Материалы для подготовки

Пошаговое выведение свойств в Isabelle/HOL

Когтенков Александр Валентинович[1]

[1] «Лаборатория Касперского»

2025-11-04

Введение

Инструмент доказательства теорем Isabelle/HOL позволяет осуществить механизированную проверку утверждений в различных областях знаний, включая формальную верификацию программного обеспечения или его моделей. Задание, подготовленное в рамках соревнований VeHa-2025, нацелено не на знание инструмента и доказательство сложных утверждений, а на понимание логического вывода и обучение его использованию. Данный материал позволяет подготовиться к соревнованиям и покрывает практически все необходимые конструкции языка доказательств и особенности их применения. Рассматриваемые структуры данных ограничены списками, что, тем не менее, даёт возможность моделировать более сложные структуры с требуемым поведением. Доказательство таких свойств включено в задание соревнований.

Среда Isabelle/jEdit

Для соревнований используется версия Isabelle2025. Она может быть предустановлена на виртуальной машине либо установлена самостоятельно согласно инструкциям по адресу: https://isabelle.in.tum.de/website-Isabelle2025/installation.html. Далее предполагается, среда Isabelle установлена и настроена, а путь к исполняемому файлу `Isabelle2025` включён в пути поиска программ.

Установка.

Определения типов данных и функций, утверждения об их свойствах, а также более сложные конструкции организованы в Isabelle в модули, называемые теориями. Код теорий хранится в текстовых файлах с расширением `.thy`, который можно открыть в любом текстовом редакторе. Однако он использует множество специальных символов, задаваемых с помощью собственного языка разметки. Поэтому на практике используют адаптированные для этого инструменты. Одним из них является текстовый редактор jEdit, входящий в поставку инструментария Isabelle. Для его запуска используется команда `Isabelle2025` *имя_файла_теории*. Подготовительные материалы используют теорию ***`Odd_Property`***, поэтому запуск производится командой

```
Isabelle2025 Odd_Property.thy
```

Запуск.

В окне редактора часть символов преобразуется из вводимых ASCII символов автоматически:

Ввод	Результат	Значение	Логика
=>	$\Rightarrow$	Разделение типов	Мета
==>	$\Longrightarrow$	Импликация	
/\	$\wedge$	Конъюнкция	HOL
\/	$\vee$	Дизъюнкция	
–>	$\longrightarrow$	Импликация	
<–>	$\longleftrightarrow$	Эквивалентность	

Сочетания клавиш

Isabelle представляет среду, позволяющую реализовывать разные логики. Мета-логика относится к ядру Isabelle и используется для всех логик. HOL соответствует классической логике высших порядков. Другие примеры логик — FOL (логика первого порядка) и ZF (теория множеств Цермело—Френкеля).

Другие символы можно добавить, используя специальную разметку и авто-продолжение. Специальная разметка начинается с символа \. Вводить все остальные символы не обязательно. После нескольких символов и небольшого ожидания высветится подсказка с выбором либо можно вызвать её принудительно, нажав *Ctrl* + *B*. Выбрать подходящее продолжение можно либо мышью, либо стрелками и нажатием *Tab*.

Префикс	Продолжение	Значение	Логика
\comment	—	Комментарий	Мета
”	◊	«Скобки» для выражений	
\And	$\bigwedge$	Всеобщность	
\not	¬	Отрицание	HOL
\noteq	$\neq$	Неравенство	
\notin	$\notin$	Непринадлежность	
\in	$\in$	Принадлежность	
\forall	$\forall$	Всеобщность	
\exists	$\exists$	Существование	
\lambda	λ	Лямбда	

Авто-продолжение

В таблице указаны только наиболее часто используемые продолжения.

Рекурсивные типы

Язык Isabelle/HOL чисто функциональный, поэтому в нём отсутствуют изменяемые переменные. Так называемые «схематические переменные» и «переменные», используемые в функциях и выражениях, либо являются аргументами функций, либо явно или неявно связаны кванторами. Все они имеют определённый тип. Выражения являются полиморфными, т.е. в общем случае одна и та же переменная может иметь различные типы, которые имеют смысл для данного выражения. Кроме того, типы могут быть параметризованы другими типами, аналогично тому как это сделано в традиционных языках программирования.

Один из способов задания типа заключается в перечислении всех

возможных конструкторов его значений. В задании будут использованы два заданных таким образом стандартных типа: натуральные числа и списки. Натуральные числа с типом *nat* заданы конструкторами *0* и *Suc*. Последний имеет аргумент типа *nat* и означает число, следующее за аргументом. Таким образом, определение натуральных чисел следует аксиоматике Пеано.

Аналогичным образом задаются списки, тип которых тоже имеет два конструктора: *Nil* для пустого списка и *Cons* для ячейки со значением и хвоста списка. По типам, в которых есть базовый случай (без отсылки к определяемому типу) и индуктивный случай (с отсылкой на определяемый тип), можно строить рекурсивные функции и проводить доказательства по индукции.

Задание функции

Примитивно-рекурсивная функция определяется по индукции по одному из её аргументов с помощью конструкции **primrec**. Опишем свойство нечётности натуральных чисел функцией *is_odd*. Это сделано внутри теории *Odd_Property*. Правила вычисления функции для разных конструкторов помечены метками, чтобы их на них было удобнее ссылаться в доказательствах:

Isabelle/HOL позволяет задавать функции, не являющиеся примитивно-рекурсивными, например, функцию Аккермана, с помощью более продвинутых конструкций **fun**/**function**, но в задании они не используются.

- *odd_Zero* задаёт базовый случай: *0* — чётное число;
- *odd_Suc* задаёт случай индукции: *Suc (n)* нечётное тогда и только тогда, когда *n* чётное (не является нечётным).

```
theory Odd_Property imports Main begin

primrec is_odd :: ‹nat ⇒ bool› where
 odd_Zero: ‹is_odd 0 ⟷ False› |
 odd_Suc: ‹is_odd (Suc n) ⟷ ¬ is_odd n›

end
```

Чтобы не дублировать вспомогательный код, будем считать, что последующие фрагменты теории добавляются после спецификации функции *is_odd* и перед маркером окончания теории **end**.

Доказательство

Основной целью задания является написание *прямого* формального доказательства утверждения. «Прямое» означает, что ход рассуждений выводит заключение из предпосылок. Первым примером будет доказательство леммы, утверждающей, что определение функции *is_odd* гарантирует нечётность произвольного натурального числа либо числа,

следующего за ним: `is_odd n ∨ is_odd (Suc n)`.

Текст доказательства

Проведём доказательство индукцией по `n`:

1. Базовый случай.
   (a) Пусть `n = 0`.
   (b) Применяя правило `odd_Suc` к факту `¬ is_odd 0` из правила `odd_Zero`, получаем, что единица — нечётное число: `is_odd (Suc 0)`.
   (c) Поскольку значение одного из дизъюнктов истинно, утверждение леммы верно.
2. Шаг индукции.
   (a) Пусть `is_odd n ∨ is_odd (Suc n)`.
   (b) Докажем, что `is_odd (Suc n) ∨ is_odd (Suc (Suc n))`.
      i. Рассмотрим чётность `n`.
      ii. Случай, когда `is_odd n` истинно.
         A. Пусть `n` нечётно, т.е. `is_odd n`.
         B. Тогда следующее за ним число чётно: `¬ is_odd (Suc n)`.
         C. Тогда следующее за этим число опять нечётно: `is_odd (Suc (Suc(n))`.
         D. Поскольку значение одного из дизъюнктов истинно, утверждение шага индукции верно.
      iii. Случай, когда `is_odd n` ложно.
         A. Пусть `n` чётно, т.е. `¬ is_odd n`.
         B. Тогда следующее за ним число нечётно: `¬ is_odd (Suc n)`.
         C. Поскольку значение одного из дизъюнктов истинно, утверждение шага индукции верно. □

Код доказательства

Доказательства в Isabelle можно проводить аналогично тому, как это делается в других системах доказательств — от заключений к гипотезам. Для такого стиля доказательств используются **apply**-скрипты. Чтобы понять, что происходит на каждом шаге, необходимо проанализировать текущее состояние, включая цели и доступные предпосылки. Это хорошо подходит для проверки доказательства компьютером, так как оно получается почти линейным (с учётом ветвления при разборе случаев). Но такое доказательство плохо для понимания, поскольку не структурировано. Особенно неудобно, если под рукой нет инструмента, позволяющего перемещаться по скрипту и смотреть, что происходит с текущим состоянием.

Для сохранения структуры и читаемости был разработан специальный язык доказательств Isar. Он позволяет почти дословно перевести текст доказательства в формальный код Isabelle/HOL. Доказательство ниже построчно соответствует текстовому варианту. Для удобства строки помечены номерами пунктов доказательства.

```
lemma successive_is_odd: ‹is_odd n ∨ is_odd (Suc n)›
proof (induction n)                                                    0.
 case 0                                                                1. (a)
 have ‹is_odd (Suc 0)› using odd_Zero odd_Suc by blast                 1. (b)
 then show ‹is_odd 0 ∨ is_odd (Suc 0)› by blast                        1. (c)
next
 case (Suc n)                                                          2.
 assume ‹is_odd n ∨ is_odd (Suc n)›                                    2. (a)
 show ‹is_odd (Suc n) ∨ is_odd (Suc (Suc n))›                          2. (b)
 proof (cases ‹is_odd n›)                                              i.
  case True                                                            ii.
  assume ‹is_odd n›                                                    ii. A.
  then have ‹¬ is_odd (Suc n)› using odd_Suc by blast                  ii. B.
  then have ‹is_odd (Suc (Suc n))› using odd_Suc by blast              ii. C.
  then show ‹is_odd (Suc n) ∨ is_odd (Suc (Suc n))› by blast           ii. D.
 next
  case False                                                           iii.
  assume ‹¬ is_odd n›                                                  iii. A.
  then have ‹is_odd (Suc n)› using odd_Suc by blast                    iii. B.
  then show ‹is_odd (Suc n) ∨ is_odd (Suc (Suc n))› by blast           iii. C.
 qed
qed
```

Конструкции языка кратко описаны во введении в Isabelle/HOL [1]. Отдельно доступно полное руководство по языку [2]. Вот интуитивное описание использованных конструкций:

Конструкция	Примерное значение
lemma *L*: *E*	Лемма *L*. *E* — утверждение леммы.
proof *M* ⋮	Докажем с помощью метода *M*. ⋮
qed	Утверждение доказано.
case *V*	Рассмотрим случай *V*.
fix *X*	Возьмём произвольное *X*.
assume *E*	Пусть *E*.
next	Переходим к следующему случаю.
have *E* [**using** *L*] **by** *M*	Имеем *E* [используя утверждения *L*] с помощью метода *M*.
then have …	Из предыдущего утверждения имеем …
with *F* **have** …	Из предыдущего и фактов *F* имеем …
from *F* **have** …	Из фактов *F* имеем …
show …	Докажем …

Методы описаны в разделе Методы доказательства.

При этом предыдущий случай должен быть «закрыт», используя **show**.

show ведёт себя как **have** плюс помечает цель, которую надо доказать, как доказанную.

Методы доказательства

Внутри Isabelle зашита механика проверки доказательств. Она имеет строгую теоретическую базу с очень маленьким ядром, что делает систему надёжной. Доказательства проверяются ядром, гарантируя отсутствие ошибок. Как эти доказательства получены, ядру всё равно. Это открывает дорогу к использованию автоматизированных средств, скрывающую большую часть сложности внутри инструмента. Частью таких средств являются методы, которые по собственным правилам и эвристикам ищут доказательства и предъявляют их ядру для проверки. Некоторые методы скрывают значительное число промежуточных шагов и позволяют существенно сократить код доказательств. Это сильно помогает при практическом применении, но иногда делает доказательства не очень понятными, т.к. скрывают внутренние подробности.

Ограничение на методы в задании. В сложных доказательствах даже продвинутые методы не всегда могут найти подходящие решения, и это приходится делать специалисту самостоятельно. Представленное задание нацелено как раз на понимание и умение провести доказательство максимально понятно для человека. Поэтому для решения задания достаточно использовать только следующие методы:

Использование других методов снизит оценку за выполнение задания.

- `simp`: упрощение выражений, основанное на правилах переписывания;
- `blast`: простое использование правил вывода;
- `induction` *E*: индукция по значению выражения *E*;
- `cases` *E*: анализ значений выражения *E*.

Завершающее доказательство. Первые два метода позволяют установить истинность текущего утверждения. В случае успеха можно переходить к доказательству следующего утверждения. Конструкции **by** `simp` и **by** `blast` подразумевают именно такое использование. Они являются сокращением **proof** `simp` **qed** и **proof** `blast` **qed**.

Разбиение над подцели. Вторые два метода подразумевают разбиение доказательства на две и более ветви, каждую из которых, должно быть проще доказать. Оба метода имеют как минимум один аргумент. Обязательным аргументом является выражение, по которому будет проводиться доказательство по индукции или разбором случаев. Методы с аргументами необходимо заключать в круглые скобки. Такие доказательства начинаются с конструкций **proof** `(induction E)` и **proof** `(cases E)`. При использовании индукции важно сообщить методу все факты, которые будут участвовать в индукции, используя **then** или **from**/**with** `P`.

Предположения в подцелях. Конструкция **case** используется для удобства и сокращения кода. Она вводит в контекст предположения, соответствующие выбранному значению выражения, использованному в `cases`/`induction`. Посмотреть предположения, вводимые командой **case**, можно, установив курсор после этой команды, в панели *State*. Таким образом, все явно выписанные предположения в лемме `successive_is_odd`, т.е. все строки с **assume**, можно удалить. Можно сделать и наоборот: удалить строки с **case**, оставив строки с **assume**. При этом для случая **case** `(Suc n)` необходимо будет сообщить об использовании произвольного значения `n` с помощью **fix** `n`.

Найдите панель *State* в интерфейсе Isabelle/jEdit справа.

Проделайте указанные манипуляции с кодом, наблюдая за состоянием в панели *State*, чтобы понять, как это работает.

Специальные переменные целей и подцелей. Конструкция **proof** автоматически вводит переменную `?thesis`, содержащую выражение, которое необходимо доказать. Поэтому при разборе случаев её можно использовать в конструкции **show** вместо явного выражения: **show** `?thesis` … . Аналогично, при индукции, **case** вводит переменную `?case`, соответствующую выбранному случаю, что позволяет использовать **show** `?case` … .

Убедитесь в этом, заменив выражения в строках ii.D. и iii.C. на `?thesis`, а в строках 1.(c) и 2.(b) на `?case`.

Панель состояния. Перемещая курсор по тексту теории, в панели *State* можно видеть, какие цели (и подцели) надо доказать в данном контексте. Сверху панели перечислены выбранные для данного утверждения факты, а затем указана сама цель.

Незаконченные доказательства. Сразу написать корректное доказательство удаётся не всегда. Чтобы продолжить работу над текстом, который не проходит проверку корректности, есть две конструкции. **oops** «прерывает» доказательство на любом шаге вложенности и возвращает контекст на верхний уровень, где задаются функции, вводятся леммы и т.д. **sorry** «пропускает» конкретный шаг доказательства, т.е. заменяет **proof** …**qed** и **by** … , не касаясь остальной структуры доказательства. И в том и в другом случае доказательство не считается завершённым, а лишь позволяет отложить его на будущее.

Понаблюдайте эффект от **oops** и **sorry** на доказательствах выше.

Доказательства с более сложной структурой

Последовательность фактов

В доказательстве леммы `successive_is_odd` достаточно было ссылаться на факты, доказанные непосредственно очередным утверждением, что позволяло обойтись конструкцией **then**. Обычно же утверждения зависят не от одного факта, а от нескольких. Одним из способов указать это является конструкция с ключевыми словами **moreover** и **ultimately**:

Конструкция	Примерное значение
⋮	⋮
moreover have …	Кроме того имеем…
moreover have …	Кроме того имеем…
⋮	⋮
ultimately have …	Из всего этого имеем…

Последнее утверждение может использовать **show**, если именно его надо сейчас доказать.

Ниже приведён пример «сбора» двух фактов для доказательства финального утверждения в каждой ветке разбора случаев:

```
 — Целое или следующее за ним число нечётно.
lemma successive_exclusive_is_odd: ‹is_odd n ≠ is_odd (Suc n)›
 — Рассмотрим случаи чётности n.
proof (cases ‹is_odd n›)
    — Случай, когда is_odd n истинно.
 case True
  — n нечётно.
 assume ‹is_odd n›
  — Кроме того тогда n + 1 чётно.
 moreover then have ‹¬ is_odd (Suc n)› using odd_Suc by blast
  — Из вышесказанного следует, что утверждение леммы верно.
 ultimately show ‹is_odd n ≠ is_odd (Suc n)› by blast
next
  — Случай, когда is_odd n ложно.
 case False
  — n чётно.
 assume ‹¬ is_odd n›
  — Кроме того тогда n + 1 нечётно.
 moreover then have ‹is_odd (Suc n)› using odd_Suc by blast
  — Из вышесказанного следует, что утверждение леммы верно.
 ultimately show ‹is_odd n ≠ is_odd (Suc n)› by blast
qed
```

Использование собственных лемм

Часто при доказательстве одного утверждения приходится выявлять и доказывать другие. Например, надо доказать, что произведение нечётных чисел нечётно тогда и только тогда, когда они все нечётны. Для начала можно рассмотреть случай двух множителей (вариант с одним множителем тривиален): `is_odd (x * y)` $\longleftrightarrow$ `is_odd x` $\wedge$ `is_odd y`. Поначалу может показаться, что доказательство легко провести индукцией по одному, а затем по другому множителю. Но на практике доказательство быстро становится нереально длинным и запутанным. Вместо этого можно заметить, что любое число представимо в виде $2k$ или $2k + 1$, провести операцию умножения, упростить получившиеся выражения и убедиться в корректности утверждения в зависимости от того, являются множители

Попробуйте проделать это самостоятельно.

нечётными или нет.

Таким образом, нам потребуются следующие леммы:

- `2 * k` чётно;
- `2 * k + 1` нечётно;
- любое число представимо в виде `2 * k` или `2 * k + 1`.

Первая лемма доказывается по индукции:

Попробуйте доказать её самостоятельно.

```
lemma two_multiple_is_even: <¬ is_odd (2 * n)>
```

Вторую лемму тоже можно доказать по индукции. Но она же является следствием предыдущей леммы с учётом определения функции is_odd:

```
lemma non_two_multiple_is_odd: <is_odd (Suc (2 * n))>
 using two_multiple_is_even odd_Suc by blast
```

Этот пример показывает, что завершающее доказательство можно применять непосредственно к утверждению леммы.

Использование квантора существования

Третья лемма из предыдущего раздела утверждает, что число представимо в виде `2 * k` или `2 * k + 1`:

```
lemma odd_even_representation: <∃ k. n = 2 * k ∨ n = Suc (2 * k)>
```

Доказывается она индукцией по `n`. При этом в индуктивном шаге из предположения, что число `n` представимо, необходимо вывести, что `Suc n` тоже представимо. Для этого необходимо получить значение `k`, что достигается с помощью конструкции **obtain**:

Конструкция	Примерное значение
[**from** … | **with** … | **then**] **obtain** *X* **where** *E* **by** M	[Из … | Вместе с … | Тогда] возьмём *X* такой что *E* с помощью метода *M*.

После этого в контексте появляется переменная, заданная в **obtain** и утверждение, заданное в **where**, которые можно использовать как обычно:

```
lemma odd_even_representation: ‹∃ k. n = 2 * k ∨ n = Suc (2 * k)›
proof (induction n)
 case 0
 show ?case by simp
next
 case (Suc n)
 then obtain k where ‹n = 2 * k ∨ n = Suc (2 * k)› by blast
 show ?case
 proof (cases ‹n = 2 * k›)
  case True
  then show ?thesis by blast
 next
  case False
  with ‹n = 2 * k ∨ n = Suc (2 * k)› have ‹n = Suc (2 * k)› by blast
  then have ‹Suc n = Suc (Suc (2 * k))› by blast
  then have ‹Suc n = 2 * (Suc k)› by simp
  then show ?thesis by blast
 qed
qed
```

case *(Suc n)* неявно задаёт **assume** ‹∃ *k*. *n* = 2 * *k* ∨ *n* = Suc (2 * *k*)›

Переменные *?case* и *?thesis* описаны ранее.

Почему доказательство для случая $n \neq 2 * k$ заметно длиннее?

Ссылки на предыдущие утверждения

Доказательства выше ссылались на ранее доказанные леммы или отдельные случаи из определения примитивно-рекурсивной функции по имени. Это достигалось за счёт добавления имени перед выражением: *name*: ‹*expression*›. Точно такой же механизм используется для ссылок на промежуточные утверждения, выводимые в процессе доказательства. В частности, имена можно указывать перед выражениями в **have** и **where**, а также для именования случаев после **case**. Например, доказательство для леммы *is_odd (x * y)* ⟷ *is_odd x* ∧ *is_odd y* ниже проводится разбором случаев чётности переменных *x* и *y*. Оба случая имеют значения *True* и *False*, что не позволяет использовать имена значений в качестве имён ссылок и для *x*, и для *y*. Явные имена случаев обходят это ограничение.

```
lemma is_odd_mult: ‹is_odd (x * y) ⟷ is_odd x ∧ is_odd y›
proof -
 obtain m where
  X: ‹x = 2 * m ∨ x = Suc (2 * m)› using odd_even_representation by blast
 obtain n where
  Y: ‹y = 2 * n ∨ y = Suc (2 * n)› using odd_even_representation by blast
 show ‹is_odd (x * y) ⟷ is_odd x ∧ is_odd y›
 proof (cases ‹is_odd x›)
  case oddX: True
  with X have
   M: ‹x = Suc (2 * m)› using two_multiple_is_even by blast
  show ?thesis
  proof (cases ‹is_odd y›)
   case oddY: True
   with Y have ‹y = Suc (2 * n)› using two_multiple_is_even by blast
   with M have ‹x * y = Suc (2 * m) * Suc (2 * n)› by blast
   then have ‹x * y = Suc (2 * (Suc (2 * m) * n + m))› by simp
   moreover have ‹is_odd (Suc (2 * (Suc (2 * m) * n + m)))›
    using non_two_multiple_is_odd by blast
   ultimately show ?thesis using oddX oddY by simp
  next
   case evenY: False
   with Y have ‹y = 2 * n› using non_two_multiple_is_odd by blast
   with M have ‹x * y = Suc (2 * m) * (2 * n)› by blast
   then have ‹x * y = 2 * (Suc (2 * m) * n)› by simp
   moreover have ‹¬ is_odd (2 * (Suc (2 * m) * n))›
    using two_multiple_is_even by blast
   ultimately show ?thesis using oddX evenY by simp
  qed
 next
  case evenX: False
  with X have ‹x = 2 * m› using non_two_multiple_is_odd by blast
  then have ‹x * y = 2 * (m * y)› by simp
  then have ‹¬ is_odd (x * y) ⟷ ¬ is_odd (2 * (m * y))› by (simp only:)
  moreover have ‹¬ is_odd (2 * (m * y))›
   using two_multiple_is_even by blast
  ultimately have ‹¬ is_odd (x * y)› by simp
  with evenX show ‹is_odd (x * y) ⟷ (is_odd x ∧ is_odd y)› by blast
 qed
qed
```

До перехода к доказательству формулируются дополнительные утверждения, поэтому вместо метода после **proof** стоит минус.

simp упрощает выражения, используя множество фактов. Их применение не всегда успешно, как в данном случае. Список после ***only:*** задаёт, какие правила использовать, в данном случае никакие.

Кванторы всеобщности и индукция

Во многих леммах, приведённых выше, часть переменных не связана явным образом. В лемме odd_even_representation переменная ***k*** связана квантором существования, а переменная ***n*** не связана. Самая первая лемма successive_is_odd содержит только одну переменную ***n***, и она тоже не связана. Что это означает? Свободные переменные интерпретируются как имеющие «произвольное» значение. Лемма интерпретируется как «для

произвольного значения *n*, нечётно либо *n*, либо *Suc n*». Из этого можно вывести утверждение ∀ *n. is_odd n* ∨ *is_odd (Suc n)*, где *n* будет связана. Однако выражения со свободными переменными гораздо удобнее использовать в доказательствах, т.к. требуется меньше манипуляций, чтобы подставить интересующие значения вместо свободной переменной. Выражение со свободной переменной можно переписать в связанном виде как ⋀ *n. is_odd n* ∨ *is_odd (Suc n)*. Это важно учитывать в доказательствах, где значение переменной произвольно, а не фиксировано.

Следующая лемма is_odd_foldr обобщает предыдущую лемму на любое число множителей. В её доказательстве индукцией по списку этих множителей метод *induction* содержит дополнительный параметр, указывающий, что значение переменной *y* произвольно. В данном доказательстве это несущественно, но в других случаях это может стать критически важно. При этом гипотеза индукции *IH* принимает форму с указанием «произвольности» значения переменной *y*.

Раздел Рекурсивные типы упоминает рекурсивную природу списков и связанные с ними конструкторы.

```
lemma is_odd_foldr:
 ‹is_odd (foldr (*) xs y) ⟹ z ∈ set xs ⟹ is_odd z›
proof (induction xs arbitrary: y)
 case Nil
 then show ‹is_odd z› by simp
next
 case (Cons x xs)
 assume Prem: ‹is_odd (foldr (*) (x # xs) y)›
 assume IH: ‹⋀ y. is_odd (foldr (*) xs y) ⟹ z ∈ set xs ⟹ is_odd z›
 from Prem have ‹is_odd (x * foldr (*) xs y)› by simp
 with IH have ‹z ∈ set xs ⟹ is_odd z› using is_odd_mult by blast
 moreover from ‹is_odd (x * foldr (*) xs y)› have ‹is_odd x›
  using is_odd_mult by blast
 moreover assume ‹z ∈ set (x # xs)›
 then have ‹z = x ∨ z ∈ set xs› by simp
 ultimately show ‹is_odd z› by blast
qed
```

foldr применяет указанную функцию (умножение) к каждому элементу списка *xs* и ранее вычисленному значению, начиная с *y*.

x # xs эквивалентно *Cons x xs*.

Функция *set* выдаёт множество значений всех элементов указанного списка.

Квантификации подвержены переменные не только для данных, но и для функций. В лемме ниже используется функция просеивания *P*, которая по заданному значению сообщает, подходит ли оно или нет. Библиотечная функция *filter* оставляет в списке только те элементы, которые «подходят». Хотя значение абстрактной функции *P* неизвестно, её можно использовать, чтобы вывести, при каких условиях элемент списка точно является нечётным. А именно, если произведение элементов списка, отфильтрованных по признаку *P*, нечётно, то произвольный элемент этого списка с указанным признаком является нечётным.

```
lemma is_odd_filter:
 ‹is_odd (foldr (*) (filter P xs) y) ⟹ x ∈ set xs ⟹ P x ⟹ is_odd x›
proof -
 assume ‹is_odd (foldr (*) (filter P xs) y)›
 then have ‹∀ x ∈ set (filter P xs). is_odd x› using is_odd_foldr by blast
 moreover assume ‹x ∈ set xs› ‹P x›
 ultimately show ‹is_odd x› by simp
qed
```

В связке ⟹ последнее выражение — цель, а все предыдущие — предположения.

Правила `blast` позволяют переходить от «произвольный» (⋀) к «для всех» (∀).

Ещё про функции

Функции свёртки

Стандартная библиотека содержит не только функцию свёртки `foldr`, но и две другие: `fold` и `foldl`. Все они применяют функцию объединения к элементам списка, чтобы в конце концов получить на выходе результат объединения этих элементов. Отдельным аргументом передаётся начальное значение, которое как используется для первого объединения с элементом списка, так и является непосредственным результатом, когда список пуст. Обработка элементов списка идёт следующим образом:

Стандартная библиотека загружается в теорию командой **imports** `Main`, как видно во фрагменте определения функции is_odd.

`fold` сначала свёртывает голову списка, используя результат в качестве правого аргумента для последующей свёртки;
`foldr` сначала свёртывает хвост списка, а потом уже голову;
`foldl` сначала свёртывает голову списка, используя результат в качестве левого аргумента последующей свёртки.

Все функции свёртки принимают функцию объединения элементов в качестве первого аргумента. Для первых двух вариантов вторым аргументом идёт список элементов, а третьим — начальное значение. Для последнего варианта порядок второго и третьего аргумента обратный.

Функция объединения элементов — бинарная. Типы её операндов могут различаться. Тип одного операнда всегда соответствует типу элемента списка, на котором делается свёртка. Тип другого операнда всегда соответствует типу результата свёртки. Если функцию объединения обозначить через ·, то формулы для каждой из функций свёртки будут следующие:

$$\texttt{fold}\ (\cdot)\ [x_1, x_2, ..., x_{n-1}, x_n]\ y_0 = x_n \cdot (x_{n-1} \cdot ... \cdot (x_2 \cdot (x_1 \cdot y_0))...)$$
$$\texttt{foldr}\ (\cdot)\ [x_1, x_2, ..., x_{n-1}, x_n]\ y_0 = x_1 \cdot (x_2 \cdot ... \cdot (x_{n-1} \cdot (x_n \cdot y_0))...)$$
$$\texttt{foldl}\ (\cdot)\ y_0\ [x_1, x_2, ..., x_{n-1}, x_n] = (...((y_0 \cdot x_1) \cdot x_2) \cdot ... \cdot x_{n-1}) \cdot x_n$$

Безымянные функции

Можно заметить, что функции `fold` и `foldl` из предыдущего подраздела различаются лишь порядком следования аргументов. Это подтверждает

следующая лемма:

```
lemma ‹fold f xs y₀ = foldl (λ y x. f x y) y₀ xs›
 by (simp add: foldl_conv_fold)
```

Стандартная лемма `foldl_conv_fold` выражает `foldl` через `fold`.

В её формулировке использовано лямбда-выражение. Оно позволяет ввести функцию, не задавая её отдельно и не указывая её имя. После символа λ перечисляются аргументы неименованной функции, а после символа точки идёт определение этой функции.

Несколько других стандартных функций также могут быть выражены через функцию свёртки. Например, функция `map`, которая полностью сохраняет структуру списка, но заменяет каждый элемент на значение заданной функции для этого элемента:

```
lemma ‹map f xs = foldr (λ y ys. Cons (f y) ys) xs []›
lemma ‹map f xs = foldr (λ y. Cons (f y)) xs []›
lemma ‹map f xs = foldr (Cons ∘ f) xs []›
```

Доказательства всех трёх лемм идентичны и проводятся по индукции. Проведите их самостоятельно.

В первой лемме все аргументы передаваемой функции выписаны явно. При этом последний аргумент под λ также является последним для вызова `Cons`, поэтому его можно опустить, как сделано во второй лемме. Наконец, последовательное применение двух функций является композицией, что учтено в третьей лемме, где лямбда-выражение полностью пропало.

Анализ значений

Предположим, необходимо задать функцию, которая сохраняет от списка только префикс заданной длины или весь список, если его длина меньше заданной. Функция принимает два аргумента, по обоим из которых возможна рекурсия. Однако, декларация с использованием **primrec** допускает рекурсию только по одному аргументу. Для анализа другого аргумента можно использовать конструкцию разбора значений индуктивного типа:

```
(case Выражение of
  Конструктор₁ парам₁₁ … парам₁ₘ₁ ⇒ Выражение₁ |
  Конструктор₂ парам₂₁ … парам₂ₘ₂ ⇒ Выражение₂ |
    ⋮
  Конструкторₙ paramₙ₁ … paramₙₘₙ ⇒ Выражениеₙ
)
```

В зависимости от значения *Выражения* выбирается ветка с подходящим конструктором, означиваются соответствующие параметры конструктора, и вычисляется выражение в выбранной ветке. При вычислении можно использовать ранее означенные параметры. С помощью такого анализа

необходимую функцию можно задать с рекурсией по второму аргументу (списку) и анализом по первому (длине префикса):

```
primrec keep :: <nat ⇒ 'a list ⇒ 'a list> where
 <keep n [] = []> |
 <keep n (x # xs) = (case n of 0 ⇒ [] | Suc m ⇒ x # keep m xs)>
```

Разбор для непустого списка ведётся по конструкторам натуральных чисел: `0` и `Suc m`. Если не `0`, сохраняем голову списка и добавляем на единицу короче префикс от хвоста.

Оказывается, именно так и задана стандартная функция `take`. Следующая лемма подтверждает, что обе функции ведут себя одинаково:

```
lemma keep_take_eq: <keep n xs = take n xs>
```

Доказательство по индукции с разбором случаев. Проведите его самостоятельно.

Область определения функций

В Isabelle/HOL все функции всюду определены. Звучит очень странно, но это позволяет не заботиться, как писать выражения, и даёт возможность задавать «удобные» значения функций в точках, где они обычно не имеют смысла, так, чтобы формулы не приходилось усложнять. В большинстве случаев, если функция на каких-то аргументах не имеет смысла, то она возвращает специальное значение `undefined`.

Есть исключения. Например, деление на ноль даёт ноль. Вопрос, почему так, об этом периодически всплывает в почтовой рассылке, посвящённой инструменту.

Для списков, некоторые функции не имеют смысла, если список пуст:

- `hd` возвращает «голову» списка: `hd (x # xs) = x`;
- `tl` возвращает «хвост» списка: `tl (x # xs) = xs`.

Для пустого списка эти функции имеют значение `undefined`, поэтому доказать `hd xs # tl xs = xs` не получится.

Но получится доказать `hd (x # xs) # tl (x # xs) = x # xs`.

Аналогично, функция доступа к элементам списка по индексу `xs ! i`, которая возвращает `i`-ый элемент списка `xs` (нумерация начинается с нуля), осмыслена только для индексов, меньших длины этого списка `length xs`. Чтобы ей пользоваться, необходимо добавлять условие, что функция будет иметь смысл, например,

- `length xs > 0 ⟹ xs ! 0 = hd xs`
- `xs ≠ [] ⟹ xs ! 0 = hd xs`
- `0 < i ⟹ i < length xs ⟹ xs ! i = tl xs ! (i - 1)`

Задание для самоподготовки

Теория `Even_Property` содержит пример для самостоятельного решения при подготовке к соревнованиям. Необходимо завершить теорию с учётом следующих ограничений:

Запуск:
`Isabelle2025 Even_Property.thy`

- список импортируемых теорий должен остаться неизменным;
- нельзя использовать определения и леммы из теории `Odd_Property`;

- допустимыми методами доказательства являются только *blast*, *cases*, *induction*, *simp*;
- доказательство должно быть прямым и не использовать **apply**-скрипты.

Термин «прямое» рассмотрен в разделе Доказательство.

В теории *Even_Property* необходимо:

1. Определить примитивно-рекурсивную функцию *is_even*, сообщающую, является ли заданное натуральное число чётным.
2. Доказать лемму *successive_is_even*, что произвольное натуральное число или следующее за ним число чётно.
3. Доказать лемму *successive_exclusive_is_even*, что два последовательных натуральных числа имеют разную чётность в соответствии с функцией *is_even*.
4. Доказать лемму *is_even_mult*, что произведение двух произвольных натуральных чисел чётно тогда и только тогда, когда хотя бы одно из них чётно.

   Возможно, для доказательства потребуются вспомогательные леммы.
5. Доказать лемму *is_odd_foldr*, использующую функцию *is_even*, что если произведение списка множителей нечётно, то произвольный элемент этого списка — нечётный.
6. Доказать лемму *is_odd_filter*, что если произведение множителей, отфильтрованных про некоторому свойству, нечётно, то произвольный множитель из этого списка с этим свойством — нечётный.
7. Доказать лемму *odd_expr_even_mult*, что если значение выражения *(x + y) * z* нечётно, то произведение *x * y* чётно.

   Возможно, для доказательства потребуются вспомогательные леммы.

Список литературы

[1]Tobias Nipkow. *Programming and Proving in Isabelle/HOL*. 13 марта 2025. url: https://isabelle.in.tum.de/dist/Isabelle2025/doc/prog-prove.pdf.

[2]Makarius Wenzel. *The Isabelle/Isar Reference Manual*. 13 марта 2025. url: https://isabelle.in.tum.de/dist/Isabelle2025/doc/isar-ref.pdf.