Инструмент интерктивного доказательства теорем Isabelle/HOL

К. В. Зиборов

Московский Государственный Университет им. М. В. Ломоносова

2023

1 Введение

2 λ-исчисление

3 Термы и типы в Isabelle

4 Доказательства в Isabelle

5 FOL

6 Автоматизация

7 HOL

8 Симплификатор

9 Индукция и рекурсия

10 Isar

Мотивация

- Сложные системы почти наверняка содержат ошибки
- Критически важные системы (например, ПО ВС) должны соответствовать очень высоким стандартам. На практике их невозможно достичь простым тестированием

Формальная верификация

Формальная верификация программы — это математические методы доказательства того, что модель программы удовлетворяет заданной спецификации или свойствам корректности.

Основные методы:

- Дедуктивная верификация (в средствах интерактивного доказательства теорем)
- Model Checking
- Статический, динамический анализ, и т. п.

Формальная верификация

Формальная верификация программы — это математические методы доказательства того, что модель программы удовлетворяет заданной спецификации или свойствам корректности.

Основные методы:

- Дедуктивная верификация (в средствах интерактивного доказательства теорем)
- Model Checking
- Статический, динамический анализ, и т. п.

Формальная верификация может обеспечить отсутствие ошибок в модели (не в реальной системе). Поэтому, тестирование и верификация дополняют друг друга.

Формальные доказательства

Формальное доказательство

- проводится в рамках используемой логики
- проверяется компьютером
- надежное, однако могут быть ошибки в устройстве, ОС, рантайме, компиляторе и самой модели
- не содержит красивых математических идей
- в основном тривиальные, однако могут быть очень трудозатратными

Математическое доказательство

- неформальное
- проверяется людьми, которые не всегда могут найти ошибки
- обычно содержит новую красивую идею

Формальные доказательства

Доказательство (формальное) вручную

- трудозатратно, приходится доказывать тривиальные вещи вручную
- тяжело держать в уме все предположения и предусловия
- тяжело модифицировать

Автоматическое доказательство

- сильно упрощает работу
- все ещё бессильно для многих утверждений
- тяжело понять что пошло не так в случае провала

Формальные доказательства

Пример формального доказательства

$$A \wedge B \longrightarrow B \wedge A$$

Имеются правила:
$\frac{X \in S}{S \vdash X}$ **(assumption)**, $\frac{S \cup \{X\} \vdash Y}{S \vdash X \longrightarrow Y}$ **(implI)**, $\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}$ **(conjI)**, $\frac{S \cup \{X, Y\} \vdash Z}{S \cup \{X \wedge Y\} \vdash Z}$ **(conjE)**
Доказательство:
1. $\{A, B\} \vdash B$ by **assumption**
2. $\{A, B\} \vdash A$ by **assumption**
3. $\{A, B\} \vdash B \wedge A$ using 1, 2 by **conjI**
4. $\{A \wedge B\} \vdash B \wedge A$ using 3 by **conjE**
5. $\{\} \vdash B \wedge A \longrightarrow A \wedge B$ using 4 by **implI**

Интерактивное доказательство = автоматическое + вручную

- Задачи пользователя:
  - Формализация проблемы
  - Структурирование доказатеств
- Задачи компьютера:
  - Автоматическое доказательство простых утверждений
  - Сохранение предположений
  - Проверка доказательств в целом

Инструменты интерактивного доказательства

Перечислим основные инструменты интерактивного доказательства:

- LCF-подход
  - Isabelle
  - HOL family
- Явные доказательства + proof checker:
  - Coq
  - Lean
  - Isabelle
  - Agda
  - HOL4
- И другие...

Isabelle

Isabelle — интерактивный инструмент для доказательства теорем, реализующий логику высшего порядка.

- Содержит компактное логическое ядро Pure, которое можно принимать в качестве истинного без дополнительных доказательств.
- Реализует металогику, которая используется для реализации нескольких вариантов объектной логики Isabelle, таких как логика первого порядка (FOL), логика высшего порядка (HOL) и теория множеств Цермело-Френкеля (ZFC). Чаще всего используется Isabelle/HOL.

Мета-логика

Мета-логика — логика, используемая для формализации других логик.
Мета-язык (ML) — язык для описания других языков.
Мета-логика Isabelle = $\{\bigwedge, \implies, \lambda\}$

Архитектура Isabelle

1) Prover IDE (jEdit) — пользовательский интерфейс
2) HOL, ZF — объектные логики
3) Isabelle — средство интерактивного доказательства
4) Standart ML — логический уровень

Мы можем взаимодействовать с инструментом на всех уровнях!

Архитектура Isabelle

1) Prover IDE (jEdit) — пользовательский интерфейс
2) HOL, ZF — объектные логики
3) Isabelle — средство интерактивного доказательства
4) Standart ML — логический уровень
Мы можем взаимодействовать с инструментом на всех уровнях!

Установить можно с официальной страницы https://isabelle.in.tum.de

λ-исчисление

Мета-логика Isabelle = $\{\bigwedge, \Longrightarrow, \lambda\}$

$\bigwedge$ — квантор всеобщности на мета-уровне,
$\Longrightarrow$ — мета-импликация,
Что такое λ?

λ-исчисление

λ-исчисление (англ. lambda calculus) — формальная система, придуманная в 1930-х годах Алонзо Чёрчем.
λ-исчисление — основа функционального программирования.

- Запишем функцию $f(x) = x + 3$ как $\lambda x.\ x + 3$
  $\lambda x.\ x + 3$ — терм и безымянная функция.

Вычисление $(\lambda x.\ t)\ y$: в терме t заменяем x на y

λ-исчисление

Синтаксис

Мн-во λ-термов — минимальное мн-во Λ такое, что:

0) $\mathit{Const} \subseteq \Lambda$,

1) $\mathit{Var} \subseteq \Lambda$,

2) Если $t \in \Lambda$, $x \in \mathit{Var}$, то $(\lambda x.\ t) \in \Lambda$

3) Если $t, z \in \Lambda$, то $(tz) \in \Lambda$

λ-исчисление

Соглашение о скобках:

- внешняя пара скобок терма обычно опускается;
- вместо $(\ldots(X_1 X_2) X_3)\ldots X_n)$ пишут $X_1 X_2 \ldots X_n$ (скобки по умолчанию группируются влево);
- вместо $\lambda.\ (AB)$ пишут $\lambda.\ AB$ (квантор λ имеет меньший приоритет, чем аппликация);
- вместо $(\lambda x_1.(\lambda x_2.(\ldots\lambda x_n.A)\ldots))$ пишут $\lambda x_1 x_2 \ldots x_n.\ A$

β-редукция

Замена параметра на значение аргумента называется β **- редукцией**:

- $(\lambda x.\ A)B \longrightarrow_\beta A[B/x]$ (все свободные вхождения переменной x в A заменяются на терм B (при этом предполагается, что при подстановке в A свободные переменные терма B не попадают в область действия кванторов по одноименным переменным);
- если $A \longrightarrow_\beta B$ есть β-редукция, то $(AC) \longrightarrow_\beta (BC)$ и $(CA) \longrightarrow_\beta (CB)$ тоже β - редукции;
- если $A \longrightarrow_\beta B$ есть β-редукция, то $\lambda x.\ A \longrightarrow_\beta \lambda x.\ B$ тоже β-редукция.

β-редукция

Замена параметра на значение аргумента называется β **- редукцией**:

- $(\lambda x.\ A)B \longrightarrow_\beta A[B/x]$ (все свободные вхождения переменной x в A заменяются на терм B (при этом предполагается, что при подстановке в A свободные переменные терма B не попадают в область действия кванторов по одноименным переменным);
- если $A \longrightarrow_\beta B$ есть β-редукция, то $(AC) \longrightarrow_\beta (BC)$ и $(CA) \longrightarrow_\beta (CB)$ тоже β - редукции;
- если $A \longrightarrow_\beta B$ есть β-редукция, то $\lambda x.\ A \longrightarrow_\beta \lambda x.\ B$ тоже β-редукция.

Пример β-редукции

$(\lambda x\, y.\, f(yx))\, 5\, (\lambda x.\, x) \longrightarrow_\beta (\lambda y.\, f(y\, 5))\, (\lambda x.\, x) \longrightarrow_\beta f((\lambda x.\, x)\, 5) \longrightarrow_\beta f 5$

β-редукция

- β-эквивалентность: $P =_\beta S \iff \exists N.\ P \longrightarrow_\beta^* N \land S \longrightarrow_\beta^* N$
- терм редуцируем, если для него существует β-редукция
- редуцируемое выражение — *редекс*

Свободные и связанные переменные

Связанными переменными называются все переменные, по которым выше в дереве разбора были абстракции. Все остальные переменные называются **свободными**.

Например, в $\lambda x.\ y\ x$, x – связана, а y – свободна.
В терме $\lambda y.\ x\ (\lambda x.\ x)$ в своём первом вхождении переменная x свободна, а во втором — связана.

Связанные переменные — это аргументы функции, т. е. для функции они являются локальными. Терм без свободных переменных называют замкнутым (или комбинатором).

Свободные и связанные переменные

Рассмотрим замену $(\lambda x.\ x\ y)\ [x/y]$.
В этом случае, возникает проблема так как связанная переменная x является свободной при замене: $(\lambda x.\ x\ x)$.
Такая проблема бы не возникла при переименовании переменных $(\lambda z.\ z\ y)\ [x/y] = (\lambda z.\ z\ x)$.
Поэтому, в подобных случаях необходимо переименовывать связанные переменные.

Нотация де-Брауна

Вместо имени переменной хранится натуральное число — количество абстракций в дереве разбора, на которое нужно подняться, чтобы найти ту лямбду, с которой данная переменная связана.

Standart	de Bruijn
$\lambda x.\, x$	$\lambda.0$
$\lambda z.\, z$	$\lambda.0$
$\lambda x.\, \lambda y.\, x$	$\lambda.\, \lambda.1$
$\lambda x.\, \lambda y.\, \lambda s.\, \lambda z.\, x\ s\ (y\ s\ z)$	$\lambda.\, \lambda.\, \lambda.\, \lambda.3\ 1(2\ 1\ 0)$
$(\lambda x.\, x\ x)(\lambda x.\, x\ x)$	$(\lambda.0\ 0)(\lambda.0\ 0)$
$(\lambda x.\, \lambda x.\, x)(\lambda y.\, y)$	$(\lambda.\, \lambda.0)(\lambda.0)$

- Переменная будет свободной, если ей соответствует число, которое больше количества абстракций на пути до неё в дереве разбора.
- При β-редукции ко всем свободным переменным заменяющего дерева при каждой замене прибавляем число, равное разницы уровней раньше и сейчас. Тогда эта переменная продолжит «держаться» за ту же лямбду, что и раньше.

α-конверсия

На самом деле, нам не важны имена связанных переменных, $\lambda x.\ x = \lambda y.\ y$.
Скажем, что два терма α-эквиваленты, если они совпадают с точностью до переименования связанных переменных.

α-конверсия:

- $\lambda x.\ A \longrightarrow_\alpha \lambda y.\ A[y/x]$ (все свободные вождения переменной x в A заменяются на y; при этом переменная y не должна входить в исходный терм A);
- если $A \longrightarrow_\alpha B$ есть α-конверсия, то $(AC) \longrightarrow_\alpha (BC)$ и $(CA) \longrightarrow_\alpha (CB)$ тоже α-конверсии;
- если $A \longrightarrow_\alpha B$ есть α-конверсия, то $\lambda x.\ A \longrightarrow_\alpha \lambda x.\ B$ тоже α-конверсия.

α-конверсия

На самом деле, нам не важны имена связанных переменных,
$\lambda x.\ x = \lambda y.\ y$.
Скажем, что два терма α-эквиваленты, если они совпадают с точностью до переименования связанных переменных.

α-конверсия:

- $\lambda x.\ A \longrightarrow_\alpha \lambda y.\ A[y/x]$ (все свободные вождения переменной x в A заменяются на y; при этом переменная y не должна входить в исходный терм A);
- если $A \longrightarrow_\alpha B$ есть α-конверсия, то $(AC) \longrightarrow_\alpha (BC)$ и $(CA) \longrightarrow_\alpha (CB)$ тоже α-конверсии;
- если $A \longrightarrow_\alpha B$ есть α-конверсия, то $\lambda x.\ A \longrightarrow_\alpha \lambda x.\ B$ тоже α-конверсия.

  Тогда $P =_\alpha S \iff P \longrightarrow_\alpha^* S$

η-конверсия

Рассмотрим термы: $y = (\lambda x.\ y\ x)$.

η-конверсия:

- $(\lambda x.\ y\ x) \longrightarrow_\eta y$ если x не является свободной в y;
- если $A \longrightarrow_\eta B$ есть η-конверсия, то $(AC) \longrightarrow_\eta (BC)$ и $(CA) \longrightarrow_\eta (CB)$ тоже η - конверсия;
- если $A \longrightarrow_\eta B$ есть η - конверсия, то $\lambda x.\ A \longrightarrow_\eta \lambda x.\ B$ тоже η - конверсия.

$$P =_\eta S \iff \exists N.\ P \longrightarrow_\eta^* N \wedge S \longrightarrow_\eta^* N$$

η-конверсия терминируема и приводит к единому результату

Равенство λ-термов

λ-термы A и B называются равными, если существует цепочка λ-термов
$A \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots \longrightarrow C_n \longrightarrow B$,
где каждый следующий терм получается из предыдущего с помощью $\alpha\beta\eta$-редукции.
(Кроме того, каждый терм считается равным самому себе.)

Равенство в Isabelle $=$ равенство в смысле $\alpha\beta\eta$

Теорема Чёрча-Россера

- Говорят, что λ-терм A имеет нормальную форму, если к нему нельзя применить β-редукцию (даже предварительно применив несколько раз α-конверсию).
- Не все λ-термы имеют нормальную форму. Пример: $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. Следовательно, *$\lambda$-исчисление не терминируемо!*

Теорема Чёрча-Россера

- Говорят, что λ-терм A имеет нормальную форму, если к нему нельзя применить β-редукцию (даже предварительно применив несколько раз α-конверсию).
- Не все λ-термы имеют нормальную форму. Пример: $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. Следовательно, *$\lambda$-исчисление не терминируемо!*

Теорема Чёрча-Россера

Для любых λ-термов P, Q и R таких, что $P \longrightarrow_{\alpha\beta\eta} Q$ и $P \longrightarrow_{\alpha\beta\eta} R$, существует λ-терм S такой, что $Q \longrightarrow_{\alpha\beta\eta} S$ и $R \longrightarrow_{\alpha\beta\eta} S$.

Следствие: единственность нормальной формы.

Каррирование

Каррирование (carrying) — преобразование функции от многих переменных в функцию, берущую свои аргументы по одному.
$\Lambda : (A \star B \to C) \to (A \to (B \to C))$

Применение

С помощью λ-исчисления можно выразить логические операции:

- $\mathit{true} \equiv \lambda x\ y.\ x$
- $\mathit{false} \equiv \lambda x\ y.\ y$
- $\mathit{if} \equiv \lambda z\ x\ y.\ z\ x\ y$
- $\mathit{conj} \equiv \lambda x\ y.\ \mathit{if}\ x\ y\ \mathit{false}$
- $\mathit{disj} \equiv \lambda x\ y.\ \mathit{if}\ x\ \mathit{true}\ y$
- $\mathit{neg} \equiv \lambda x.\ \mathit{if}\ x\ \mathit{false}\ \mathit{true}$

Нумералы Чёрча

Определим на основе лямбда-исчисления натуральные числа:

- $0 \equiv \lambda s\ z.\ z$
- $1 \equiv \lambda s\ z.\ s\ z$
- $2 \equiv \lambda s\ z.\ s\ (s\ z)$
- Число n будет n раз применять функцию s к начальному значению z и возвращать результат.

$$succ = \lambda n\ s\ z.\ s\ (n\ s\ z)$$
$$plus = \lambda n\ m\ s\ z.\ n\ s\ (m\ s\ z)$$
$$mult = \lambda n\ m\ s\ z.\ n\ (m\ s)\ z$$

Неподвижные точки

Неподвижной точкой λ-терма f назовём такой терм X, что $F\ X \longrightarrow_\beta^\star X$.

Теорема о неподвижной точке

Для любого λ-терма существует неподвижная точка.

Неподвижные точки

Неподвижной точкой λ-терма f назовём такой терм X, что $F\ X \longrightarrow_{\beta}^{\star} X$.

Теорема о неподвижной точке

Для любого λ-терма существует неподвижная точка.

Доказательство

Пусть F - исходный λ-терм. Тогда введём $W := \lambda x.\ F\ (x\ x)$ и $X := W\ W$.
Получим $X \equiv W\ W\ \equiv (\lambda x.\ F\ (x\ x))\ W =_{\beta} F\ (W\ W) \equiv F\ X$

Неподвижные точки

Неподвижной точкой λ-терма f назовём такой терм X, что $F\ X \longrightarrow_\beta^\star X$.

Теорема о неподвижной точке

Для любого λ-терма существует неподвижная точка.

Доказательство

Пусть F - исходный λ-терм. Тогда введём $W := \lambda x.\ F\ (x\ x)$ и $X := W\ W$.
Получим $X \equiv W\ W\ \equiv (\lambda x.\ F\ (x\ x))\ W =_\beta F\ (W\ W) \equiv F\ X$

Теорема о комбинаторе неподвижной точки

Существует комбинатор Y, такой что для любого λ-терма F
$F\ (Y\ F) \longrightarrow_\beta Y\ F$.

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Рекурсия

Y-комбинатор позволяет ввести рекурсию в λ-исчисление.

Пример

$fac = \lambda x.\ if\ (isZero\ x)\ 1\ (fact\ (pred\ x))$

Перепишем (преобразование, обратное β-редукции):

$fac = (\lambda f\ x.\ if\ (isZero\ x)\ 1\ (f\ (pred\ x)))fac = fac' fac$

Получаем рекурсивную функцию, с помощью которой можем посчитать факториал числа:

$fac = Y\ fac'$

Применение

Также можем ввести пары, списки и операции на множествах. Следовательно, с помощью λ-исчисления можно реализовать эмулятор машины Тьюринга: с помощью пар, списков чисел можно хранить состояния. С помощью рекурсии можно обрабатывать переходы.

Противоречивость

Парадокс Рассела в λ-исчислении:

$R = \{x \mid x \notin x\}$
В терминах λ-исчисления:
$R \equiv \lambda x.\ \neg\ (x\ x)$
$(R\ R) = (\lambda x.\ \neg(x\ x))\ R \longrightarrow_{\beta} \neg(R\ R)$

Типизированное λ-исчисление

Модифицируем λ-исчисление, введя понятие типа:

- Каждый λ-терм должен иметь тип ($t :: \alpha$)
- Для λ-терма ($s\ t$):
  - $s :: \alpha \Rightarrow \beta$
  - $t :: \alpha$
  - $s\ t :: \beta$

$\lambda^{\longrightarrow}$

Синтаксис

- Мн-во λ-термов остаётся тем же
- Мн-во типов: $\tau ::= b \mid v \mid \tau \Rightarrow \tau$,
  где b - тип, v - переменная типа ($\alpha, \beta, \ldots$)
- Контекст Γ – функция, сопоставляющая имена переменным и константам

$\Gamma \vdash t :: \alpha$

t – корректно типизированный (well-typed), если $\exists \Gamma\ \alpha.\ \Gamma \vdash t :: \alpha$

Типизация по Чёрчу и по Карри

- Типизация по Чёрчу: каждому терму сопоставляется единственный тип.
- Типизация по Карри: терм может иметь или не иметь тип. Если тип имеется, он может быть не единственным (полиморфизм).

Подход Карри используется в Haskell и StandartML (т. е. и в Isabelle).

Типизация по Карри

- Система типов по Карри обеспечивает полиморфизм и перегрузку.
- При перегрузке терм может иметь различные типы, не связанные структурным сходством.
- Тип σ – более общий чем τ, если существует такая замена S, что $\tau = S(\sigma)$.
- Каждый корректно типизированный терм имеет наиоблее общий тип.
- Проверка и вывод типов (type checking и type inference в Isabelle) разрешимы.

β-редукция

- Определение β-редукции для $\lambda^{\longrightarrow}$ такое же, как и для обычно λ-исчисления.
- Корректно типизированный терм остаётся корректно типизированным при β-редукции.
- В $\lambda^{\longrightarrow}$ β-редукция терминируема!

Сильная нормализация

- $=_{\alpha\beta\eta}$ разрешимо
  Поэтому Isabelle может автоматически редуцировать каждый терм к $\beta\eta$-нормальной форме.

Теорема

Каждый типизируемый терм имеет нормальную форму, и каждая возможная последовательность редукций, начинающаяся с типизируемого терма, завершается.

Полнота по Тьюрингу

- λ-исчисление полно по Тьюрингу
- $\lambda^{\longrightarrow}$ не полно по Тьюрингу, так как мы не можем создавать незавершающиеся программы
- но функциональные языки, основанные на $\lambda^{\longrightarrow}$, Тьюринг-полны. Как это возможно?

Полнота по Тьюрингу

- λ-исчисление полно по Тьюрингу
- $\lambda^{\longrightarrow}$ не полно по Тьюрингу, так как мы не можем создавать незавершающиеся программы
- но функциональные языки, основанные на $\lambda^{\longrightarrow}$, Тьюринг-полны. Как это возможно?

Чтобы обеспечить полноту по Тьюрингу, добавим полиморфный оператор фиксированной точки Y.

$Y :: (\sigma \Rightarrow \tau) \Rightarrow (\sigma \Rightarrow \tau) \Rightarrow \sigma \Rightarrow \tau$

$Y\ F \longrightarrow_{\beta} F\ (Y\ F)$

Термы и типы в Isabelle

- **Типы:** $\tau ::= b \mid v \mid v :: C \mid \tau \Rightarrow \tau \mid (\tau, ..., \tau)\ K$
  b – базовые типы (bool, int, ...)
  v – переменные типа (α, β, ...)
  K – конструкторы типа (list, set, Suc, ...)
  C – type classes
- **Термы:** $\tau ::= c \mid v \mid ?v \mid (t\ t) \mid (\lambda x.\ t)$
  $v, x \in V$, где V – множество имён
  $?v$ – схематические переменные, т.е. переменные, которые могут быть инстанцированы

Type Classes

Type Classes (аналогично Haskell) ограничивают переменные типа классом, определённым аксиомами.

- Мы можем доказывать теоремы для класса;
- Мы можем делать подклассы (подтипы);
- Мы можем инстанцировать класс переменной типа. При этом необходимо доказать корректность инстанцирования.

Схмеатические переменные

Унификация – нахождение замены переменных σ для термов s, t, такой что: $\sigma(s) = \sigma(t)$

Унификация в Isabelle – нахождение замены схематических переменных σ для термов s, t, такой что: $\sigma(s) =_{\alpha\beta\eta} \sigma(t)$
Так как речь идёт об унификации высшего порядка, то схематические переменные могут быть функциями.

Примеры унификации:

- $?P \wedge ?Q =_{\alpha\beta\eta} x \wedge y \; [?P \leftarrow x, ?Q \leftarrow y]$
- $P\ (?f\ x) =_{\alpha\beta\eta} ?Q\ x \; [?f \leftarrow \lambda y.\ y, ?Q \leftarrow P]$

Доказательства в Isabelle

Схема apply-скрипта:

lemma name: "goal"
apply <tactic (method)>
apply <tactic (method)>
...
done

Proof State

Proof State (состояниye доказательства) в Isabelle:

$\bigwedge x_1 \ldots x_n . [[A_1; \ldots; A_n]] \Longrightarrow G$

$x_1 \ldots x_n$ – параметры

$A_1; \ldots; A_n$ – предположения

G – цель доказательства

Простейший метод

apply assumption

С помощью метода assumption мы можем доказать цель вида $[\![A_1; \ldots; A_n]\!] \Longrightarrow G$, если G унифицирутеся с одним из предположений A_i

Правила интродукции

apply (rule <intro-rule>)
apply (intro <intro-rule>)

Правила интродукции имеют вид $[[A_1; \ldots; A_n]] \Longrightarrow A$ и означают, что для вывода A нам нужно показать $A_1; \ldots; A_n$.
Применение rule $[[A_1; \ldots; A_n]] \Longrightarrow A$ к цели G влечёт за собой:

- унификацию A и G
- замену G на n новых подцелей $A_1; \ldots; A_n$

Правила интродукции

Рассмторим пример применения правила интродукции $\frac{X \quad Y}{X \wedge Y}$ (**conjI**) к цели $B \Longrightarrow A \Longrightarrow A \wedge B$
(В Isabelle: **conjI** :$?P \Longrightarrow ?Q \Longrightarrow ?P \wedge ?Q$)

- унификация $?P \wedge ?Q$ с $A \wedge B$
- замена цели $A \wedge B$ на 2 цели: A и B
- Proof State до применения: $B \Longrightarrow A \Longrightarrow A \wedge B$
- Proof State после применения:
  1. $B \Longrightarrow A \Longrightarrow A$
  2. $B \Longrightarrow A \Longrightarrow B$

Правила элиминации

apply (erule <elim-rule>)

Правила элиминации имеют вид $[[A_1; \ldots; A_n]] \Longrightarrow A$ и означают, что если мы имеем A_1 и хотим показать A, то нам нужно показать $A_2; \ldots; A_n$. Применение erule $[[A_1; \ldots; A_n]] \Longrightarrow A$ к цели G - это применение rule, а также:

- унификация одной из посылок (premises) правила A_i с предположением (assumption)
- элиминация (удаление) этого предположения из Proof State

Правила элиминации

Рассмотрим пример применения правила элиминации $\frac{P \wedge Q \quad [[P;Q]] \Longrightarrow R}{R}$ (**conjE**) к цели $[[A \wedge B]] \Longrightarrow A$

(В Isabelle: **conjE** : $[[?P \wedge ?Q; [[?P; ?Q]] \Longrightarrow ?R]] \Longrightarrow ?R$)

- унификация $?R$ с A
- унификация премиса $?P \wedge Q$ с предположением $A \wedge B$ и его удаление
- замена цели $[[A \wedge B]] \Longrightarrow A$ на $[[A; B]] \Longrightarrow A$

Safe and unsafe rules

- **Небезопасные правила (unsafe rules)** заменяют текущую цель доказательства на одну или несколько новых целей, не эквивалентных исходной. И таким образом, могут сделать доказуемую цель недоказуемой. Примеры: **disjI1, impE, notE**.
- **Безопасные правила (safe rules)** текущую цель доказательства на одну или несколько новых целей, совместно эквивалентных исходной. Примеры: **conjI, disjE, classical**.
- Необходимо сначала применять безопасные правила, а затем небезопасные!

Правила:

- **allI** (safe)
- **allE** (unsafe)
- **exI** (unsafe)
- **exE** (safe)

allI и exE вводят новый параметр (Λx)
allE и exI вводят новую переменную ($?x$).
Сначала нужно вводить параметры, а затем переменные!

Инстанцирование правил

apply (rule_tac x = "*term*" in *rule*)
apply (erule_tac x = "*term*" in *erule*)

Тактики работают анлогично rule и erule, при этом происходит инстанцирование переменной ?x термом *term* перед применением.

Имена параметров

- Имена параметров выбираются Isabelle автоматически.
- Мы можем вручную переименовывать параметры с помощью тактики **rename_tac**.

rename_tac $x_1 \ldots x_n$ переименовывает n параметров справа-налево в $\mathrm{x}_1 \ldots x_n$.

Атрибуты of и where

- С помощью атрибута of мы можем инстанцировать термы в теореме слева направо.
- С помощью атрибута **where** мы можем инстанцировать терм в теореме, явно указав его.

Forward Proof

apply (frule <*rule*>)

Правило rule имеет вид $[[A_1; ...; A_m]] \Longrightarrow A$. Если цель имеет вид $[[B_1; ...; B_n]] \Longrightarrow C$, то происходит замена $\sigma(B_i) \equiv \sigma(A_1)$.
В итоге, получаем m подцелей:
1. $\sigma[[B_1; ...; B_n]] \Longrightarrow A_2$.
.
.
m-1. $\sigma[[B_1; ...; B_n]] \Longrightarrow A_m$
m. $\sigma[[B_1; ...; B_n]] \Longrightarrow C$

drule работает как frule, при этом удаляя B_i.

Атрибуты OF и THEN

$$r\ [\mathbf{OF}\ r_1..r_n]$$

С помощью данного атрибута мы доказываем 1-ое предположение теоремы r с помощью теоремы r_1, 2-ое предположние - с помощью етеоремы r_2 и т.д.

r: $[[A_1; ...; A_m]] \Longrightarrow A$

r_1 :$[[\mathrm{B}_1; ...; B_n]] \Longrightarrow B$

r [OF r_1] :$[[\mathrm{B}_1; ...; B_n; A_2; ...; A_m]] \Longrightarrow A$

$r_1\ [THEN\ r_2]$ означает $r_2\ [OF\ r_1]$

Методы автоматического доказательства

- **apply** (intro <intro-rules>) – многократно применяет указанные правила интродукции;
- **apply** (elim <elim-rules>) – многократно применяет казанные правила элиминации;
- **apply** (clarify) – применяет все безопасные правила, не разбивающие цель;
- **apply** (safe) – применяет все безопасные правила;
- **apply** (blast) – автоматический прувер (разрешает все цели первого порядка);
- **apply** (fast) – автоматический прувер.

Автоматизация

Использование автоматических методов может быть настроено пользователем с помощью атрибутов.

- Добавление атрибута: **declare** conjI[intro!], iffE[elim].
- Удаление атрибута: **declare** allE[rule del].
- Локальное использование правила: **apply** (blast intro: <intro-rule>).
- Локальное удаление правила: **apply** (blast del: conjI).

Другие методы автоматизации

- **apply** (fastforce) – fast + simp.
- **apply** (clarsimp) – clarify + simp. (Напомним, что **apply** (clarify) – применяет все безопасные правила, не разбивающие цель)
- **apply** (auto) – автоматический решатель: simp + classical + многое другое.

Перечисленные выше методы переписывают сразу все цели. Мы можем указать лимит (n) целей с помощью специального атрибута: **apply** (auto)[n].

Логика высшего порядка

- Логика высказываний
  - без кванторов;
  - все переменные булевы.
- FOL – логика первого порядка
  - квантификация только переменных;
  - термы и формулы синтаксически отличаются.
- HOL – логика высшего порядка
  - квантификация над всем, в т.ч. над предикатами;
  - формула – это терм типа bool;
  - построена на типизированном λ-исчислении с определенными типами и константами.

ϵ-оператор Гильберта

$\epsilon\ x.\ P\ x$ – значение, которое удовлетворяет P. (если такое сущетсвует)
ϵ – **ϵ-оператор Гильберта**.

В Isabelle он записывается как $SOME\ x.P\ x$

$$\frac{P\ ?x}{P\ (SOME\ x.\ P\ x)}\ \textbf{someI}$$

HOL

Базовые типы:

- bool
- $\_ \Rightarrow \_$ (fun)
- ind

Базовые константы:

- $\longrightarrow :: \ \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$
- $= :: \ \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$
- $\epsilon :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$

Higher Order Abstract Syntax

Higher Order Abstract Syntax (HOAS) – это техника использования HOL как мета-языка для объектных языков с операторами связывания ($\forall$, $\exists$, ϵ, ...).

- Для задания операторов связывания удобно использовать оператор λ.
- Isabelle автоматически транслирует классический синтаксис в HOAS, используя синтаксические декларации.

Синтаксические декларации

- **mixfix**

  $const\ drvbl\ ::\ ctxt\ \Rightarrow\ ctxt\ \Rightarrow\ f\ \Rightarrow\ bool\ (''\_\,,\ \_\ \vdash\ \_'')$

  Используемый синтаксис: $\Gamma,\ \Pi\ \vdash\ F$
- Мы можем задать приоритеты в шаблоне для указания силы биндинга:

  $const\ drvbl\ ::\ ctxt\ \Rightarrow\ ctxt\ \Rightarrow\ f\ \Rightarrow\ bool\ (''\_\,,\ \_\ \vdash\ \_)''$
  $[30, 0, 20]\ 60)$
- **infixl/infixr**: удобная форма для лево/право-ассоциативного бинарного оператора

  $conj\ ::\ ''[bool,\ bool] \Rightarrow\ bool''(infixr\ ''\ \wedge\ ''35)$
- **binders**:

  $c\ ::\ (\tau_1\ \Rightarrow\ \tau_2)\ \Rightarrow\ \tau_3\ (binder\ ''\ B''\ \ p)$

  Тогда $B\ x.\ P\ x$ транслируется в $c\ P$.

$definition\ All\ ::\ (\alpha\ \Rightarrow\ bool)\ \Rightarrow\ bool\ (binder''\forall''\,10)\ where$
"All P $\equiv\ (P\ = (\lambda x.\ True))''$
И терм $\forall x.\ x = 0$ в HOAS выглядит как $ALL\ (\lambda x.\ x = 0)$

Базовые определения в HOL

Аналогично можем задать оставшиеся операторы:

$True \equiv (\lambda x :: bool.\, x) = (\lambda x.\, x)$

$Ex\ P \equiv \forall Q.\, (\forall x.\, P\ x \longrightarrow Q) \longrightarrow Q$

$False \equiv (\forall P.\, P)$

$\neg\, P \equiv P \longrightarrow False$

$P \wedge Q \equiv \forall R.\, (P \longrightarrow Q \longrightarrow R) \longrightarrow R$

$P \vee Q \equiv \forall R.\, (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

$If\ P\ x\ y \equiv SOME\ z.\, (P = True \longrightarrow z = x) \wedge (P = False \longrightarrow z = y)$

$inj\ f \equiv \forall x\ y.\, f\ x = f\ y \longrightarrow x = y$

$surj\ f \equiv \forall y.\, \exists x.\, y = f\ x$

Аксиомы HOL

$$\frac{}{t = t}\ \text{refl} \qquad \frac{s = t \quad P\ s}{P\ t}\ \text{subst} \qquad \frac{\bigwedge x.\ f\ x = g\ x}{(\lambda x.\ f\ x) = (\lambda x.\ g\ x)}\ \text{ext}$$

$$\frac{P \Longrightarrow Q}{P \longrightarrow Q}\ \text{impl} \qquad \frac{P \longrightarrow Q \quad P}{Q}\ \text{mp}$$

$$\frac{}{(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)}\ \text{iff}$$

$$\frac{}{P = \mathsf{True} \lor P = \mathsf{False}}\ \text{True\_or\_False}$$

$$\frac{P\ ?x}{P\ (\mathsf{SOME}\ x.\ P\ x)}\ \text{someI}$$

$$\frac{}{\exists f :: \mathit{ind} \Rightarrow \mathit{ind}.\ \mathsf{inj}\ f \land \neg \mathsf{surj}\ f}\ \text{infty}$$

HOL

Итого, по определению в HOL есть:

- 3 базовых константы
- 3 базовых типа
- 9 аксиом

С их помощью мы можем выводить другие правила и вводить новые определения.

См. файл $ISABELLE_HOME/src/HOL/HOL.thy

assumes-shows

При формулировании утверждения теоремmы можно использовать именованные предположения с помощью шаблона **assumes-shows**

lemma [name:]
assumes $[n_1:]$ "P_1"
assumes $[n_2:]$ "P_2"
...
shows "G"
<proof>

Proof state: $[[P_1; P_2; ...]] \Longrightarrow G$
Такой шаблон удобно использовать для вывода правил.

Симплификация

Пусть нам дано множество уравнений:

$$\begin{aligned} l_1 &= r_1 \\ l_2 &= r_2 \\ &\ldots \\ l_n &= r_n \end{aligned}$$

Можно ли доказать равенство $l = r$, используя их, и как?

Симплификация

Рассмотрим уравнения как правила редукции:

$$
\begin{gathered}
l_1 \longrightarrow r_1 \\
l_2 \longrightarrow r_2 \\
\dots \\
l_n \longrightarrow r_n
\end{gathered}
$$

Докажем $l = r$, показав, что $l \longleftrightarrow^* r$.
То есть, что l и r лежат в рефлексивно-транзитивном симметрическом замыкании отношения шага редукции.

Симплификация

Вспомним теорему Чёрча-Россера, чтобы понять основную идею:

Теорема Чёрча-Россера

Для любых λ-термов P, Q и R таких, что $P \longrightarrow_{\alpha\beta\eta} Q$ и $P \longrightarrow_{\alpha\beta\eta} R$, существует λ-терм S такой, что $Q \longrightarrow_{\alpha\beta\eta} S$ и $R \longrightarrow_{\alpha\beta\eta} S$.

То есть, необходимо найти такое S, чтобы $l \longrightarrow^* S$ и $r \longrightarrow^* S$.
Такое S найдется не всегда. Это работает в формальных системах, в которых выполняется аналог теоремы Черча-Россера (например, λ -исчисление).

Симплификация

В общем случае, процесс переписывания терма с помощью набора правил редукций может быть незавершаемым.

$\longrightarrow$ завершается, когда мы можем задать порядок редукции $<$ на термах ($s < t$ при $s \longrightarrow t$)

Пример завершаемой системы: $f\ (g\ x) \longrightarrow g\ x, g\ (f\ x) \longrightarrow f\ x$
Порядок редукции для системы: $s < t$, тогда и только тогда, когда количество функциональнх символов в s меньше чем в t.

Симплификатор

Механизм переписывания термов в Isabelle называется симплификатором. Его можно использовать с помощью **apply** simp.

- симлификатор использует правила симпилифкации;
- переписывание термов происходит слева-направо;
- переписывание происходит пока есть применимые правила симплификации;
- процесс может зацикливаться (нет завершаемости);
- результат может различаться в зависимости от того, в каком порядке мы используем правила.

Атрибуты симплификатора

Механизм переписывания термов в Isabelle называется симплификатором. Его можно использовать с помощью **apply** simp.

- уравнения можно добавлять в глобальное множество симпилификаци с помощью атрибута [simp];
- уравнения можно добавлять/удалять во/из множество симпилификаци локально:
  apply (simp add: <rules>)
  apply (simp del: <rules>);
- можно задать необходимое множество симплификации:
  apply (simp only: <rules>)

Симплификатор

- Правило переписывания $l \longrightarrow r$ применимо к терму $t[s]$ если существует такая замена σ, что $\sigma\ l = s$
- В результате переписывания получим терм $t[\sigma\ r]$

Пример:

Правило: $0 + n \longrightarrow n$
Терм: $a + (0 + (b + c))$
Замена: $\sigma = \{n \mapsto b + c\}$
Результат: $a + b + c$

Симплификатор

Правила переписывания могут быть с условиями:
$[\![P_1; \ldots; P_n]\!] \Longrightarrow l = r$ применимо к терму $t[s]$, если

- $\sigma\ l = s$
- $\sigma\ P_1, \ldots, \sigma\ P_n$ доказуемы переписыванием.

Симплификатор

Симплификатор Isabelle использует предположения, что также влияет на завершаемость. Атрибуты для использования и упрощения предположений при симплификации:

- simp (no_asm) – полностью игнорирует предположения;
- simp (no_asm_use) – упрощает, но не использует предположения;
- simp (no_asm_simp) – использует, но не упрощает предположения;

Разбиение на случаи

- Isabelle может автоматически упрощать такие операторы как let и if-then-else (в цели);
  $P\ (if\ \ A\ then\ x\ else\ y) = (A \longrightarrow P\ x) \wedge (\neg A \longrightarrow P\ y)$
- С помощью атрибута симплификатора split можно вручную задавать правила разбиения на случаи. Для произвольного типа t оно хранится под именем t.split.

Правила конгруэнции

Примеры:

- **conj_cong** $[[P = P'; P' \Longrightarrow Q \Longrightarrow Q']] \Longrightarrow (P \wedge Q) = (P' \wedge Q')$
- **if_cong** $[[b = c; c \Longrightarrow x = u; \neg c \Longrightarrow y = v]] \Longrightarrow$
  $(\mathit{if}\ \ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ \ c\ \mathit{then}\ u\ \mathit{else}\ v)$

Данные правила могут быть полезны, но не используются автоматически, так как замедляет симплификатор.
Мы можем добавлять и использовать правила конгруэнции с помощью **apply** (simp cong: <cong_rule>).

Упорядоченное переписывание

Мы понимаем, что переписывание $x + y \longrightarrow y + x$ незавершаемо. Поэтому нужно использовать подобные правила только если они уменьшают лексикографический порядок терма.

- Пример правил для типов nat, int: **add_ac**, **mult_ac**

Объявление типов

- Мы можем задать аббревиатуру типа с помощью команды **type_synonym**
  Пример: **type_synonym** string = "char list"
- Мы можем объявить тип с помощью команды **typedecl**
  Пример: **typedecl** myType

Datatype

datatype $(\alpha_1, ..., \alpha_n)\ \tau = C_1\ \tau_{1,1} ... \tau_{1,n_1} \mid C_2\ \tau_{2,1} ... \tau_{2,n_2} \mid ... \mid C_k\ \tau_{k,1} ... \tau_{k,n_k}$

Свойства:

- Конструкторы: $C_i\ \tau_{i,1} ... \tau_{i,n_i} :: \tau_{i,1} \Rightarrow ... \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, ..., \alpha_n)\ \tau$
- Отличимоть конструкторов: $C_i \neq C_j$ если $i \neq j$
- Инъективность:
  $(C_i\ x_1 ... x_{n_i} = C_i\ y_1 ... y_{n_i}) \Longleftrightarrow (x_1 = y_1 \wedge ... \wedge x_{n_i} = y_{n_i})$

Datatype

Примеры использования:

- Перечисление:

  datatype Status = NotStarted | Active | Finished

- Полиморфные типы:

  datatype ’a option = None | Some ’a

- Рекурсивные типы:

  datatype ’a list = Nil | Cons ’a “a list"
  datatype ’a tree = Tip | Node ’a “a tree’a tree"

- Взаинмая рекурсия:

  datatype even = EvenZero | EvenSuc odd
  and odd = OddZero | OddSuc even

- Вложенная рекурсия:

  datatype ’a tree = Tip | Node ’a “a tree list"

В результате объявления **datatype** вводится конструкция **case**:

$$(case\ xs\ of\ [] \Rightarrow \ldots |\ y\#ys \Rightarrow \ldots y \ldots ys)$$

Одному конструктору соответствует 1 случай.

apply (case_tac t)

Создаёт k подцелей для каждого конструктора типа t.

Структурная индукция

Для того чтобы доказать, что $P(xs)$ выполнено для любого списка xs, необходимо доказать

- $P([])$;
- для произвольных фиксированных x, xs: $P(xs) \implies P(x\#xs)$

Метод **induction** x производит индукцию по переменной x, если x задана с помощью **datatype**

fun

С помощью ключевого слова **fun** можно определять рекурсивные функции. Её ключевые особенности:

- pattern matching по конструкторам рекурсивного типа;
- порядок равенств важен;
- завершаемость должна доказываться автоматически;
- автоматически доказывается индуктивное правило *.induct*.

primrec (примитивная рекурсия) – упрощёная версия **fun**.

- большинство функций примитивно рекурсивные;
- часто используется.

Basic induction heuristics

- **Теоремы о рекурсивных функциях доказываются по индукции!**
- Если функция f рекурсивна по аргументу номер i, то индукция должна проводиться тоже по аргументу i.

Пример

Функция rev разворота списка:

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

Версия с использованием хвостовой рекурсии:

fun itrev :: 'a list ⇒ 'a list ⇒ 'a list **where**
itrev [] ys = ys |
rev (x#xs) ys = itrev xs (x#ys)

lemma itrev xs [] = rev xs

Обобщение теорем

lemma itrev xs [] = rev xs

- Замена констант переменными:
  lemma itrev xs ys = rev xs@ys
- Обобщаение свободных переменных
  - Квантификация;
    lemma ∀ ys. itrev xs ys = rev xs@ys
  - Использование метода *arbitrary*:
    (induction xs arbitrary: ys)

inductive

Определим индуктивно чётные числа:

- 0 – чётное число
- Если n чётно, то и n + 2 чётно

Мы можем формализовать это определение в Isabelle/HOL с помощью ключевого слова **inductive**:

inductive ev :: nat ⇒ bool **where**

ev 0 |

ev n $\Longrightarrow$ *ev (n + 2)*

Другое определение: **fun** evn :: nat $\Rightarrow$ bool **where**
$evn\ 0 = True$ |
$evn\ (Suc\ 0) = False$ |
$evn\ (Suc\ (Suc\ n)) = evn\ n$

Докажем $ev\ m \Longrightarrow evn\ m$.
2 случая для $ev\ m$:

- rule $ev\ 0 \Longrightarrow m = 0 \Longrightarrow evn\ m = 0 = True$
- rule $ev\ n \Longrightarrow ev\ (n + 2)$
  $\Longrightarrow m = n + 2$ и $ev\ n$ (IH)
  $\Longrightarrow evn\ m = evn\ (n + 2) = evn\ n = True$

inductive

Общий формат **inductive**:

inductive I :: $\tau \Rightarrow$ bool **where**

$[[I\ a_1, \ldots, I\ a_n]] \Longrightarrow I\ a$ |

...|

...

Замечания:

- I может иметь несколько аргументов;
- Каждое правило может включать в себя условие, не связанное с I

inductive

Чтобы доказать утверждение вида

$$I\ x \Longrightarrow Px$$

с помозью rule induction по I x необходимо:
для каждого правила

$$[[I\ a_1, ..., I\ a_n]] \Longrightarrow I\ a$$

показать

$$[[I\ a_1, P\ a_1 ..., I\ a_n, P\ a_n]] \Longrightarrow P\ a$$

Inductive_set

inductive_set I :: τ set $\Rightarrow$ bool **where**
$[\![a_1 \in I, \ldots, a_n \in I]\!] \Longrightarrow a \in I$ |
...|
...

Отличия от **inductive**:

- аргументы I не каррированы;
- I может быть использовано с теоретико-множественными операторами

Isar(Intelligible semi-automated reasoning) — язык структурированных доказательств.

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!

Apply scripts versus Isar proofs

Apply script = assembly language program

Isar proof = structured program with assertions

But: **apply** still useful for proof exploration

A typical Isar proof

proof
  assume $formula_0$
  have $formula_1$ **by** $simp$
  $\vdots$
  have $formula_n$ **by** $blast$
  show $formula_{n+1}$ **by** $\dots$
qed

proves $formula_0 \Longrightarrow formula_{n+1}$

Isar core syntax

proof = **proof** [method] step* **qed**
| **by** method

method = $(simp \dots) \mid (blast \dots) \mid (induction \dots) \mid \dots$

step = **fix** variables $(\bigwedge)$
| **assume** prop $(\Longrightarrow)$
| [**from** fact$^+$] (**have** | **show**) prop proof

prop = [name:] "formula"

fact = name | . . .

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

Example: Cantor's theorem

lemma $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof default proof: assume *surj*, show *False*
  assume a: $surj\ f$
  from a **have** b: $\forall\ A.\ \exists\ a.\ A = f\ a$
    by$(simp\ add{:}\ surj\_def)$
  from b **have** c: $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$
    by $blast$
  from c **show** $False$
    by $blast$
qed

Isar_Demo.thy

Cantor and abbreviations

Abbreviations

$this$	=	the previous proposition proved or assumed
then	=	**from** $this$
thus	=	**then show**
hence	=	**then have**

using and **with**

(**have**|**show**) prop **using** facts

=

from facts (**have**|**show**) prop

with facts

=

from facts $this$

Structured lemma statement

lemma
  fixes $f :: 'a \Rightarrow 'a\ set$
  assumes s: $surj\ f$
  shows $False$
proof $-$ no automatic proof step
  have $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$ **using** s
    by$(auto\ simp\!: surj\_def)$
  thus $False$ **by** $blast$
qed

Proves $surj\ f \Longrightarrow False$
but $surj\ f$ *becomes local fact* s *in proof.*

The essence of structured proofs

Assumptions and intermediate facts
can be named and referred to explicitly and selectively

Structured lemma statements

fixes $x :: \tau_1$ **and** $y :: \tau_2 \ldots$
assumes *a:* P **and** *b:* $Q \ldots$
shows R

- **fixes** and **assumes** sections optional
- **shows** optional if no **fixes** and **assumes**

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

Case distinction

show R
proof *cases*
  assume P
  $\vdots$
  show R $\langle proof \rangle$
next
  assume $\neg\ P$
  $\vdots$
  show R $\langle proof \rangle$
qed

have $P \vee\ Q$ $\langle proof \rangle$
then show R
proof
  assume P
  $\vdots$
  show R $\langle proof \rangle$
next
  assume Q
  $\vdots$
  show R $\langle proof \rangle$
qed

Contradiction

show $\neg\ P$
proof
  assume P
  $\vdots$
  show $\mathit{False}\ \langle \mathit{proof} \rangle$
qed

show P
proof $(\mathit{rule}\ \mathit{ccontr})$
  assume $\neg P$
  $\vdots$
  show $\mathit{False}\ \langle \mathit{proof} \rangle$
qed

$\longleftrightarrow$

show $P \longleftrightarrow Q$
proof
  assume P
  $\vdots$
  show Q $\langle proof \rangle$
next
  assume Q
  $\vdots$
  show P $\langle proof \rangle$
qed

$\forall$ and $\exists$ introduction

show $\forall x.\ P(x)$
proof
  fix x local fixed variable
  show $P(x)$ $\langle proof \rangle$
qed

show $\exists x.\ P(x)$
proof
  $\vdots$
  show $P(witness)$ $\langle proof \rangle$
qed

$\exists$ elimination: **obtain**

have $\exists\, x.\ P(x)$
then obtain x **where** *p:* $P(x)$ **by** *blast*
$\vdots$ $\quad x$ fixed local variable

Works for one or more x

obtain example

lemma $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof
  assume $surj\ f$
  hence $\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a$ **by**$(auto\ simp{:}\ surj\_def)$
  then obtain a **where** $\{x.\ x \notin f\ x\} = f\ a$ **by** $blast$
  hence $a \notin f\ a \longleftrightarrow a \in f\ a$ **by** $blast$
  thus $False$ **by** $blast$
qed

Set equality and subset

show $A = B$
proof
  show $A \subseteq B$ $\langle proof \rangle$
next
  show $B \subseteq A$ $\langle proof \rangle$
qed

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  $\vdots$
  show $x \in B$ $\langle proof \rangle$
qed

`Isar_Demo.thy`

Exercise

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

12 Streamlining Proofs
Pattern Matching and Quotations
Top down proof development
moreover
Local lemmas

Example: pattern matching

show $formula_1 \longleftrightarrow formula_2$ (**is** $?L \longleftrightarrow ?R$)
proof
  assume $?L$
  $\vdots$
  show $?R$ $\langle proof \rangle$
next
  assume $?R$
  $\vdots$
  show $?L$ $\langle proof \rangle$
qed

$?thesis$

show $formula$ (is $?thesis$)
proof -
  $\vdots$
  show $?thesis$ $\langle proof \rangle$
qed

Every show implicitly defines $?thesis$

let

Introducing local abbreviations in proofs:

> **let** $?t$ = *"some-big-term"*
> $\vdots$
> **have** "$\ldots$ $?t$ $\ldots$ "

Quoting facts by value

By name:

have *x0:* "$x > 0$" ...

$\vdots$

from *x0* ...

By value:

have "$x > 0$" ...

$\vdots$

from $\langle x > 0 \rangle$ ...

`\<open>` `\<close>`

Isar_Demo.thy

Pattern matching and quotations

12 Streamlining Proofs

Pattern Matching and Quotations
Top down proof development
moreover
Local lemmas

Example

lemma

$$\exists\, ys\ zs.\ xs = ys\ @\ zs \wedge$$
$$(length\ ys = length\ zs \vee length\ ys = length\ zs + 1)$$

proof ???

`Isar_Demo.thy`

Top down proof development

When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

have ... **using** ...	
apply -	to make incoming facts part of proof state
apply *auto*	or whatever
apply ...	

At the end:

- **done**
- Better: convert to structured proof

12 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas

moreover—ultimately

have P_1 ...		**have** lab_1: P_1 ...
moreover		**have** lab_2: P_2 ...
have P_2 ...		$\vdots$
moreover	$\approx$	**have** lab_n: P_n ...
$\vdots$		**from** lab_1 lab_2 ...
moreover		**have** P ...
have P_n ...		
ultimately		
have P ...		

With names

12 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas

Local lemmas

have B **if** *name:* $A_1 \ldots A_m$ **for** $x_1 \ldots x_n$
$\langle proof \rangle$

proves $[\![A_1; \ldots ; A_m]\!] \Longrightarrow B$
where all x_i have been replaced by $?x_i$.

Proof state and Isar text

In general: **proof** *method*

Applies *method* and generates subgoal(s):

$$\bigwedge x_1 \ \ldots \ x_n. \ \llbracket \ A_1; \ \ldots \ ; \ A_m \ \rrbracket \Longrightarrow B$$

How to prove each subgoal:

fix $x_1 \ \ldots \ x_n$
assume $A_1 \ \ldots \ A_m$
$\vdots$
show B

Separated by **next**

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

Isar_Induction_Demo.thy

Proof by cases

Datatype case analysis

datatype $t \;=\; C_1\ \vec{\tau}\ \mid\ \ldots$

proof $($*cases* "*term*"$)$
  case $(C_1\ x_1\ \ldots\ x_k)$
  $\ldots\ x_j\ \ldots$
next
$\vdots$
qed

where **case** $(C_i\ x_1\ \ldots\ x_k)$ $\equiv$

fix $x_1\ \ldots\ x_k$
assume $\underbrace{C_i\!:}_{\text{label}}\ \underbrace{term = (C_i\ x_1\ \ldots\ x_k)}_{\text{formula}}$

Isar_Induction_Demo.thy

Structural induction for nat

Structural induction for nat

show $P(n)$		
proof $(induction\ n)$		
case 0	$\equiv$	**let** $?case = P(0)$
$\vdots$		
show $?case$		
next		
case $(Suc\ n)$	$\equiv$	**fix** n **assume** Suc: $P(n)$
$\vdots$		**let** $?case = P(Suc\ n)$
$\vdots$		
show $?case$		
qed		

Structural induction with $\Longrightarrow$

show $A(n) \Longrightarrow P(n)$		
proof $(induction\ n)$		
case 0	$\equiv$	**assume** 0: $A(0)$
$\vdots$		**let** $?case = P(0)$
show $?case$		
next		
case $(Suc\ n)$	$\equiv$	**fix** n
$\vdots$		**assume** Suc: $A(n) \Longrightarrow P(n)$ $A(Suc\ n)$
$\vdots$		**let** $?case = P(Suc\ n)$
show $?case$		
qed		

Named assumptions

In a proof of

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by structural induction:

In the context of

case C

we have

$C.IH$	the induction hypotheses
$C.prems$	the premises A_i
C	$C.IH + C.prems$

A remark on style

- **case** $(Suc\ n)$ $\ldots$ **show** $?case$
  is easy to write and maintain
- **fix** n **assume** $formula$ $\ldots$ **show** $formula'$
  is easier to read:
  - all information is shown locally
  - no contextual references (e.g. $?case$)

13 Proof by Cases and Induction

Rule Induction

Rule Inversion

Isar_Induction_Demo.thy

Rule induction

Rule induction

inductive $I :: \tau \Rightarrow \sigma \Rightarrow bool$
where
$rule_1$: ...
⋮
$rule_n$: ...

show $I\ x\ y \Longrightarrow P\ x\ y$
proof (*induction rule: I.induct*)
  case $rule_1$
  ...
  show *?case*
next
⋮
next
  case $rule_n$
  ...
  show *?case*
qed

Fixing your own variable names

case $(rule_i\ x_1\ \dots\ x_k)$

Renames the first k variables in $rule_i$ (from left to right) to $x_1\ \dots\ x_k$.

Named assumptions

In a proof of

$$I \ldots \Longrightarrow A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by rule induction on $I \ldots$:
In the context of
case R

we have

$R.IH$	the induction hypotheses
$R.hyps$	the assumptions of rule R
$R.prems$	the premises A_i
R	$R.IH + R.hyps + R.prems$

13 Proof by Cases and Induction

Rule Induction

Rule Inversion

Rule inversion

inductive $ev :: nat \Rightarrow bool$ **where**
$ev0$: $ev\ 0\ |$
$evSS$: $ev\ n \Longrightarrow ev(Suc(Suc\ n))$

What can we deduce from $ev\ n$?
That it was proved by either $ev0$ or $evSS$!

$$ev\ n \Longrightarrow n = 0 \lor (\exists k.\ n = Suc\ (Suc\ k) \land ev\ k)$$

Rule inversion = case distinction over rules

Isar_Induction_Demo.thy

Rule inversion

Rule inversion template

from ‘$ev\ n$‘ **have** P
proof $cases$
  case $ev0$ $\qquad\qquad n = 0$
  $\vdots$
  show $?thesis$ $\ldots$
next
  case $(evSS\ k)$ $\qquad\qquad n = Suc\ (Suc\ k),\ ev\ k$
  $\vdots$
  show $?thesis$ $\ldots$
qed

Impossible cases disappear automatically