# Interactive Theorem Proving in Higher-Order Logics

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

Jasmin Blanchette

VRIJE UNIVERSITEIT AMSTERDAM

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

# 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.

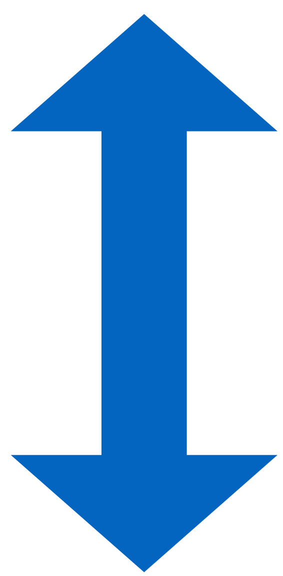
Proof assistants help catch almost all flaws in pen-and-paper proofs, but they are tedious to use.

# 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.

less expressive
easier to automate



more expressive
harder to automate

First-Order Logic

Higher-Order Logic

Set Theory

Type Theory

# 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?

| | | Small kernel | Proof objects |
|---|---|---|---|
| First-Order Logic | ACL2 | | |
| Higher-Order Logic | HOL4 | ✓ | |
| | HOL Light | ✓ | (✓) |
| | Isabelle/HOL | ✓ | ✓ |
| | PVS | (✓) | (✓) |
| Set Theory | Isabelle/ZF | ✓ | ✓ |
| | Mizar | (✓) | |
| Type Theory | Agda | | ✓ |
| | Coq | | ✓ |
| | Lean | (✓) | ✓ |
| | Matita | | ✓ |

# Are proof assistants toys?

| | | |
|---|---|---|
| Mathematics | Four-color theorem | Coq |
| | Feit-Thompson theorem | Coq |
| | Kepler conjecture | HOL Light & Isabelle/HOL |
| Hardware | AMD | ACL2 |
| | Intel | HOL Light |
| Software | Compiler | Coq |
| | Operating system | Isabelle/HOL |
| Programming languages | Program semantics courses | Coq & Isabelle/HOL |
| | POPL conference | Coq & Agda |

# Are proof assistants toys?

| | | |
|---|---|---|
| Automated reasoning | Completeness of FOL | Coq, Isabelle/HOL, Mizar, ... |
| | SAT proof checkers | ACL2, Coq, Isabelle/HOL |
| | SAT solver with 2WL | Isabelle/HOL |
| | Resolution | Isabelle/HOL |
| | Superposition | Isabelle/HOL |

# 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* and *B* implies *B* and *A*** using tactics.

Goal: ***A* and *B* implies *B* and *A***
Tactic: rule and-left
Goal: ***A*, *B* implies *B* and *A***
Tactic: rule and-right
Goals: ***A*, *B* implies *B*** and ***A*, *B* implies *A***
Tactics: rule implies-trivial rule implies-trivial
No goals left

# What do proofs look like?

Let us prove ***A* and *B* implies *B* and *A*** declaratively.

**proof**
  **assume** ***A* and *B***
  **from** *A* and *B* **have** ***A*** **by** (rule and-get-left)
  **from** *A* and *B* **have** ***B*** **by** (rule and-get-right)
  **from** *B*, *A* **show** ***B* and *A*** **by** (rule and-right)
**qed**

# Can proofs be automated?

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. $2 < 3$, $k > n$ or $k = 0$ or $[k \leq n$ 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
(`!f:real->real. ~(!x y. f(x + y * y) - f(x) >= y)`,
 REWRITE_TAC[real_ge] THEN REPEAT STRIP_TAC THEN
 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]);;
```

John Harrison

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] $f(x + n y^2) - f(x) \geq n y$ 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])

[4] Contradiction of [3] and the Archimedean property of the reals

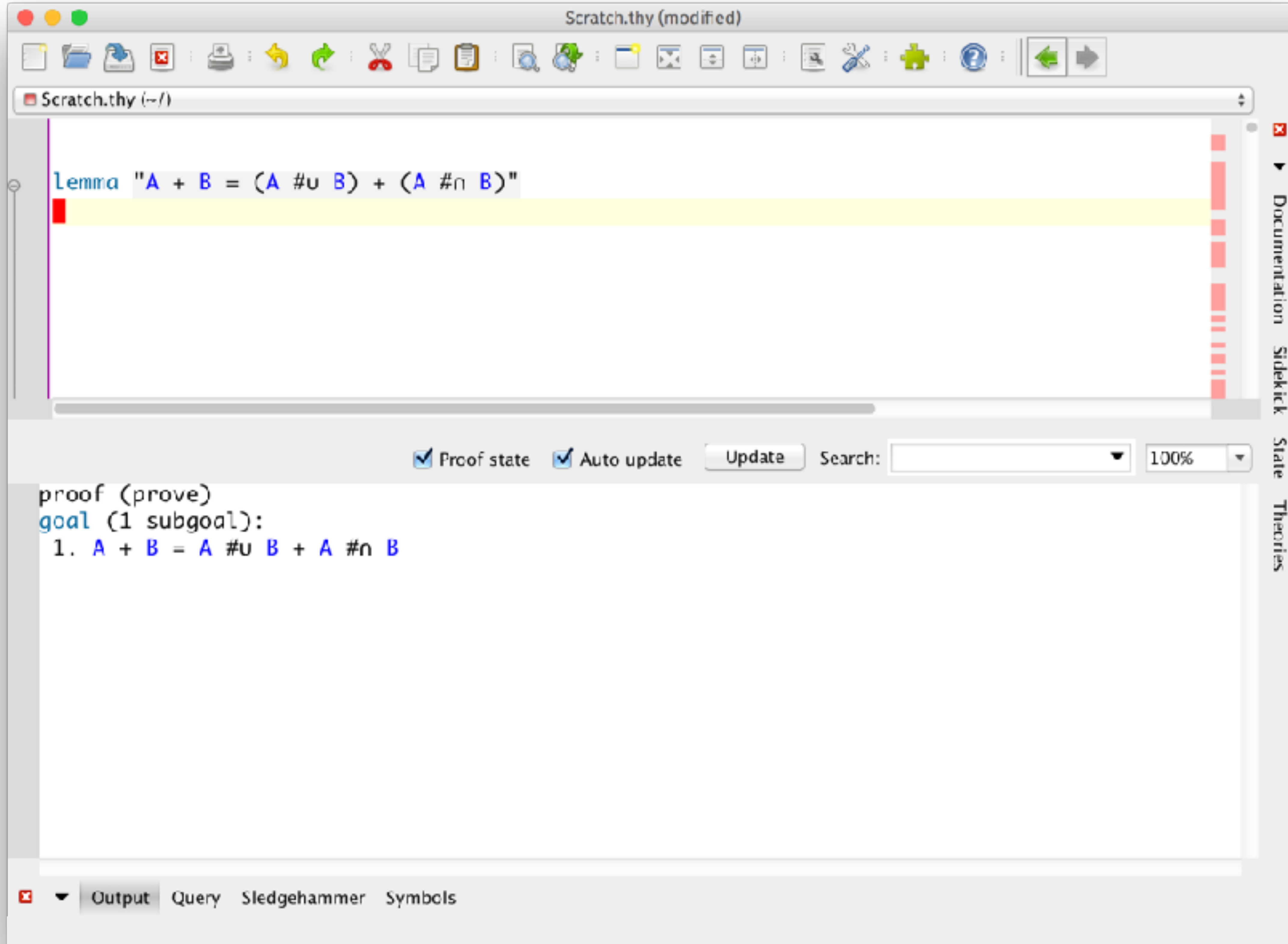
John Harrison

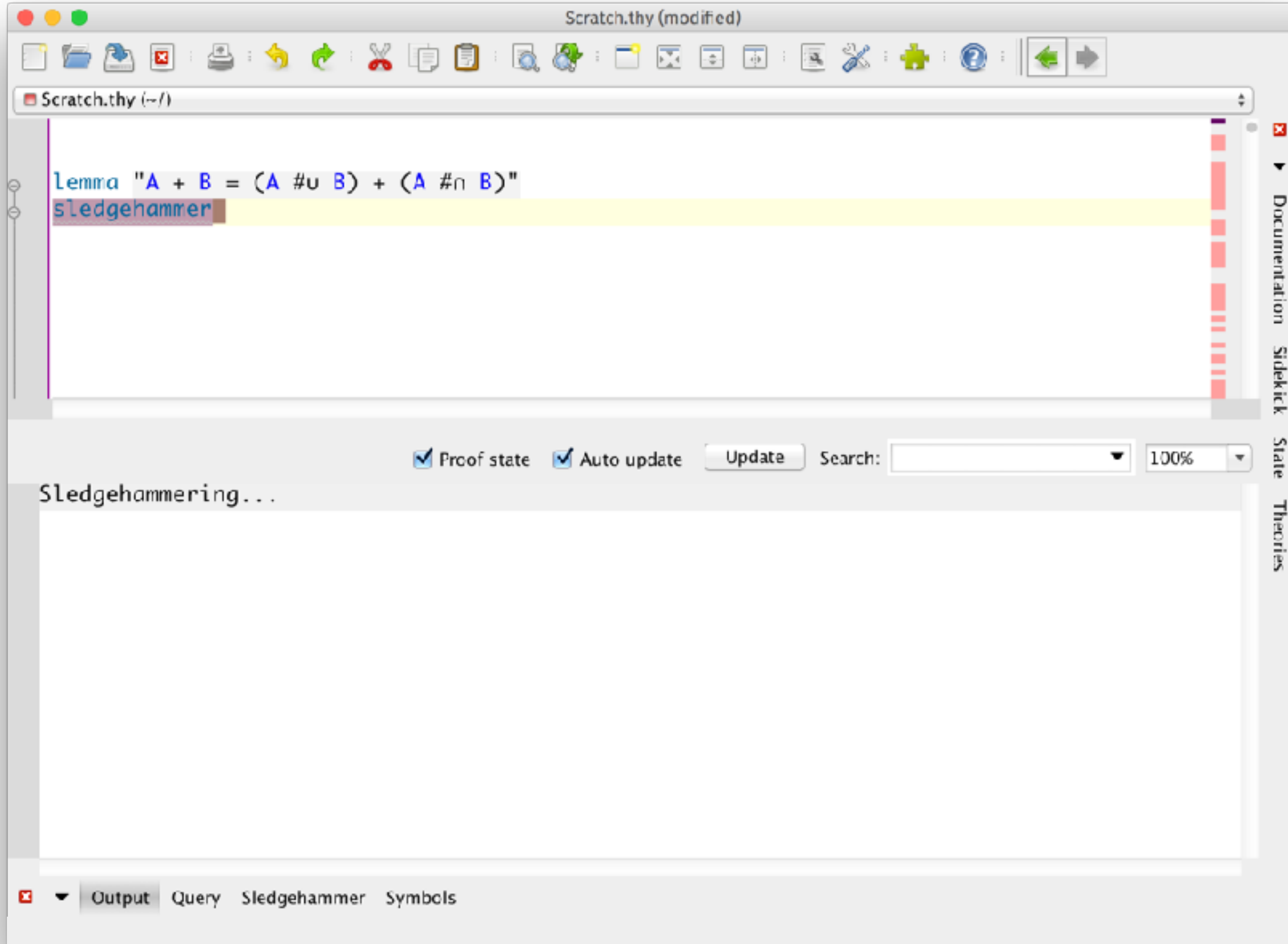
```
lemma
  shows "¬ (∃f :: real ⇒ real. ∀x y. f (x + y * y) - f x ≥ y)"
proof
  assume "∃f :: real ⇒ real. ∀x y. f (x + y * y) - f x ≥ y"
  then obtain f :: "real ⇒ real" where f: "⋀x y. f (x + y * y) - f x ≥ y"
    by blast
  have nf: "⋀(n :: nat) x y. f (x + real n * y * y) - f x ≥ real n * y"
  proof -
    fix n x y
    show "f (x + real n * y * y) - f x ≥ real n * y"
    proof (induct n)
      case 0 thus ?case by simp
    next
      case (Suc n) show ?case
      proof simp
        have "∃r. y ≤ f (y * y + (x + y * (y * real n))) - r ∧ y * real n ≤ r - f x"
          by (metis Suc.hyps add.commute f mult.commute)
      then have "y + y * real n ≤ f (y * y + (x + y * (y * real n))) - f x"
          by linarith
      then show "(1 + real n) * y ≤ f (x + (1 + real n) * y * y) - f x"
          by (simp add: add.left_commute distrib_left mult.commute)
      qed
    qed
  qed
  have min: "⋀m. f 1 - f 0 ≥ real m + 1"
  proof -
    fix m
    show "f 1 - f 0 ≥ real m + 1"
    proof -
      have "⋀r ra rb. (r :: real) / ra * rb = r * (rb / ra)"
        by simp
      then have "real (m + 1) * (real (m + 1) / real (m + 1)) ≤
          f (real (m + 1) * (real (m + 1) / (real (m + 1) * real (m + 1)))) - f 0"
        using nf[where n = "(m + 1) * (m + 1)" and x = 0 and y = "1 / (m + 1)"]
        by (metis (no_types) add.left_neutral divide_divide_eq_left mult.right_neutral of_nat_mult
          times_divide_eq_right)
      then have "real (m + 1) ≤ f 1 - f 0"
        by simp
      then show ?thesis
        by simp
    qed
  qed
  then show False
    by (metis add.commute add_le_imp_le_diff add_le_same_cancel2 add_mono diff_add_cancel
      ex_le_of_nat not_one_le_zero)
qed
```

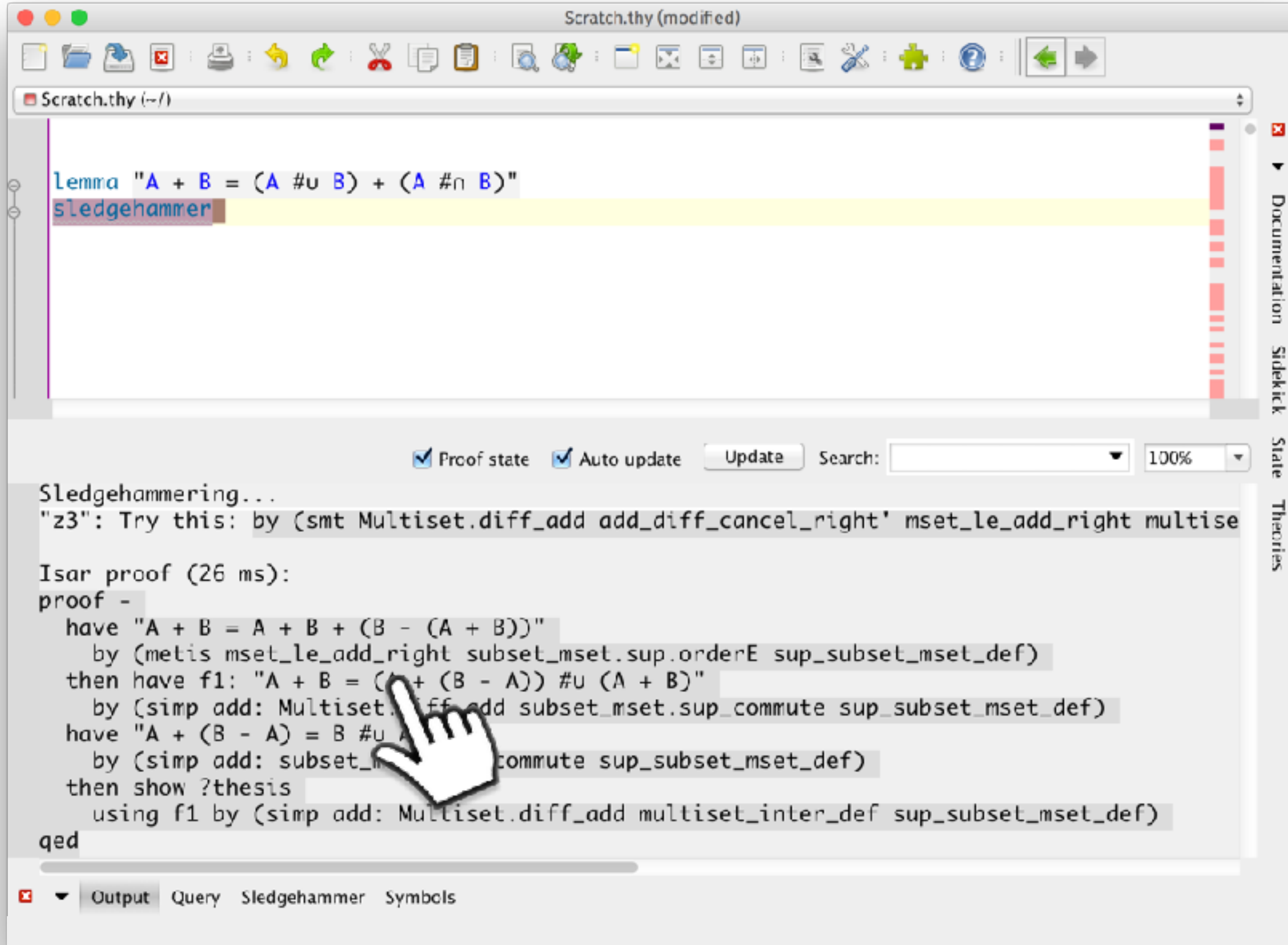
intermediate properties

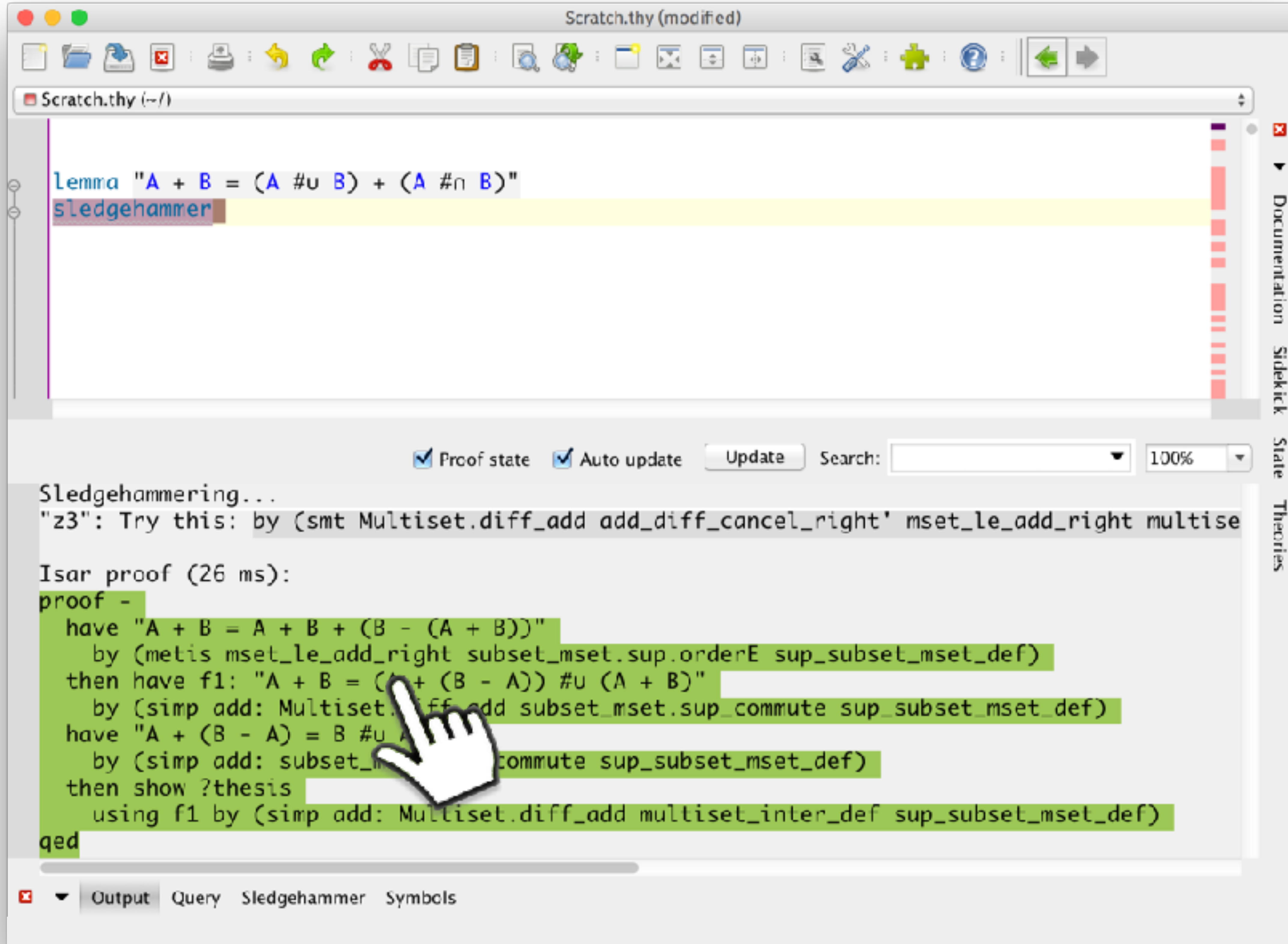
manual

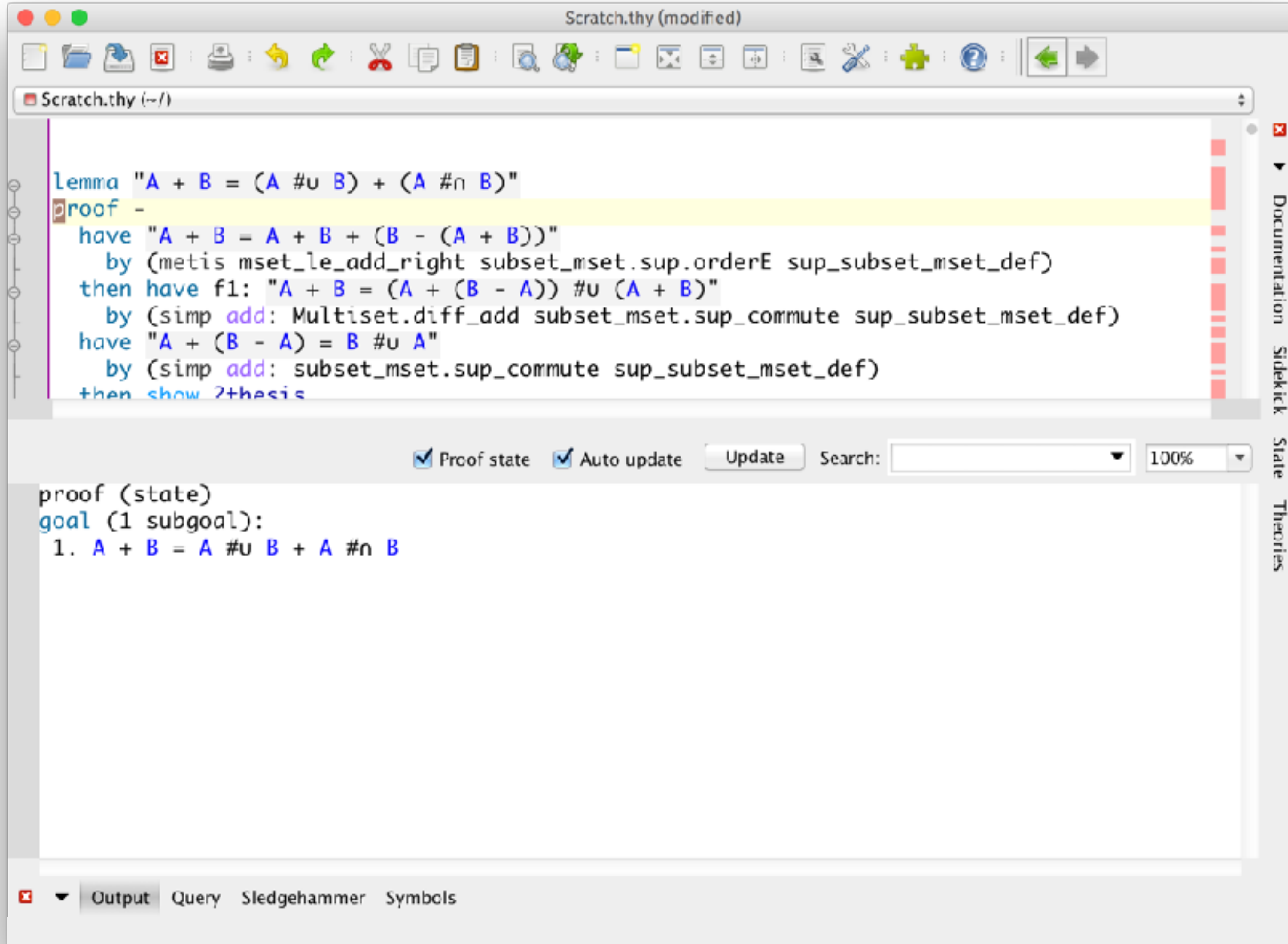
generated automatically


Scratch.thy (modified)
Scratch.thy (~/)
lemma "A + B = (A #∪ B) + (A #∩ B)"
Proof state
Auto update
Update
Search:
100%
proof (prove)
goal (1 subgoal):
1. A + B = A #∪ B + A #∩ B
Output
Query
Sledgehammer
Symbols
Documentation
Sidekick
State
Theories


Scratch.thy (modified)
Scratch.thy (~/)
lemma "A + B = (A #∪ B) + (A #∩ B)"
sledgehammer
Proof state
Auto update
Update
Search:
100%
Sledgehammering...
Output
Query
Sledgehammer
Symbols
Documentation
Sidekick
State
Theories


Scratch.thy (modified)
Scratch.thy (~/)
lemma "A + B = (A #∪ B) + (A #∩ B)"
sledgehammer
Documentation
Sidekick
State
Theories
Proof state
Auto update
Update
Search:
100%
Sledgehammering...
"z3": Try this: by (smt Multiset.diff_add add_diff_cancel_right' mset_le_add_right multise
Isar proof (26 ms):
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 f1: "A + B = (A + (B - A)) #∪ (A + B)"
by (simp add: Multiset.diff_add subset_mset.sup_commute sup_subset_mset_def)
have "A + (B - A) = B #∪ A"
by (simp add: subset_mset.sup_commute sup_subset_mset_def)
then show ?thesis
using f1 by (simp add: Multiset.diff_add multiset_inter_def sup_subset_mset_def)
qed
Output
Query
Sledgehammer
Symbols


Scratch.thy (modified)
Scratch.thy (~/)
lemma "A + B = (A #∪ B) + (A #∩ B)"
sledgehammer
Proof state
Auto update
Update
Search:
100%
Sledgehammering...
"z3": Try this: by (smt Multiset.diff_add add_diff_cancel_right' mset_le_add_right multise
Isar proof (26 ms):
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 f1: "A + B = (A + (B - A)) #∪ (A + B)"
by (simp add: Multiset.diff_add subset_mset.sup_commute sup_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
Output
Query
Sledgehammer
Symbols
Documentation
Sidekick
State
Theories


Scratch.thy (modified)
Scratch.thy (~/)
lemma "A + B = (A #∪ B) + (A #∩ 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 f1: "A + B = (A + (B - A)) #∪ (A + B)"
by (simp add: Multiset.diff_add subset_mset.sup_commute sup_subset_mset_def)
have "A + (B - A) = B #∪ A"
by (simp add: subset_mset.sup_commute sup_subset_mset_def)
Proof state
Auto update
Update
Search:
100%
proof (state)
goal (1 subgoal):
1. A + B = A #∪ B + A #∩ B
Output
Query
Sledgehammer
Symbols
Documentation
Sidekick
State
Theories

**Proof assistants**

Isabelle

well suited for
large formalizations

but

require intensive
manual labor

**Sledge-
hammer**

**Automatic provers**

SPASS

Z3

Vampire

LEO-II

E

SATALLAX

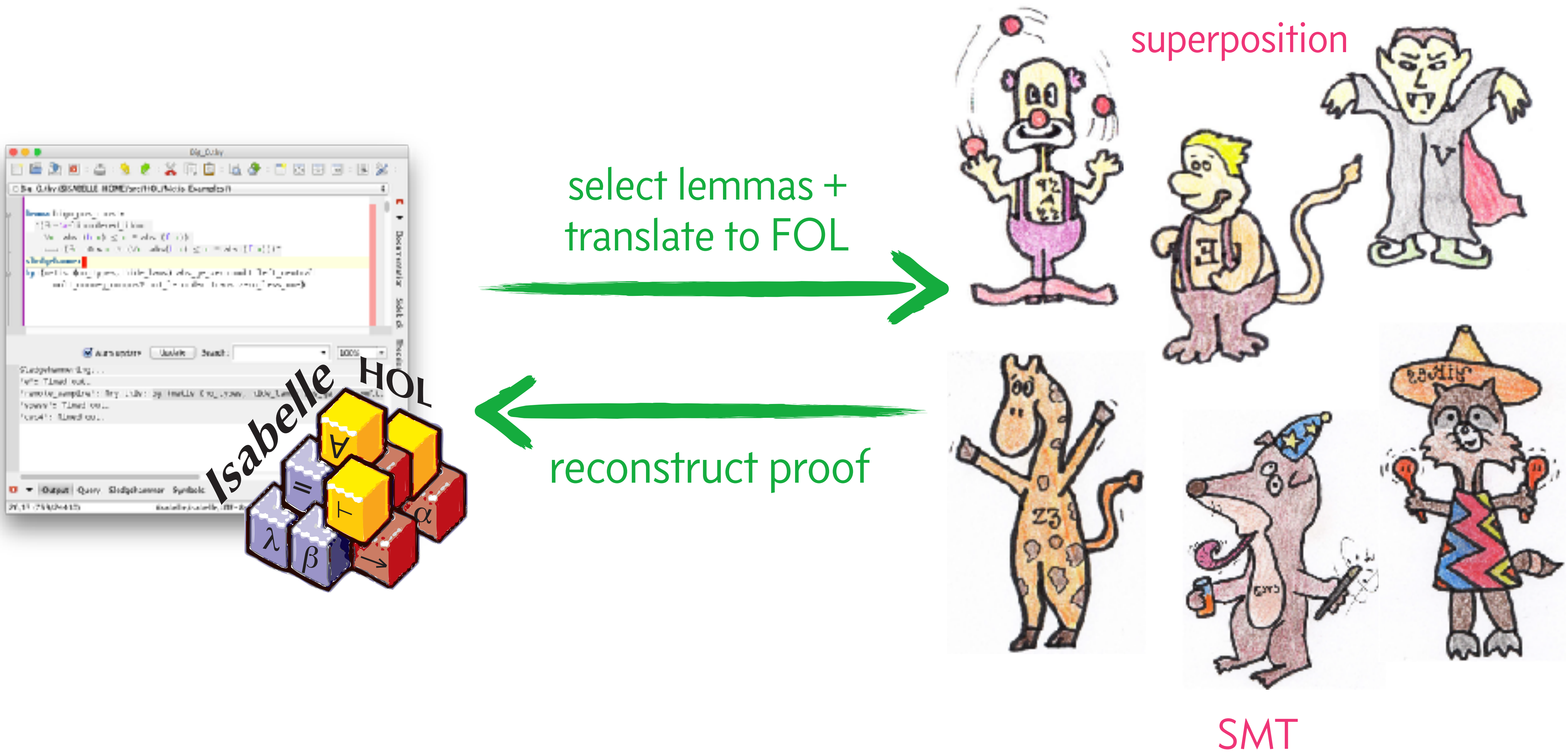
veriT

CVC4

fully automatic

but

no proof
management


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?

*pre-Sledgehammer*

Otter in ACL2
Bliksem in Coq
Gandalf in HOL98
DISCOUNT, SPASS, etc., in ILF
Otter, SPASS, etc., in KIV
LEO, SPASS, etc., in ΩMEGA
E, Vampire, etc., in Naproche
…

*post-Sledgehammer*

HOLyHammer for HOLs
MizAR for Mizar
SMTCoq/CVC4Coq for Coq
SMT integration in TLAPS
…

Developing proofs without Sledgehammer is like **walking as opposed to running**.

Tobias Nipkow

Larry Paulson

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**.

Thomas Hales

- ⊕ 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?

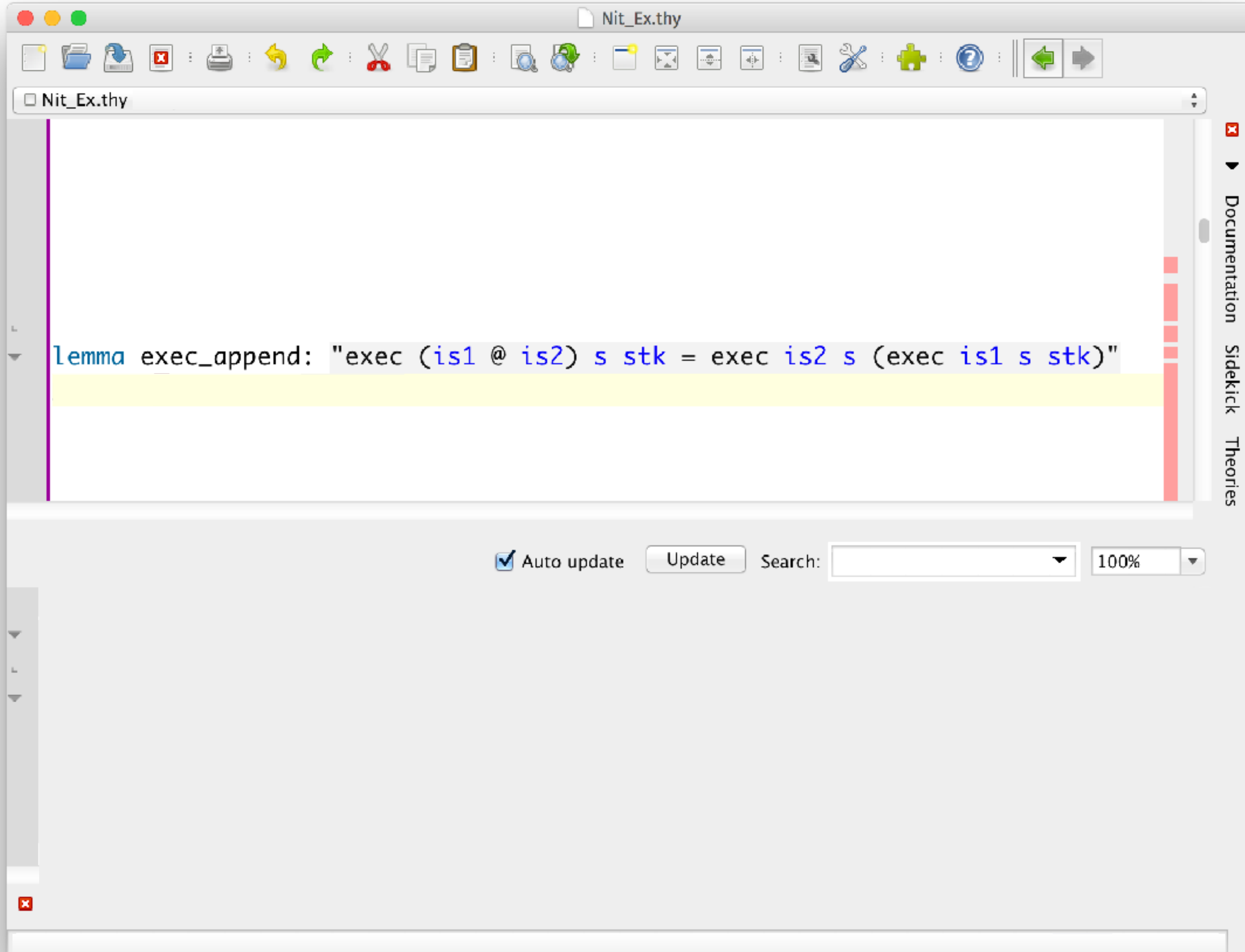
# What if the formula is wrong?

***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: $i = n = \mathbf{0}$ and $j = m = \mathbf{1}$

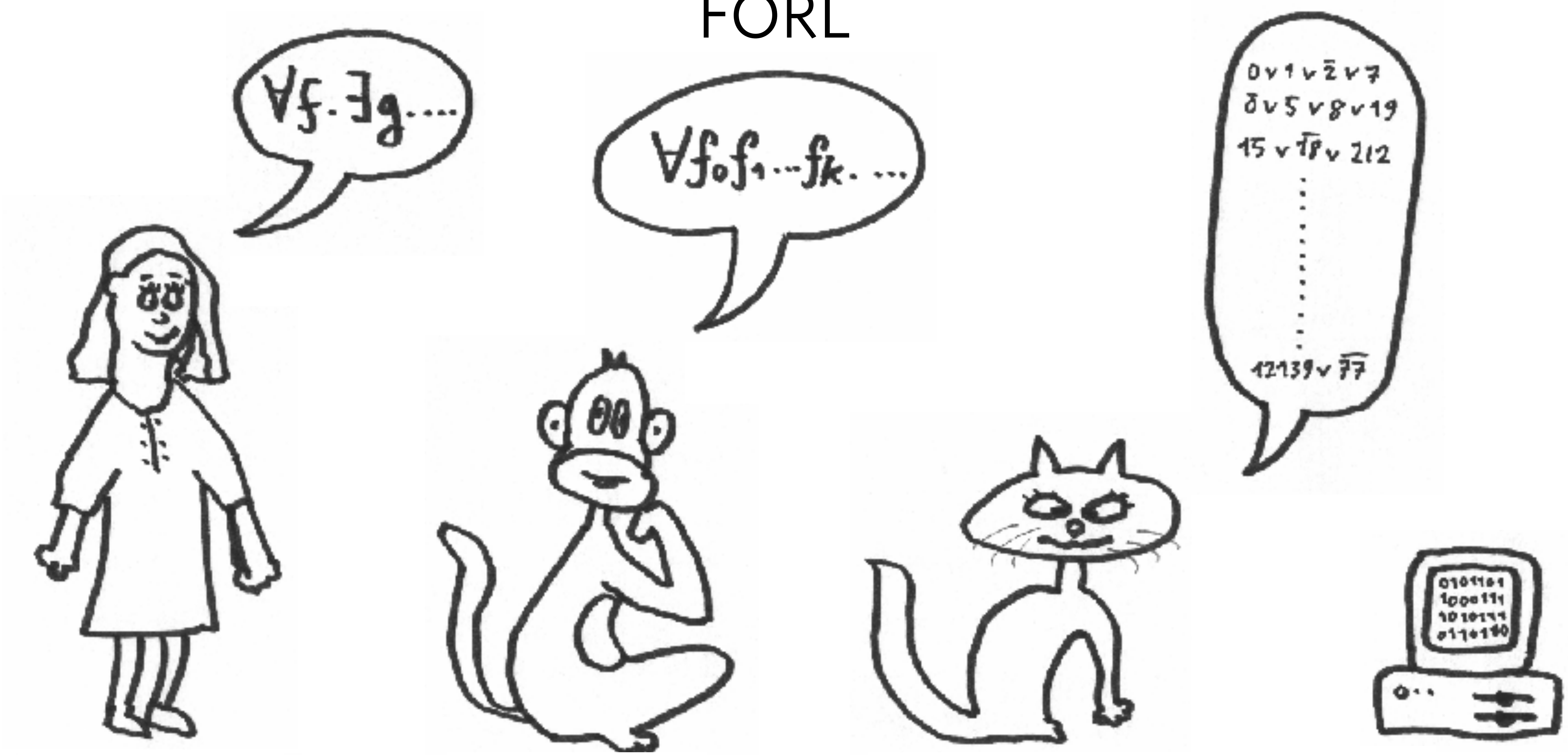

Nit_Ex.thy
Nit_Ex.thy
lemma exec_append: "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
Auto update
Update
Search:
100%
Documentation
Sidekick
Theories

# Under the hood

HOL

FORL

SAT



Isabelle

Nitpick

Kodkod

SAT solver

Isabelle HOL

# 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

| | | | |
|---|---|---|---|
| $\tau$ | $::=$ | $(\tau)$ | |
| | $\mid$ | $bool \mid nat \mid int \mid \ldots$ | 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: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

# Syntax of terms

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

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:

$$\frac{}{(\lambda x.\ t[x]) \equiv (\lambda y.\ t[y])}\ \alpha \qquad \frac{}{(\lambda x.\ t[x])\ u \equiv t[u]}\ \beta \qquad \frac{}{t^{\sigma \to \tau} \equiv (\lambda x^{\sigma}.\ t\ x)}\ \eta$$

# Notations

Implication and function arrows associate to the right:

$a \Rightarrow b \Rightarrow c$ means $a \Rightarrow (b \Rightarrow c)$

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

$$\frac{a \qquad b}{c}$$

# Typing rules

## Terms must be well typed

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

$$\frac{}{\vdash x^\sigma : \sigma} \qquad \frac{}{\vdash \mathsf{c}^\sigma : \sigma}$$

$$\frac{\vdash t : \tau}{\vdash \lambda x^\sigma .\, t : \sigma \to \tau} \qquad \frac{\vdash t : \sigma \to \tau \quad \vdash u : \sigma}{\vdash t\, u : \tau}$$

# 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:

e.g. $f\ (x :: nat)$

# Currying

"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. (op +) 1

# Metalogical propositions

Propositions have type ***prop*** (intuitionistic).

Built-in operators:

| | |
|---|---|
| $\Longrightarrow^{prop \to prop \to prop}$ | implication |
| $\bigwedge^{(\alpha \to prop) \to prop}$ | universal quantification |
| $\equiv^{\alpha \to \alpha \to prop}$ | equality |

# The HOL object logic

Propositions have type ***bool*** (classical).

The familiar operators are defined on *bool*
(False, True, =, $\forall$, $\exists$, $\neg$, $\rightarrow$, $\wedge$, $\vee$, ...).

$\longleftrightarrow$ is syntactic sugar for =.

*Trueprop* is a special implicit constant that converts a *bool* to a *prop*.

e.g. $a \wedge b \Rightarrow c$ is really

$\text{Trueprop } (a \wedge b) \Rightarrow \text{Trueprop } c$

# Predefined syntactic sugar

*Infix:* $+$, $-$, $*$, $\#$, @, ...

*Mixfix:* $if \_ \; then \_ \; else \_$, $case \_ \; of$, ...

Prefix binds more strongly than infix:

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

Enclose *if* and *case* in parentheses:

$$(if \_ \; then \_ \; else \_)$$

# Theory = Isabelle file

```
theory MyTh
imports T1 ... Tn
begin
(definitions, theorems, proofs, ...)*
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

**typedecl** ’a dlist

## (Term) Constants

**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.

```
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 *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* ...)
  ...
  **have** $P_k$ **by** (*method* ...)
  **show** Q **by** (*method* ...)
**qed**

Instead of **by**: nested proof block.

Instead of **have**: **obtain** $y_1 \ldots y_m$ **where** $P$.

Extensional equality is axiomatized, the other logical constants are definable.

$$\begin{aligned} \text{True} &:= ((\lambda x.\ x) = (\lambda x.\ x)) \\ \text{All} &:= (\lambda P.\ P = (\lambda x.\ \text{True})) \end{aligned}$$

$$\forall x.\ P\,x \ :\equiv\ \text{All}\,(\lambda x.\ P\,x)$$

# 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.