AUTO2, a saturation-based heuristic prover for higher-order logic

Bohua Zhan

Massachusetts Institute of Technology

bzhan@mit.edu

August 23, 2016

Table of Contents

Motivation

- With the increasing depth and complexity of proofs, automation in interactive theorem provers becomes ever more important.

Motivation

- With the increasing depth and complexity of proofs, automation in interactive theorem provers becomes ever more important.
- Despite enormous progress, computers still cannot prove many statements that humans consider to be routine.

Motivation

- With the increasing depth and complexity of proofs, automation in interactive theorem provers becomes ever more important.
- Despite enormous progress, computers still cannot prove many statements that humans consider to be routine.
- Obstacle to both the QED project (formalization of mathematics) and more widespread adoption of formal software verification.

Motivation

- `auto2`: a new approach to automation that combines the best features of tactics and SMT.

Motivation

- auto2: a new approach to automation that combines the best features of tactics and SMT.
- From the tactics framework:
  - Incorporate heuristics that humans use when proving theorems.
  - Make it easy for users to add new heuristics while maintaining soundness (LCF architecture).
  - Work with higher order logic and (simple) types.

Motivation

- `auto2`: a new approach to automation that combines the best features of tactics and SMT.
- From the tactics framework:
  - Incorporate heuristics that humans use when proving theorems.
  - Make it easy for users to add new heuristics while maintaining soundness (LCF architecture).
  - Work with higher order logic and (simple) types.
- From SMT:
  - Have a robust search mechanism.

Motivation

- auto2: a new approach to automation that combines the best features of tactics and SMT.
- From the tactics framework:
  - Incorporate heuristics that humans use when proving theorems.
  - Make it easy for users to add new heuristics while maintaining soundness (LCF architecture).
  - Work with higher order logic and (simple) types.
- From SMT:
  - Have a robust search mechanism.
- auto2 is *not* designed to:
  - Be fully automatic.
  - Have good completeness properties.

Table of Contents

Overall architecture

- Implemented in Isabelle/ML, using Isabelle/HOL as base logic.

Overall architecture

- Implemented in Isabelle/ML, using Isabelle/HOL as base logic.
- Transform input problem into contradiction form:

$$[A_1, A_2, \ldots, A_n] \implies C \quad \text{becomes} \quad [A_1, A_2, \ldots, A_n, \neg C] \implies \text{False}.$$

Overall architecture

- Implemented in Isabelle/ML, using Isabelle/HOL as base logic.
- Transform input problem into contradiction form:

$$[A_1, A_2, \ldots, A_n] \implies C \quad \text{becomes} \quad [A_1, A_2, \ldots, A_n, \neg C] \implies \mathsf{False}.$$

- **Important:** Previously proved theorems are *not* among the assumptions.
  - Instead they are encoded into set of allowed actions.

Overall architecture

- Saturation based: maintain a list of **items** — facts derived from the initial assumptions. This list is initially $A_1, \ldots, A_n, \neg C$.

Overall architecture

- Saturation based: maintain a list of **items** — facts derived from the initial assumptions. This list is initially $A_1, \ldots, A_n, \neg C$.
- New items are generated from existing ones using **proof steps**. Proof steps are ML functions that accept as input one or two existing items, and output (among other possibilities) a list of items that logically follow from the input items.

Overall architecture

- Saturation based: maintain a list of **items** — facts derived from the initial assumptions. This list is initially $A_1, \ldots, A_n, \neg C$.
- New items are generated from existing ones using **proof steps**. Proof steps are ML functions that accept as input one or two existing items, and output (among other possibilities) a list of items that logically follow from the input items.
- Best-first search: each item is assigned a **score**. New items are put into a priority queue, and are inserted into the main list (and considered by proof steps) in order of their score.

Overall architecture

- Saturation based: maintain a list of **items** — facts derived from the initial assumptions. This list is initially $A_1, \ldots, A_n, \neg C$.
- New items are generated from existing ones using **proof steps**. Proof steps are ML functions that accept as input one or two existing items, and output (among other possibilities) a list of items that logically follow from the input items.
- Best-first search: each item is assigned a **score**. New items are put into a priority queue, and are inserted into the main list (and considered by proof steps) in order of their score.
- Algorithm ends when False is derived, or when there are no more items waiting to be processed, or if a timeout condition is reached.

A simple example

- Given an infinite sequence $X = (X_0, X_1, X_2, \ldots)$, assume X is monotone increasing, show $-X$ is monotone decreasing, where $-X$ is defined by $(-X)_i = -(X_i)$.

A simple example

- Given an infinite sequence $X = (X_0, X_1, X_2, \ldots)$, assume X is monotone increasing, show $-X$ is monotone decreasing, where $-X$ is defined by $(-X)_i = -(X_i)$.
- In Isabelle:

$$\text{monotone\_incr}\ X \implies \text{monotone\_decr}\ (-X).$$

- In contradiction form:

$$[\text{monotone\_incr} X, \neg\text{monotone\_decr}\ (-X)] \implies \text{False}.$$

A simple example

monotone_incr X

$\neg$monotone_decr $(-X)$

A simple example

$$\text{monotone\_incr } X$$
$$\downarrow$$
$$\forall m\, n.\; m \le n \longrightarrow X_m \le X_n$$

$$\neg\text{monotone\_decr } (-X)$$
$$\downarrow$$
$$\neg\forall m\, n.\; m \le n \longrightarrow (-X)_n \le (-X)_m$$

A simple example

$$\text{monotone\_incr } X$$

$$\downarrow$$

$$\forall m\, n.\; m \le n \longrightarrow X_m \le X_n$$

$$\neg\text{monotone\_decr } (-X)$$

$$\downarrow$$

$$\neg\forall m\, n.\; m \le n \longrightarrow (-X)_n \le (-X)_m$$

$$\swarrow \qquad \searrow$$

$$m \le n \qquad \neg(-X)_n \le (-X)_m$$

A simple example

$\text{monotone\_incr}\ X$

$\forall m\, n.\ m \le n \longrightarrow X_m \le X_n$

$\neg\text{monotone\_decr}\ (-X)$

$\neg\forall m\, n.\ m \le n \longrightarrow (-X)_n \le (-X)_m$

$m \le n$

$\neg(-X)_n \le (-X)_m$

$X_m \le X_n$

A simple example

$$\text{monotone\_incr } X$$

$$\forall m\, n.\ m \le n \longrightarrow X_m \le X_n$$

$$\neg \text{monotone\_decr } (-X)$$

$$\neg \forall m\, n.\ m \le n \longrightarrow (-X)_n \le (-X)_m$$

$$m \le n$$

$$\neg (-X)_n \le (-X)_m$$

$$X_m \le X_n$$

$$(-X)_m = -(X_m)$$
$$(-X)_n = -(X_n)$$

A simple example

$\text{monotone\_incr } X$

$\neg\text{monotone\_decr } (-X)$

$\forall m\, n.\ m \leq n \longrightarrow X_m \leq X_n$

$\neg\forall m\, n.\ m \leq n \longrightarrow (-X)_n \leq (-X)_m$

$m \leq n$

$\neg(-X)_n \leq (-X)_m$

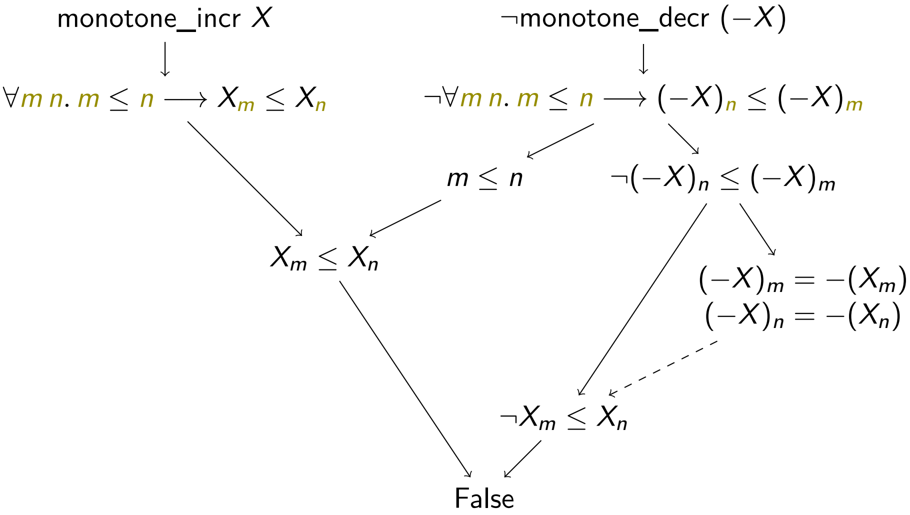
$X_m \leq X_n$

$(-X)_m = -(X_m)$

$(-X)_n = -(X_n)$

$\neg X_m \leq X_n$

A simple example



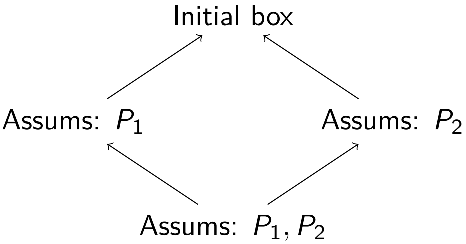
A simple example: linearized trace

1. monotone_incr X (assumption)
2. $\neg$monotone_decr $(-X)$ (assumption)
3. $\forall m\, n.\ m \leq n \longrightarrow X_m \leq X_n$ (1, def. of monotone_incr)
4. $\neg\forall m\, n.\ m \leq n \longrightarrow (-X)_n \leq (-X)_m$ (2, def. of monotone_decr)
5. $m \leq n, \neg(-X)_n \leq (-X)_m$ (4, skolemization)
6. $X_m \leq X_n$ (3 and 5a, quantifier instantiation)
7. $(-X)_m = -(X_m), (-X)_n = -(X_n)$ (5b, def. of $-X$)
8. $\neg X_m \leq X_n$ (5b and 7, inequalities)
9. False (6 and 8, contradiction)

Table of Contents

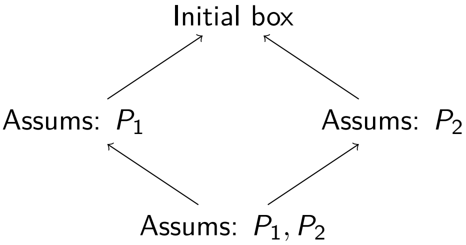
Details: case analysis

- A **box** corresponds to a subcase of the problem. They are specified by a list of additional assumptions. The boxes are organized into a lattice:



Details: case analysis

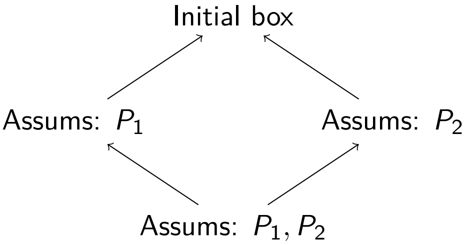
- A **box** corresponds to a subcase of the problem. They are specified by a list of additional assumptions. The boxes are organized into a lattice:



- New boxes are created by proof steps.

Details: case analysis

- A **box** corresponds to a subcase of the problem. They are specified by a list of additional assumptions. The boxes are organized into a lattice:



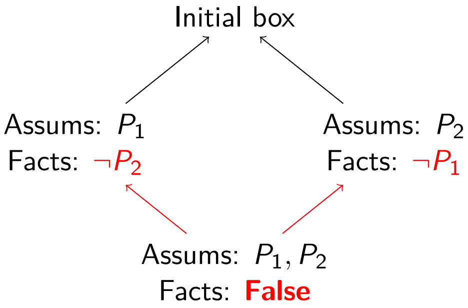
- New boxes are created by proof steps.
- Each item is placed in a box. Placing item P into box with additional assumptions P_1, P_2 is the same as deriving fact $[A_1, \ldots, A_n, \neg C, P_1, P_2] \implies P$.

Details: case analysis

- When a contradiction is derived in a box, the box is called *resolved*, and appropriate facts are added to its parent boxes:

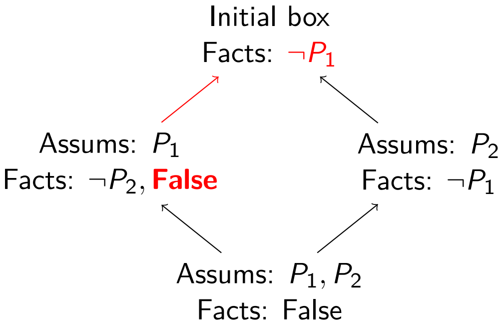
Details: case analysis

- When a contradiction is derived in a box, the box is called *resolved*, and appropriate facts are added to its parent boxes:
- On resolving the **bottom** box:



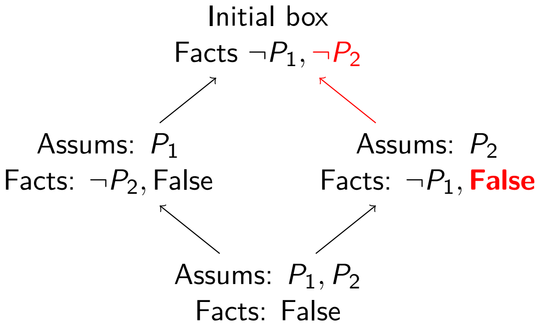
Details: case analysis

- When a contradiction is derived in a box, the box is called *resolved*, and appropriate facts are added to its parent boxes:
- On resolving the **left** box:



Details: case analysis

- When a contradiction is derived in a box, the box is called *resolved*, and appropriate facts are added to its parent boxes:
- On resolving the **right** box:



Details: case analysis

- This is similar to how case splitting is handled in the DPLL algorithm, but there are a few differences:
  - Case splits are generated by proof steps, which produce them based on what facts are currently derived.
  - Case splits are not necessarily in sequential order.
  - Derivation in different subcases proceed in parallel.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.
- Examples:
  - $S = \{x(y+z) = xy + xz\}, p = ?a + ?b, t = x(y+z)$.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.
- Examples:
  - $S = \{x(y+z) = xy + xz\}, p = ?a + ?b, t = x(y+z)$.
    Result: $\sigma = \{?a := xy, ?b := xz\}, p(\sigma) = t' = xy + xz$.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.
- Examples:
  - $S = \{x(y+z) = xy + xz\}, p = ?a + ?b, t = x(y+z)$.
    Result: $\sigma = \{?a := xy, ?b := xz\}, p(\sigma) = t' = xy + xz$.
  - $S = \{y = f(x), z = g(y)\}, p = g(f(?a)), t = z$.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.
- Examples:
  - $S = \{x(y+z) = xy + xz\}, p = ?a + ?b, t = x(y+z)$.
    Result: $\sigma = \{?a := xy, ?b := xz\}, p(\sigma) = t' = xy + xz$.
  - $S = \{y = f(x), z = g(y)\}, p = g(f(?a)), t = z$.
    Result: $\sigma = \{?a := x\}, p(\sigma) = t' = g(f(x))$.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.
- Examples:
  - $S = \{x(y+z) = xy + xz\}, p = ?a + ?b, t = x(y+z)$.
    Result: $\sigma = \{?a := xy, ?b := xz\}, p(\sigma) = t' = xy + xz$.
  - $S = \{y = f(x), z = g(y)\}, p = g(f(?a)), t = z$.
    Result: $\sigma = \{?a := x\}, p(\sigma) = t' = g(f(x))$.
  - $S = \{x = y, z = f(y)\}, p = f(x), t = z$.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.
- Examples:
  - $S = \{x(y+z) = xy + xz\}, p = ?a + ?b, t = x(y+z)$.
    Result: $\sigma = \{?a := xy, ?b := xz\}, p(\sigma) = t' = xy + xz$.
  - $S = \{y = f(x), z = g(y)\}, p = g(f(?a)), t = z$.
    Result: $\sigma = \{?a := x\}, p(\sigma) = t' = g(f(x))$.
  - $S = \{x = y, z = f(y)\}, p = f(x), t = z$.
    Result: $\sigma = \{\}, p(\sigma) = t' = f(x)$.

Details: E-matching

- The E-matching problem: given a set of equalities S, a pattern p, and a term t, find all instantiations σ of arbitrary variables in p, so that $p(\sigma) = t'$, where $t' \sim t$ according to equalities in S.
- Examples:
  - $S = \{x(y+z) = xy + xz\}, p = ?a + ?b, t = x(y+z)$.
    Result: $\sigma = \{?a := xy, ?b := xz\}, p(\sigma) = t' = xy + xz$.
  - $S = \{y = f(x), z = g(y)\}, p = g(f(?a)), t = z$.
    Result: $\sigma = \{?a := x\}, p(\sigma) = t' = g(f(x))$.
  - $S = \{x = y, z = f(y)\}, p = f(x), t = z$.
    Result: $\sigma = \{\}, p(\sigma) = t' = f(x)$.
- Widely used for quantifier instantiation in SMT solvers.

Details: E-matching

- E-matching is often used as the first step of a proof step function.

Details: E-matching

- E-matching is often used as the first step of a proof step function.
- Example: given a previously proved theorem of form $[A, B] \implies C$, where $vars(C) \subseteq vars(A) \cup vars(B)$, can write proof step that
  - Perform E-matching on two facts against patterns A and B.
  - For each match, output the instantiated C.

Details: Proof steps

- Proof steps encapsulate how to use each previously proved theorem, various heuristics, how to reason with logic, sets, arithmetic, etc.

Details: Proof steps

- Proof steps encapsulate how to use each previously proved theorem, various heuristics, how to reason with logic, sets, arithmetic, etc.
- Simple proof steps can be added in one line of code (for example, apply a theorem $[A, B] \implies C$ in the forward direction).

Details: Proof steps

- Proof steps encapsulate how to use each previously proved theorem, various heuristics, how to reason with logic, sets, arithmetic, etc.
- Simple proof steps can be added in one line of code (for example, apply a theorem $[A, B] \implies C$ in the forward direction).
- However, arbitrarily complex proof steps can be written in ML, with soundness guaranteed by the LCF architecture (similar to tactics in Isabelle).

Details: Proof scripts

- For a more difficult theorem, users can supply intermediate steps. `auto2` then tries to fill in the gaps between the steps.
  - `OBTAIN` P: prove P, then add P to the set of derived items.
  - `CASE` P: prove a contradiction from P, then add $\neg P$ to the set of derived items.
  - `CHOOSE` $x, P(x)$: prove $\exists x.P(x)$, then obtain new variable x satisfying $P(x)$.
  - C_1 `THEN` C_2: perform C_1, then perform C_2 after C_1 is finished.
  - C_1 `WITH` C_2: perform C_1, and perform C_2 as a part of proving the goal in C_1.

Details: Proof scripts

- For a more difficult theorem, users can supply intermediate steps. `auto2` then tries to fill in the gaps between the steps.
  - `OBTAIN` P: prove P, then add P to the set of derived items.
  - `CASE` P: prove a contradiction from P, then add $\neg P$ to the set of derived items.
  - `CHOOSE` $x, P(x)$: prove $\exists x.P(x)$, then obtain new variable x satisfying $P(x)$.
  - C_1 `THEN` C_2: perform C_1, then perform C_2 after C_1 is finished.
  - C_1 `WITH` C_2: perform C_1, and perform C_2 as a part of proving the goal in C_1.
- Similar to Isar, but with simpler structure, and no need to reference names of previous lemmas or tactics.

Table of Contents

Case studies

- Formalizations performed using `auto2`:
  - Elementary theory of prime numbers, up to infinitude of prime numbers and the unique factorization theorem.
  - Functional data structures, including red-black trees.
  - Parts of Hoare logic.
  - Verification of imperative programs (based on Imperative/HOL, without using Hoare or separation logic).
  - Construction of real numbers using Cauchy sequences.
  - Arrow's impossibility theorem.

Case studies

- Formalizations performed using `auto2`:
  - Elementary theory of prime numbers, up to infinitude of prime numbers and the unique factorization theorem.
  - Functional data structures, including red-black trees.
  - Parts of Hoare logic.
  - Verification of imperative programs (based on Imperative/HOL, without using Hoare or separation logic).
  - Construction of real numbers using Cauchy sequences.
  - Arrow's impossibility theorem.
- In all case studies, we aim to use `auto2` to prove all major theorems, using proof scripts at a level of detail comparable to usual mathematical exposition.

Case studies: prime numbers

- Lemma `larger_prime` (for proving infinitude of prime numbers):

  $\exists p.\, \mathrm{prime}\ p \wedge n < p$

  with proof script

  CHOOSE p, prime $p \wedge p$ dvd fact $n + 1$ THEN
  CASE $p \leq n$ WITH OBTAIN p dvd fact n

Case studies: prime numbers

- Lemma `factorization_unique_aux` (for proving uniqueness of factorization):

$$\forall p \in \text{set } M.\ \text{prime } p \implies \forall p \in \text{set } N.\ \text{prime } p \implies \prod_{i \in M} i \ \texttt{dvd} \ \prod_{i \in N} i \implies M \subseteq N$$

with proof script

CASE $M = \emptyset$ THEN
CHOOSE $M', m, M = M' + \{m\}$ THEN
OBTAIN $m \ \texttt{dvd} \ \prod_{i \in N} i$ THEN
CHOOSE $n, n \in N \wedge m \ \texttt{dvd} \ n$ THEN
CHOOSE $N', N = N' + \{n\}$ THEN
OBTAIN $m = n$ THEN
OBTAIN $\prod_{i \in M'} i \ \texttt{dvd} \ \prod_{i \in N'} i$ THEN
STRONG_INDUCT $(M, [\text{Arbitrary } N])$

Case studies: verification of imperative programs

- Based on Imperative-HOL in the Isabelle library. Reason directly from the semantics of commands (no use of Hoare or separation logic).

Case studies: verification of imperative programs

- Based on Imperative-HOL in the Isabelle library. Reason directly from the semantics of commands (no use of Hoare or separation logic).
- Data structures and algorithms verified:
  - Arrays: reverse, quicksort.
  - Linked list: insert, remove, reverse, merge.
  - Binary trees: insert, delete-min.

Case studies: verification of imperative programs

- Based on Imperative-HOL in the Isabelle library. Reason directly from the semantics of commands (no use of Hoare or separation logic).
- Data structures and algorithms verified:
  - Arrays: reverse, quicksort.
  - Linked list: insert, remove, reverse, merge.
  - Binary trees: insert, delete-min.
- Most proofs are either automatic or only need specifying the induction scheme, whereas corresponding proofs using tactics can run for several dozen lines. The theorems also appear to be beyond the reach of Sledgehammer tools.

Case studies: Arrow's impossibility theorem

- One of the examples used as Sledgehammer benchmarks.

Case studies: Arrow's impossibility theorem

- One of the examples used as Sledgehammer benchmarks.
- Important result in social choice theory: it is impossible to design a voting system for more than two candidates that satisfy a set of reasonable conditions.

Case studies: Arrow's impossibility theorem

- One of the examples used as Sledgehammer benchmarks.
- Important result in social choice theory: it is impossible to design a voting system for more than two candidates that satisfy a set of reasonable conditions.
- Proofs of all major lemmas / theorems are done using `auto2`, with slightly fewer subgoals than in the tactics version.

Table of Contents

Future work

- Present partial or completed proofs in a way that is easy for humans to read and navigate (for debugging or understanding the proof).

Future work

- Present partial or completed proofs in a way that is easy for humans to read and navigate (for debugging or understanding the proof).
- Systematic, well-tested library of proof steps for:
  - Equality and inequality reasoning on natural numbers, integers, rationals, and real numbers.
  - Reasoning about sets and partial functions.

Future work

- Present partial or completed proofs in a way that is easy for humans to read and navigate (for debugging or understanding the proof).
- Systematic, well-tested library of proof steps for:
  - Equality and inequality reasoning on natural numbers, integers, rationals, and real numbers.
  - Reasoning about sets and partial functions.
- For formalization of mathematics: systems of heuristics for real and complex analysis, abstract algebra, number theory, discrete mathematics, etc.

Future work

- Present partial or completed proofs in a way that is easy for humans to read and navigate (for debugging or understanding the proof).
- Systematic, well-tested library of proof steps for:
  - Equality and inequality reasoning on natural numbers, integers, rationals, and real numbers.
  - Reasoning about sets and partial functions.
- For formalization of mathematics: systems of heuristics for real and complex analysis, abstract algebra, number theory, discrete mathematics, etc.
- For verification of imperative programs: incorporating separation logic, as well as further techniques such as symbolic execution and shape analysis.

Conclusion

- After one and a half years of development, `auto2` already provides automation that compares favorably with both tactics and Sledgehammer in Isabelle.

Conclusion

- After one and a half years of development, `auto2` already provides automation that compares favorably with both tactics and Sledgehammer in Isabelle.
- Improvements are still to be made on all fronts.

Conclusion

- After one and a half years of development, `auto2` already provides automation that compares favorably with both tactics and Sledgehammer in Isabelle.
- Improvements are still to be made on all fronts.
- Opportunity to carry automation in Isabelle (and possibly other proof assistants) to the next level.

Conclusion

- After one and a half years of development, `auto2` already provides automation that compares favorably with both tactics and Sledgehammer in Isabelle.
- Improvements are still to be made on all fronts.
- Opportunity to carry automation in Isabelle (and possibly other proof assistants) to the next level.
- Link to code:

  `https://github.com/bzhan/auto2`