Лабораторная работа #2 Рекурсия, индукция и контрпримеры

Данный материал основан на:

- Книге Тобиаса Нипкоу, Лоуренса Паулсон и Маркуса Уэйнзель «Isabelle/HOL: A Proof Assistant for Higher Logic»
- Материалах сайта, посвященного Isabelle http://isabelle.in.tum.de/
- Книге Кэннета Роузен «Discrete Mathematics and Its Applications»

Подборку материалов их компиляцию и адаптацию к курсу формальных методов выполнили:

- Якимов И.А. ivan.yakimov.research@yandex.ru
- Кузнецов А.С. askuznetsov@sfu-kras.ru

Иммутабельность

Одним из ключевых принципов в математике является *иммутабельность переменных*. Поэтому в Isabelle/HOL (и в Haskell, и в ML и в Lisp) вы не встретите циклов — все вычисления производятся при помощи рекурсивных вызовов. Именно иммутабельность переменных позволяет проводить доказательства в системе Isabelle.

Пример индуктивного доказательства — app и rev

Рассмотрим простейшую теорию, в которой будет определен тип данных list, функции app и rev а также рассмотрены их свойства.

Сначала отменим определения, встроенные в HOL:

```
no_notation Nil("[]") and Cons (infix "#" 65) and append (infixr "@" 65)
hide_type list
hide_const rev
```

- no_notation отменяет заданные нотации — например, для конструктора Cons был задан псевдоним # — инфиксный правоассоциативный оператор с приоритетом 65
- hide_type скрывает тип

Далее переопределяем тип данных list:

```
datatype 'a list = Nil ("[]")
| Cons 'a "'a list" (infixr "#" 65)
```

Тип данных list содержит два конструктора — Nil, создающий пустой список и Cons, который добавляет новый элемент к уже существующему списку. Использование псевдонимов сокращает запись. Например, список [True, False] имеет вид Cons True (Cons False Nil), и с использованием псевдонимов записывается компактно True # False # [].
Правоассоциотивность оператора # означает, что выражение x # y # z читается как x # (y # z), а не (x # y) # z.

Далее идет два определения функций:

app склеивает два списка

```
primrec app :: "'a list ⇒ 'a list ⇒ 'a list" (infixr "@" 65) where
```

```
"[] @ ys = ys" |                    (*app.1 – первое уравнение*)
"(x # xs) @ ys = x # (xs @ ys)"     (*app.2 - второе*)
```

Алгоритм ее работы достаточно прост. В уравнении **app.1** говорится, что присоединение к списку **ys** пустого списка **[]** дает исходный список **ys**, что является интуитивно понятным.

Во втором уравнении мы видим, что склеивание двух списков происходит за счет процесса постепенного «перетаскивания» по одному элементов **x** из списка **x#xs** в список **ys**. При этом когда в левом списке заканчиваются элементы, он обращается в пустой список **[]**, происходит вызов **app.1** и завершение рекурсии.

Пример работы

```
(1#2#[]) @ 3#4#[]            (app.2)       "(x # xs) @ ys = x # (xs @ ys)"
= 1#(2#[] @ 3#4#[])          (app.2)       "(x # xs) @ ys = x # (xs @ ys)"
= 1#(2#([] @ 3#4#[]))        (app.1)       "[] @ ys = ys"
= 1#(2#(3#4#[])) = 1#2#3#4#[]
```

rev разворачивает список

```
primrec rev :: "'a list ⇒ 'a list" where
"rev [] = []" |
"rev (x # xs) = (rev xs) @ (x # [])"
```

Алгоритм работы rev схож с app.

Пример работы

```
rev 1#2#3#[] =               (rev.2)       "rev (x # xs) = (rev xs) @ (x # [])"
= (rev 2#3#[]) @ (1#[])      (rev.2)       "rev (x # xs) = (rev xs) @ (x # [])"
= (rev 3#[]) @ (2#(1#[]))    (rev.2)       "rev (x # xs) = (rev xs) @ (x # [])"
= (rev []) @ (3#(2#(1#[])))  (rev.1)       "rev [] = []"
= [] @ (3#(2#(1#[])))        (app.1)       "[] @ ys = ys"
= (3#(2#(1#[]))) = 3#2#1#[]
```

Каждое определение имеет вид:

primrec *name :: type (опциональный синтаксис)* **where** уравнения

Запись primrec означает что функция задана через «примитивную рекурсию». Примитивная рекурсия характеризуется тем, что во время каждого вызова ровно от *одного* из аргументов «отщепляется» ровно *один* конструктор. Примитивнорекурсивные функции всегда завершают работу и являются *всюду определенными* — для них не существует хорошо типизированного входного значения, для которого они не завершили бы свою работу.

После того как мы дали определения типу данных — list и двум функциям над ним — app и rev мы уже получили половину нашей теории. Чтобы закончить построение, мы должны исследовать свойства функций rev и app. Это достигается через доказательство теорем. По своей сути теорема это высказывание, которое всегда истинное для определенного класса объектов и которое при этом описывает некоторое (интересное нам) свойство данных объектов.

Теорема 1 — двойная реверсия списка

Теорема — если список дважды развернуть, получится исходных список.

```
value "a # b # c # []"
```

```
> "a # b # c # []"
value "rev (a # b # c # [])"
> "c # b # a # []"
value "rev (rev (a # b # c # []))"
> "a # b # c # []"
```

Формально:

```
theorem rev_rev [simp]: "rev (rev xs) = xs"
```

Данную теорему будем доказывать по индукции по длине списка. Сначала «пробежимся» по структуре доказательства, а потом внимательно разберем его по шагам.

```
theorem rev_rev [simp]: "rev (rev xs) = xs" (* формулируем теорему *)
apply (induct_tac xs)        (* применяем индукцию по длине списка xs*)
apply auto    (* автоматически доказываем подцели *)
sorry  (* для ее доказательства нужно доказать еще три леммы *)
```

Первая строка формирует цель:

```
theorem rev_rev [simp]: "rev (rev xs) = xs"
> 1. rev (rev xs) = xs
```

Применение тактики induct_tac во второй строке автоматически порождает нужное число подцелей:

```
apply (induct_tac xs)
goal (2 subgoals):
1. rev (rev []) = []  (* База *)
```

Первая подцель — если мы развернем пустой список, то получим пустой список, это база индукции. База индукции это шаг доказательства, показывающий что условие теоремы верно для базового случая, в нашей теореме это случай пустого списка (списка нулевой длины). Вспомните метафоры из лекции — это как первая ступень бесконечной лестницы, на которую нужно забраться, или как первая кость домино, которую нужно толкнуть. Мы как бы показываем, что действительно можем шагнуть на первую ступень или уронить первую костяшку.

```
2. ∧x1 x2. rev (rev x2) = x2 ⇒ rev (rev (x1 # x2)) = x1 # x2     (* Переход *)
```

Вторая подцель — индуктивный переход. Здесь x1 — это элемент списка x2. Пусть у нас есть список x2 длины k. Мы предполагаем, что теорема верна для данного списка x2 длины k, то есть **rev (rev x2) = x2**, это называется *индуктивной гипотезой*. Индуктивная гипотеза — это утверждение вида «теорема верна для любого списка длины 1,2...k». Снова обратимся к нашей метафоре. Мы говорим, что можем забраться, начиная с первой, на k-ю ступень лестницы, или мы говорим, что если толкнем первую кость домино, то по цепной реакции упадут все следующие кости вплоть до k-й.

Нам нужно показать, что индуктивный переход верен. То есть из того, что дважды развернув список x2 длины k мы снова получим тот же список x2 длины k *следует*, что дважды развернув список x1#x2 длины k+1 мы получим тот же список x1#x2 длины k+1.

$$rev\left(rev\underbrace{[...]}_{k}\right)=\underbrace{[...]}_{k}\Rightarrow rev\left(rev\underbrace{[....]}_{k+1}\right)=\underbrace{[....]}_{k+1}$$

Применим тактику auto, которая применяет упрощение и простые логические преобразования. Isabelle из двух имеющихся целей сформирует одну новую. То есть она

докажет базу и упростит переход.

Доказательство базы:

// Прим. мы можем использовать команду using [[simp_trace]] для трейсинга доказательства.

```
rev (rev []) = []     (rev.1)        "rev [] = []"
rev [] = []           (rev.1)        "rev [] = []"
[] = []               (=)            определение отношения эквивалентности
True
```

Упрощение перехода:

```
∧x1 x2. rev (rev x2) = x2 ⇒ rev (rev (x1 # x2)) = x1 # x2        (*Переход B1*)
```

Переписывается в:

```
∧x1 x2. rev (rev x2) = x2 ⇒ rev (rev x2 @ x1 # []) = x1 # x2     (*Переход B2*)
```

Isabelle использовала определение функции rev, а именно rev.2

```
"rev (x # xs) = (rev xs) @ (x # [])"
```

Вас может ввести в некоторое заблуждение второй вариант B2 так как кажется, что он не совсем соответствует результату применения rev.2 к варианту B1. В действительности, Isabelle упускает «лишние скобки» в записи целей и подцелей. Согласно правилу применения аргумента к функции, а также из правоассоциативности функции @ и # имеем соответствие:

```
(rev xs) @ (x # []) = rev xs @ x # []
```

Isabelle не может продолжить доказательство, так как не хватает Леммы, описывающей свойства взаимодействия rev и @. То есть Isabelle не может раскрыть выражение

```
rev xs @ x # []
```

Давайте взглянем на него более подробно. Здесь у **rev** есть два аргумента: **xs** и **x#[]**. Первый аргумент, **xs** является списком, **x#[]** также является списком. Следовательно, в нашей новой лемме в левой части уравнения должно стоять выражение

lemma «rev_app»: rev (xs @ ys) = … (* читайте далее *)

Лемма 1 — распределение rev между аргументами @

Нам осталось понять, что должно стоять в правой части. Можно предположить, что если мы соединим два списка, а затем их развернем (что соответствует **rev (xs @ ys)**), то получим тоже самое, как если бы мы соединили два развернутых списка (взятых в обратном порядке). Рассмотрим пример:

```
rev (1#2#[] @ 3#4#[]) = 4#3#2#1#[]
(rev 3#4#[]) @ (rev 1#2#[]) = (4#3#[]) @ (2#1#[]) = 4#3#2#1#[]
```

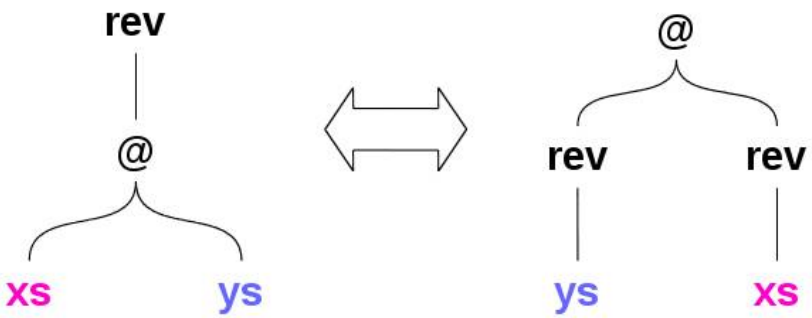
Мы видим, что для частного случая результат соответствует нашим ожиданиям. Это свойство похоже на дистрибутивность, однако отличается тем, что при вынесении rev за скобки порядок аргументов @ меняется:

rev (xs @ ys) = (rev ys) @ (rev xs),

или что тоже самое

(rev ys) @ (rev xs) = rev (xs @ ys)

На рисунке:



Сформулируем условие новой леммы формально и проведем доказательство в Isabelle:

```
lemma rev_app [simp]: "rev (xs @ ys) = (rev ys) @ (rev xs)"
apply (induct_tac xs)
apply auto
sorry
```

Для доказательства этой леммы нужно еще две леммы, однако мы используем ключевое слово **sorry** сообщая Isabelle что берем это факт без доказательства. Вернувшись к 1-й теореме обнаруживаем, что теперь доказательство прошло успешно.

```
lemma rev_app [simp]: "rev (xs @ ys) = (rev ys) @ (rev xs)"
apply (induct_tac xs)
apply auto
sorry

theorem rev_rev [simp]: "rev (rev xs) = xs" (*первая теорема – свойства rev*)
using [[simp_trace]]
apply (induct_tac xs)
apply auto
done
```

Третья и четвертая леммы

Обратив внимание на вывод Isabelle после применения тактики auto во время доказательства теоремы rev_app вы обнаружите

```
lemma rev_app [simp]: "rev (xs @ ys) = (rev ys) @ (rev xs)"
apply (induct_tac xs)
apply auto (* здесь *)
```

дает две подцели

```
1. rev ys = rev ys @ []
2. ∧x1 x2. rev (x2 @ ys) = rev ys @ rev x2 ⇒ (rev ys @ rev x2) @ x1 # [] = rev
ys @ rev x2 @ x1 # []
```

Для доказательства первой из них не хватает простого факта, а именно того, что если мы добавим ко списку **ys** справа пустой список, то получим **ys**.

```
rev ys = rev ys @ [] (*здесь справа*)
```

Формально это можно доказать при помощи простой леммы:

```
lemma app_Nil2 [simp]: "xs @ [] = xs"
apply (induct_tac xs)
apply auto
done
```

Для второй подцели нужно доказать равенство вида

(rev ys @ rev x2) @ x1 # [] = rev ys @ rev x2 @ x1 # []

Это ничто иное, как свойство ассоциативности:

$$x \mathbin{@} (y \mathbin{@} z) = (x \mathbin{@} y) \mathbin{@} z$$

Докажем простую лемму:

```
lemma app_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
apply (induct_tac xs)
apply auto
done
```

Готовая теория

Готовая теория имеет вид:

```
theory lab2
imports Main
begin
(*note: from Isabelle Tutorial*)
no_notation Nil("[]") and Cons (infix "#" 65) and append (infixr "@" 65)
hide_type list
hide_const rev

datatype 'a list = Nil ("[]")
| Cons 'a "'a list" (infixr "#" 65)

primrec app :: "'a list ⇒ 'a list ⇒ 'a list" (infixr "@" 65) where
"[] @ ys = ys" |
"(x # xs) @ ys = x # (xs @ ys)"

primrec rev :: "'a list ⇒ 'a list" where
"rev [] = []" |
"rev (x # xs) = (rev xs) @ (x # [])"

value "a # b # c # []"
value "rev (a # b # c # [])"
value "rev (rev (a # b # c # []))"

lemma app_Nil2 [simp]: "xs @ [] = xs"
apply (induct_tac xs)
apply auto
done

lemma app_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
apply (induct_tac xs)
apply auto
done

lemma rev_app [simp]: "rev (xs @ ys) = (rev ys) @ (rev xs)"
apply (induct_tac xs)
apply auto
done

theorem rev_rev [simp]: "rev (rev xs) = xs"
using [[simp_trace]]
apply (induct_tac xs)
apply auto
done

end
```

Поиск контрпримеров

Предположим, мы хотим проверить следующую «теорему»:

Пусть **xs** и **ys** — списки произвольной длины и пусть **xs @ ys** — конкатенация данных списков, тогда головой полученной конкатенации будет голова списка **xs**, то есть:

lemma "hd (xs @ ys) = hd xs"

Выглядит вполне правдоподобно. Начнем доказательство по индукции:

```
lemma "hd (xs @ ys) = hd xs"
apply (induct_tac xs)
apply auto
```

После применения auto получаем одну подцель:

```
goal (1 subgoal):
 1. hd ys = hd []
```

Данная цель утверждает: «для любого списка ys головой данного списка будет пустой список []», очевидно, что это абсурд. Для того чтобы доказать ложность нашей «теоремы» достаточно найти контрпример. Для этого можно использовать инструменты nitpick или quickcheck:

```
lemma "hd (xs @ ys) = hd xs"
apply (induct_tac xs)
apply auto
nitpick
oops
value "hd ([] @ [a1])"
```

После nitpick:

```
Nitpicking goal:
  hd ys = hd [].
Nitpick found a counterexample for card 'a = 2:

  Free variable:
    ys = [a₁]          // Контрпример, достаточно подставить [a1] вместо ys
```

После value:

```
"a1"
  :: "'a"
```

Использовать nitpick можно было сразу после условия леммы:

```
lemma "hd (xs @ ys) = hd xs"
nitpick
oops
```

Дает вывод:

```
Nitpicking formula...
Nitpick found a counterexample for card 'a = 2:

  Free variables:
    xs = []            // Вот он,
    ys = [a₁]          // контрпример
```

Также вы можете применять quickcheck:

```
lemma "hd (xs @ ys) = hd xs"
quickcheck
oops
```

Дает

```
Quickchecking...
Testing conjecture with Quickcheck-exhaustive...
Quickcheck found a potentially spurious counterexample due to underspecified
functions:
  xs = []      // Вот он,
  ys = []      // контрпример
Evaluated terms:
  hd (xs @ ys) = ?
  hd xs = ?
Quickcheck continues to find a genuine counterexample...
Quickcheck found no counterexample.
```

Полная теория

```
theory lab2b
imports Main
begin

(* note: from Isabelle NitPick tutorial *)
lemma "hd (xs @ ys) = hd xs"
apply (induct_tac xs)
apply auto
nitpick
oops
value "hd ([] @ [a1])"

lemma "hd (xs @ ys) = hd xs"
nitpick
oops

lemma "hd (xs @ ys) = hd xs"
quickcheck
oops

end
```

Заключение

В данной работе мы познакомились с базовыми техниками доказательств теорем о списках по индукции. Также был рассмотрен пример поиска контрпримеров для опровержения ложных теорем.

Упражнения

Определите функцию

- replace, такую что "replace x y zs" возвращает zs, где каждое вхождение x подменено y.

```
primrec replace :: "'a ⇒ 'a ⇒ 'a list ⇒ 'a list" where
  "replace x y []     = []"
| "replace x y (z#zs) = (if z=x then y else z)#(replace x y zs)"
```

Определите две функции для удаления элементов из списка:

- del1 x xs удаляет первое вхождение (слева) элемента x в списке xs
- delall x xs удаляет все вхождения x xs

```
primrec del1 :: "'a ⇒ 'a list ⇒ 'a list" where
"del1 x [] = []" |
"del1 x (y#ys) = (if (y=x) then ys else (y # (del1 x ys)))"
primrec delall :: "'a ⇒ 'a list ⇒ 'a list" where
"delall x [] = []" |
"delall x (y#ys) = (if (y=x) then (delall x ys) else (y # (delall x ys)))"
```

Выполните для нескольких теорем из списка ниже (соответственно варианту) задания

1. theorem "del1 x (delall x xs) = delall x xs"
2. theorem "delall x (delall x xs) = delall x xs"
3. theorem "del1 x (del1 y zs) = del1 y (del1 x zs)"
4. theorem "delall x (delall y zs) = delall y (delall x zs)"
5. theorem "del1 y (replace x y xs) = del1 x xs"
6. theorem "delall y (replace x y xs) = delall x xs"
7. theorem "replace x y (delall z zs) = delall z (replace x y zs)"
8. theorem "rev(del1 x xs) = del1 x (rev xs)"

Варианты и номера теорем:

1. 1,5
2. 2,5
3. 3,5
4. 4,5
5. 1,6
6. 2,6
7. 3,6
8. 4,6
9. 1,7
10. 2,7
11. 3,7
12. 4,7
13. 1,8
14. 2,8
15. 3,8
16. 4,8

*Если вам не хватило варианта — обратитесь к преподавателю. Для каждой теоремы из задания выполните следующее:

- Покажите на конкретных значениях, что функции replace, del1 и delall работают.
- Сформулируйте теорему на естественном языке.
- Выделите Базу и Переход, сформулируйте их на естественном языке.
- Как База и Переход связаны с определением списка?
- Как База и Переход связаны с определениями функций, задействованных в теореме?
- Если теорема верна, покажите это на конкретных значениях для Базы и Перехода.
- Если теорема неверна, объясните почему, покажите на конкретных значениях контрпример.