Alexander Bentkamp
Mathematisches Institut
Heinrich-Heine-Universität Düsseldorf
alexander.bentkamp@hhu.de

Jasmin Blanchette
Institut für Informatik
Ludwig-Maximilians-Universität München
jasmin.blanchette@ifi.lmu.de

Research

Finding mathematical proofs using computers

In the 1960s, some researchers believed that computers would one day replace mathematicians: Computers would autonomously suggest conjectures and prove them automatically. Despite recent progress in artificial intelligence, this vision has not yet materialized. Even if they are assisted by computers (via, e.g., computer algebra systems), mathematicians remain needed for doing mathematics. Even so, there has been substantial progress in the area of automated reasoning in the past 60 years. In this article, Alexander Bentkamp and Jasmin Blanchette describe the development of automatic proof methods and automatic theorem provers based on those methods. Broadly understood, the area also encompasses interactive theorem provers, which provide a graphical user interface for developing formal proofs.

In combination, automatic and interactive theorem provers help users develop rigorous, computer-checked proofs. The easiest parts are carried out automatically, and the more difficult parts require user intervention. For example, the following lemma, which originates from a mathematics paper, can be proved by automatic theorem provers:

$$\begin{aligned} & \gcd(a,b) = 1 \\ \wedge\ & d \mid ab \\ \wedge\ & a' \mid \gcd(d,a) \\ \wedge\ & b' \mid \gcd(d,b) \\ \Rightarrow\ & a'b' \mid d \wedge d \mid a'b' \end{aligned}$$

The symbols $\wedge$ and $\Rightarrow$ mean 'and' and 'implies', $\gcd$ is the greatest common divisor, and $\mid$ is the 'divides' relation. Such a lemma is easy for mathematicians, so it is a relief that it can be proved automatically; the alternative, a fully rigorous, interactive proof, could easily take 15 to 30 minutes.

In this article, we will review two leading proof methods implemented in automatic theorem provers: *resolution* [5] and *superposition* [2]. Although they are not based on machine learning, as descendants of the pioneering Logic Theorist system these methods constitute a form of artificial intelligence. Resolution can be used to prove problems formulated in predicate logic (also called first-order logic). Superposition is a further development of resolution with special support for the equality symbol ($=$), which is ubiquitous in mathematics. One method that we will not cover but that is also very successful is satisfiability modulo theories [3].

Automatic theorem provers based on resolution and superposition can be used on their own, but often they are invoked as backends from the comfort of an interactive theorem prover. This relies on bridges between automatic and interactive provers.

This article is based on an eponymous presentation by Blanchette at the KWG Wintersymposium 2023 on 14 January 2023 in Utrecht. That presentation in turn drew from a paper written by the present authors together with colleagues and recently published in the *Communications of the ACM*.

Our logical formalism: Predicate logic

There are hundreds, perhaps thousands of logics, but when people simply say 'logic', they often mean predicate logic. Predicate logic is parameterized by a *signature*, which consists of *function symbols* (e.g., 1, ..., gcd) and *predicate symbols* (e.g., $=$, $\mid$, prime). Thus, strictly speaking, predicate logic is a family of logics indexed by a signature. The signature also associates an *arity* with each symbol: the number of arguments the symbol may take.

Based on a fixed signature, the *terms* are defined by these two rules:

- A variable x is a term.
- If f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms, then $\mathrm{f}(t_1, \ldots, t_n)$ is a term. As a special case, if $n = 0$, we write f instead of $\mathrm{f}()$ and we call f a constant.

Finally, the *formulas* are defined by these rules:

- If p is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms, then $\mathrm{p}(t_1, \ldots, t_n)$ is a formula. As a special case, if $n = 0$, we write p instead of $\mathrm{p}()$.
- If F and G are formulas and x is a variable, then $\bot$ (falsity), $\top$ (truth), $\neg F$ (negation), $F \wedge G$ (conjunction), $F \vee G$ (disjunction), $F \Rightarrow G$ (implication), $\forall x.\, F$ (universal quantification), and $\exists x.\, F$ (existential quantification) are formulas.

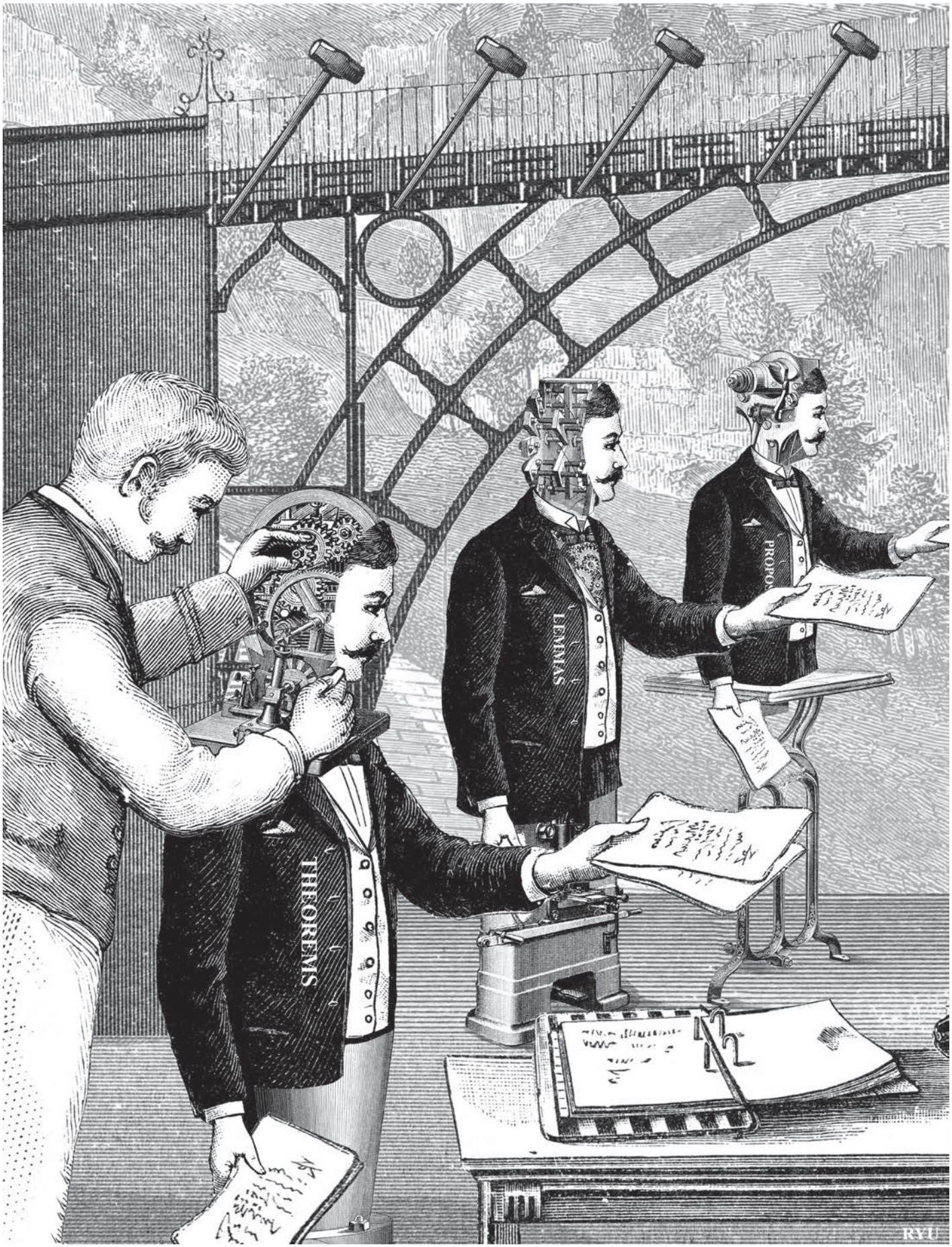

Illustration: Ryu Tajiri

Terms represent mathematical objects, whereas formulas represent mathematical statements. An example of a complex formula follows:

$$\begin{aligned} & (\forall x.\, \mathrm{food}(x) \Rightarrow \mathrm{likes}(\mathrm{johanna}, x)) \\ \wedge\ & (\forall x.\, (\exists y.\, \mathrm{eats}(y,x) \wedge \neg \mathrm{wasKilledBy}(y,x)) \Rightarrow \mathrm{food}(x)) \\ \wedge\ & \mathrm{eats}(\mathrm{bill}, \mathrm{peanuts}) \\ \wedge\ & \mathrm{alive}(\mathrm{bill}) \\ \wedge\ & (\forall y.\, \mathrm{alive}(y) \Rightarrow \forall x.\, \neg \mathrm{wasKilledBy}(y,x)) \\ \Rightarrow\ & \mathrm{likes}(\mathrm{johanna}, \mathrm{peanuts}) \end{aligned}$$

It can be a useful exercise to try to identify the function symbols and the predicate symbols above.

A general framework: Proof by saturation

Resolution and superposition are instances of a general framework called 'proof by saturation', which is a form of proof by refutation. It works as follows. Given a problem $F_1 \wedge \cdots \wedge F_n \Rightarrow G$, perform these steps:

1. Put $F_1, \ldots, F_n$ and the negation of G in clausal form, yielding a set $\mathcal{F}$ of clauses (i.e., of formulas in clausal form).
2. Repeat forever:
   2.1. Apply an inference rule to $\mathcal{F}$ and add the conclusion to $\mathcal{F}$. Stop if no new inference is possible.
   2.2. Possibly remove redundant clauses. For example, $\mathrm{p} \vee \mathrm{q}$ is redundant in the presence of the clause p.
   2.3. Stop if $\bot$ (falsity) is in $\mathcal{F}$.

Both resolution and superposition work with clauses. These are made of *literals*, which are themselves made of *atoms*:

- If p is a predicate symbol and $t_1, \ldots, t_n$ are terms, then $\mathrm{p}(t_1, \ldots, t_n)$ is an atom.
- If A is an atom, then A and $\neg A$ are literals.

The clauses are then defined by this rule:

- If $L_1, \ldots, L_n$ are literals, then $L_1 \vee \cdots \vee L_n$ is a clause. Clauses are considered equal up to associativity and commutativity of $\vee$. If $n = 0$, we write $\bot$ (which is appropriate since $\bot$ is the neutral element for $\vee$).

For example, by commutativity, $\mathrm{p} \vee \mathrm{q}(x)$ is the same clause as $\mathrm{q}(x) \vee \mathrm{p}$. It may help to think of clauses as multisets of literals. In keeping with this view, we call $\bot$ the empty clause.

Note that the logical symbols $\top$, $\wedge$, $\Rightarrow$, $\forall$, and $\exists$ cannot occur in a clause. Any variable occurring in a clause is understood as a $\forall$ variable; for example, the clause $\mathrm{p}(x) \vee \mathrm{q}(x)$ is understood to mean the same as the formula $\forall x.\, \mathrm{p}(x) \vee \mathrm{q}(x)$.

Any problem can be transformed into a set of clauses. For example, the problem

$$\begin{aligned} & (\forall x.\, \mathrm{human}(x) \Rightarrow \mathrm{mortal}(x)) \\ \wedge\ & \mathrm{human}(\mathrm{anne}) \\ \Rightarrow\ & \mathrm{mortal}(\mathrm{anne}) \end{aligned}$$

can be translated to the following three clauses:

$$\begin{aligned} & \neg \mathrm{human}(x) \vee \mathrm{mortal}(x) \\ & \mathrm{human}(\mathrm{anne}) \\ & \neg \mathrm{mortal}(\mathrm{anne}) \end{aligned}$$

The original problem is clearly provable. Correspondingly, its translation to clauses is inconsistent, meaning that it admits a proof by refutation, as we will see below.

It may seem surprising that even formulas containing $\exists$ can be converted to the clausal format. This is possible thanks to a technique called *Skolemization* [1], whereby the $\exists$ variables are replaced by new symbols representing unknown witnesses. For example, $\forall x.\, \exists y.\, \mathrm{p}(x,y)$ is translated to $\mathrm{p}(x, \mathrm{wit}(x))$, where $\mathrm{wit}(x)$ represents the witness associated with x ("for every x, there exists a witness $\mathrm{wit}(x)$ such that $\mathrm{p}(x, \mathrm{wit}(x))$").

A first proof method: Resolution

Resolution is an instance of the saturation framework that consists of one main inference rule (and of a side rule, which we will not cover). Ignoring for a moment that clauses may contain variables, we can state the rule as follows:

> If the clauses $C \vee A$ and $\neg A \vee D$ are contained in $\mathcal{F}$, then add the clause $C \vee D$ to $\mathcal{F}$.

Here, C and D stand for clauses, and A stands for an atom. Because clauses are defined up to associativity and commutativity of $\vee$, the literal A may actually occur anywhere in $C \vee A$, and similarly for $\neg A$ in $\neg A \vee D$. Thus, $C \vee A$ denotes the clause that contains the literal A and all the literals from C, whereas $\neg A \vee D$ denotes the clause that contains the literal $\neg A$ and all the literals from D.

Suppose $\mathcal{F}$ consists of the following clauses:

$$\begin{aligned} & \neg \mathrm{human}(\mathrm{anne}) \vee \mathrm{mortal}(\mathrm{anne}) \\ & \mathrm{human}(\mathrm{anne}) \\ & \neg \mathrm{mortal}(\mathrm{anne}) \end{aligned}$$

This is the same set as above, except that we instantiated x with anne to avoid variables. A first inference is possible involving the first and second clauses, taking C to be $\bot$ (the empty clause), A to be $\mathrm{human}(\mathrm{anne})$, and D to be $\mathrm{mortal}(\mathrm{anne})$. The conclusion, $C \vee D$, consists of the literals of $\bot$ and those of $\mathrm{mortal}(\mathrm{anne})$. Thus, the conclusion is

$$\mathrm{mortal}(\mathrm{anne})$$

and we add it to $\mathcal{F}$.

Next, we can perform a second inference involving the newly added clause with the clause $\neg \mathrm{mortal}(\mathrm{anne})$. This time, C and D are both $\bot$, and the conclusion to add to $\mathcal{F}$ is $\bot$. Once the empty clause is added to $\mathcal{F}$, the saturation loop stops and the contradiction is reported. The proof by refutation is successful.

The resolution inference would appear to be working correctly on this example, but is it correct in general? And is it complete, meaning: Will saturation always find a contradiction from an inconsistent clause set? The answer to both questions is yes.

We start with correctness. We will assume the two premises $C \vee A$ and $\neg A \vee D$ are true and show that $C \vee D$ is true. We proceed by case distinction on A:

- If A is true, then $\neg A$ is false, and $\neg A \vee D$ can be true only if D is true. If D is true, then the conclusion $C \vee D$ is true as well, as desired.
- If A is false, then $C \vee A$ can be true only if C is true. If C is true, then the conclusion $C \vee D$ is true as well, as desired.

Resolution (as presented in the literature) is also complete in the following sense: If the clause set $\mathcal{F}$ is initially inconsistent and

inferences are performed fairly, then $\mathcal{F}$ will eventually contain $\bot$. The fairness requirement is vital; without it, a necessary inference might be delayed forever, preventing the derivation of $\bot$.

So far, we have ignored the difficulties arising from the presence of variables in clauses, but resolution can actually cope with variables. Suppose we have the two clauses $C(y) \vee \mathrm{p}(\mathrm{a}, y)$ and $\neg\mathrm{p}(x, \mathrm{b}) \vee D(x)$, where $C(y)$ denotes a clause that depends on the variable y and $D(x)$ a clause that depends on x. The atoms $\mathrm{p}(\mathrm{a}, y)$ and $\mathrm{p}(x, \mathrm{b})$ are not syntactically identical, but they can be made identical by taking x to be a and y to be b, following the German saying "was nicht passt, wird passend gemacht". This process of instantiating variables to make atoms identical is called *unification* [5]. Once we unify the two atoms, we get the two clauses

$$C(\mathrm{b}) \vee \mathrm{p}(\mathrm{a}, \mathrm{b}) \qquad \neg\mathrm{p}(\mathrm{a}, \mathrm{b}) \vee D(\mathrm{a})$$

From these, a resolution inference derives $C(\mathrm{b}) \vee D(\mathrm{a})$.

We can now consider the 'Anne is mortal' example in its full generality:

$$\neg\mathrm{human}(x) \vee \mathrm{mortal}(x)$$
$$\mathrm{human}(\mathrm{anne})$$
$$\neg\mathrm{mortal}(\mathrm{anne})$$

From $\mathrm{human}(\mathrm{anne})$ and $\neg\mathrm{human}(x) \vee \mathrm{mortal}(x)$, we instantiate x with anne and compute the conclusion $\mathrm{mortal}(\mathrm{anne})$. From this conclusion and $\neg\mathrm{mortal}(\mathrm{anne})$, we derive $\bot$.

Although resolution is complete, it is not always efficient. Consider the inconsistent set $\mathcal{F}$ consisting of the six clauses

$$\mathrm{a} \vee \mathrm{b} \vee \mathrm{c} \vee \mathrm{d} \vee \mathrm{e} \qquad \neg\mathrm{a} \qquad \neg\mathrm{b} \qquad \neg\mathrm{c} \qquad \neg\mathrm{d} \qquad \neg\mathrm{e}$$

A *particularly inefficient strategy* would first derive all four-literal clauses that can be derived from this set. There are five of them:

$$\mathrm{b} \vee \mathrm{c} \vee \mathrm{d} \vee \mathrm{e} \qquad \mathrm{a} \vee \mathrm{c} \vee \mathrm{d} \vee \mathrm{e} \qquad \mathrm{a} \vee \mathrm{b} \vee \mathrm{d} \vee \mathrm{e}$$
$$\mathrm{a} \vee \mathrm{b} \vee \mathrm{c} \vee \mathrm{e} \qquad \mathrm{a} \vee \mathrm{b} \vee \mathrm{c} \vee \mathrm{d}$$

Then it would derive the three-literal clauses:

$$\mathrm{c} \vee \mathrm{d} \vee \mathrm{e} \qquad \mathrm{b} \vee \mathrm{d} \vee \mathrm{e} \qquad \mathrm{b} \vee \mathrm{c} \vee \mathrm{e} \qquad \mathrm{b} \vee \mathrm{c} \vee \mathrm{d} \qquad \mathrm{a} \vee \mathrm{d} \vee \mathrm{e}$$
$$\mathrm{a} \vee \mathrm{c} \vee \mathrm{e} \qquad \mathrm{a} \vee \mathrm{c} \vee \mathrm{d} \qquad \mathrm{a} \vee \mathrm{b} \vee \mathrm{e} \qquad \mathrm{a} \vee \mathrm{b} \vee \mathrm{d} \qquad \mathrm{a} \vee \mathrm{b} \vee \mathrm{c}$$

Then the one-literal clauses:

$$\mathrm{e} \qquad \mathrm{d} \qquad \mathrm{c} \qquad \mathrm{b} \qquad \mathrm{a}$$

Finally, from any one-literal clause (e.g., a) and its negation (e.g., $\neg\mathrm{a}$), it would derive $\bot$.

A *more efficient strategy* would derive, from the initial set $\mathcal{F}$, a single four-literal clause (e.g., $\mathrm{a} \vee \mathrm{b} \vee \mathrm{c} \vee \mathrm{d}$), then a single three-literal clause (e.g., $\mathrm{a} \vee \mathrm{b} \vee \mathrm{c}$), then a single two-literal clause (e.g., $\mathrm{a} \vee \mathrm{b}$), then a single one-literal clause (e.g., a), and finally $\bot$. Such a strategy can be programmed in the automatic provers, but we can do better: We can enforce it in the proof method itself by introducing an order.

Specifically, *ordered resolution* is a variant of resolution that is parameterized by an order on literals. For our example, we will simply take the alphabetical order, requiring $\mathrm{e} > \mathrm{d} > \mathrm{c} > \mathrm{b} > \mathrm{a}$. Ignoring that clauses may contain variables, we can state the main inference rule of ordered resolution as follows:

> If the clauses $C \vee A$ and $\neg A \vee D$ are contained in $\mathcal{F}$, A is maximal in $C \vee A$, and $\neg A$ is maximal in $\neg A \vee D$, then add the clause $C \vee D$ to $\mathcal{F}$.

This rule is less explosive than the standard resolution rule. It is also complete; for example, the 'more efficient strategy' above would be allowed, whereas the 'particularly inefficient strategy' would be disallowed. Intuitively, ordered resolution focuses on the largest literals first and tries to eliminate them by performing inferences. If they cannot be eliminated, the other literals are never considered.

The situation is analogous to a scenario often encountered by mathematicians. Suppose we want to use a lemma of the form "If F_1, F_2, and F_3, then G" in a proof, we can without loss of generality focus on F_1 and try to prove it first, then proceed with F_2, then with F_3, and finally retrieve the conclusion G. If we fail at proving the condition F_1 (i.e., at 'eliminating' F_1), we can immediately give up; there is no point in trying to prove F_2 and F_3.

How to deal with equality

Equality is ubiquitous in mathematical problems. With resolution, to reason about equality we need to specify its properties as axioms to be included in the problem:

reflexivity:	$\forall x.\ x = x$
symmetry:	$\forall x, y.\ x = y \Rightarrow x = z$
transitivity:	$\forall x, y, z.\ (x = y \wedge y = z) \Rightarrow x = z$
congruence:	$\forall x, y.\ x = y \Rightarrow \mathrm{f}(x) = \mathrm{f}(y)$

Congruence is shown for a single unary function symbol f, but it needs to be repeated for all function and predicate symbols.

An alternative, which is the approach taken by superposition, is to treat equality specially in the proof method. Reflexivity, symmetry, transitivity, and congruence are then not axiomatized. Moreover, once equality is available, we can eliminate all other predicates. Specifically, for any predicate symbol p other than equality, we can use a function symbol f instead that returns a truth value. A literal $\mathrm{p}(x)$ can be coded as $\mathrm{f}(x) = \mathrm{true}$, and a literal $\neg\mathrm{p}(x)$ can be coded as $\neg(\mathrm{f}(x) = \mathrm{true})$. Literals then have the form $t = t'$ and $\neg(t = t')$, and we write $t \neq t'$ to abbreviate $\neg(t = t')$. Finally, we consider literals equal up to commutativity of $=$; thus, $\mathrm{b} = \mathrm{a}$ and $\mathrm{a} = \mathrm{b}$ are the same literal.

A more advanced proof method: Superposition

Superposition resembles resolution, but it works on clauses whose atoms are equalities $t = t'$, and it performs inferences that embody the four characteristic properties of equality (reflexivity, symmetry, transitivity, and congruence). It consists of three rules, of which we will review two.

Ignoring variables and multiple-literal clauses, we can state the main inference rule as follows:

> If the clauses $t = t'$ and $L[t]$ are contained in $\mathcal{F}$, then add the clause $L[t']$ to $\mathcal{F}$.

Here, $L[t]$ denotes a literal that contains t as a subterm, and $L[t']$ denotes the same literal in which the singled-out occurrence of t is replaced by t'. The rule essentially allows the replacement of equals with equals.

Clause sets rarely consist exclusively of single-literal clauses, so we need to generalize the above inference rule to allow multiple literals:

> If the clauses $C \vee t = t'$ and $D \vee L[t]$ are contained in $\mathcal{F}$, then add the clause $C \vee D \vee L[t']$ to $\mathcal{F}$.

The C and D components play a similar role as in resolution.

Next to the main rule, the following side rule invariably appears in successful refutations:

If the clause $C \vee t \neq t$ is contained in $\mathcal{F}$, then add the clause C to $\mathcal{F}$.

Both rules are stated above without worrying about variables, but superposition, like resolution, unifies terms as necessary to make them syntactically equal, as we will see with an example. Informally, the problem is as follows:

Assuming that $\pi \neq 0$ and that $x^{-1} = 1/x$ for all $x \neq 0$, we have $|\pi^{-1}| = |1/\pi|$.

Expressed as a formula, the problem becomes

$$\begin{aligned} &(\forall x.\, x \neq \mathrm{zero} \Rightarrow \mathrm{inv}(x) = \mathrm{div}(\mathrm{one}, x)) \\ &\wedge\, \mathrm{pi} \neq \mathrm{zero} \\ \Rightarrow\quad &\mathrm{abs}(\mathrm{inv}(\mathrm{pi})) = \mathrm{abs}(\mathrm{div}(\mathrm{one}, \mathrm{pi})) \end{aligned}$$

Conversion into clausal form yields three clauses:

$$\begin{aligned} &x = \mathrm{zero} \vee \mathrm{div}(\mathrm{one}, x) = \mathrm{inv}(x) \\ &\mathrm{pi} \neq \mathrm{zero} \\ &\mathrm{abs}(\mathrm{div}(\mathrm{one}, \mathrm{pi})) \neq \mathrm{abs}(\mathrm{inv}(\mathrm{pi})) \end{aligned}$$

Looking at the first and third clauses, we notice that we can unify the term $\mathrm{div}(\mathrm{one}, x)$ in the first clause with the subterm $\mathrm{div}(\mathrm{one}, \mathrm{pi})$ in the third clause, by taking x to be pi. The main inference rule is applicable and adds the clause

$$\mathrm{pi} = \mathrm{zero} \vee \mathrm{abs}(\mathrm{inv}(\mathrm{pi})) \neq \mathrm{abs}(\mathrm{inv}(\mathrm{pi}))$$

to $\mathcal{F}$. At this point, the side rule applies to eliminate the second literal, resulting in

$$\mathrm{pi} = \mathrm{zero}$$

Now, we can apply the main rule on this clause and on the clause $\mathrm{pi} \neq \mathrm{zero}$, resulting in

$$\mathrm{zero} \neq \mathrm{zero}$$

Finally, an application of the side rule eliminates the literal, yielding $\bot$.

Superposition is correct and complete. Moreover, like resolution, superposition can take an order into account to restrict its search space. The main inference rule then focuses on the larger side of the largest literal of each of the two premises, trying to rewrite larger clauses into smaller clauses. That superposition is complete despite such drastic restrictions is far from obvious [2, Section 4].

Although superposition is not strictly speaking a generalization of resolution, in practice superposition provers have replaced resolution provers. Resolution is nowadays seen mostly as a stepping stone, as within this article.

Bridges between automatic and interactive provers

Automatic theorem provers, including those based on superposition, are integrated in proof assistants via bridges. These bridges are called 'hammers' [4] in honor of Sledgehammer, possibly the most successful such bridge.

Automatic provers work best when their input does not contain too many axioms. If all of the proof assistant's definitions, lemmas, theorems, and actual axioms were exported as axioms to the automatic provers, these would have to find their way among perhaps 10 000 formulas and would not perform very well. Hence, the first step of a hammer is to filter the available facts (definitions, lemmas, et cetera) to a reasonable number — typically less than a thousand.

The second step is to translate the problem. Interactive provers typically work in a richer logic (such as higher-order logic and dependent type theory) than automatic provers. There is work on reducing the gap between the two types of systems, including by the present authors, but for most combinations of interactive and automatic theorem provers a translation is necessary.

At this point, the automatic provers run on the translated problem. Sledgehammer works with a default time limit of 30 seconds. If a proof is found within that time, the last step is to import this proof into the interactive proof assistant, where it is independently rechecked.

By building on the strengths of automatic theorem provers, hammers make users more productive. One user claims he is three to five times more productive thanks to Sledgehammer. Another compared working with it to running as opposed to walking. As developers of automatic provers further improve their systems, hammers become even stronger, benefiting users of interactive provers.

Acknowledgment

We thank Mark Summerfield and Mark Timmer for suggesting several textual improvements.

References

1 M. Baaz, U. Egly and A. Leitsch, Normal form transformations, in J.A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning* (in 2 volumes), Elsevier and MIT Press, 2001, pp. 273–333.

2 L. Bachmair and H. Ganzinger, Rewrite-based equational theorem proving with selection and simplification, *J. Log. Comput.* 4(3) (1994), 217–247.

3 C.W. Barrett, R. Sebastiani, S.A. Seshia and C. Tinelli, Satisfiability modulo theories, in A. Biere, M. Heule, H. van Maaren and T. Walsh, eds., *Handbook of Satisfiability*, Second Edition, Vol. 336 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2021, pp. 1267–1329.

4 J.C. Blanchette, C. Kaliszyk, L.C. Paulson and J. Urban, Hammering towards QED, *J. Formaliz. Reason.* 9(1) (2016), 101–148.

5 J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM* 12(1) (1965), 23–41.