Interactive Theorem Proving in Higher-Order Logics

Partly based on material by Mike Gordon, Tobias Nipkow, and Andrew Pitts

Jasmin Blanchette
<table>
<thead>
<tr>
<th>Time</th>
<th>Monday July 31</th>
<th>Tuesday August 1</th>
<th>Wednesday August 2</th>
<th>Thursday August 3</th>
<th>Friday August 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:00</td>
<td>Registration</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08:45</td>
<td>Opening</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09:00</td>
<td>State-of-the-art SAT solving</td>
<td>Practical Session on SAT/SMT</td>
<td>Symbolic Computation (Quantifier Elimination)</td>
<td>Cylindrical Algebraic Decomposition and Real Polynomial Constraints</td>
<td>Industrial Applications and Challenges for Verifying Reactive Embedded Software</td>
</tr>
<tr>
<td>10:30</td>
<td>Coffee break</td>
<td>Coffee break</td>
<td>Coffee break</td>
<td>Coffee break</td>
<td>Coffee break</td>
</tr>
<tr>
<td>11:00</td>
<td>State-of-the-art SAT solving</td>
<td>Practical Session on SAT/SMT</td>
<td>Symbolic Computation (Quantifier Elimination)</td>
<td>Cylindrical Algebraic Decomposition and Real Polynomial Constraints</td>
<td>Formal Verification in an Industrial Setting</td>
</tr>
<tr>
<td>11:45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Closing</td>
</tr>
<tr>
<td>12:30</td>
<td>Lunch</td>
<td>Lunch</td>
<td>Lunch</td>
<td>Lunch</td>
<td>Lunch</td>
</tr>
<tr>
<td>14:00</td>
<td>Foundations of Satisfiability Modulo Theories</td>
<td>State-of-the-art FOL Solving</td>
<td>Syntax-Guided Synthesis</td>
<td>Symbolic Computation through Maple and Reduce</td>
<td>Lunch</td>
</tr>
<tr>
<td>15:30</td>
<td>Coffee break</td>
<td>Coffee break</td>
<td>Coffee break</td>
<td>Coffee break</td>
<td>Coffee break</td>
</tr>
<tr>
<td>16:00</td>
<td>Foundations of Satisfiability Modulo Theories</td>
<td>Interactive Theorem Proving in Higher-Order Logics</td>
<td>Syntax-Guided Synthesis</td>
<td>Symbolic Computation through Maple and Reduce</td>
<td></td>
</tr>
<tr>
<td>17:30</td>
<td>Break</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:00</td>
<td>Formal Verification of Financial Algorithms</td>
<td>Some information on Saarland, its history, and the restaurant</td>
<td>Bus transfer to the restaurant (from MPI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:00</td>
<td>Get together</td>
<td></td>
<td></td>
<td>Dinner</td>
<td></td>
</tr>
</tbody>
</table>
What are proof assistants?

**Proof assistants** (or **interactive theorem provers**) are programs with a graphical user interface designed for proving logical formulas.

The logical formulas may represent mathematical theorems but also the correctness of hardware or software.

Proof assistants help catch almost all flaws in pen-and-paper proofs, but they are tedious to use.
What are they based on?

Different proof assistants are based on different logics.

Some logics are more expressive (flexible) than others; others are easier to automate.

- **First-Order Logic**
  - less expressive
  - easier to automate

- **Higher-Order Logic**
  - more expressive
  - harder to automate

- **Set Theory**
  - less expressive
  - easier to automate

- **Type Theory**
  - more expressive
  - harder to automate
Why should we trust them?

Some proof assistants are designed around a **small inference kernel**, with simple logical primitives.

Some generate detailed **proof objects**, which can be rechecked independently by small programs.

And some just have to be trusted.
## What are the main systems?

<table>
<thead>
<tr>
<th>System Type</th>
<th>System</th>
<th>Small kernel</th>
<th>Proof objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>First-Order Logic</td>
<td>ACL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higher-Order Logic</td>
<td>HOL4</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HOL Light</td>
<td>✓</td>
<td>(√)</td>
</tr>
<tr>
<td></td>
<td>Isabelle/HOL</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>PVS</td>
<td>(√)</td>
<td>(√)</td>
</tr>
<tr>
<td>Set Theory</td>
<td>Isabelle/ZF</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Mizar</td>
<td>(√)</td>
<td></td>
</tr>
<tr>
<td>Type Theory</td>
<td>Agda</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Coq</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Lean</td>
<td>(√)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Matita</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
### Are proof assistants toys?

<table>
<thead>
<tr>
<th>Mathematics</th>
<th>Four-color theorem</th>
<th>Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Feit-Thompson theorem</td>
<td>Coq</td>
</tr>
<tr>
<td></td>
<td>Kepler conjecture</td>
<td>HOL Light &amp; Isabelle/HOL</td>
</tr>
<tr>
<td>Hardware</td>
<td>AMD</td>
<td>ACL2</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>HOL Light</td>
</tr>
<tr>
<td>Software</td>
<td>Compiler</td>
<td>Coq</td>
</tr>
<tr>
<td></td>
<td>Operating system</td>
<td>Isabelle/HOL</td>
</tr>
<tr>
<td>Programming languages</td>
<td>Program semantics courses</td>
<td>Coq &amp; Isabelle/HOL</td>
</tr>
<tr>
<td></td>
<td>POPL conference</td>
<td>Coq &amp; Agda</td>
</tr>
</tbody>
</table>
Are proof assistants toys?

<table>
<thead>
<tr>
<th>Automated reasoning</th>
<th>Completeness of FOL</th>
<th>SAT proof checkers</th>
<th>SAT solver with 2WL</th>
<th>Resolution</th>
<th>Coq, Isabelle/HOL, Mizar, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ACL2, Coq, Isabelle/HOL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Isabelle/HOL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Isabelle/HOL</td>
</tr>
</tbody>
</table>
What do proofs look like?

*Tactical proofs* apply tactics to the *proof goal* to produce a new proof goal, proceeding in a backward fashion.

*Declarative proofs* state intermediate properties, proceeding in a forward fashion.
What do proofs look like?

Let us prove \(A \text{ and } B \implies B \text{ and } A\) using tactics.

**Goal:** \(A \text{ and } B \implies B \text{ and } A\)

**Tactic:** rule and-left

**Goal:** \(A, B \implies B \text{ and } A\)

**Tactic:** rule and-right

**Goals:** \(A, B \implies B \text{ and } A, B \implies A\)

**Tactics:** rule implies-trivial \quad rule implies-trivial

No goals left
What do proofs look like?

Let us prove \( A \text{ and } B \) implies \( B \text{ and } A \) declaratively.

proof

\begin{itemize}
  \item assume \( A \text{ and } B \)
  \item from \( A \text{ and } B \) have \( A \) by (rule and-get-left)
  \item from \( A \text{ and } B \) have \( B \) by (rule and-get-right)
  \item from \( B, A \) show \( B \text{ and } A \) by (rule and-right)
\end{itemize}

qed
Most proof assistants offer a variety of general and specialized automatic tactics.

The **Simplifier** rewrites by applying equations left-to-right; e.g. the equation \( x + 0 = x \) can be used to simplify the goal \( 2 + 0 < 3 \) to \( 2 < 3 \).

The **Arithmetic Procedure** can prove formulas involving linear arithmetic, e.g. \( 2 < 3, \, k > n \text{ or } k = 0 \text{ or } [k \leq n \text{ and } k \neq 0] \).

The **General Reasoner** performs a systematic, bounded proof search, applying rules like and-left and and-right.
Can proofs be automated?

In addition, automatic theorem provers can be invoked via tools such as Sledgehammer for Isabelle/HOL and HOLyHammer for HOL Light and HOL4.

These provers perform a systematic search in first-order logic and are designed to be very efficient.

There are also integrations of computer algebra systems.
Does there exist a function \( f \) from reals to reals such that for all \( x \) and \( y \), \( f(x + y^2) - f(x) \geq y \)?

let lemma = prove
(`~(!x y. f(x + y * y) - f(x) >= y)`,
 REWRITE_TAC[real_ge] THEN REPEAT STRIP_TAC THENL
 SUBGOAL_THEN `∀n x y. &n * y <= f(x + &n * y * y) - f(x)` MP_TAC THENL
 [MATCH_MP_TAC num_INDUCTION THEN SIMP_TAC[REAL_MUL_LZERO; REAL_ADD_RID] THEN
  REWRITE_TAC[REAL_SUB_REFL; REAL_LE_REFL; GSYM REAL_OF_NUM_SUC] THEN
  GEN_TAC THEN REPEAT(MATCH_MP_TAC MONO_FORALL THEN GEN_TAC) THEN
  FIRST_X_ASSUM(MP_TAC o SPECL [`x + &n * y * y`; `y:real`]) THEN
  SIMP_TAC[REAL_ADD_ASSOC; REAL_ADD_RDISTRIB; REAL_MUL_LID] THEN
  REAL_ARITH_TAC;
 X_CHOOSE_TAC `m:num` (SPEC `f(&1) - f(&0):real` REAL_ARCH_SIMPLE) THEN
 DISCH_THEN(MP_TAC o SPECL [`SUC m EXP 2`; `&0`; `inv(&SUC m)`]) THEN
 REWRITE_TAC[REAL_ADD_LID; GSYM REAL_OF_NUM_SUC; GSYM REAL_OF_NUM_POW] THEN
 REWRITE_TAC[REAL_FIELD `(&m + &1) pow 2 * inv(&m + &1) = &m + &1`;
   REAL_FIELD `(&m + &1) pow 2 * inv(&m + &1) * inv(&m + &1) = &1`] THEN
 ASM_REAL_ARITH_TAC);
Does there exist a function $f$ from reals to reals such that for all $x$ and $y$, $f(x + y^2) - f(x) \geq y$?

[1] $f(x + y^2) - f(x) \geq y$ for any $x$ and $y$ (given)

[2] $f(x + ny^2) - f(x) \geq ny$ for any $x$, $y$, and natural number $n$ (by an easy induction using [1] for the step case)

[3] $f(1) - f(0) \geq m + 1$ for any natural number $m$ (set $n = (m + 1)^2$, $x = 0$, $y = 1/(m + 1)$ in [2])


John Harrison
Lemma shows
\( \forall x y. f (x + y * y) - f x \geq y \)
Lemma: \( A + B = (A \cup B) + (A \cap B) \)
Lemma: \( A + B = (A \cup B) + (A \cap B) \)
Lemma "A + B = (A ∪ B) + (A ∩ B)"

Sledgehammer

Sledgehammering...
"z3": Try this: by (smt Multiset.diff_add add_diff_cancel_right subst mset_le_add_right multiset_add_right multiset_distrib)

Isar proof (26 ms):
proof -
  have "A + B = A + B + (B - (A + B))"
    by (melis mset_le_add_right subst_mset.sup.orderE sup_subset_mset_def)
  then have f1: "A + B = (A + (B - A)) ∪ (A + B)"
    by (simp add: Multiset.diff_add add_diff_cancel_right subst_mset.sup_commute sup_subset_mset_def)
  have "A + (B - A) = B ∪ (A + B)"
    by (simp add: subst_mset.sup_commute sup_subset_mset_def)
  then show ?thesis
    using f1 by (simp add: Multiset.diff_add add_multiset_inter_def sup_subset_mset_def)
qed
Lemma "A + B = (A \cup B) + (A \cap B)"

Proof:

Isar proof (26 ms):

proof -

  have "A + B = A + B + (B - (A + B))"
  by (melis mset_le_add_right subset_mset.sup.orderE sup_subset_mset_def)
  then have f1: "A + B = (A + (B - A)) \cup (A + B)"
  by (simp add: Multiset.diff_add subset_mset.sup commute sup_subset_mset_def)
  have "A + (B - A) = B \cup (A + B)"
  by (simp add: add_difference_subset_mset_def commute sup_subset_mset_def)
  then show thesis using f1 by (simp add: Multiset.diff_add Multiset_inter_def sup_subset_mset_def)
qed
lemma "A + B = (A \cup B) + (A \cap B)"

proof -
have "A + B = A + B + (B - (A + B))"
  by (metis mset_le_add_right subset_mset.sup.orderE sup_subset_mset_def)
then have fl: "A + B = (A + (B - A)) \cup (A + B)"
  by (simp add: Multiset.diff_add subset_mset.sup.commute sup_subset_mset_def)
have "A + (B - A) = B \cup A"
  by (simp add: subset_mset.sup.commute sup_subset_mset_def)
then show thesis

proof (state)
goal (1 subgoal):
  1. A + B = A \cup B + A \cap B
Proof assistants

Isabelle

well suited for large formalizations
but require intensive manual labor

Automatic provers

Sledgehammer

fully automatic but no proof management

Z3
Vampire
SPASS
LEO-II
Satallax
VeriT
CVC4
Isabelle HOL

select lemmas +
translate to FOL

reconstruct proof

superposition

SMT
superposition

refutational resolution rule
term ordering
equality reasoning
redundancy criterion

E, SPASS, Vampire, ...

SMT

refutational SAT solver
+ congruence closure
+ quantifier instantiation
+ other theories (e.g. LIA, LRA)

CVC4, veriT, Yices, Z3, ...
How many hammers are there?

<table>
<thead>
<tr>
<th>pre-Sledgehammer</th>
<th>post-Sledgehammer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Otter in ACL2</td>
<td>HOLyHammer for HOLs</td>
</tr>
<tr>
<td>Bliksem in Coq</td>
<td>MizAR for Mizar</td>
</tr>
<tr>
<td>Gandalf in HOL98</td>
<td>SMTCOq/CVC4Coq for Coq</td>
</tr>
<tr>
<td>DISCOUNT, SPASS, etc., in ILF</td>
<td>SMT integration in TLAPS</td>
</tr>
<tr>
<td>Otter, SPASS, etc., in KIV</td>
<td>...</td>
</tr>
<tr>
<td>LEO, SPASS, etc., in ΩMEGA</td>
<td>...</td>
</tr>
<tr>
<td>E, Vampire, etc., in Naproche</td>
<td>...</td>
</tr>
</tbody>
</table>
I have recently been working on a new development. Sledgehammer has found some simply incredible proofs. I would estimate the improvement in productivity as a factor of at least three, maybe five. Sledgehammers … have led to visible success. Fully automated procedures can prove … 47% of the HOL Light/Flyspeck libraries, with comparable rates in Isabelle. These automation rates represent an enormous saving in human labor.
⊕ productivity
⊕ teaching revolution:
   Isar + auto + induct + Sledgehammer
⊕ lemma search

⊖ higher-order (induction)
⊖ other logical mismatches
⊖ too much search, not enough computation/intuition
⊖ end-game/transparency
⊖ what about nontheorems?
What if the formula is wrong?

**Counterexample generators** automatically test the goal with different values.

They are useful to detect errors early, whether they are in the formula to prove or in the concepts on which it builds.

**lemma** \( i \leq j \) and \( n \leq m \) implies \( in + jm \leq im + jn \)

**nitpick**

Counterexample: \( i = n = 0 \) and \( j = m = 1 \)
lemma exec_append: "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
Under the hood

HOL

FORL

SAT

Isabelle

Nitpick

Kodkod

SAT solver
What is HOL?

HOL = higher-order logic
    = Church’s simple theory of types + polymorphism
    = logic of Gordon’s HOL88 and successors
    = logic of HOL Light, HOL Zero, ProofPower–HOL
    = logic of PVS (+ dependent types)
    = logic of Isabelle/HOL (+ type classes)
Syntax of types

\[
\begin{align*}
\tau & ::= (\tau) \\
& \mid \text{bool} | \text{nat} | \text{int} | \ldots \quad \text{base types} \\
& \mid \text{'a} | \text{'b} | \ldots \quad \text{type variables} \\
& \mid \tau \Rightarrow \tau \quad \text{functions} \\
& \mid \tau \times \tau \quad \text{pairs (ASCII: *)} \\
& \mid \tau \text{ list} \quad \text{lists} \\
& \mid \tau \text{ set} \quad \text{sets} \\
& \mid \ldots \quad \text{user-defined types}
\end{align*}
\]

Convention: \( \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3) \)
Syntax of terms

\[ t ::= (t) \]
\[ a \quad \text{constant or variable (identifier)} \]
\[ t t \quad \text{function application} \]
\[ \lambda x. t \quad \text{function abstraction} \]
\[ \ldots \quad \text{lots of syntactic sugar} \]

Convention: \[ f t_1 t_2 t_3 \equiv ((f t_1) t_2) t_3 \]
Isabelle’s metalogic

The HOL types and terms are part of the metalogic.

Alpha-, beta-, eta-equivalence is built-in:

\[
(\lambda x. t[x]) \equiv (\lambda y. t[y]) \quad (\lambda x. t[x]) u \equiv t[u] \quad t^{\sigma \to \tau} \equiv (\lambda x^\sigma. t x)
\]
Notations

Implication and function arrows associate to the right:

\[ a \Rightarrow b \Rightarrow c \text{ means } a \Rightarrow (b \Rightarrow c) \]

The rule format is sometimes used instead of \( \Rightarrow \), e.g.:

\[ \frac{a \quad b}{c} \]
Terms must be well typed
(the argument of every function call must be of the right type).

\[ \vdash x^\sigma : \sigma \quad \vdash c^\sigma : \sigma \]
\[ \vdash t : \tau \quad \vdash t : \sigma \rightarrow \tau \quad \vdash u : \sigma \]
\[ \vdash \lambda x^\sigma . t : \sigma \rightarrow \tau \quad \vdash t u : \tau \]
Type inference

Isabelle computes types of variables (and polymorphic constants) automatically.

In the presence of overloaded functions, this is not always possible.

Users can provide type annotations inside the terms:

\[
\text{e.g. } f (x :: \text{nat})
\]
"Thou shalt curry thy functions."

Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$

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

Currying allows partial applications:

e.g. $(\text{op } +) 1$
Metalogical propositions

Propositions have type $\text{prop}$ (intuitionistic).

Built-in operators:

\[ \iff \, \text{prop} \rightarrow \text{prop} \rightarrow \text{prop} \]
\[ \land \, (\alpha \rightarrow \text{prop}) \rightarrow \text{prop} \]
\[ \equiv \, \alpha \rightarrow \alpha \rightarrow \text{prop} \]

implication
universal quantification
equality
Propositions have type \( \text{bool} \) (classical).

The familiar operators are defined on \( \text{bool} \) (False, True, \( = \), \( \forall \), \( \exists \), \( \neg \), \( \rightarrow \), \( \land \), \( \lor \), …).

\( \leftrightarrow \) is syntactic sugar for \( = \).

\( \text{Trueprop} \) is a special implicit constant that converts a \( \text{bool} \) to a \( \text{prop} \).

\( \text{e.g.} \quad a \land b \Rightarrow c \) is really

\[ \text{Trueprop} \ (a \land b) \Rightarrow \text{Trueprop} \ c \]
Predefined syntactic sugar

\textit{Infix}: +, -, *, #, @, \ldots

\textit{Mixfix}: \texttt{if \_ then \_ else \_}, \texttt{case \_ of}, \ldots

Prefix binds more strongly than infix:

\[ f \ x + y \equiv (f \ x) + y \neq f (x + y) \]

Enclose \texttt{if} and \texttt{case} in parentheses:

\[(\texttt{if \_ then \_ else \_})\]
Theory = Isabelle file

```
theory Myth
imports T_1 \ldots T_n
begin
  (definitions, theorems, proofs, \ldots)^
end
```

Types and terms must be enclosed in quotes ("), except for single identifiers.
Extensions

Definitional

New types are carved out of an old type.
New constants are defined in terms of old ones.

Axiomatic

New types are declared and characterized by axioms.
New constants are introduced by axioms.

Locales

Parameterized by types, terms, and assumptions.
Assumptions discharged upon instantiation.
Definitional

Type Constructors

typedef 'a dlists = {xs : 'a list | distinct xs}
(co)datatype, quotient_type are built on typedef

(Term) Constants

definition id :: 'a => 'a where id x = x
(co)inductive, fun, prim(co)rec, corec, lift_definition
are built on definition
Axiomatic

Type Constructors

```plaintext
typedecl 'a dlist
```

(Term) Constants

```plaintext
axiomatization
  Abs_dlist :: 'a list => 'a dlist and
  Rep_dlist :: 'a dlist => 'a list
where Abs_Rep: distinct xs ==> Rep (Abs xs) = xs
```
Locales

They combine
  – type parameters
  – term parameters
  – assumptions.

```plaintext
locale semigroup = 
  fixes f :: 'a => 'a => 'a (infixl "*" 70)
  assumes assoc: a * b * c = a * (b * c)
begin
  ...
end
```

They are not part of Isabelle’s kernel.
Proof styles

Tactical (apply-style)
Tactics directly modify the proof state.
Backward: reduction of goal to True.

Declarative (Isar-style)
Textual, linearized \textit{natural deduction}.
Forward: intermediate steps towards final goal.
Apply-style proofs

apply method
apply (method arg1 … argN)
by method
done

Main methods: simp
              auto
              blast
             metis
              arith
              rule

Also: try0 and try tools
Isar-style proofs

proof method [or -]
  fix \( x_1 \ldots x_n \)
  assume \( A_1 \ldots A_n \)
  have \( P_1 \) by (method \ldots)
  \ldots
  have \( P_k \) by (method \ldots)
  show \( Q \) by (method \ldots)
qed

Instead of \textbf{by}: nested proof block.
Instead of \textbf{have}: obtain \( y_1 \ldots y_m \) \textbf{where} \( P \).
Extensional equality is axiomatized, the other logical constants are definable.

\[
\begin{align*}
\text{True} & := ((\lambda x. x) = (\lambda x. x)) \\
\text{All} & := (\lambda P. P = (\lambda x. \text{True})) \\
\forall x. P \, x & := \text{All} (\lambda x. P \, x)
\end{align*}
\]
Demo
In conclusion

Proof assistants are wonderful and dreadful.

In some areas (esp. of computer science), they are more wonderful than dreadful. (And they are very addictive.)

They can serve as the glue between automatic theorem provers, computer algebra systems, and the human.

Exhortation: Try them out, and see for yourself if they make sense for your research.