Functional Data Structures

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2022-4-26

Chapter 1

Introduction

What the course is about

Data Structures and Algorithms
for Functional Programming Languages

The code is not enough!

Formal Correctness and Complexity Proofs
with the Proof Assistant *Isabelle*

Proof Assistants

- You give the structure of the proof
- The PA checks the correctness of each step

Government health warnings:

Time consuming
Potentially addictive
Undermines your naive trust in informal proofs

Terminology

Formal = machine-checked
Verification = formal correctness proof

Two landmark verifications

C compiler
Competitive with `gcc -O1`

Xavier Leroy
INRIA Paris
using Coq

Operating system
microkernel (L4)

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

Overview of course

- Week 1–5: Introduction to Isabelle
- Rest of semester: Search trees, priority queues, etc and their (amortized) complexity

What we expect from you

Functional programming experience with an ML/Haskell-like language

First course in data structures and algorithms

First course in discrete mathematics

You will not survive this course without doing the time-consuming homework

Part I

Isabelle

Chapter 2

Programming and Proving

1. Overview of Isabelle/HOL

2. Type and function definitions

3. Induction Heuristics

4. Simplification

Notation

Implication associates to the right:

$$A \Longrightarrow B \Longrightarrow C \quad \text{means} \quad A \Longrightarrow (B \Longrightarrow C)$$

Similarly for other arrows: $\Rightarrow$, $\longrightarrow$

$$\frac{A_1 \quad \ldots \quad A_n}{B} \quad \text{means} \quad A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$$

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:

- For the moment: only $term = term$, e.g. $1 + 2 = 4$
- Later: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\ldots$

1. Overview of Isabelle/HOL
   - Types and terms
   - Interface
   - By example: types $bool$, nat and $list$
   - Summary
   - Numeric Types

Types

Basic syntax:

$$\begin{array}{lcll}
\tau & ::= & (\tau) & \\
& \mid & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \dots & \text{base types} \\
& \mid & 'a \mid 'b \mid \dots & \text{type variables} \\
& \mid & \tau \Rightarrow \tau & \text{functions} \\
& \mid & \tau \times \tau & \text{pairs (ascii: } *\text{)} \\
& \mid & \tau\ \mathit{list} & \text{lists} \\
& \mid & \tau\ \mathit{set} & \text{sets} \\
& \mid & \dots & \text{user-defined types}
\end{array}$$

Terms

Basic syntax:

$$
\begin{array}{rcll}
t & ::= & (t) & \\
 & \mid & a & \text{constant or variable (identifier)} \\
 & \mid & t\ t & \text{function application} \\
 & \mid & \lambda x.\ t & \text{function abstraction} \\
 & \mid & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

λ-calculus

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation:
$t\ ::\ \tau$ means "t is a well-typed term of type τ".

$$\frac{t :: \tau_1 \Rightarrow \tau_2 \qquad u :: \tau_1}{t\ u :: \tau_2}$$

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

User can help with *type annotations* inside the term.
Example: $f\ (x{::}nat)$

Currying

Thou shalt Curry your functions

- Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Predefined syntactic sugar

- *Infix:* $+$, $-$, $*$, $\#$, $@$, $\ldots$
- *Mixfix:* *if* _ *then* _ *else* _, *case* _ *of*, $\ldots$

Prefix binds more strongly than infix:

$$! \quad f\,x + y \;\equiv\; (f\,x) + y \;\not\equiv\; f\,(x + y) \quad !$$

Enclose *if* and *case* in parentheses:

! (*if* _ *then* _ *else* _) !

Theory = Isabelle Module

Syntax:

```
theory MyTh
imports T_1 ... T_n
begin
(definitions, theorems, proofs, ...)*
end
```

$MyTh$: name of theory. Must live in file $MyTh$`.thy`

T_i: names of *imported* theories. Import transitive.

Usually: `imports Main`

Concrete syntax

In `.thy` files:

Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides

isabelle jedit

- Based on *jEdit* editor
- Processes Isabelle text automatically
  when editing `.thy` files (like modern Java IDEs)

`Overview_Demo.thy`

1 Overview of Isabelle/HOL

- Types and terms
- Interface
- By example: types $bool$, nat and $list$
- Summary
- Numeric Types

Type $bool$

datatype $bool \;=\; True \;|\; False$

Predefined functions:
$\wedge, \vee, \longrightarrow, \ldots \;::\; bool \Rightarrow bool \Rightarrow bool$

A *formula* is a term of type $bool$

if-and-only-if: $=$

Type nat

datatype $nat = 0 \mid Suc\ nat$

Values of type nat: $0,\ \ Suc\ 0,\ \ Suc(Suc\ 0),\ \ldots$

Predefined functions: $+,\ *,\ ...\ ::\ nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0,1,2,...\ ::\ 'a, \qquad +\ ::\ \ 'a \Rightarrow 'a \Rightarrow 'a$

You need type annotations: $1\ ::\ nat,\ x + (y::nat)$
unless the context is unambiguous: $Suc\ z$

`Nat_Demo.thy`

An informal proof

Lemma $add\ m\ 0 = m$
Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$,
  the induction hypothesis (IH).
  We need to show $add\ (Suc\ m)\ 0 = Suc\ m$.
  The proof is as follows:
  $$\begin{array}{rcll} add\ (Suc\ m)\ 0 & = & Suc\ (add\ m\ 0) & \text{by def. of } add \\ & = & Suc\ m & \text{by IH} \end{array}$$

Type $'a\ list$

Lists of elements of type $'a$

datatype $'a\ list\ =\ Nil\ \mid\ Cons\ 'a\ \ ('a\ list)$

Some lists: Nil, $Cons\ 1\ Nil$, $Cons\ 1\ (Cons\ 2\ Nil)$, $\ldots$

Syntactic sugar:

- $[] = Nil$: empty list
- $x\ \#\ xs = Cons\ x\ xs$:
  list with first element x (*"head"*) and rest xs (*"tail"*)
- $[x_1, \ldots, x_n] = x_1\ \#\ \ldots\ x_n\ \#\ []$

Structural Induction for lists

To prove that $P(xs)$ for all lists xs, prove

- $P([])$ and
- for arbitrary but fixed x and xs,
  $P(xs)$ implies $P(x\#xs)$.

$$\frac{P([]) \qquad \bigwedge x\ xs.\ P(xs) \Longrightarrow P(x\#xs)}{P(xs)}$$

List_Demo.thy

An informal proof

Lemma $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$

Proof by induction on xs.

- Case Nil: $app\ (app\ Nil\ ys)\ zs = app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$ holds by definition of app.
- Case $Cons\ x\ xs$: We assume $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$ (IH), and we need to show $app\ (app\ (Cons\ x\ xs)\ ys)\ zs = app\ (Cons\ x\ xs)\ (app\ ys\ zs)$.

  The proof is as follows:

  $$\begin{aligned}
  & app\ (app\ (Cons\ x\ xs)\ ys)\ zs \\
  &= Cons\ x\ (app\ (app\ xs\ ys)\ zs) && \text{by definition of } app \\
  &= Cons\ x\ (app\ xs\ (app\ ys\ zs)) && \text{by IH} \\
  &= app\ (Cons\ x\ xs)\ (app\ ys\ zs) && \text{by definition of } app
  \end{aligned}$$

Large library: `HOL/List.thy`

Included in `Main`.

Don't reinvent, reuse!

Predefined: $xs\ @\ ys$ (append), $length$, map, $filter$
$set :: 'a\ list \Rightarrow 'a\ set$, ...

- **datatype** defines (possibly) recursive data types.
- **fun** defines (possibly) recursive functions by pattern-matching over datatype constructors.

Proof methods

- $induction$ performs structural induction on some variable (if the type of the variable is a datatype).

- $auto$ solves as many subgoals as it can, mainly by simplification (symbolic evaluation):

  "$=$" is used only from left to right!

Proofs

General schema:

lemma *name*: `"..."`
apply (...)
apply (...)
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*`[simp]: "..."`

Top down proofs

Command

sorry

"completes" any proof.

Allows top down development:

Assume lemma first, prove it later.

The proof state

$$1.\ \bigwedge x_1 \ \dots \ x_p.\ \ A \Longrightarrow B$$

$x_1 \ \dots \ x_p$	fixed local variables
A	local assumption(s)
B	actual (sub)goal

Multiple assumptions

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

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

$;\quad \approx$ "and"

1. Overview of Isabelle/HOL
    - Types and terms
    - Interface
    - By example: types $bool$, nat and $list$
    - Summary
    - Numeric Types

Numeric types: nat, int, $real$

Need conversion functions (inclusions):

$$\begin{array}{rcl} int & :: & nat \Rightarrow int \\ real & :: & nat \Rightarrow real \\ real\_of\_int & :: & int \Rightarrow real \end{array}$$

If you need type $real$,
import theory $Complex\_Main$ instead of $Main$

Numeric types: nat, int, $real$

Isabelle inserts conversion functions automatically
(with theory $Complex\_Main$)
If there are multiple correct completions,
Isabelle chooses an arbitrary one

Examples

$$(i{::}int) + (n{::}nat) \rightsquigarrow i + int\ n$$

$$((n{::}nat) + n) :: real \rightsquigarrow real(n{+}n),\ real\ n + real\ n$$

Numeric types: nat, int, $real$

Coercion in the other direction:

$$\begin{array}{rcl} nat & :: & int \Rightarrow nat \\ floor & :: & real \Rightarrow int \\ ceiling & :: & real \Rightarrow int \end{array}$$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+,\ -,\ * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$
- Division on nat and int:
  $div,\ mod :: 'a \Rightarrow 'a \Rightarrow 'a$
- Division on $real$: $/ :: 'a \Rightarrow 'a \Rightarrow 'a$
- Exponentiation with nat: $\hat{}\ :: 'a \Rightarrow nat \Rightarrow 'a$
- Exponentiation with $real$: $powr :: 'a \Rightarrow 'a \Rightarrow 'a$
- Absolute value: $abs :: 'a \Rightarrow 'a$

Above all binary operators are infix

2 Type and function definitions

Type definitions

Function definitions

datatype — the general case

$$\textbf{datatype}\ (\alpha_1,\ldots,\alpha_n)t \ = \ C_1\ \tau_{1,1}\ldots\tau_{1,n_1} \ \mid\ \ldots \ \mid\ C_k\ \tau_{k,1}\ldots\tau_{k,n_k}$$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1,\ldots,\alpha_n)t$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:* $(C_i\ x_1\ldots x_{n_i} = C_i\ y_1\ldots y_{n_i}) = (x_1 = y_1 \wedge \cdots \wedge x_{n_i} = y_{n_i})$

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Case expressions

Like in functional languages:

$$(\textit{case } t \textit{ of } \ pat_1 \Rightarrow t_1 \ \mid \ \ldots \ \mid \ pat_n \Rightarrow t_n)$$

Complicated patterns mean complicated proofs!

Need ( ) in context

Tree_Demo.thy

The $option$ type

datatype $'a\ option = None \mid Some\ 'a$

If $'a$ has values a_1, a_2, $\ldots$
then $'a\ option$ has values $None$, $Some\ a_1$, $Some\ a_2$, $\ldots$

Typical application:

fun $lookup :: ('a \times\ 'b)\ list \Rightarrow\ 'a \Rightarrow\ 'b\ option$ **where**
$lookup\ []\ x = None\ |$
$lookup\ ((a, b)\ \#\ ps)\ x =$
  $($*if* $a = x$ *then* $Some\ b$ *else* $lookup\ ps\ x)$

2 Type and function definitions

Type definitions

Function definitions

Non-recursive definitions

Example

definition $sq :: nat \Rightarrow nat$ **where** $sq\ n\ =\ n{*}n$

No pattern matching, just $f\ x_1\ \dots\ x_n\ =\ \dots$

The danger of nontermination

How about $f\,x = f\,x + 1$?

! All functions in HOL must be total **!**

Key features of **fun**

- Pattern-matching over datatype constructors
- Order of equations matters
- Termination must be provable automatically by size measures
- Proves customized induction schema

Example: separation

fun $sep :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$sep\ a\ (x\#y\#zs) = x\ \#\ a\ \#\ sep\ a\ (y\#zs)\ \ |$
$sep\ a\ xs = xs$

primrec

- A restrictive version of **fun**
- Means *primitive recursive*
- Most functions are primitive recursive
- Frequently found in Isabelle theories

The essence of primitive recursion:

$$\begin{array}{lll} f(0) & = \dots & \text{no recursion} \\ f(Suc\ n) & = \dots f(n) \dots & \end{array}$$

$$\begin{array}{lll} g([]) & = \dots & \text{no recursion} \\ g(x\#xs) & = \dots g(xs) \dots & \end{array}$$

Basic induction heuristics

Theorems about recursive functions
are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

A tail recursive reverse

Our initial reverse:

fun $rev :: 'a\ list \Rightarrow 'a\ list$ **where**
$rev\ [] \quad = [] \quad |$
$rev\ (x\#xs) \quad = rev\ xs\ @\ [x]$

A tail recursive version:

fun $itrev :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$itrev\ [] \quad ys = ys \quad |$
$itrev\ (x\#xs) \quad ys =$

lemma $itrev\ xs\ [] = rev\ xs$

`Induction_Demo.thy`

Generalisation

Generalisation

- Replace constants by variables
- Generalize free variables
  - by $arbitrary$ in induction proof
  - (or by universal quantifier in formula)

So far, all proofs were by structural induction because all functions were primitive recursive.

In each induction step, 1 constructor is added.
In each recursive call, 1 constructor is removed.

Now: induction for complex recursion patterns.

Computation Induction

Example

fun $div2 :: nat \Rightarrow nat$ **where**
$div2\ 0 = 0 \quad |$
$div2\ (Suc\ 0) = 0 \quad |$
$div2\ (Suc(Suc\ n)) = Suc(div2\ n)$

$\leadsto$ induction rule `div2.induct`:

$$\frac{P(0) \quad P(Suc\ 0) \quad \bigwedge n.\ \ P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

for each defining equation

$$f(e) \;=\; \ldots f(r_1) \ldots f(r_k) \ldots$$

prove $P(e)$ *assuming* $P(r_1), \ldots, P(r_k)$.

Induction follows course of (terminating!) computation
Motto: properties of f are best proved by rule $f.induct$

How to apply $f.induct$

If $f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau'$:

$$(\textit{induction } a_1 \ \ldots \ a_n \ \textit{rule: f.induct})$$

Heuristic:

- there should be a call $f\ a_1\ \ldots\ a_n$ in your goal
- ideally the a_i should be variables.

`Induction_Demo.thy`

Computation Induction

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification

Simplification means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\rightsquigarrow$ *simplification rule*

Simplification $=$ (Term) Rewriting

An example

Equations:

$$
\begin{array}{rcll}
0 + n & = & n & (1) \\
(Suc\ m) + n & = & Suc\ (m + n) & (2) \\
(Suc\ m \leq Suc\ n) & = & (m \leq n) & (3) \\
(0 \leq m) & = & True & (4)
\end{array}
$$

Rewriting:

$$
\begin{array}{rcll}
0 + Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(1)}{=} \\
Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(2)}{=} \\
Suc\ 0 & \leq & Suc\ (0 + x) & \stackrel{(3)}{=} \\
0 & \leq & 0 + x & \stackrel{(4)}{=} \\
 & True & &
\end{array}
$$

Conditional rewriting

Simplification rules can be conditional:

$$\llbracket\ P_1;\ \ldots;\ P_k\ \rrbracket \Longrightarrow l = r$$

is applicable only if all P_i can be proved first, again by simplification.

Example

$$\begin{array}{rcl} p(0) & = & True \\ p(x) \Longrightarrow\ f(x) & = & g(x) \end{array}$$

We can simplify $f(0)$ to $g(0)$ but we cannot simplify $f(1)$ because $p(1)$ is not provable.

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x),\ g(x) = f(x)$

Principle:

$$\llbracket\ P_1; \ldots; P_k\ \rrbracket \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

$$n < m \Longrightarrow (n < Suc\ m) = True \quad \text{YES}$$
$$Suc\ n < m \Longrightarrow (n < m) = True \quad \text{NO}$$

Proof method *simp*

Goal: 1. $[\![P_1; \ldots; P_m]\!] \Longrightarrow C$

apply$(simp\ add\colon\ eq_1\ \ldots\ eq_n)$

Simplify $P_1\ \ldots\ P_m$ and C using

- lemmas with attribute $simp$
- rules from **fun** and **datatype**
- additional lemmas $eq_1\ \ldots\ eq_n$
- assumptions $P_1\ \ldots\ P_m$

Variations:

- $(simp\ \ldots\ del\colon\ \ldots)$ removes $simp$-lemmas
- add and del are optional

auto versus *simp*

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1

- *auto* applies *simp* and more

- *auto* can also be modified:

  $(auto\ simp\ add{:}\ \ldots\ simp\ del{:}\ \ldots)$

Rewriting with definitions

Definitions (**definition**) must be used explicitly:

$$(simp\ add\colon f\_def \ldots)$$

f is the function whose definition is to be unfolded.

Case splitting with $simp/auto$

Automatic:

$$P\ (\textit{if}\ A\ \textit{then}\ s\ \textit{else}\ t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

By hand:

$$P\ (\textit{case}\ e\ \textit{of}\ 0 \Rightarrow a \mid Suc\ n \Rightarrow b)$$
$$=$$
$$(e = 0 \longrightarrow P(a)) \wedge (\forall\, n.\ e = Suc\ n \longrightarrow P(b))$$

Proof method: $(simp\ split\colon\ nat.split)$
Or $auto$. Similar for any datatype t: $t.split$

Splitting pairs with *simp/auto*

How to replace

$$P\ (\textsf{\textit{let}}\ (x,\ y) = t\ \textsf{\textit{in}}\ u\ x\ y)$$

or

$$P\ (\textsf{\textit{case}}\ t\ \textsf{\textit{of}}\ (x,\ y) \Rightarrow u\ x\ y)$$

by

$$\forall\, x\ y.\ t = (x,\ y) \longrightarrow P\ (u\ x\ y)$$

Proof method: $(simp\ split{:}\ prod.split)$

Simp_Demo.thy

Chapter 3

Case Study: Binary Search Trees

Preview: sets

Type: $'a\ set$

Operations: $a \in A$, $A \cup B$, ...

Bounded quantification: $\forall\, a{\in}A.\ P$

Proof method $auto$ knows (a little) about sets.

The (binary) tree library

`imports "HOL-Library.Tree"`

(File: `isabelle/src/HOL/Library/Tree.thy`)

datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

Abbreviations:

$$\begin{array}{rcl} \langle\rangle & \equiv & Leaf \\ \langle l,\ a,\ r\rangle & \equiv & Node\ l\ a\ r \end{array}$$

The (binary) tree library

Size = number of nodes:

$size :: {}'a\ tree \Rightarrow nat$

$size\ \langle\rangle = 0$
$size\ \langle l,\ \_,\ r\rangle = size\ l + size\ r + 1$

Height:

$height :: {}'a\ tree \Rightarrow nat$

$height\ \langle\rangle = 0$
$height\ \langle l,\ \_,\ r\rangle = max\ (height\ l)\ (height\ r) + 1$

The (binary) tree library

The set of elements in a tree:

$set\_tree :: {}'a\ tree \Rightarrow {}'a\ set$

$set\_tree\ \langle\rangle = \{\}$
$set\_tree\ \langle l,\ a,\ r\rangle = set\_tree\ l \cup \{a\} \cup set\_tree\ r$

Inorder listing:

$inorder :: {}'a\ tree \Rightarrow {}'a\ list$

$inorder\ \langle\rangle = []$
$inorder\ \langle l,\ x,\ r\rangle = inorder\ l\ @\ [x]\ @\ inorder\ r$

The (binary) tree library

Binary search tree invariant:
$bst :: {'a}\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$
$bst\ \langle l,\ a,\ r\rangle =$
$((\forall\, x{\in}set\_tree\ l.\ x < a) \wedge$
$(\forall\, x{\in}set\_tree\ r.\ a < x) \wedge bst\ l \wedge bst\ r)$

For any type $'a$?

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)
- and a set of axioms about those functions

Example: class $linorder$: linear orders with $\leq$, $<$

A type belongs to some class if

- the interface functions are defined on that type
- and satisfy the axioms of the class (proof needed!)

Notation: $\tau :: C$ means type τ belongs to class C

Example: $bst :: ('a :: linorder)\ tree \Rightarrow bool$

$\Longrightarrow$ $'a$ must be a linear order!

Case study

`BST_Demo.thy`

This was easy!

Because we chose easy problems.

Difficult problems need more than $induction$+$auto$.

We need more automation
and a more expressive proof language

Chapter 4

Logic and Proof Beyond Equality

5. Logical Formulas

6. Proof Automation

7. Single Step Proofs

Syntax (in decreasing precedence):

$$
\begin{array}{rclclcl}
\mathit{form} & ::= & (\mathit{form}) & \mid & \mathit{term} = \mathit{term} & \mid & \neg\mathit{form} \\
& \mid & \mathit{form} \wedge \mathit{form} & \mid & \mathit{form} \vee \mathit{form} & \mid & \mathit{form} \longrightarrow \mathit{form} \\
& \mid & \forall x.\ \mathit{form} & \mid & \exists x.\ \mathit{form} & &
\end{array}
$$

Examples:

$$
\begin{array}{rcl}
\neg\ A\ \wedge\ B \vee\ C & \equiv & ((\neg\ A)\ \wedge\ B)\ \vee\ C \\
s\ =\ t\ \wedge\ C & \equiv & (s\ =\ t)\ \wedge\ C \\
A\ \wedge\ B = B \wedge\ A & \equiv & \color{red}{A\ \wedge\ (B\ =\ B)\ \wedge\ A} \\
\forall\, x.\ P\ x \wedge\ Q\ x & \equiv & \forall\, x.\ (P\ x \wedge\ Q\ x)
\end{array}
$$

Input syntax: $\longleftrightarrow$ (same precedence as $\longrightarrow$)

Variable binding convention:

$$\forall\, x\ y.\ P\ x\ y\ \equiv\ \forall\, x.\ \forall\, y.\ P\ x\ y$$

Similarly for $\exists$ and λ.

Warning

Quantifiers have low precedence
and need to be parenthesized (if in some context)

$$! \quad P \wedge \forall x.\ Q\ x \ \rightsquigarrow\ P \wedge (\forall x.\ Q\ x) \quad !$$

Mathematical symbols

and their ascii representations

$\forall$	\<forall>	ALL
$\exists$	\<exists>	EX
λ	\<lambda>	%
$\longrightarrow$	-->	
$\longleftrightarrow$	<->	
$\wedge$	/\	&
$\vee$	\/	|
$\neg$	\<not>	~
$\neq$	\<noteq>	~=

Sets over type $'a$

$'a\ set$

- $\{\}$, $\{e_1, \ldots, e_n\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-A$
- $\{x.\ P\}$ where x is a variable
- $\ldots$

$\in$	\<in>	:
$\subseteq$	\<subseteq>	<=
$\cup$	\<union>	Un
$\cap$	\<inter>	Int

simp and *auto*

simp: rewriting and a bit of arithmetic

auto: rewriting and a bit of arithmetic, logic and sets

- Show you where they got stuck
- highly incomplete
- Extensible with new *simp*-rules

Exception: *auto* acts on all subgoals

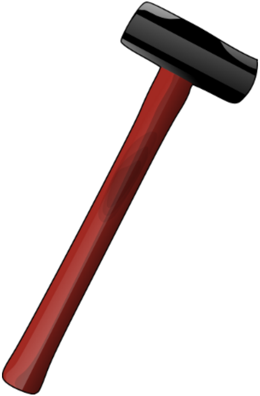
fastforce

- rewriting, logic, sets, relations and a bit of arithmetic.
- incomplete but better than *auto*.
- Succeeds or fails
- Extensible with new *simp*-rules

blast

- A complete proof search procedure for FOL ...
- ... but (almost) without "="
- Covers logic, sets and relations
- Succeeds or fails
- Extensible with new deduction rules

Sledgehammer



Architecture:

Isabelle

Goal
& filtered library ↓ ↑ Proof

external
ATPs[1]

Characteristics:

- Sometimes it works,
- sometimes it doesn't.

Do you feel lucky?

[1] Automatic Theorem Provers

by($proof\text{-}method$)

$$\approx$$

apply($proof\text{-}method$)
done

Auto_Proof_Demo.thy

6 Proof Automation

Automating Arithmetic

Linear formulas

Only:

variables

numbers

number $*$ variable

$+, -$

$=, \leq, <$

$\neg, \wedge, \vee, \longrightarrow, \longleftrightarrow$

Examples

Linear: $3 * x + 5 * y \leq z \longrightarrow x < z$

Nonlinear: $x \leq x * x$

Extended linear formulas

Also allowed:

min, max

$even$, odd

$t\ div\ n$, $t\ mod\ n$ where n is a number

conversion functions

nat, $floor$, $ceiling$, abs

Automatic proof of arithmetic formulas

by *arith*

Proof method *arith* tries to prove arithmetic formulas.

- Succeeds or fails
- Decision procedure for extended linear formulas
- Nonlinear subterms are viewed as (new) variables.
  Example: $x \leq x * x + f\ y$ is viewed as $x \leq u + v$

Automatic proof of arithmetic formulas

by (*simp add: algebra_simps*)

- The lemmas list *algebra_simps* helps to simplify arithmetic formulas
- It contains associativity, commutativity and distributivity of $+$ and $*$.
- This may prove the formula, may make it simpler, or may make it unreadable.

Automatic proof of arithmetic formulas

by ($simp\ add$: $field\_simps$)

- The lemmas list $field\_simps$ extends $algebra\_simps$ by rules for /
- Can only cancel common terms in a quotient, e.g. $x * y \ / \ (x * z)$, if $x \neq 0$ can be proved.

Numerals

Numerals are syntactically different from Suc-terms.
Therefore numerals do not match Suc-patterns.

Example

Exponentiation $x \ \hat{}\ n$ is defined by Suc-recursion on n.
Therefore $x \ \hat{}\ 2$ is not simplified by $simp$ and $auto$.

Numerals can be converted into Suc-terms with rule
$numeral\_eq\_Suc$

Example

$simp\ add: numeral\_eq\_Suc$ rewrites $x \ \hat{}\ 2$ to $x * x$

`Auto_Proof_Demo.thy`

Arithmetic

5 Logical Formulas

6 Proof Automation

7 Single Step Proofs

Step-by-step proofs can be necessary if automation fails and you have to explore where and why it failed by taking the goal apart.

What are these *?-variables* ?

After you have finished a proof, Isabelle turns all free variables V in the theorem into $?V$.

Example: theorem `conjI`: $[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$

These ?-variables can later be instantiated:

- By hand:
  `conjI[of "a=b" "False"]` $\leadsto$
  $[\![a = b;\ False]\!] \Longrightarrow a = b \wedge False$
- By unification:
  unifying $?P \wedge ?Q$ with $a{=}b \wedge False$
  sets $?P$ to $a{=}b$ and $?Q$ to $False$.

Rule application

Example: rule: $[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

Result: $1.\ \ldots \Longrightarrow A$

$2.\ \ldots \Longrightarrow B$

The general case: applying rule $[\![\ A_1; \ldots\ ; A_n\ ]\!] \Longrightarrow A$ to subgoal $\ldots \Longrightarrow C$:

- Unify A and C
- Replace C with n new subgoals $A_1\ \ldots\ A_n$

apply($rule\ xyz$)

"Backchaining"

Typical backwards rules

$$\frac{?P \quad ?Q}{?P \land ?Q}\ \texttt{conjI}$$

$$\frac{?P \Longrightarrow ?Q}{?P \longrightarrow ?Q}\ \texttt{impI} \qquad \frac{\bigwedge x.\ ?P\ x}{\forall x.\ ?P\ x}\ \texttt{allI}$$

$$\frac{?P \Longrightarrow ?Q \quad ?Q \Longrightarrow ?P}{?P = ?Q}\ \texttt{iffI}$$

They are known as introduction rules
because they *introduce* a particular connective.

Forward proof: OF

If r is a theorem $A \Longrightarrow B$
and s is a theorem that unifies with A then

$$r[OF\ s]$$

is the theorem obtained by proving A with s.

Example: theorem `refl`: $?t = ?t$

$$\texttt{conjI[OF refl[of "a"]]}$$

$$\leadsto$$

$$?Q \Longrightarrow a = a \land\ ?Q$$

The general case:

If r is a theorem $\llbracket\ A_1; \ldots; A_n\ \rrbracket \Longrightarrow A$
and $r_1, \ldots, r_m$ $(m \le n)$ are theorems then

$$r[OF\ r_1\ \ldots\ r_m]$$

is the theorem obtained
by proving $A_1\ \ldots\ A_m$ with $r_1\ \ldots\ r_m$.

Example: theorem `refl`: $?t = ?t$

```
conjI[OF refl[of "a"] refl[of "b"]]
```

$$\leadsto$$

$$a = a \wedge b = b$$

From now on: ? mostly suppressed on slides

Single_Step_Demo.thy

$\Longrightarrow$ versus $\longrightarrow$

$\Longrightarrow$ is part of the Isabelle framework. It structures theorems and proof states: $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$

$\longrightarrow$ is part of HOL and can occur inside the logical formulas A_i and A.

Phrase theorems like this $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$
not like this $A_1 \wedge \ldots \wedge A_n \longrightarrow A$

Chapter 5

Isar: A Language for Structured Proofs

8. Isar by example

9. Proof patterns

10. Streamlining Proofs

11. 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** $\ldots$
qed

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

Isar core syntax

$$\begin{array}{lcl} \text{proof} & = & \textbf{proof}\ [\text{method}]\ \ \text{step}^*\ \ \textbf{qed} \\ & | & \textbf{by}\ \text{method} \end{array}$$

$$\text{method} \ = \ (simp\ \dots) \mid (blast\ \dots) \mid (induction\ \dots) \mid \dots$$

$$\begin{array}{lcll} \text{step} & = & \textbf{fix}\ \text{variables} & (\bigwedge) \\ & | & \textbf{assume}\ \text{prop} & (\Longrightarrow) \\ & | & [\textbf{from}\ \text{fact}^+]\ \ (\textbf{have} \mid \textbf{show})\ \text{prop}\ \ \text{proof} & \end{array}$$

$$\text{prop} \ = \ [\text{name:}]\ \ \text{"formula"}$$

$$\text{fact} \ = \ \text{name} \mid \dots$$

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$(simp\ add\colon surj\_def)$
  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

$$
\begin{array}{rcl}
\color{blue}{this} & = & \text{the previous proposition proved or assumed} \\
\color{blue}{\textsf{then}} & = & \textbf{from } this \\
\color{blue}{\textsf{thus}} & = & \textbf{then show} \\
\color{blue}{\textsf{hence}} & = & \textbf{then have}
\end{array}
$$

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** $y :: \tau_2 \dots$
assumes *a:* P **and** *b:* $Q \dots$
shows R

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

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 $False\ \langle proof\rangle$
qed

show P
proof $(rule\ ccontr)$
  assume $\neg P$
  $\vdots$
  show $False\ \langle proof\rangle$
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

9 Proof patterns

Chains of (In)Equations

Chains of equations

Textbook proof

$$
\begin{aligned}
t_1 &= t_2 \quad \langle\text{justification}\rangle\\
&= t_3 \quad \langle\text{justification}\rangle\\
&\ \vdots\\
&= t_n \quad \langle\text{justification}\rangle
\end{aligned}
$$

In Isabelle:

have $t_1 = t_2$ $\langle proof\rangle$
also have $... = t_3$ $\langle proof\rangle$
$\vdots$
also have $... = t_n$ $\langle proof\rangle$
finally show $t_1 = t_n$.

"..." is literally three dots

Chains of equations and inequations

Instead of $=$ you may also use $\leq$ and $<$.

Example

have $t_1 < t_2$ $\langle proof \rangle$
also have $\ldots = t_3$ $\langle proof \rangle$
$\vdots$
also have $\ldots \leq t_n$ $\langle proof \rangle$
finally show $t_1 < t_n$.

How to interpret "..."

$$\begin{array}{rl} \textbf{have} & t_1 \leq t_2 \;\; \langle proof \rangle \\ \textbf{also have} & \ldots = t_3 \;\; \langle proof \rangle \end{array}$$

Here "..." is internally replaced by t_2

In general, if $this$ is the formula $p\ t_1\ t_2$ where p is some constant, then "..." stands for t_2.

`Isar_Demo.thy`

Example & Exercise

10 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

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 `x>0` ...

$\uparrow \quad \uparrow$

back quotes

Isar_Demo.thy

Pattern matching and quotations

10 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

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

10 Streamlining Proofs

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):

$$\bigwedge x_1 \ldots x_n.\ [\![A_1; \ldots ; A_m]\!] \Longrightarrow B$$

How to prove each subgoal:

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

Separated by **next**

8 Isar by example

9 Proof patterns

10 Streamlining Proofs

11 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** $0\colon\ A(0)$
$\vdots$		**let** $?case = P(0)$
show $?case$		
next		
case $(Suc\ n)$	$\equiv$	**fix** n
$\vdots$		**assume** $Suc\colon\ A(n) \Longrightarrow P(n)$ $A(Suc\ n)$
$\vdots$		**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. $?case$)

`Isar_Induction_Demo.thy`

Computation induction

Computation induction

If function f is defined by **fun** with n equations:

proof($induction\ s\ t\ ...\ rule$: $f.induct$)

Generates cases named $i = 1 \ldots\ n$:

case $(i\ x\ y\ ...)$

Isabelle/jEdit generates Isar template for you!

Computation induction

Naming

- i is a name, but not $i.IH$
- Needs double quotes: $"i.IH"$
- Indexing: $i(1)$ and $"i.IH"(1)$
- If defining equations for f overlap:
  $\leadsto$ Isabelle instantiates overlapping equations
  $\leadsto$ case names of the form $"i\_j"$