе будут представлены несколько способов реализации объектов общей алгебры. Один из них будет основан на реализованном нами базовом множестве.

Цель - показать, как можно реализовать более сложные объекты в Coq и использовать все те возможности, которые мы уже реализовали для множества, в том числе автоматическое доказательство теорем.

Способов для реализации данной задачи много, поэтому будет представлен один из варинтов, который во-первых является достаточно наглядным, во-вторых расширяемым, т.е., при необходимости реализовать тот или иной объект, достаточно добавить некоторые условия к уже имеющимся определениям, и мы получим желаемое, причем сможем использовать все наработки, сделанные ранее.

Показанные алгоритмы должны позволить понять принцип реализации объектов и подтолкнуть студентов к более глубокому изучению данного языка и его возможностей, так как описанные здесь принципы оставляют большой простор для творчества и плавно вводят в мир автоматизации и компьютерных доказательств математических объектов.

4.1. Множество как объект общей алгебры

В данном подразделе мы попробуем рассмотреть множество как объект общей алгебры. Реализация практически всех объектов общей алгебры основывается на данном понятии, и умение управлять множеством и доказывать разнообразные факты о нем, является ключом к обобщению и введению более сложных структур в рассмотрение.

Множество будет представлять собой фундамент, на котором мы будем строить другие, более сложные объекты. А для начальных манипуляций попробуем посмотреть на множество под другим углом.

4.1.1. Альтернативное определение множества

Для начала работы необходимо определить интерфейс, с помощью которого мы будем взаимодействовать с данными объектами. В Coq есть специальный инструмент для более привычного восприятия формализуемых понятий - таким понятием является **структура**. Определим базовую структуру, которая будет полностью эквивалентна нашему базовому множеству:

```
Structure SetS :=
{
  As :> BSet T;
}.
```

Данная конструкция представляет собой своего рода контейнер. Ядром данного объекта является конструктор As, который имеет тип $BSetT$. Т.е. данный контейнер и есть наше базовое множество.

Таким образом, мы просто сделали обертку для нашего базового множества. И теперь будем использовать данный объект $SetS$ как единицу общей алгебры.

Приведем пример леммы для данного объекта:

```
Goal : forall s : SetS T, |s| = |s|.
_____________________________________ baseT
No more subgoals.
```

Как видим, для данного объекта также работают автоматические тактики. Все наработки, которые мы реализовали для объекта типа $BSet\,T$, также применимы для объекта типа $SetS\,T$, т.к. объект $BSet\,T$ является контейнером для объекта $SetS\,T$, т.е. фактически определяет его структуру.

4.1.2. Пустое множество

Теперь попробуем реализовать немного другой объект на основе уже имеющегося. Мы реализуем пустое множество, но не будем

давать определения, которые полностью описывают данную структуру. Используем базовое множество $BSet\ T$ в качестве основы и наложим ограничение, сказав, что это множество с количеством элементов равным 0.

```
Structure ESetS :=
{
  Aes :> BSet T;

  power_cond : forall (s : BSet T), |s| = 0
}.
```

В данном определении снова в качестве контейнера используется базовое множество $BSet\ T$, но помимо этого указывается дополнительное ограничение, которому должны удовлетворять все объекты, принадлежащие типу $ESetS\ T$. Данное ограничение $power\_cond$ описывается с помощью условия-леммы, это - аксиома, которой должен удовлетвоять данный объект.

При реализации базового множества мы также определяли аксиомы, которым должен удовлетворять тот или иной объект $Ident1$ и $Ident2$. Поэтому, по сути, это тот же самый принцип, только записывается по-другому. Но по функциональности объект, реализованный таким способом, более гибок в использовании, так как данная аксиома $power\_cond$ заложена в самом определении объекта, а не будет вводится дополнительно от него.

Но, в целом, можно использовать любой удобный вариант для реализации подобных объектов.

Попробуем доказать какой-нибудь факт, чтобы убедиться, что наши автоматические тактики работают и для этого объекта тоже.

```
Goal : forall s : BSet T, s = Empty <-> | s | = 0.
_____________________________________________ baseT
No more subgoals.
```

Как видно из доказательства данной теоремы, тактика $baseT$ все сделала автоматически. Таким образом, мы добились того, что автоматическая тактика, определенная для базового множества, также работает и для данного объекта.

При даказательстве данной теоремы условие $power$ автоматически добавляется в секцию гипотез, и нет необходимости указывать отдельно, что данное условие выполняется.

Благодаря этому мы сможем реализовывать любой объект общей алгебры и использовать автоматические тактики для него. При необходимости можно добавить и другие тактики доказательства, если это будет необходимо в какой-то конкретной теореме.

4.1.3. Взаимодействие между объектами

Теперь попробуем показать, как разные объекты, которые мы определяем, могут взаимодействовать между собой. Докажем несколько простых фактов о множестве и пустом множестве.

Первый факт, который мы формализуем будет о том, что любой объект, принадлежащий классу пустого множества, будет вложен в любой объект обычного множества. Другими словами пустое множество вложено в любое другое. Для доказательства данного факта нам понадобиться доказать вспомогательную лемму, доказательство которой тривиальное, оно использует предыдущую лемму, доказанную в разделе 4.1.2.

```
Lemma Trivial : forall s : ESetS T, Aes s = Empty.
```

В данной лемме говориться, что базовое множество-контейнер является пустым. Что собственно говоря очевидно. Теперь приведем теорему о вложенности пустого множества в любое другое:

```
Goal : forall (s1 : ESetS T)
              (s2 : SetS T), s1 [= s2.
_____________________________________ baseT
Goal : s1 [= s2
_____________________________________ rewrite Trivial
Goal : { # } [= s2
_____________________________________ done
No more subgoals.
```

В данном доказательстве также показано использование автоматической тактики $baseT$, и единственное, что делает программист - это указывает, что в доказательстве следует применить лемму $Trivial$ (последнии две тактики можно объединить в одну с помощью $by\ rewrite\ Trivial$).

Теперь должно сложиться представление о том, как взаимодействуют объекты между собой, и как формализовывать теоремы с помощью инструментов, представленных в среде разработки Coq.

4.2. Реализация группы

Группа - это множество, на котором определена ассоциативная бинарная операция, причём для этой операции имеется нейтральный элемент (аналог единицы для умножения), и каждый элемент множества имеет обратный.

На примере группы мы покажем еще один из альтернативных способов формализации объектов общей алгебры, которые могут быть представлены в Coq. У каждого из определений есть свои плюсы и минусы, поэтому удобно использовать разные способы формализации в зависимости от задачи.

В данном варианте формализации будет использование той же структуры, которую мы использовали в предыдущем разделе. Но в отличие от реализации пустого множества, здесь мы не будем использовать внутри определения реализованное множество $BSet\ T$, а попробуем использовать стандартный тип Set.

Использование структур наглядно показывает как выгядит объект, т.е. она обеспечивает понимание, как воспринимать то или иное определение.

```
Structure group :=
{
  G :> Set;

  id : G;
  op : G -> G -> G;
  inv : G -> G;

  op_assoc : forall (x y z : G), op x (op y z) = op (op x y) z;
  op_inv_l : forall (x : G), id = op (inv x) x;
  op_id_l : forall (x : G), x = op id x
}.
```

В данном примере мы определяем носитель G, который имеет стандартный тип Set. Данный носитель - это фактически набор элементов, которые есть в группе.

Помимо носителя определен единичный элемент, который имеет тип носителя - G, т.е. фактически мы указываем, что данный объект является элементом носителя, т.е. элементом группы.

Элемент op - это бинарная операция, определенная на носителе G. Она принимает два элемента типа G и возвращает элемент этого же типа.

Последнее определение inv - это функция, которая принимает элемент носителя G и возвращает такой же тип. Данная операция описывает наличие обратного элемента для каждого элемента из группы G.

Помимо операций, определены аксиомы, которые гарантируют корректую формализацию группы. Также как и при реализации пустого множества, аксиомы находятся внутри структуры, что позволяет учитывать их при доказательстве теорем.

Первая аксиома op_assoc свидетельствует о том, что введенная нами бинарная операция должна быть ассоциативной. Вторая аксиома op_inv_l, говорит о том, что, если применить бинарную операцию к элементу из группы и его обратному, то мы должны получить единичный элемент id. И последнння аксиома op_id_l говорит, что применение бинарной операции к единичному элементу и любому элементу из группы не должно менять данный элемент.

Проверим типы наших элементов:

```
Check (id).   (* => id : forall g : group, g *)
Check (op).   (* => op : forall g : group, g -> g -> g *)
Check (inv).  (* => id : forall g : group, g -> g *)
```

После проверки типов видно, что они определены для любого объекта типа $group$. Для удобства использования можно сделать некоторые аргументы неявными, чтобы можно было определить нотацию для них.

```
Arguments id {g}.
Arguments op {g} _ _.
Arguments inv {g} _.
```

Тогда, после указания того, что аргументы будут неявными, мы можем ввести следующую нотацию, например, для бинарной операции:

```
Notation "x $ y" := (op x y).
```

Теперь покажем, как будет формализовывать теорема для данного объекта. Докажем следующий факт: если бинарная операция применяется для одного и того же объекта и результатом является тот же самый объект, то этот объект является единичным элементом.

```
Theorem ExGSimple (G : group) : forall (f : G),
                    f <.> f = f -> f = id.
```

В доказательстве данной теоремы мы также можем использовать определённую нами ранее автоматическую тактику $baseT$. Помимо этого в данной теореме используются внутренние аксиомы, и, при необходимости, их тоже можно автоматизировать аналогичным способом как и для аксиом $Ident1$ и $Ident2$.

Аналогиными способами можно реализовывать любые другие объекты общей алгебры и использовать автоматизацию для доказательства теорем, которая была разработана для множеств.

Далее открывается широкий простор для творчества. Можно пробовать усложнять уже имеющиеся объекты или пытаться ввести в рассмотрение более сложные структуры общей алгебры.

Также можно начать изучение уже реализованных библиотек, созданных для работы с объектами общей алгебры и пытаться разобрать используемые принципы работы там.

Заключение

В данной работе были показаны основные способы реализации объектов общей алгебры. Представлены принципы формализации и доказательства теоремы на примере факта того, что операция вложенности множеств является отношением частисного порядка. Для этого мы доказали, что отношение является рефлексивным, транзитивным и антисимметричным.

Ключевой задачей было показать как можно автоматизировать процесс доказательства теорем для сложных математических объектов с минимальной вовлеченностью человека в процесс доказательства. Была разработана специальная тактика $baseT$, которая позволяет автоматизировать процесс доказательства.

Также были показаны принципы реализации математических объектов на основе базового множества на примере пустого множества и групп. Показаны два варианта реализцации с явным использованием объекта типа $BaseT$ и косвенным.

После освоения данного введения в автоматизацию математических объектов у читателя должна сложиться общая картина о внутреннем устройстве системы Coq и библиотеки $ssreflect$, а также о том как можно их использовать в собственных исследованиях с максимальной эффективностью.

Список литературы

[1] Chlipala A. Certified Programming with Dependent Types. — MIT Press, 2019

[2] Programs and Proofs. — URL: http://ilyasergey.net/pnp/ (последнее посещение 05/24/2019).

[3] Репозиторий данной работы. — URL: https://github.com/arrival3000/Master-Work (последнее посещение 05/24/2019).

[4] Software Foundations. — URL: http://www.seas.upenn.edu/bcpierce/sf/current/index.html (последнее посещение 05/24/2019).

[5] Документация по библиотеке Ssreflect. — URL: http : / / ssr.msr-inria.inria.fr/doc/ssreflect-1.5/ (последнее посещение 05/24/2019).

[6] Курош А. Г. Лекции по общей алгебре. — 2-е изд . — М.: Физматлит, 1973.