# Automation and Sledgehammer in Isabelle

Simon Foster Jim Woodcock
University of York

20th August 2022

# Overview

1. SAT solvers, resolution provers, and SMT solvers
2. Tool integration with `sledgehammer`
3. Counterexample generators

# Overview

1. SAT solvers, resolution provers, and SMT solvers
2. Tool integration with `sledgehammer`
3. Counterexample generators

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\, y.\, 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.)
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\ y.\ 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\; y.\; 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\, y.\, 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\ y.\ 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (simp)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\ y.\ 2x + 3y > n$).
- Classical Reasoner (blast, auto, force, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\; y.\; 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (simp)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\; y.\, 2x + 3y > n$).
- Classical Reasoner (blast, auto, force, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\ y.\ 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (simp)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\ y.\ 2x + 3y > n$).
- Classical Reasoner (blast, auto, force, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (simp)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\ y.\ 2x + 3y > n$).
- Classical Reasoner (blast, auto, force, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# Automated Proof so Far: Simplifier (`simp`)

- Equational rewriting of propositions ($f(x, y) = g(x, y)$).
- Very fast, and can prove many statements.
- Can't generally handle formulae with quantifiers ($\exists\, x\ y.\ 2x + 3y > n$).
- Classical Reasoner (`blast`, `auto`, `force`, etc.).
- Employs the tableau method to find natural deduction proofs.
- Slower than the simplifier, due to backtracking, but more powerful.
- Can prove many statements involving quantifiers with witnesses.
- However, these tools are often not enough, for example:
- Finding of suitable witnesses for variables in quantified formulae.
- Determining the right set of rules to use from HOL's theorem library.
- Isabelle/HOL integrates reasoning tools for constrained problems.

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :
  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CORG).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT

- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT

- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# SAT Solvers

- Determine satisfiability of proposition with only Boolean variables.
- $P \wedge \neg Q$ – satisfiable, with $P = True$ and $Q = False$ (the assignment).
- $P \wedge \neg P$ – not satisfiable, no value for $P$ that yields $True$.
- A SAT solver typically returns :

  1. SAT + an assignment or 2. UNSAT
- NP-complete problem.
- Very efficient solvers exist handling millions of variables.
- MiniSat, zChaff, PicoSAT, BerkMin, Lingeling, Glucose, SAT4J.
- Widely used in hardware verification and circuit design.
- Instance of constraint satisfaction problems (see module CONS).

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st-order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method.
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st-order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method.
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st-order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method.
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st-order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method.
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st-order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method.
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st-order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method.
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution Provers

- Resolution: deduction rule + algorithm for proving 1st-order predicates.
- Modern SAT solvers can produce a resolution-based proof of UNSAT.
- Resolution implemented in some 1st-order automated theorem provers:
- Prover9, E, Vampire, SPASS, Waldmeister, and the `metis` proof method.
- Clause normal form represents knowledge: conjunction of disjunctions.
- Inherently classical in nature, as it depends on proof by contradiction.

# Resolution

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions.
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions.
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions.
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions:
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions:
$$\frac{\neg P \vee Q \qquad P}{Q}$$
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions:
$$\frac{\neg P \vee Q \qquad P}{Q}$$
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions:
$$\frac{\neg P \vee Q \qquad P}{Q}$$
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions:
$$\frac{\neg P \vee Q \qquad P}{Q}$$
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution

## Algorithm (Outline)

1. Hypotheses $A_1 \cdots A_n$ and goal $G$: produce $A_1 \wedge \cdots \wedge A_n \wedge \neg G$.
2. Rewrite $A_i$ and $G$ into CNF: $P \wedge (P \longrightarrow Q)$ becomes $P \wedge (\neg P \vee Q)$.
3. Apply the resolution rule over and over to make all possible deductions:
$$\frac{\neg P \vee Q \qquad P}{Q}$$
4. If a contradiction emerges, then $G$ must follow from the assumptions.

- `metis` converts HOL proposition into required form, applies resolution.
- Requires all background theorems needed to be passed as hypotheses.
- If a refutation of $\neg G$ emerges, this is translated to a HOL theorem.

# Resolution Example

- We want to prove $man(S), \forall x.\ man(x) \longrightarrow mortal(x) \vdash mortal(S)$.
- Rewrite to CNF gives

  $A_1 = man(S),\ A_2 = \neg man(x) \vee mortal(x),\ C = mortal(S)$.
- We need to show $A_1 \wedge A_2 \wedge \neg C$ yields a contradiction.
- Resolving $A_1$ and $A_2$ yields the additional clause $A_3 = mortal(S)$.
- Resolving $A_3$ and $C$ yields the empty set, thus the proof is complete.

# Resolution Example

- We want to prove $man(S), \forall\, x.\, man(x) \longrightarrow mortal(x) \vdash mortal(S)$.
- Rewrite to CNF gives

$$A_1 = man(S),\ A_2 = \neg man(x) \vee mortal(x),\ G = mortal(S).$$

- We need to show $A_1 \wedge A_2 \wedge \neg G$ yields a contradiction.
- Resolving $A_1$ and $A_2$ yields the additional clause $A_3 = mortal(S)$.
- Resolving $A_3$ and $G$ yields the empty set, thus the proof is complete.

# Resolution Example

- We want to prove $man(S), \forall\, x.\, man(x) \longrightarrow mortal(x) \vdash mortal(S)$.
- Rewrite to CNF gives

$$A_1 = man(S),\ A_2 = \neg man(x) \lor mortal(x),\ G = mortal(S).$$

- We need to show $A_1 \land A_2 \land \neg G$ yields a contradiction.
- Resolving $A_1$ and $A_2$ yields the additional clause $A_3 = mortal(S)$.
- Resolving $A_3$ and $G$ yields the empty set, thus the proof is complete.

# Resolution Example

- We want to prove $man(S), \forall\, x.\, man(x) \longrightarrow mortal(x) \vdash mortal(S)$.
- Rewrite to CNF gives

  $$A_1 = man(S),\ A_2 = \neg man(x) \vee mortal(x),\ G = mortal(S).$$

- We need to show $A_1 \wedge A_2 \wedge \neg G$ yields a contradiction.
- Resolving $A_1$ and $A_2$ yields the additional clause $A_3 = mortal(S)$.
- Resolving $A_3$ and $G$ yields the empty set, thus the proof is complete.

# Resolution Example

- We want to prove $man(S), \forall\, x.\, man(x) \longrightarrow mortal(x) \vdash mortal(S)$.
- Rewrite to CNF gives

  $A_1 = man(S)$, $A_2 = \neg man(x) \vee mortal(x)$, $G = mortal(S)$.

- We need to show $A_1 \wedge A_2 \wedge \neg G$ yields a contradiction.
- Resolving $A_1$ and $A_2$ yields the additional clause $A_3 = mortal(S)$.
- Resolving $A_3$ and $G$ yields the empty set, thus the proof is complete.

# Resolution Example

- We want to prove $man(S), \forall\, x.\, man(x) \longrightarrow mortal(x) \vdash mortal(S)$.
- Rewrite to CNF gives

$$A_1 = man(S),\ A_2 = \neg man(x) \vee mortal(x),\ G = mortal(S).$$

- We need to show $A_1 \wedge A_2 \wedge \neg G$ yields a contradiction.
- Resolving $A_1$ and $A_2$ yields the additional clause $A_3 = mortal(S)$.
- Resolving $A_3$ and $G$ yields the empty set, thus the proof is complete.

# Resolution Example

- We want to prove $man(S), \forall\, x.\, man(x) \longrightarrow mortal(x) \vdash mortal(S)$.
- Rewrite to CNF gives

  $$A_1 = man(S),\ A_2 = \neg man(x) \vee mortal(x),\ G = mortal(S).$$

- We need to show $A_1 \wedge A_2 \wedge \neg G$ yields a contradiction.
- Resolving $A_1$ and $A_2$ yields the additional clause $A_3 = mortal(S)$.
- Resolving $A_3$ and $G$ yields the empty set, thus the proof is complete.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures:
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the smt proof method
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# SMT Solvers

- SMT = Satisfiability Modulo Theories.
- Extends SAT with further types and decision procedures.
- Quantifiers, linear arithmetic, bit vectors, arrays, datatypes, records, etc.
- UNSAT: proof term often returned using accompanying ATP.
- More readily applicable to program verification.
- Examples: Z3, Yices, CVC3, CVC4, veriT.
- Z3 is the backend for Microsoft's Boogie verification language.
- Isabelle/HOL integrates several SMT solvers in the `smt` proof method.
- Reconstructs proof terms output from CVC4, veriT, and Z3.

# Arithmetic Decision Procedures

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\; y : \mathbb{N}.\, 2x + 5y = 3)$ is solvable with `presburger`.

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\ y : \mathbb{N}.\ 2x + 5y = 3)$ is solvable with `presburger`.

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\ y : \mathbb{N}.\ 2x + 5y = 3)$ is solvable with `presburger`.

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\ y : \mathbb{N}.\ 2x + 5y = 3)$ is solvable with `presburger`.

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\ y : \mathbb{N}.\, 2x + 5y = 3)$ is solvable with `presburger`.

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\ y : \mathbb{N}.\ 2x + 5y = 3)$ is solvable with `presburger`.

## Linear Arithmetic (`linarith`)

- Proof method for systems of linear inequalities, e.g. $a \cdot x + b \cdot y \leq c$.
- Employs an algorithm called Fourier-Motzkin elimination.

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\ y : \mathbb{N}.\, 2x + 5y = 3)$ is solvable with `presburger`.

## Linear Arithmetic (`linarith`)

- Proof method for systems of linear inequalities, e.g. $a \cdot x + b \cdot y \leq c$.
- Employs an algorithm called Fourier-Motzkin elimination.

- `arith` tries both `presburger` and/or `linarith`.

# Arithmetic Decision Procedures

## Presburger Arithmetic (presburger)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- presburger handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\; y : \mathbb{N}.\, 2x + 5y = 3)$ is solvable with presburger.

## Linear Arithmetic (linarith)

- Proof method for systems of linear inequalities, e.g. $a \cdot x + b \cdot y \leq c$.
- Employs an algorithm called Fourier-Motzkin elimination.

- arith tries both presburger and/or linarith.

# Arithmetic Decision Procedures

## Presburger Arithmetic (`presburger`)

- Arithmetic formulae involving only $0$, $1$, and $+$, and equality.
- Decidable with exponential complexity.
- `presburger` handles quantifiers using quantifier elimination (QE).
- E.g. $\neg(\exists\, x\ y : \mathbb{N}.\ 2x + 5y = 3)$ is solvable with `presburger`.

## Linear Arithmetic (`linarith`)

- Proof method for systems of linear inequalities, e.g. $a \cdot x + b \cdot y \leq c$.
- Employs an algorithm called Fourier-Motzkin elimination.

- `arith` tries both `presburger` and/or `linarith`.

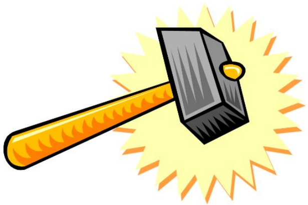
# Overview

1. SAT solvers, resolution provers, and SMT solvers
2. Tool integration with `sledgehammer`
3. Counterexample generators

# Sledgehammer

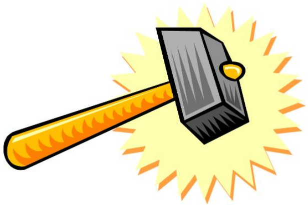
- Integration of resolution provers and SMT solvers into Isabelle/HOL.
- Use the “Sledgehammer” pane to apply the tool on an open subgoal.
- Runs several tools in parallel, reconstructs output as Isabelle proof.
- Brings benefits of proof automation to interactive theorem proving.
- Does so in a safe way – there is no need to trust the external tools.

# Sledgehammer



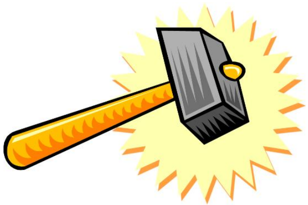
- Integration of resolution provers and SMT solvers into Isabelle/HOL.
- Use the “Sledgehammer” pane to apply the tool on an open subgoal.
- Runs several tools in parallel, reconstructs output as Isabelle proof.
- Brings benefits of proof automation to interactive theorem proving.
- Does so in a safe way – there is no need to trust the external tools.

# Sledgehammer



- Integration of resolution provers and SMT solvers into Isabelle/HOL.
- Use the "Sledgehammer" pane to apply the tool on an open subgoal.
- Runs several tools in parallel, reconstructs output as Isabelle proof.
- Brings benefits of proof automation to interactive theorem proving.
- Does so in a safe way – there is no need to trust the external tools.

# Sledgehammer

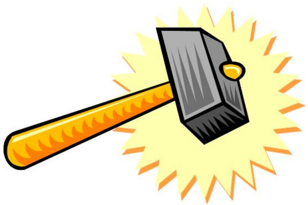


- Integration of resolution provers and SMT solvers into Isabelle/HOL.
- Use the "Sledgehammer" pane to apply the tool on an open subgoal.
- Runs several tools in parallel, reconstructs output as Isabelle proof.
- Brings benefits of proof automation to interactive theorem proving.
- Does so in a safe way – there is no need to trust the external tools.

# Sledgehammer

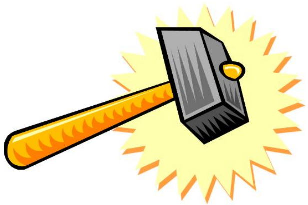
- Integration of resolution provers and SMT solvers into Isabelle/HOL.
- Use the “Sledgehammer” pane to apply the tool on an open subgoal.
- Runs several tools in parallel, reconstructs output as Isabelle proof.
- Brings benefits of proof automation to interactive theorem proving.
- Does so in a safe way – there is no need to trust the external tools.

# Sledgehammer



- Integration of resolution provers and SMT solvers into Isabelle/HOL.
- Use the “Sledgehammer” pane to apply the tool on an open subgoal.
- Runs several tools in parallel, reconstructs output as Isabelle proof.
- Brings benefits of proof automation to interactive theorem proving.
- Does so in a safe way – there is no need to trust the external tools.

# Sledgehammer



- Integration of resolution provers and SMT solvers into Isabelle/HOL.
- Use the "Sledgehammer" pane to apply the tool on an open subgoal.
- Runs several tools in parallel, reconstructs output as Isabelle proof.
- Brings benefits of proof automation to interactive theorem proving.
- Does so in a safe way – there is no need to trust the external tools.

# Sledgehammer Workflow

1. Formulate a theorem we want to prove:

```
lemma "(A < B)= (A !≠! B !∧! (!∀! x!∈!A. x !∈! B))"
```

2. Run sledgehammer with set of provers and solvers (click "Apply").

3. Click on one of the options returned to prove the theorem:

# Sledgehammer Workflow

1. Formulate a theorem we want to prove:

   ```
   lemma "(A < B)= (A !≠! B !∧! (!∀! x!∈!A. x !∈! B))"
   ```

2. Run sledgehammer with set of provers and solvers (click “Apply”).

3. Click on one of the options returned to prove the theorem:

# Sledgehammer Workflow

1. Formulate a theorem we want to prove:

```
lemma "(A < B)= (A !≠! B !∧! (!∀! x!∈!A. x !∈! B))"
```

2. Run sledgehammer with set of provers and solvers (click “Apply”).

3. Click on one of the options returned to prove the theorem:

# Sledgehammer Workflow

1. Formulate a theorem we want to prove:

   ```
   lemma "(A < B)= (A !≠! B !∧! (!∀! x!∈!A. x !∈! B))"
   ```

2. Run sledgehammer with set of provers and solvers (click “Apply”).

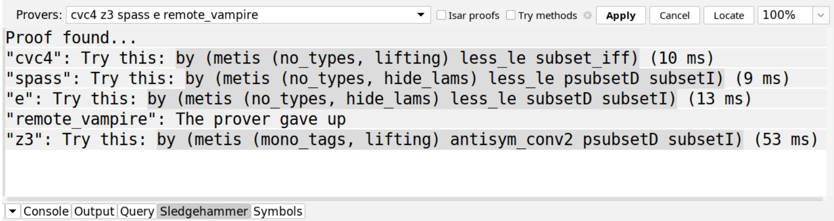
3. Click on one of the options returned to prove the theorem:

# Sledgehammer Workflow

1. Formulate a theorem we want to prove:

   ```
   lemma "(A < B)= (A !≠! B !∧! (!∀! x!∈!A. x !∈! B))"
   ```

2. Run sledgehammer with set of provers and solvers (click “Apply”).



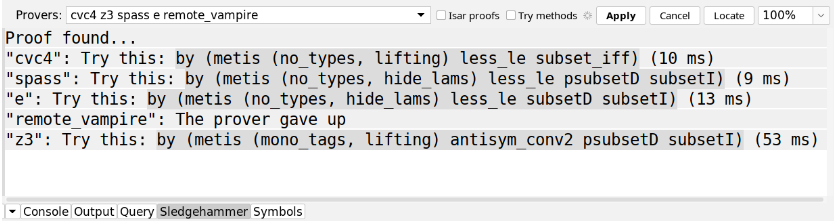

3. Click on one of the options returned to prove the theorem:

# Sledgehammer Workflow

1. Formulate a theorem we want to prove:

   ```
   lemma "(A < B)= (A !≠! B !∧! (!∀! x!∈!A. x !∈! B))"
   ```

2. Run sledgehammer with set of provers and solvers (click “Apply”).



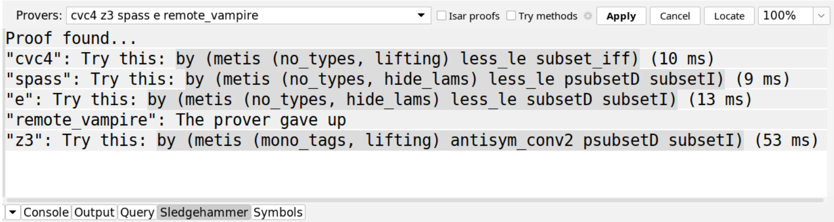

3. Click on one of the options returned to prove the theorem:

# Sledgehammer Workflow

1. Formulate a theorem we want to prove:

```
lemma "(A < B)= (A !≠! B !∧! (!∀! x!∈!A. x !∈! B))"
```

2. Run sledgehammer with set of provers and solvers (click “Apply”).




3. Click on one of the options returned to prove the theorem:

```
lemma "(A < B) = (A ≠ B ∧ (∀ x∈A. x ∈ B))"
  by (metis (no_types, lifting) less_le subset_iff)
```

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

## Proof Reconstruction Tactics

- `metis`: built-in resolution prover. Solves goal with small theorem set.
- `smt`: converts proof objects from Z3, CVC4, veriT into Isabelle proofs.
- `presburger` and `linarith`: solve arithmetic conjectures.
- `simp` and `fastforce`: sometimes normal proof methods are used.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

## Proof Reconstruction Tactics

- `metis`: built-in resolution prover. Solves goal with small theorem set.
- `smt`: converts proof objects from Z3, CVC4, veriT into Isabelle proofs.
- `presburger` and `linarith`: solve arithmetic conjectures.
- `simp` and `fastforce`: sometimes normal proof methods are used.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

## Proof Reconstruction Tactics

- `metis`: built-in resolution prover. Solves goal with small theorem set.
- `smt`: converts proof objects from Z3, CVC4, veriT into Isabelle proofs.
- `presburger` and `linarith`: solve arithmetic conjectures.
- `simp` and `fastforce`: sometimes normal proof methods are used.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

## Proof Reconstruction Tactics

- `metis`: built-in resolution prover. Solves goal with small theorem set.
- `smt`: converts proof objects from Z3, CVC4, veriT into Isabelle proofs.
- `presburger` and `linarith`: solve arithmetic conjectures.
- `simp` and `fastforce`: sometimes normal proof methods are used.

# How Sledgehammer Works

1. Find background theorems important for proof with relevance filtering.
2. Convert goal, assumptions, theorems into input format for proof tools.
3. Execute the specified tools, possibly in parallel and remotely.
4. For tools returning a solution, optimise proof by minimising theorem set.
5. Reconstruct and check proofs in Isabelle/HOL. Ensures soundness.

## Proof Reconstruction Tactics

- `metis`: built-in resolution prover. Solves goal with small theorem set.
- `smt`: converts proof objects from Z3, CVC4, veriT into Isabelle proofs.
- `presburger` and `linarith`: solve arithmetic conjectures.
- `simp` and `fastforce`: sometimes normal proof methods are used.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

## Typical Proof Procedure

1. Use high-level proof pattern, like structural induction or case analysis.
2. Use `simp` and `auto` to breakdown resulting subgoals.
3. Apply `sledgehammer` to residual proof obligations.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

## Typical Proof Procedure

1. Use high-level proof pattern, like structural induction or case analysis.
2. Use `simp` and `auto` to breakdown resulting subgoals.
3. Apply `sledgehammer` to residual proof obligations.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

## Typical Proof Procedure

1. Use high-level proof pattern, like structural induction or case analysis.
2. Use `simp` and `auto` to breakdown resulting subgoals.
3. Apply `sledgehammer` to residual proof obligations.

# Sledgehammer Applications

- Best suited to first-order, algebraic, and equational proof obligations.
- `sledgehammer` doesn't handle higher-order techniques like induction.
- Useful for finding suitable lemmas for proofs.
- `Sledgehammer` is invaluable discharging verification conditions.

**Typical Proof Procedure**

1. Use high-level proof pattern, like structural induction or case analysis.
2. Use `simp` and `auto` to breakdown resulting subgoals.
3. Apply `sledgehammer` to residual proof obligations.

# Sledgehammer Examples (1)

# Sledgehammer Examples (1)

```
lemma rat_prop:
  fixes x :: rat
  shows "x² - 3*x + 2 < 0 ⟶ x > 0"
  by (metis add_less_zeroD add_neg_neg
    diff_add_cancel less_iff_diff_less_0
    mult_less_cancel_right_disj
    not_numeral_less_zero power2_eq_square)
```

# Sledgehammer Examples (1)

```
lemma rat_prop:
  fixes x :: rat
  shows "x² - 3*x + 2 < 0 ⟶ x > 0"
  by (metis add_less_zeroD add_neg_neg
    diff_add_cancel less_iff_diff_less_0
    mult_less_cancel_right_disj
    not_numeral_less_zero power2_eq_square)

lemma tl_element:
  assumes "x ∈ set xs" "x ≠ hd(xs)"
  shows "x ∈ set(tl(xs))"
  by (metis assms(1) assms(2) list.exhaust_sel
     list.sel(2) set_ConsD)
```

# Sledgehammer Examples (2)

# Sledgehammer Examples (2)

```
lemma sorted_distinct:
  assumes "sorted xs" "distinct xs"
  shows "(∀ i<length xs - 1. xs i < xs(i + 1))"
using assms proof (induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case
    by (simp, metis Suc_leI Suc_le_lessD diff_less
      less_nat_zero_code linorder_le_less_linear
      not_one_le_zero nth_Cons' nth_Cons_Suc
      nth_equal_first_eq order_less_le sorted_wrt_nth_less
      strict_sorted_iff)
qed
```

# Overview

1. SAT solvers, resolution provers, and SMT solvers
2. Tool integration with `sledgehammer`
3. Counterexample generators

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem:

```
lemma wrong: "(x::nat) > 5"
```

- This states that all natural numbers are greater than 5. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true.
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem.

```
lemma wrong:"(x::nat)> 5"
```

- This states that all natural numbers are greater than $5$. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these.

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true.
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem.

```
lemma wrong:"(x::nat)> 5"
```

- This states that all natural numbers are greater than $5$. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these.

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true.
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem.

```
lemma wrong:"(x::nat)> 5"
```

- This states that all natural numbers are greater than 5. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these.

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true.
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem.

```
lemma wrong:"(x::nat)> 5"
```

- This states that all natural numbers are greater than 5. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these.

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true.
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem.

```
lemma wrong:"(x::nat)> 5"
```

- This states that all natural numbers are greater than $5$. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these.

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true.
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem.

```
lemma wrong:"(x::nat)> 5"
```

- This states that all natural numbers are greater than $5$. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these.

# Counterexamples

- Automated theorem provers may fail to prove a goal you believe is true.
- Could be (1) some lemmas are missing or (2) the goal is actually false.
- A counterexample is a variable assignment that falsifies a theorem.

```
lemma wrong:"(x::nat)> 5"
```

- This states that all natural numbers are greater than $5$. Not provable.
- Possible counterexamples are $x = 0$, $x = 1$, $x = 2$ etc.
- Counterexample generators automatically generate these.

# Quickcheck

- Generates counterexamples using the code generator
- Inspired by the Haskell random testing tool QuickCheck
- Tests randomly, exhaustively (up to a bound), or symbolically
- Run automatically when theorem is specified, or by [illegible]

```
lemma wrong: "(x::nat) > 5"
```

- `Auto Quickcheck found a counterexample: x = 0`
- [illegible] quick debugger for theorem specifications

# Quickcheck

- Generates counterexamples using the code generator.
- Inspired by the Haskell random testing tool QuickCheck.
- Tests randomly, exhaustively (up to a bound), or symbolically.
- Run automatically when theorem is specified, or by `quickcheck`.

```
lemma wrong:"(x::nat)> 5"
```

- `Auto Quickcheck found a counterexample: x = 0`.
- `quickcheck`: quick debugger for theorem specifications.

# Quickcheck

- Generates counterexamples using the code generator.
- Inspired by the Haskell random testing tool QuickCheck.
- Tests randomly, exhaustively (up to a bound), or symbolically.
- Run automatically when theorem is specified, or by `quickcheck`.

```
lemma wrong:"(x::nat)> 5"
```

- `Auto Quickcheck found a counterexample: x = 0.`
- `quickcheck`: quick debugger for theorem specifications.

# Quickcheck

- Generates counterexamples using the code generator.
- Inspired by the Haskell random testing tool QuickCheck.
- Tests randomly, exhaustively (up to a bound), or symbolically.
- Run automatically when theorem is specified, or by `quickcheck`.

```
lemma wrong:"(x::nat)> 5"
```

- `Auto Quickcheck found a counterexample: x = 0.`
- `quickcheck`: quick debugger for theorem specifications.

# Quickcheck

- Generates counterexamples using the code generator.
- Inspired by the Haskell random testing tool QuickCheck.
- Tests randomly, exhaustively (up to a bound), or symbolically.
- Run automatically when theorem is specified, or by **quickcheck**.

```
lemma wrong:"(x::nat)> 5"
```

- Auto Quickcheck found a counterexample: x = 0.
- **quickcheck**: quick debugger for theorem specifications.

# Quickcheck

- Generates counterexamples using the code generator.
- Inspired by the Haskell random testing tool QuickCheck.
- Tests randomly, exhaustively (up to a bound), or symbolically.
- Run automatically when theorem is specified, or by **quickcheck**.

  **lemma** wrong:"(x::nat)> 5"
- Auto Quickcheck found a counterexample: x = 0.
- **quickcheck**: quick debugger for theorem specifications.

# Quickcheck

- Generates counterexamples using the code generator.
- Inspired by the Haskell random testing tool QuickCheck.
- Tests randomly, exhaustively (up to a bound), or symbolically.
- Run automatically when theorem is specified, or by **quickcheck**.

```
lemma wrong:"(x::nat)> 5"
```

- `Auto Quickcheck found a counterexample: x = 0`.
- **quickcheck**: quick debugger for theorem specifications.

# Quickcheck

- Generates counterexamples using the code generator.
- Inspired by the Haskell random testing tool QuickCheck.
- Tests randomly, exhaustively (up to a bound), or symbolically.
- Run automatically when theorem is specified, or by **quickcheck**.

```
lemma wrong:"(x::nat)> 5"
```

- `Auto Quickcheck found a counterexample: x = 0`.
- **quickcheck**: quick debugger for theorem specifications.

# Quickcheck and Lists

- This theorem is not correct – we forgot to reorder xs and ys.
- [illegible] quickly finds the following counterexample:

# Quickcheck and Lists

- This theorem is not correct – we forgot to reorder `xs` and `ys`.
- `quickcheck` quickly finds the following counterexample:

# Quickcheck and Lists

- This theorem is not correct – we forgot to reorder `xs` and `ys`.

```
lemma rev_app: "rev (xs @ ys) = rev xs @ rev ys"
```

- `quickcheck` quickly finds the following counterexample:

# Quickcheck and Lists

- This theorem is not correct – we forgot to reorder `xs` and `ys`.

  ```
  lemma rev_app: "rev (xs @ ys) = rev xs @ rev ys"
  ```

- **quickcheck** quickly finds the following counterexample:

# Quickcheck and Lists

- This theorem is not correct – we forgot to reorder `xs` and `ys`.

```
lemma rev_app: "rev (xs @ ys) = rev xs @ rev ys"
```

- **quickcheck** quickly finds the following counterexample:

```
Auto Quickcheck found a counterexample:
  xs = [a]
  ys = [b]
Evaluated terms:
  rev (xs @ ys) = [b, a]
  rev xs @ rev ys = [a, b]
```

# Constraint Solving with Quickcheck

- No list has five distinct elements that are all even:
- `quickcheck` finds the following counterexample:

  `xs = [8, 6, 4, 2, 0]`.

# Constraint Solving with Quickcheck

- No list has five distinct elements that are all even:
- quickcheck finds the following counterexample:

  xs = [8, 6, 4, 2, 0].

# Constraint Solving with Quickcheck

- No list has five distinct elements that are all even:

```
lemma list_constraint:
  fixes xs :: "nat list"
  shows "¬ (length xs = 5 ∧ distinct xs
         ∧ (∀ i<length xs. even (xs!i)))"

  quickcheck [tester=narrowing, size=100]
```

- `quickcheck` finds the following counterexample:

  `xs = [8, 6, 4, 2, 0]`.

# Constraint Solving with Quickcheck

- No list has five distinct elements that are all even:

```
lemma list_constraint:
  fixes xs :: "nat list"
  shows "¬ (length xs = 5 ∧ distinct xs
          ∧ (∀ i<length xs. even (xs!i)))"

  quickcheck [tester=narrowing, size=100]
```

- **quickcheck** finds the following counterexample:

```
xs = [8, 6, 4, 2, 0].
```

# Constraint Solving with Quickcheck

- No list has five distinct elements that are all even:

```
lemma list_constraint:
  fixes xs :: "nat list"
  shows "¬ (length xs = 5 ∧ distinct xs
          ∧ (∀ i<length xs. even (xs!i)))"

  quickcheck [tester=narrowing, size=100]
```

- `quickcheck` finds the following counterexample:

  `xs = [8, 6, 4, 2, 0]`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.
- More suited to set theoretic problems than `quickcheck`.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.

```
lemma "(x::nat) > 5" nitpick
(* Nitpick found a counterexample: x = 5 *)
```

- More suited to set theoretic problems than quickcheck.

# Nitpick

- Counterexample generator based on a SAT solver.
- Negate the theorem, and pass to the Kodkod relational constraint solver.
- KodKod successfully applied to software verification.
- Example: Hotel Key Card problem.
- Tries to find a finite model that falsifies the theorem.
- Converts the constraint problem into a SAT solver.
- Tries SAT4J, MiniSat to solve the problem.

```
lemma "(x::nat) > 5" nitpick
(* Nitpick found a counterexample: x = 5 *)
```

- More suited to set theoretic problems than **quickcheck**.

# Conclusion

### Next Lecture

- Formal specification of railway signalling equipment

# Conclusion

### This Lecture

- Overview of automated reasoning in Isabelle/HOL.
- SAT solvers, resolution provers, and SMT solvers.
- Tool integration with `sledgehammer`.
- Counterexample generators.

### Next Lecture

- Formal specification of railway signalling equipment.

# Conclusion

## This Lecture

- Overview of automated reasoning in Isabelle/HOL.
- SAT solvers, resolution provers, and SMT solvers.
- Tool integration with `sledgehammer`.
- Counterexample generators.

## Next Lecture

- Formal specification of railway signalling equipment.

# Conclusion

## This Lecture

- Overview of automated reasoning in Isabelle/HOL.
- SAT solvers, resolution provers, and SMT solvers.
- Tool integration with `sledgehammer`.
- Counterexample generators.

## Next Lecture

- Formal specification of railway signalling equipment.

# Conclusion

## This Lecture

- Overview of automated reasoning in Isabelle/HOL.
- SAT solvers, resolution provers, and SMT solvers.
- Tool integration with `sledgehammer`.
- Counterexample generators.

## Next Lecture

- Formal specification of railway signalling equipment.

# Conclusion

## This Lecture

- Overview of automated reasoning in Isabelle/HOL.
- SAT solvers, resolution provers, and SMT solvers.
- Tool integration with `sledgehammer`.
- Counterexample generators.

## Next Lecture

- Formal specification of railway signalling equipment.

# Conclusion

## This Lecture

- Overview of automated reasoning in Isabelle/HOL.
- SAT solvers, resolution provers, and SMT solvers.
- Tool integration with `sledgehammer`.
- Counterexample generators.

## Next Lecture

- Formal specification of railway signalling equipment.

# Conclusion

## This Lecture

- Overview of automated reasoning in Isabelle/HOL.
- SAT solvers, resolution provers, and SMT solvers.
- Tool integration with `sledgehammer`.
- Counterexample generators.

## Next Lecture

- Formal specification of railway signalling equipment.