Доказательство теорем в системе Isabelle / HOL. Интенсивный курс.

- Данный курс подготовлен на основе курса формальных методов за авторством Тобиаса Нипкова (Tobias Nipkow), профессора Мюнхенского Технологического Университета.
- Note: this material is based on the original course developed by Prof. Tobias Nipkow from TUM University. For more details see http://isabelle.in.tum.de/coursematerial/PSV2009-1/

Примечание*

Переведено и озвучено:

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

Слайды, отмеченные звездочкой*, добавлены переводчиками, так же добавлены некоторые примечания.

На момент перевода и чтения курса в СФУ оригинальный курс отрыт и доступен публично http://isabelle.in.tum.de/coursematerial/PSV2009-1/

Мета-логика в Isabelle

Базовые конструкции

- Логическое следование* $\Longrightarrow$
  - для отделения посылок и заключений в теоремах
- Эквивалентность $\equiv$
  - для определений
- Универсальный квантификатор $\bigwedge$
  - для привязки локальных переменных

*Логическое следование **не** тоже что импликация.
Подробнее читайте на stackexchange и здесь

Логическое следование vs Импликация*

Импликация (→), это функция на утверждениях (высказываниях, которые могут быть либо истинны либо ложны).

Импликация (φ → ψ) истинна iff (¬φ ∨ ψ) истинно.

Отсюда следует классический подход к импликации →. Чтобы получить значение (φ → ψ), достаточно вычислить φ, проверить его значение: если оно ложно, результат в любом случае истина; если же оно истинно, вычислить ψ и вернуть его в качестве результата.

Логическое следование
vs
Импликация*

Логическое следование (⊨) это отношение между множеством логических высказываний и одним логическим высказыванием.

Логическое следование ($\Gamma \vDash \psi$) истинно iff каждая интерпретация, которая делает истинными все $\varphi \in \Gamma$, делает ψ истинным.

Таков классический взгляд на логическое следование. Чтобы вычислить ($\Gamma \vDash \psi$), достаточно рассмотреть произвольную интерпретацию логических символов выраженных на языке выражений в Г которая делает все эти утверждения истинными, и затем проверить, делает ли данная (произвольная) интерпретация истинным ψ. Другими словами, : ψ является логическим следованием из Г если невозможно сделать одновременно все утверждения в Г истинными и ψ ложным.

Источник: http://philosophy.stackexchange.com/questions/12816/difference-between-implica

Нотация

- [[A1; …; An]] ==> B
  - тоже, что и
- A1 ==> … ==> An ==> B
  - ; примерно тоже что и «И»

Структура доказательства

1. $\bigwedge x_1 \ldots x_p . [[A_1; \ldots; A_n]] \Longrightarrow B$

$x_1 \ldots x_p$ // локальные константы

$A_1; \ldots; A_n$ // локальные предположения

B // актуальная (под) цель

Определение функций и типов в Isabelle/HOL

Определение типов в Isabelle/HOL

Определение нового типа

- typedecl // объявление типа
- type_synonym // задание псевдонима
- datatype // определение типа

typedecl

typedecl name

Вводит новый тип без его определения

Пример:

typedecl addr — абстрактный тип адреса

type_synonym

type_synonym *name* = *τ*

Вводит новое обозначение *name* для существующего типа *τ*

Пример:

type_synonym id = nat

Синонимы типов раскрываются немедленно после парсинга и не используются в окне вывода Isabelle

datatype

Пример

datatype 'a list Nil | Cons 'a ('a list)

Свойства:

- Типы:
  - Nil   :: 'a list
  - Cons   :: 'a => 'a list => 'a list
- Различность:
  - Nil != Cons
- Инъективность:
  - (Cons x xs = Cons y ys) = (x = y /\ xs = ys)

Общий случай

$$\begin{array}{lcl} \textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau & = & C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \\ & | & \ldots \\ & | & C_k\ \tau_{k,1} \ldots \tau_{k,n_k} \end{array}$$

Типы: $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$

Различность: $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$

Инъективность:

$$(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$$

Определение функций в Isabelle/HOL

Проблема зацикливания

Рассмотрим функцию

f x = f x + 1

ее вычисление никогда не завершится

Вычтем f x и получим явное противоречие:

==> 0 = 1

! Функции в Isabelle должны быть всюду определенными !

Различия в определениях функций в Isabelle/HOL

- Нерекурсивное определение через definition
  - нет проблем
- Примитивно-рекурсивное с primrec
  - всегда завершается по построению
- Рекурсивная функция с fun
  - автоматическое доказательство завершения
- Рекурсия с function
  - пользовательское доказательство

definition

Нерекурсивное определение функции на примере

definition sq :: nat => nat **where** sq n = n*n

Возможные проблемы

definition prime :: nat => bool where

"prime p = (1 < p ∧ (m dvd p ⟶ m = 1 ∨ m = p))"

не является определением

! каждая свободная переменная в правой части уравнения должна быть в его левой части !

"prime p = (1 < p ∧ (**∀ m**. m dvd p ⟶ m = 1 ∨ m = p))"

Использование определений

Определения <u>не</u> используются автоматически

Для использования определения sq его нужно развернуть:

apply (simp add: sq_def)

`primrec`

Определение примитивно-рекурсивных функций осуществляется через **primrec**

// Прим.: примитивно-рекурсивную функцию всегда можно заменить циклом for

Пример

```
primrec app :: 'a list => 'a list => 'a list where
app Nil            ys = ys |
app (Cons x xs)    ys = Cons x (app xs ys)
```

Общий случай

Если τ это тип данных (с конструкторами $C_1 \dots C_k$), тогда функция $f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ может

быть задана через примитивную рекурсию:

$$
\begin{aligned}
f\ x_1 \dots (C_1\ y_{1,1} \dots y_{1,n_1}) \dots x_p &= r_1 \mid \\
&\vdots \\
f\ x_1 \dots (C_k\ y_{k,1} \dots y_{k,n_k}) \dots x_p &= r_k
\end{aligned}
$$

Рекурсивный вызов в r_i должен быть структурно меньшим, т. е. формы $f\ a_1 \dots y_{i,j} \dots a_p$

nat определен через datatype

datatype nat = 0 | Suc nat

Функции на *nat* определяемы через primrec!

primrec f::nat => …

f 0 = …

f (Suc n) = … f n ...

Больше предопределенных типов и функций

option

datatype 'a option = None | Some 'a

важное применение:

… => 'a option ~ частичная функция

None ~ нет результата

Some a ~ результат a

Пример:

```
primrec lookup :: "'k ⇒ ('k × 'v) list ⇒ 'v option" where
"lookup k [] = None" |
"lookup k (x#xs) = (if fst x = k then Some (snd x) else lookup k xs)"
```

case

Значения типа данных могут быть взяты через ключевое слово case

(case xs of [] => … | y#ys => y … ys …)

Вайлдкарды:

(case xs of [] => [] | y # _ => [y])

Вложенные шаблоны:

(case xs of [0] => 0 | [Suc n] => n | _ = 2)

! Сложные шаблоны предполагают сложные доказательства !

! Требуют () в контексте !

Доказательство перебором вариантов

Если $t :: \tau$ и τ — тип данных

apply (case_tac t)

создает *k* подцелей

$$t = C_i\ x_1 \ldots x_p \Longrightarrow \ldots$$

по одной для каждого конструктора C_i типа τ

Демо: деревья

- http://isabelle.in.tum.de/coursematerial/PSV2009-1/session2/Demo.thy

`fun`

от примитивной рекурсии к сопоставлению с произвольными образцами

Пример: числа Фибоначчи

```
fun fib :: "nat ⇒ nat" where
"fib 0 = 0" |
"fib (Suc 0) = 1" |
"fib (Suc (Suc n)) = fib (n + 1) + fib n"

value "int (fib 7)"      (* приводим nat к int *)
```

Пример: функция Аккермана

```
fun ack :: "nat ⇒ nat ⇒ nat" where
"ack 0 n = Suc n" |
"ack (Suc m) 0 = ack m (Suc 0)" |
"ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

value "int (ack 1 2)"
```

// Прим.: функция Аккермана это пример функции, которую *невозможно* вычислить при помощи цикла for!

// https://www.youtube.com/watch?v=i7sm9dzFtEI

Основные особенности `fun`

- Сопоставление с произвольным образцом
- Порядок уравнений *важен*
- Терминация доказуема через использование лексикографического порядка

Размер

- size (n::nat) = n
- size (xs) = length xs
- size подсчитывает кол-во конструкторов

Лексикографический* порядок

Либо уменьшается первый элемент, либо второй и т. д.:

Для кортежей из трех чисел:

(1,2,3) < (1,2,4) < … < (2,3,4) < (2,3,5) < … < (3,4,5) < ...

Если по мере рекурсивных вызовов функции ее аргументы убывают в лексикографическом порядке, то функция завершит работу.

* он же словарный, см.: https://en.wikipedia.org/wiki/Lexicographical_order

Функция Аккермана завершит работу

fun ack :: "nat ⇒ nat ⇒ nat" where

"ack 0 n = Suc n" |

"ack (Suc m) 0 = ack m (Suc 0)" |

"ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

т. к. аргументы каждого рекурсивного вызова лексикографически меньше чем аргументы левой части уравнения

Вычислительная индукция

Если $f :: \tau \Rightarrow \tau'$ определена через fun, то применяется специальная схема индукции, нужная для доказательства $P(x)$ для всех $x :: \tau$:

- для каждого уравнения $f\ (e) = t$, доказать $P(e)$ предполагая $P(r)$ для всех рекурсивных вызовов $f\ (r)$ в t .

// Прим.: lhs всегда лексикографически больше rhs, т. к. мы за каждый рекурсивный вызов «отщепляем» один или несколько конструкторов от аргумента в lhs. Если рассмотреть уравнения справа налево, то получится «присоединение» → а т. к. размер определен через количество конструкторов, то lhs больше rhs.

Вычислительная индукция: Пример

fun div2 :: "nat ⇒ nat" where

"div2 0 = 0" |

"div2 (Suc 0) = 0" |

"div2 (Suc (Suc n)) = Suc (div2 n)"

$$\frac{P(0) \qquad P(Suc\ 0) \qquad P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

n,m — произвольные числа

P(m) = теорема верна для всех P