Notation

Implication associates to the right:

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

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

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

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: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, ...

3. Overview of Isabelle/HOL

   Types and terms

   Interface

   By example: types *bool*, *nat* and *list*

   Summary

Types

Basic syntax:

τ	$::=$	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	$\tau\ list$	lists
	$\mid$	$\tau\ set$	sets
	$\mid$	$\ldots$	user-defined types

Convention: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \;\equiv\; \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

Terms

Terms can be formed as follows:

- *Function application:* $f\ t$
  is the call of function f with argument t.
  If f has more arguments: $f\ t_1\ t_2\ \ldots$
  Examples: $sin\ \pi$, $plus\ x\ y$
- *Function abstraction:* $\lambda x.\ t$
  is the function with parameter x and result t,
  i.e. "$x \mapsto t$".
  Example: $\lambda x.\ plus\ x\ x$

Terms

Basic syntax:

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

Examples: $f\ (g\ x)\ \ y$

$h\ (\lambda x.\ f\ (g\ x))$

Convention: $f\ t_1\ \ t_2\ \ t_3\ \equiv\ ((f\ t_1)\ \ t_2)\ \ t_3$

This language of terms is known as the *λ-calculus*.

The computation rule of the λ-calculus is the replacement of formal by actual parameters:

$$(\lambda x.\ t)\ u \;=\; t[u/x]$$

where $t[u/x]$ is "t with u substituted for x".

Example: $(\lambda x.\ x + 5)\ 3 \;=\; 3 + 5$

- The step from $(\lambda x.\ t)\ u$ to $t[u/x]$ is called *β-reduction*.
- Isabelle performs β-reduction automatically.

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: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Advantage:

Currying allows *partial application*

$f\ a_1$ where $a_1 :: \tau_1$

Predefined syntactic sugar

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

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

3 Overview of Isabelle/HOL

isabelle jedit

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

Overview_Demo.thy

3 Overview of Isabelle/HOL

Type $bool$

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

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:
  $$\begin{array}{lcll} add\ (Suc\ m)\ 0 & = & Suc\ (add\ m\ 0) & \text{by def. of } add \\ & = & Suc\ m & \text{by IH} \end{array}$$

Type $'a\ list$

Lists of elements of type $'a$

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

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

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:
  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs$
  $= Cons\ x\ (app\ (app\ xs\ ys)\ zs)$ by definition of app
  $= Cons\ x\ (app\ xs\ (app\ ys\ zs))$ by IH
  $= app\ (Cons\ x\ xs)\ (app\ ys\ zs)$ by definition of app

Large library: `HOL/List.thy`

Included in `Main`.

Don't reinvent, reuse!

Predefined: $xs\ @\ ys$ (append), $length$, and map

3 Overview of Isabelle/HOL

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

- **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

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

$;\quad \approx \quad$ "and"

3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Examples

type_synonym $string = char\ list$

type_synonym $('a,'b)foo = 'a\ list \times\ 'b\ list$

Type synonyms are expanded after parsing
and are not present in internal representation and output

datatype — the general case

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

- *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

Datatype values can be taken apart with *case*:

$$(\textit{case } xs \textit{ of } [] \Rightarrow \ldots \mid y\#ys \Rightarrow \ldots\, y \ldots\, ys \ldots)$$

Wildcards: _

$$(\textit{case } m \textit{ of } 0 \Rightarrow Suc\ 0 \mid Suc\ \_ \Rightarrow 0)$$

Nested patterns:

$$(\textit{case } xs \textit{ of } [0] \Rightarrow 0 \mid [Suc\ n] \Rightarrow n \mid \_ \Rightarrow 2)$$

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

Typical application:

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

4 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 $f\ x_1\ \ldots\ x_n\ =\ \ldots$

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**
$sep\ a\ (x\#y\#zs) = x\ \#\ a\ \#\ sep\ a\ (y\#zs)\ \ |$
$sep\ a\ xs = xs$

Example: Ackermann

fun $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**

$$
\begin{aligned}
&ack\ 0\ n &&= Suc\ n \quad | \\
&ack\ (Suc\ m)\ 0 &&= ack\ m\ (Suc\ 0) \quad | \\
&ack\ (Suc\ m)\ (Suc\ n) &&= ack\ m\ (ack\ (Suc\ m)\ n)
\end{aligned}
$$

Terminates because the arguments decrease *lexicographically* with each recursive call:

- $(Suc\ m,\ 0) > (m,\ Suc\ 0)$
- $(Suc\ m,\ Suc\ n) > (Suc\ m,\ n)$
- $(Suc\ m,\ Suc\ n) > (m,\ \_)$

primrec

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

The essence of primitive recursion:

$$
\begin{aligned}
f(0) &= \ldots && \text{no recursion} \\
f(Suc\ n) &= \ldots f(n) \ldots
\end{aligned}
$$

$$
\begin{aligned}
g([]) &= \ldots && \text{no recursion} \\
g(x \# xs) &= \ldots g(xs) \ldots
\end{aligned}
$$

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**
$rev\ [] = []\ |$
$rev\ (x\#xs) = rev\ xs\ @\ [x]$

A tail recursive version:

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

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

$$(induction\ a_1\ \ldots\ a_n\ rule\colon f.induct)$$

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

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:

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

Rewriting:

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

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:

$$[\![\ P_1; \ldots; P_k\ ]\!] \Longrightarrow l = r$$

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. $\llbracket P_1; \ldots; P_m \rrbracket \Longrightarrow C$

apply$(simp\ add{:}\ eq_1\ \ldots\ eq_n)$

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:

- $(simp\ \ldots\ del{:}\ \ldots)$ 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:

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

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: $(simp\ split\text{:}\ nat.split)$
Or $auto$. Similar for any datatype t: $t.split$

Simp_Demo.thy