Mechanical Mathematicians

A new generation of automatic theorem provers eliminate bugs in software and mathematics.

Alexander Bentkamp
bentkamp@ios.ac.cn
State Key Lab. of Computer Science, Institute of Software, CAS
Beijing, China

Jasmin Blanchette
j.c.blanchette@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Visa Nummelin
visa.nummelin@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Sophie Tourret
sophie.tourret@inria.fr
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France

Petar Vukmirović
p.vukmirovic@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Uwe Waldmann
uwe@mpi-inf.mpg.de
Max-Planck-Institut für Informatik
Saarbrücken, Germany

ABSTRACT

For several decades, the focus of research in automated deduction was on developing theorem provers—such as resolution provers and satisfiability (SAT) solvers—for classical first-order logic and weaker logics. Recently, there has been more work on developing higher-order automatic theorem provers. These are based on higher-order logics, which support functions as arguments, quantification over functions, and binders (e.g., the notations for summation and integration). These logics are more suitable than first-order logic for expressing a wide range of mathematics, and they are useful for hardware and software verification as well. Recent successes lead us to envision a world in which mathematicians and computer scientists can develop computer-checked proofs faster than LaTeX proofs.

CCS CONCEPTS

• **Theory of computation** → **Automated reasoning**; **Logic and verification**; **Higher order logic**.

KEYWORDS

Theorem proving, higher-order logic, superposition

In the 1960s, researchers dreamed of automatic theorem provers so powerful that these would develop lengthy proofs of open conjectures completely on their own. The hope was that these "mechanical mathematicians" would replace flesh-and-blood mathematicians.

As we now know, this did not come to pass. For many decades, automatic provers generally remained too weak to help, let alone replace, mathematicians. Instead, mathematicians turned to computer algebra systems—which help not so much with proofs as with computations—and automatic provers turned to other application areas. These areas include hardware and software verification, either directly or via an interactive verification platform (e.g., Atelier B, Dafny, F*, Frama-C, SPARK 2014, Spec#, VCC, Why3). In addition, automatic provers are used as backends to general-purpose proof assistants (e.g., ACL2, Coq, HOL, Isabelle, Lean, Mizar). As mathematicians are slowly embracing proof assistants, automatic provers are finally becoming useful to them, for discharging straightforward but tedious proof tasks.

As a simple example from elementary mathematics, consider the formula

$$\big(\gcd(a,b) = 1 \ \wedge\ d \mid ab \ \wedge\ a' = \gcd(d,a) \ \wedge\ b' = \gcd(d,b)\big) \\ \Rightarrow\ (a'b' \mid d \ \wedge\ d \mid a'b')$$

The logical symbols $\wedge$ and $\Rightarrow$ mean "and" and "implies," and the other symbols have their usual mathematical meanings—thus, gcd is the greatest common divisor, and $\mid$ is the "divides" relation between two numbers (e.g., $d \mid ab$ means ab is divisible by d). Let us check a special case: If $a = 12$, $b = 35$, and $d = 5$, we have $a' = 1$, $b' = 5$, and indeed $a'b' = 5$ divides $d = 5$ and vice versa. Mathematicians can easily see that the formula holds in general, but to write a formal proof with an acceptable level of detail for a proof assistant can easily take 15 or 30 minutes. In contrast, automatic theorem provers can prove such formulas within seconds.

It would be nice if automatic provers could always prove or disprove whatever formulas we give them in reasonable time. This is, however, impossible because provability is undecidable; we need to accept that provers might run forever on some unprovable inputs. And even for provable formulas, provers might be hopelessly slow. Here are some easy-looking formulas that could not be proved automatically in reasonable time only a few years ago:

$$\big(\textstyle\sum_{i=1}^{n} i^2 + 2i + 1\big) = \big(\sum_{i=1}^{n} i^2\big) + \big(\sum_{i=1}^{n} 2i\big) + \big(\sum_{i=1}^{n} 1\big)$$

$$\big((\forall x.\ f(x) = g(x)) \ \wedge\ f \in S\big) \ \Rightarrow\ g \in S$$

$$f(n) + g(n) \in \mathrm{O}(h(n)) \ \Rightarrow\ f(k(n)) + g(k(n)) \in \mathrm{O}(h(k(n)))$$

Intuitively, it seems reasonable that the formulas should hold: The first formula follows by distributivity of summation over addition. The second formula states that if two functions f and g coincide at all points x, then they are in fact equal, and therefore they will be members of the same sets S. The third formula states that membership in big O is invariant under application of a function k on n.

The reason why older provers time out on these formulas is that they are higher-order: They contain binders such as summation and big O as well as first-class functions (e.g., the unapplied occurrences of f and g). In contrast, the most commonly used provers for this kind of problems are first-order. This includes resolution and superposition provers and satisfiability-modulo-theories solvers. Higher-order formulas need to be translated before they can be understood by first-order provers. In principle, these encodings

work; in practice, they are so heavy and clumsy that the first-order provers are overwhelmed and time out. The higher-order constructs are effectively lost in translation.

Next to the first-order provers, native higher-order provers have been researched and developed since the 1970s. They were designed to prove smaller and sometimes tricky formulas, but not to tackle larger, mostly first-order problems emerging from interactive verification platforms and proof assistants.

In the past five years or so, a new generation of higher-order provers has emerged. Their characteristic feature is that they build on the best first-order technologies and extend them. With one of these provers, our system Zipperposition [19], we can now prove the three problematic formulas above in a few seconds as well as much more difficult higher-order problems.

Our leading design principle has been to focus on a *graceful* extension of standard superposition, following Bjarne Stroustrup's zero-overhead principle: "What you don't use, you don't pay for." Accordingly, a strong higher-order prover should behave like a first-order prover on first-order problems, perform mostly like a first-order prover on mildly higher-order problems, and scale up to arbitrary higher-order problems.

In this article, we will retrace the steps that led to the development of our prover Zipperposition and its proof system, called λ-superposition [5]. We start by describing the underlying logical language.

WHAT IS A LOGIC?

Every prover is based on a logic, which can be used to express mathematical definitions and arguments. Indeed, one of the roles of logic is to provide a rigorous foundation for mathematics. Today, even if most mathematicians do not consciously use logic on a daily basis, they feel reassured to know that their definitions and arguments can be encoded in it.

In computer science, we can use logic to specify systems and to express properties about them. Sometimes two languages are used for these two purposes, but with an expressive logic, we can use the same language for both. For example, you can define the syntax and semantics of a programming language in a logic and then provide rigorous answers to questions such as "Is the language deterministic?" and "Is its type system sound?"

There are many logics to choose from. We briefly look at two: first-order logic and higher-order logic. Each comes with a concept of provability. By definition, a theorem is a provable formula—a formula that has a proof (whether known or not).

We start with (classical) *first-order logic*, also called predicate logic. It constitutes a sweet spot between simplicity and expressiveness. It forms the foundation of set theory, the traditional foundation of mathematics. It is also the logic underlying most automatic theorem provers. First-order logic is often what people refer to when they simply say "logic."

First-order logic distinguishes between terms and formulas. We can think of the formulas as Boolean expressions, whereas terms denote other values (e.g., numbers, lists, matrices). A term can be a variable x, a constant c, or a function f applied to arguments $\mathsf{f}(a_1, \ldots, a_n)$, where the arguments are themselves terms. Terms are used as arguments of predicates $\mathsf{p}(a_1, \ldots, a_n)$ to form formulas, which represent Boolean values. Formulas can also be predicates p without arguments, or they can be constructed from other formulas by combining them using logical operators such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$ as well the quantifiers $\forall x.\ \ldots$ and $\exists x.\ \ldots$. The equality predicate is sometimes also considered part of first-order logic. It is written in infix notation: $a_1 = a_2$. In addition, modern formulations of first-order logic often include a concept of type, similar to that found in programming languages. Then every term and subterm has a type, and only type-correct formulas are allowed. Figure 1 presents a first-order formula.

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

Figure 1: A puzzle in first-order logic

First-order logic has many limitations that curtail its expressiveness. A more expressive alternative is (classical) *higher-order logic*. "Higher-order" means that $\forall$ and $\exists$ may range over functions, and functions may be passed as arguments to other functions. Higher-order logic contains pure functional programming (as in F#, Haskell, and OCaml) as a fragment. In particular, functions are represented in curried form: They are passed one argument at a time, and partial applications are allowed. Unlike in most accounts of first-order logic, Booleans are a type, and formulas are simply terms of Boolean type. Predicates are then simply functions that return Booleans.

A key feature of higher-order logic is its support for anonymous functions $x \mapsto f(x)$ via the syntax $(\lambda x.\, f(x))$. For example, $(\lambda n.\, n + 1)$ maps any integer to its successor. What makes λ so special is that it is automatically evaluated when given an argument; for example, passing k to $(\lambda n.\, n + 1)$ yields the term $\mathsf{k} + 1$. Such implicit conversions are very convenient, but they are also the source of most difficulties when we develop tools based on higher-order logic.

The λ construct is a *binder*, because it introduces a variable that is bound within its body. Other binders can be built on top of λ. For example, you can code the sum $\sum_{i=1}^{n} \mathsf{g}(i)$ as $\mathsf{sum}(1, n, (\lambda i.\, \mathsf{g}(i)))$, and similar encodings are possible for the integral $\int_{x=0}^{\pi} \sin(x)\, dx$ and the set comprehension $\{x \mid x > \mathsf{e}\}$. With its support for binders, higher-order logic provides a convenient language for doing mathematics.

Figure 2 gives a flavor of higher-order logic by presenting a formulation of Cantor's theorem. The second line states that there exists no surjective function from any type τ to its powerset, expressed as the function space $\tau \to \mathsf{bool}$. The first line defines surjectivity. Notice that the surjective predicate takes a function as argument and that it is defined using a λ. Higher-order provers can prove this formula automatically.

PROVING BY SATURATION

There are many techniques for finding proofs in first- and higher-order logic—some more suitable for humans to use, others for computers. A very successful computer-oriented approach to theorem

$$\begin{aligned}&\mathsf{surjective} = (\lambda f.\, \forall y.\, \exists x.\, y = f(x))\\ \Rightarrow\ &\neg(\exists f : \tau \to (\tau \to \mathsf{bool}).\, \mathsf{surjective}(f))\end{aligned}$$

Figure 2: Cantor's theorem in higher-order logic

proving, which lies at the heart of Zipperposition and many other automatic provers, is to *saturate* the input problem—in other words, to exhaustively draw consequences until a contradiction is found. Saturation is a form of proof by contradiction. It consists of three steps:

(1) Put the negation of the statement to prove F and any necessary hypotheses $H_1, \ldots, H_n$ in some suitable normal form, yielding a formula set $\mathcal{F}$.
(2) Repeatedly apply an inference rule from formulas in $\mathcal{F}$ and add the inference's conclusion to $\mathcal{F}$.
(3) Stop if the false formula, denoted by $\bot$, is in $\mathcal{F}$.

In other words, a saturation prover starts with a set of formulas and expands it with consequences until $\bot$ has been derived. At that point, it has established $(H_1 \wedge \cdots \wedge H_n \wedge \neg F) \Rightarrow \bot$, which is equivalent to $(H_1 \wedge \cdots \wedge H_n) \Rightarrow F$ in classical logic. Compared with other theorem proving methods, an appealing aspect of saturation is that it never needs to backtrack.

With the above procedure, the set of formulas keeps on growing and can quickly explode. Fortunately, the prover may also remove *redundant* formulas from $\mathcal{F}$. For example, if $\mathcal{F}$ contains the formula p, the weaker formula p ∨ q might be considered redundant and removed.

Saturation is a framework can be used in tandem with many proof calculi. To use it, we need to specify the normal form of our formulas, the inference rules, and possibly rules to remove redundant formulas.

The archetypical example of a saturating proof calculus is first-order *resolution*, introduced by Alan Robinson in 1965 [15]. Resolution provers first put the problem into clausal normal form—a big conjunction of disjunctions:

$$(L_{11} \vee \cdots \vee L_{1n_1}) \wedge \cdots \wedge (L_{m1} \vee \cdots \vee L_{1n_m})$$

The quantifiers ∀ and ∃ do not appear in this formula; they are eliminated using a technique called Skolemization. Each disjunction $L_{i1} \vee \cdots \vee L_{in_i}$ is called a *clause*. Each component L_{ij} is called a *literal* and has the form either $\mathsf{p}(a_1, \ldots, a_n)$ or $\neg\mathsf{p}(a_1, \ldots, a_n)$, written p or ¬p if $n = 0$. The order of literals in a clause is immaterial; thus, p(a) ∨ ¬q and ¬q ∨ p(a) are the same clause. In addition, a clause is allowed to have no literals at all. This empty clause is false. Thus we denote it by $\bot$.

A resolution prover is an instance of a saturating prover. Clauses take the role of formulas in the saturation framework described above. The main inference rule is called resolution. If we forget for a moment that clauses may contain variables, the inference rule can be presented in this way:

$$\frac{\mathcal{D} \vee L \qquad C \vee \neg L}{\mathcal{D} \vee C}\ \textsc{Res}$$

There is a lot to explain here:

- The label on the right, Res, is the rule's short name.
- The inference rule takes two premises, displayed above the horizontal bar. The premises must be present in the set $\mathcal{F}$ for the inference to be applicable.
- The inference rule produces a conclusion, displayed below the horizontal bar. This formula gets added to $\mathcal{F}$ when the inference is performed.
- The notation $\mathcal{D} \vee L$ stands for an arbitrary clause that contains a literal L and all the literals from $\mathcal{D}$. Similarly, $C \vee \neg L$ stands for an arbitrary clause that contains the negation of L as a literal and all the literals from C.
- Finally, $\mathcal{D} \vee C$ denotes the clause obtained by combining the extra literals from $\mathcal{D}$ and C.

The following instance of Res shows the rule in action:

$$\frac{\mathsf{p} \vee \mathsf{q} \qquad \neg\mathsf{q} \vee \mathsf{r} \vee \neg\mathsf{s}}{\mathsf{p} \vee \mathsf{r} \vee \neg\mathsf{s}}\ \textsc{Res}$$

Notice how q and ¬q cancel each other out. An interesting special case of the rule arises when $\mathcal{D}$ and C are the empty clause:

$$\frac{L \qquad \neg L}{\bot}\ \textsc{Res}$$

If $\mathcal{F}$ contains both the clauses L and $\neg L$, then $\mathcal{F}$ is inconsistent, and Res can be used to derive $\bot$, thereby completing the proof by refutation. Having accomplished its mission, the prover then stops.

The resolution calculus enjoys two desirable properties for a proof calculus. First, it is *sound*. This means that the conclusion of an inference is always a logical consequence of the premises. It is easy to convince ourselves that Res is sound.[1] The other desirable property is *completeness*, meaning: If the set $\mathcal{F}$ is inconsistent, then saturation with the resolution calculus will eventually derive $\bot$, as long as inferences are performed fairly. Putting all of this together, we get:

> Given enough resources, a resolution prover will find a proof if and only if the input problem is provable.

Completeness is much more difficult to establish than soundness, and it is also a less important property in practice. Still, it is reassuring to know that our prover has no blind spots.

As an analogy, suppose you are part of a party looking for a treasure on an island. You should make sure that you stay on the island (soundness) and that you explore every square inch of it (completeness).

Figure 3 presents a saturation starting with an inconsistent clause set—clearly, if ¬p and ¬q are true, then p ∨ q is false. The saturation predictably ends with the derivation of the empty clause. The figure's sets capture the consecutive states of $\mathcal{F}$ in the saturation procedure.

The simple resolution calculus presented above can be improved in two ways to make it more useful. First, the calculus is needlessly explosive. In the example of Figure 3, we arbitrarily chose in step 2 to resolve p ∨ q against ¬q, but we could also have resolved it

[1] The justification for the general rule is as follows: Either L or $\neg L$ is true. If L is true, then $\neg L$ is false, and therefore the rest of the second premise, C, must be true. A fortiori, $\mathcal{D} \vee C$ must be true. Similarly, if $\neg L$ is true, then L is false and therefore $\mathcal{D}$ and a fortiori $\mathcal{D} \vee C$ must be true.

1. $\{p \vee q, \neg p, \neg q\}$ initial clauses
2. $\{p \vee q, \neg p, \neg q, p\}$ Res inference $\dfrac{p \vee q \quad \neg q}{p}$
3. $\{\neg p, \neg q, p\}$ redundancy
4. $\{\neg p, \neg q, p, \bot\}$ Res inference $\dfrac{p \quad \neg p}{\bot}$

Figure 3: A saturation using the resolution calculus

against $\neg p$. This kind of freedom might lead the prover to explore an excessively large search space.

To guard against this, we can fix an *order* on the predicates. The order is designed in such a way that syntactically large terms are considered large by the order as well. For terms of the same size, we can use tie-breakers; for example, we can state that p is smaller than q. Then we restrict the Res rule so that it applies only if L and $\neg L$ are larger than any other literal in their respective premises; such literals are called *maximal*. This can be done without loss of generality. It makes saturation much less explosive and is still sound and complete. This optimized calculus is called *ordered resolution* [3].

Although not needed in theory, the order is crucial to obtain good performance in a practical prover. Intuitively, the situation is analogous to when we want to apply a lemma $(H_1 \wedge H_2) \Rightarrow F$ in a pen-and-paper proof. To retrieve the conclusion F, we need to show both H_1 and H_2. Without loss of generality, we can focus on H_1 first. If we fail at proving H_1, there is no point in trying to prove H_2, because we will never manage to use the lemma anyway.

The order not only breaks symmetries in the search space, it also gives the calculus a sense of direction. The calculus tends to work on the syntactically largest part of a clause and to produce smaller clauses, working its way towards the smallest clause possible: the empty clause.

Next to the order, the other improvement to resolution concerns variables. Our Res rule assumes that the problem contains no variables. For variable-free problems, resolution competes with Boolean satisfiability solvers [14], which are better suited for the task. The true strength of resolution lies in its handling of variables. Like in mathematics, these are understood to be $\forall$-quantified. Thus, assuming that the constants a and b and the unary function f are declared, a clause such as $p(x) \vee q(x)$, where x is a variable, represents an infinite set of variable-free clauses

$$\begin{array}{cc} p(a) \vee q(a) & p(b) \vee q(b) \\ p(f(a)) \vee q(f(a)) & p(f(b)) \vee q(f(b)) \\ \vdots & \vdots \end{array}$$

where x is instantiated with all possible terms that can be built from the declared functions and constants.

Following this interpretation of variables, the Res rule can be generalized to allow inferences such as

$$\frac{p(a, y) \qquad \neg p(x, b)}{\bot}\,\textsc{Res}$$

The inference sets x to a and y to b to make the two predicates $p(a, y)$ and $p(x, b)$ identical.

The process of computing an assignment for the variables that make two expressions identical is called *unification*. First-order logic enjoys the nice property that if two expressions are unifiable, there exists a variable assignment that captures that fact in the most general manner possible. This assignment is called the *most general unifier*. Probably Robinson's most celebrated contribution to computer science is an algorithm that decides whether two expressions are unifiable and that computes the most general unifier if it exists.

REASONING ABOUT EQUALITY

The ordered resolution calculus based on unification is sound and complete for a basic version of first-order logic that does not include equality. For many applications, however, we do need to reason about equality. Since resolution does not know about equality, we need to specify its properties explicitly. First, we state that equality is reflexive, symmetric, and transitive:

$$\begin{aligned} \text{reflexivity:}\ & \forall x.\ x = x \\ \text{symmetry:}\ & \forall x, y.\ x = y \Rightarrow y = x \\ \text{transitivity:}\ & \forall x, y, z.\ (x = y \wedge y = z) \Rightarrow x = z \end{aligned}$$

In addition, we state that we can replace equals with equals in function arguments, a property called *congruence*. For example, $\forall x, y.\ x = y \Rightarrow f(x) = f(y)$. We need to add formulas like this for every function f and for every predicate p as well. Specifying equality in this way is inefficient because the great number of formulas needed to describe it gives rise to lots of unnecessary resolution inferences, making it harder to find a contradiction.

The *superposition* calculus, first described in its entirety by Bachmair and Ganzinger in 1994 [2], generalizes resolution to reason about equality efficiently with inference rules specifically tailored to this predicate. Once we have equality, other predicates p are not necessary; they can be encoded as functions returning the Boolean values true or false. Accordingly, literals are either equalities $s = t$ or disequalities $s \neq t$. To establish *symmetry* without even introducing an additional formula or inference rule, we consider $s = t$ to be the same literal as $t = s$ and $s \neq t$ to be the same literal as $t \neq s$.

The two main rules of the superposition calculus are the superposition rule Sup and the equality resolution rule EqRes. In a simplified form, they can be stated as follows:

$$\frac{\mathcal{D} \vee t = t' \qquad C \vee s[t] \neq s'}{\mathcal{D} \vee C \vee s[t'] \neq s'}\,\textsc{Sup} \qquad \frac{C \vee s \neq s}{C}\,\textsc{EqRes}$$

The Sup rule is reminiscent of Res. The right premise contains a literal $s[t] \neq s'$ and possibly some other literals C. The notation $s[t]$ stands for a term s that contains t as a subterm. If we assume for a moment that there are no extra literals $\mathcal{D}$ and C, the left premise is an equation $t = t'$ that we use to replace t by t' in the right premise $s[t] \neq s'$. In essence, we replace equals by equals, thus establishing *transitivity* and *congruence*. The extra literals $\mathcal{D}$ and C encode some conditions under which the equation and the disequation hold—for example, "$\mathcal{D} \vee t = t'$" can be read as "If $\mathcal{D}$ is false, then $t = t'$". Adding the literals from $\mathcal{D}$ and C to the conclusion is thus necessary to make the rule sound. We have only shown the variant for disequations above, but in its full generality, Sup also allows replacing terms in premises of the form $C \vee s[t] = s'$.

The rule EqRes allows us to remove any literal of the form $s \neq s$ because such a literal is clearly false and therefore it has no impact on the truth of the clause. This rule establishes *reflexivity*.

Like ordered resolution, superposition uses an order to prune the search space, working on larger literals first. In addition, Sup is restricted so that t and $s[t]$ must correspond to the larger sides of equations and disequations, and it replaces larger terms by smaller terms. Superposition is sound and complete.

To illustrate the superposition calculus, we will use it to prove the following lemma:

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

In ordinary mathematical notation, we would write: Assuming that $x^{-1} = 1/x$ for all $x \neq 0$ and that $\pi \neq 0$, we have $|\pi^{-1}| = |1/\pi|$.

Negating and clausifying yields the following clauses:

$$x = \mathsf{zero} \ \vee\ \underline{\mathsf{div}(\mathsf{one}, x)} = \mathsf{inv}(x) \tag{1}$$

$$\underline{\mathsf{pi}} \neq \mathsf{zero} \tag{2}$$

$$\underline{\mathsf{abs}(\mathsf{div}(\mathsf{one}, \mathsf{pi}))} \neq \mathsf{abs}(\mathsf{inv}(\mathsf{pi})) \tag{3}$$

In each clause, underlining indicates the maximal sides of the maximal literals according to a reasonable order. In our examples, maximal simply means syntactically largest. If we take x to be pi, the underlined term in clause (1) matches a subterm of the underlined term in clause (3). Thus, we can apply Sup using as premises the instance

$$\mathsf{pi} = \mathsf{zero} \ \vee\ \underline{\mathsf{div}(\mathsf{one}, \mathsf{pi})} = \mathsf{inv}(\mathsf{pi})$$

of clause (1) together with clause (3) to obtain

$$\mathsf{pi} = \mathsf{zero} \ \vee\ \underline{\mathsf{abs}(\mathsf{inv}(\mathsf{pi}))} \neq \underline{\mathsf{abs}(\mathsf{inv}(\mathsf{pi}))} \tag{4}$$

Next, we apply EqRes to clause (4) to generate

$$\underline{\mathsf{pi}} = \mathsf{zero} \tag{5}$$

Now that we have eliminated the larger literal of (4), we may work on the remaining smaller literal. There are multiple Sup inferences possible—e.g., between clauses (5) and (3) or between (5) and (4). The inference that leads to the desired contradiction, however, is a Sup inference between clauses (5) and (2) that produces

$$\underline{\mathsf{zero}} \neq \underline{\mathsf{zero}} \tag{6}$$

If the prover is fair, it will eventually perform this inference. Which inferences should be prioritized is an important heuristic choice in practice; this is one of the factors that distinguishes good from bad provers. Finally, an EqRes inference on clause (6) yields the empty clause, which proves that the original lemma holds.

LOST IN TRANSLATION

Resolution and superposition target first-order logic. First-order provers work well on first-order problems, but most proof assistants are higher-order. The gap between the two logics is bridged by tools called *hammers* [9].

When users of a proof assistant want to prove a formula, they can develop a detailed proof using the proof assistant's language, or they can invoke a hammer to dispatch the formula to automatic provers. The hammer (1) heuristically selects suitable lemmas from the thousands available, (2) translates them to first-order logic along with the formula to prove, and (3) imports any found proof back into the proof assistant.

Unfortunately, in practice, a lot is lost in step (2). The translation of higher-order constructs is heavy and leads to poor performance. Usually, hammers will succeed if the original problem is first-order, but they will almost always fail if some higher-order reasoning is needed.

Consider the simple formula

$$\textstyle\sum_{i=1}^{n}\big(i^2 + 2i + 1\big) = \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} 2i + \sum_{i=1}^{n} 1$$

from the introduction. In higher-order logic, the formula can be proved by rewriting the left-hand side with the distributivity rule

$$\forall f, g.\ \textstyle\sum_{i=1}^{n}\big(f(i) + g(i)\big) = \sum_{i=1}^{n} f(i) + \sum_{i=1}^{n} g(i)$$

to obtain $\sum_{i=1}^{n}\big(i^2 + 2i\big) + \sum_{i=1}^{n} 1$, followed by a second rewrite step to produce the desired right-hand side. In a first-order setting, the λ's must be encoded. A complete translation scheme is based on so-called *combinators*. Following this scheme, the formula to prove is translated to the bulky first-order equation

$$\begin{aligned}&\mathsf{sum}(1, n, \mathsf{C}(\mathsf{B}(\mathsf{plus}, \mathsf{S}(\mathsf{B}(\mathsf{plus}, \mathsf{C}(\mathsf{power}, 2)), \mathsf{app}(\mathsf{times}, 2))), 1))\\ =\ &\mathsf{app}(\mathsf{app}(\mathsf{plus}, \mathsf{app}(\mathsf{app}(\mathsf{plus}, \mathsf{sum}(1, n, \mathsf{C}(\mathsf{power}, 2))),\\ &\quad \mathsf{sum}(1, n, \mathsf{app}(\mathsf{times}, 2)))), \mathsf{sum}(1, n, \mathsf{K}(1)))\end{aligned}$$

where $\mathsf{app}(f, x)$ applies an encoded curried function f to an argument x. The combinators B, C, K, and S are characterized by equations—e.g., $\mathsf{app}(\mathsf{S}(f, g), x) = \mathsf{app}(\mathsf{app}(f, x), \mathsf{app}(g, x))$ for S. This approach produces a lot of clutter. The provers need to do extensive reasoning about the combinators just to determine whether an equation is applicable. As a result, hammers fail to prove the formula of interest in a reasonable time in practice, let alone more difficult formulas.

The solution is to adapt the provers and their underlying calculi to understand higher-order logic directly, thus avoiding translation altogether.

HIGHER-ORDER CHALLENGES

Going native, however, is easier said than done. Higher-order logic is inherently more complex than first-order logic. For decades, several aspects of the superposition calculus were considered problematic in a higher-order context, blocking progress:

The Order Used to Compare Terms For the first-order superposition calculus to be complete, the underlying order used to compare terms must meet certain requirements. One of these requirements is that a term occurring as a subterm of another term must be smaller than the surrounding term. For example, the term c must be considered smaller than g(c), which in turn must be considered smaller than f(g(c)).

A second requirement on the order is that putting two terms into a surrounding context should not affect how they are oriented. For example, if b is greater than a, then f(b, c) must be greater than f(a, c); adding the shared context "f(..., c)" should not affect the order.

For higher-order logic, these requirements cannot be met because the implicit reductions of λ's can radically change a term's appearance. For example, applying the function $(\lambda y.\ \mathsf{f}(\lambda z.\ z))$ to the argument $\mathsf{g}(\mathsf{f}(\lambda z.\ z))$ yields the term $\mathsf{f}(\lambda z.\ z)$, whereas applying a smaller function $(\lambda z.\ z)$ to the same argument yields

a *larger* term $\mathsf{g}(\mathsf{f}(\lambda z.\ z))$. This violates the second requirement with the context "$\ldots\ (\mathsf{g}(\mathsf{f}(\lambda z.\ z)))$".

Unification of Terms In first-order logic, Robinson's unification algorithm decides whether two terms can be made identical by assigning terms to variables and if so, computes the most general unifier—an assignment that subsumes all other possible assignments. For example, to unify $\mathsf{f}(x)$ and $\mathsf{f}(y)$, we must set x and y to the same value. Since there are infinitely many possible terms, there are infinitely many possible assignments that unify the two terms. However, we can assign y to x (or x to y) to describe all of these assignments in one. This is the most general unifier.

In higher-order logic, when two terms are unifiable, a most general unifier does not necessarily exist. For example, to unify $y(\mathsf{f}(\mathsf{a}))$ and $\mathsf{f}(y(\mathsf{a}))$, we could set y to any of the following infinitely many terms: $(\lambda x.\ x)$, $(\lambda x.\ \mathsf{f}(x))$, $(\lambda x.\ \mathsf{f}(\mathsf{f}(x)))$, ... After unification, this would give the terms $\mathsf{f}(\mathsf{a})$, $\mathsf{f}(\mathsf{f}(\mathsf{a}))$, $\mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{a})))$, ... Unlike above, these terms cannot be described using a single assignment that captures all of them and only them; to describe them all, we need a set of assignments. Therefore, the higher-order concept that corresponds to the most general unifier is a *complete set of unifiers*: a set of unifiers that subsumes all unifiers. The example above shows that this set can sometimes be infinite.

In addition, even the question of whether two higher-order terms are unifiable at all is undecidable. So for any algorithm we devise, there will be nonunifiable terms that it cannot determine to be nonunifiable. Instead, the algorithm may run forever.

Handling of Booleans First-order logic distinguishes between terms, which are non-Boolean expressions, and formulas, which constitute a Boolean skeleton surrounding the terms. This makes conversion into clausal normal form simple: We only need to convert the skeleton.

In higher-order logic, Booleans can occur within any term. Expressions such as $(\text{if } \exists x.\ \mathsf{p}(x) \text{ then a else b})$ are supported. Even more difficult to clausify are Booleans that occur below λ's as in $(\lambda x.\ \text{if } \mathsf{p}(x) \text{ then a else b})$. For such expressions, the first-order approach of converting all Booleans into clausal form in a preprocessing step is not available. We cannot eliminate all Boolean terms in preprocessing.

GOING HIGHER-ORDER

Facing the above challenges, the automated reasoning community focused on alternatives to superposition. The prevailing orthodoxy was that higher-order logic is very different from first-order logic and hence it must be tackled using different methods. For years, the performance of higher-order provers was disappointing on proof assistant benchmarks [18].

But now there is a new generation of higher-order provers that aim to gracefully generalize their first-order counterparts. They give up the dogma that higher-order logic is fundamentally different from first-order logic. Instead, they regard first-order logic as a fragment of higher-order logic for which we have efficient methods, and they start from that position of strength.

Possibly the most successful of these systems is our prover Zipperposition. Its underlying calculus, λ-superposition, provides answers to the challenges outlined above:

The Order Used to Compare Terms As they are, the order requirements of the superposition calculus cannot be achieved in higher-order logic. Ultimately, the order requirements are dictated by the calculus. When the Sup inference operates on a subterm with a context around it, we need the order to behave well with respect to that context. We saw earlier that the order requirements are met for the first-order-like context "$\mathsf{f}(\ldots, \mathsf{c})$" but not for the specifically higher-order context "$\ldots\ (\mathsf{g}(\mathsf{f}(\lambda z.\ z)))$".

The solution is to design the λ-superposition calculus so that it avoids operating in troublesome contexts. For example, if the problem consists of the clauses $\mathsf{g} = \mathsf{f}$ and $\mathsf{g}(\mathsf{a}) \neq \mathsf{f}(\mathsf{a})$, it would notice that the context $\ldots(\mathsf{a})$ around g is troublesome and avoid rewriting the g in $\mathsf{g}(\mathsf{a})$ into f. By restricting rewriting to first-order-like contexts, we can use similar orders to compare terms as in first-order logic. This addresses the order issue.

However, we seem to have thrown out the baby with the bathwater: For this example, we are blocked, even though the two clauses are obviously contradictory. The solution is to introduce a new inference rule that transforms the equation $\mathsf{g} = \mathsf{f}$ into $\mathsf{g}(x) = \mathsf{f}(x)$. The new equation can then be used to rewrite $\mathsf{g}(\mathsf{a}) \neq \mathsf{f}(\mathsf{a})$ into $\mathsf{f}(\mathsf{a}) \neq \mathsf{f}(\mathsf{a})$, as in first-order logic. In simplified form, the new inference rule—called argument congruence—can be stated as follows:

$$\frac{C \vee s = s'}{C \vee s(x) = s'(x)}\ \textsc{ArgCong}$$

Unification of Terms As for the inconvenient explosive behavior of higher-order unification, we cannot avoid it and must adapt our prover architecture to cope with it. To deal with the potentially infinite sets of unifiers, we have devised an implementation of the unification algorithm that regularly pauses and returns control to the main program [19]. The main program may decide to perform inferences or to focus on other unification problems first before resuming to work on the given unification problem. To ensure completeness, we must be careful not to ignore any inference or unification problem indefinitely. This can be achieved by interweaving the different computations (i.e., by dovetailing).

Handling of Booleans If we do not eliminate Boolean terms in preprocessing, we need to interleave clausification with the main proving procedure to eliminate Booleans lazily. We accomplish this by means of dedicated inference rules that perform clausification steps. This idea is not new; it is present in several higher-order provers, and even in some first-order provers. The main difficulty is to find suitable side conditions for the rules (e.g., order conditions). The conditions should restrict the rules as much as possible to prevent an explosion of new clauses, but not too much to prevent the calculus from becoming incomplete.

Our λ-superposition calculus that implements these ideas is sound and complete, although both claims come with some caveats. Since we have rules for clausification in our calculus, also the elimination of $\forall$ and $\exists$ is performed by inference rules. These rules may introduce new constants, called Skolem constants. For example the term $\exists x.\ \mathsf{p}(x)$ can be transformed into $\mathsf{p}(\mathsf{c})$, where c is a new Skolem constant—if there exists an x such that $\mathsf{p}(x)$, then we may postulate the existence of a c such that $\mathsf{p}(\mathsf{c})$. Technically, however, this rule is not strictly speaking sound, because $\mathsf{p}(\mathsf{c})$ is not a logical

consequence of $\exists x.\ \mathrm{p}(x)$; after all, $\exists x.\ \mathrm{p}(x)$ says nothing about c. Even so, the Skolem constants cannot introduce a logical contradiction from noncontradictory assumptions, and it is even possible to give them a semantics that actually makes Skolemization sound. This is sufficient for our prover to work correctly.

Regarding completeness, the caveat is that as a consequence of Gödel's first incompleteness theorem, there exist true statements for which there are no proofs. Naturally, our calculus cannot prove these statements, but nor can traditional higher-order proof systems or proof assistants—these are theorem provers, not truth provers. To avoid this issue, our completeness result is stated with respect not to the standard semantics of higher-order logic, which is subject to the worst consequences of Gödel's incompleteness theorem, but to a weaker notion called Henkin semantics, which considers only those statements to be true that have a proof. Thus, if a problem is *provably* inconsistent (according to a traditional notion of provability), our calculus can be used to derive the empty clause $\bot$.

To demonstrate the λ-superposition calculus, we will use it to prove the equation

$$\textstyle\sum_{i=1}^{n}\big(i^2 + 2i + 1\big) = \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} 2i + \sum_{i=1}^{n} 1$$

from the introduction. The only axiom we need is the distributivity law

$$\textstyle\sum_{i=1}^{n}\big(f(i) + g(i)\big) = \sum_{i=1}^{n} f(i) + \sum_{i=1}^{n} g(i)$$

where the variables f and g range over functions. First, for a proof by contradiction, we negate the formula we want to prove. In addition, we express the sums using λ's. The problem becomes

$$\begin{aligned}&\mathsf{sum}(1, n, (\lambda i.\ i^2 + 2i + 1))\\ \neq\ &\underline{\mathsf{sum}(1, n, (\lambda i.\ i^2)) + \mathsf{sum}(1, n, (\lambda i.\ 2i)) + \mathsf{sum}(1, n, (\lambda i.\ 1))}\end{aligned} \tag{7}$$

$$\begin{aligned}&\mathsf{sum}(1, n, (\lambda i.\ f(i) + g(i)))\\ =\ &\underline{\mathsf{sum}(1, n, (\lambda i.\ f(i))) + \mathsf{sum}(1, n, (\lambda i.\ g(i)))}\end{aligned} \tag{8}$$

As before, underlining indicates the maximal sides of the maximal literals according to a reasonable order. If we take f to be $(\lambda x.\ x^2)$ and g to be $(\lambda x.\ 2x)$, the right-hand side of clause (8) matches the term $\mathsf{sum}(1, n, (\lambda i.\ i^2)) + \mathsf{sum}(1, n, (\lambda i.\ 2i))$ in clause (7). Here it is crucial that λ's automatically reduce—e.g., instantiating f with $(\lambda x.\ x^2)$ in $(\lambda i.\ f(i))$ first results in $(\lambda i.\ (\lambda x.\ x^2)(i))$ but then reduces to $(\lambda i.\ i^2)$. Thus, a Sup inference between clauses (7) and (8) is possible and yields

$$\begin{aligned}&\mathsf{sum}(1, n, (\lambda i.\ i^2 + 2i + 1))\\ \neq\ &\underline{\mathsf{sum}(1, n, (\lambda i.\ i^2 + 2i)) + \mathsf{sum}(1, n, (\lambda i.\ 1))}\end{aligned} \tag{9}$$

Next, we need to use clause (8) a second time. If we take f to be $(\lambda x.\ x^2 + 2x)$ and g to be $(\lambda x.\ 1)$, the right-hand side of clause (8) matches the right-hand side of clause (9). A Sup inference between clauses (8) and (9) yields

$$\underline{\mathsf{sum}(1, n, (\lambda i.\ i^2 + 2i + 1))} \neq \underline{\mathsf{sum}(1, n, (\lambda i.\ i^2 + 2i + 1))} \tag{10}$$

Now both sides are equal, and an EqRes inference yields the empty clause, proving the original equation. This derivation is typical for the superposition calculus: To prove that an equality follows from other equalities, we state it as a disequality, we apply Sup inferences on either sides of $\neq$ until both sides are identical, and finally we apply EqRes to derive $\bot$.

This native approach is fast and direct. In contrast, with the first-order encoding of λ's as combinators discussed above, two Sup inferences would not have sufficed. We would have needed several Sup inferences with the equations characterizing the combinators to get the terms in the negated lemma and the distributivity rule to match. In addition, each Sup inference would have competed with many others that are also applicable. The automatic reduction of λ's is what enables λ-superposition to prove this example equation efficiently. It replaces blind exploration by directed computation.

The Zipperposition prover served primarily as an experimentation vehicle for λ-superposition. Now that we are convinced of its value, we have started reimplementing its key ideas in the high-performance E prover [16].

ALTERNATIVE APPROACHES

In half a century of native higher-order calculus and prover development, many approaches have been proposed. We briefly review the main ones.

Resolution Higher-order automated reasoning really took off with Gérard Huet's extension of resolution to higher-order logic [11]. The key idea was to use higher-order unification, which Huet had to develop as well [12], in place of Robinson's first-order unification. The main limitation of higher-order resolution is that it does not natively support equality reasoning. This was added by Christoph Benzmüller and his colleagues in the Leo series of provers: LEO [6], Leo-II [7], and Leo-III [17].

Superposition Leo-III implements a calculus that is quite similar to an incomplete version of λ-superposition. Its predecessors LEO and Leo-II, on the other hand, did not feature an order to compare terms, leading to many unnecessary inferences. Another related approach consists in using combinators to represent λ-expressions, in combination with a superposition-like calculus. This is the route taken by Ahmed Bhayat and Giles Reger in Vampire [8].

Matings Another approach that has been studied extensively relies on matings, or connections. Peter Andrews's TPS [1] is based on this approach. The approach translates logical formulas to matrices and looks for paths with certain properties in these matrices. Once the prover successfully closes a path, a contradiction is found. The calculus is comparatively easy to implement, but it does not scale very well, especially in the presence of superfluous axioms.

Tableaux Roughly speaking, the tableau calculus emulates what a rigorous mathematician would do on paper. It analyzes the formula systematically, building a tree that keeps track of the remaining cases to prove. The Satallax prover [10] developed by Chad Brown and colleagues implements a higher-order tableau calculus guided by a satisfiability (SAT) solver. Tableaux are simple and intuitive, and they can work directly on unclausified formulas. On the negative side, they can end up repeating work in separate branches of the tree, harming performance, and equality reasoning is not their strong suit. A related approach, called focused sequent calculus, is implemented in Fredrik Lindblad's agsyHOL system [13].

Satisfiability Modulo Theories Solvers based on satisfiability modulo theories (SMT) have enjoyed a lot of success for verifying hardware and software in the past two decades. Some support

Table 1: Empirical prover comparison

	CASC 2021 (500 problems)	Seventeen (5000 problems)
agsyHOL	–	814
cvc5	239	2522
E	300	2557
Leo-II	95	–
Leo-III	357	1632
Satallax	–	1695
Vampire	386	2179
Zipperposition	**467**	**2718**

full first-order logic, including quantifiers. Among these, the solvers cvc5 and veriT have been extended by Barbosa et al. [4] to provide some support for higher-order logic. The strength of SMT solvers is their native support for decidable theories, such as linear arithmetic on integers and real numbers. Their main weakness is their handling of quantifiers, for which they rely heavily on trial and error.

PERFORMANCE ON BENCHMARKS

Designing a proof calculus that is sound and complete is relatively easy. The real difficulty is to design a calculus that is sound, complete, and efficient. In theorem proving, the proof of the pudding is in the eating. To give a flavor of the performance of the different calculi and provers, we present the results of two empirical evaluations. Table 1 summarizes the results.

The first evaluation is the higher-order theorem division of the CADE ATP System Competition, or CASC. This competition takes place every year at the CADE or IJCAR conference. The middle column of Table 1 presents the number of proved problems, out of 500 problems, by each prover that participated in the 2021 edition of CASC.[2] The problems come from a variety of sources and aim to be representative of the applications of automatic provers. A wall-clock time limit of 120 s per problem was used. Zipperposition won (indicated in bold), and Vampire came in second.

The second evaluation is an unpublished study by Martin Desharnais, Makarius Wenzel, and two of the present authors, called Seventeen.[3] For this study, 5000 problems were generated based on a proof assistant's library. These problems tend to be larger but less higher-order than the CASC problems. A CPU time limit of 30 s was used. Zipperposition won again, followed by E. All in all, the competition confirms the newly won supremacy of superposition-based approaches.

PROSPECTS

For many years, higher-order reasoning was something of a fringe topic. Perhaps researchers thought that first-order logic was already sufficiently challenging. In recent years, the picture has changed dramatically with the emergence of higher-order provers based on superposition and SMT. The leading approaches for reasoning about classical first-order logic are finally finding their way into higher-order provers. These provers are integrated in hammers, providing valuable automation to users of proof assistants.

One possible criticism of higher-order provers is that they still automate only what mathematicians would consider very easy or trivial—a far cry from the dream of fully mechanical mathematicians. However, as users of proof assistants know, a lot of time is wasted proving very easy or trivial results. Having little mechanical mathematicians that take care of the easy mathematics means that users can focus on more challenging, and interesting, problems.

We see three main avenues for future work. First, SMT is a very successful technology in the first-order world, and we expect it to catch up with λ-superposition. Second, higher-order logic can express induction principles natively, but so far little work has gone into automating higher-order proofs by induction. Third, several proof assistants, such as Coq and Lean, use a logic more powerful than higher-order logic—called dependent type theory; extending higher-order proof calculi to support this logic would increase proof automation for these systems.

ACKNOWLEDGMENTS

We first thank Simon Cruanes for allowing us to extend his prover Zipperposition and helping us in the process. We thank the entire Matryoshka team, including Pascal Fontaine, Stephan Merz, and Stephan Schulz, for the fruitful collaboration. We thank the other researchers working on automating higher-order logic, including Christoph Benzmüller, Ahmed Bhayat, Chad Brown, Giles Reger, and Alexander Steen, for stimulating discussions over the years. We thank Wan Fokkink, for suggesting that we write this article. Finally, we thank Anne Baanen, Lukas Bentkamp, Vincent François-Lavet, Valentin Hartmann, and Mark Summerfield for suggesting textual improvements.

The research on λ-superposition has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). It has also received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward). Bentkamp's work on this article has received funding from a Chinese Academy of Sciences President's International Fellowship for Postdoctoral Researchers (grant No. 2021PT0015).

REFERENCES

[1] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. 1996. TPS: A theorem-proving system for classical type theory. *J. Autom. Reason.* 16, 3 (1996), 321–353.
[2] Leo Bachmair and Harald Ganzinger. 1994. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.* 4, 3 (1994), 217–247.
[3] Leo Bachmair and Harald Ganzinger. 2001. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, John Alan Robinson and Andrei Voronkov (Eds.). Vol. I. Elsevier and MIT Press, 19–99.
[4] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark W. Barrett. 2019. Extending SMT solvers to higher-order logic. In *CADE-27 (LNCS, Vol. 11716)*, Pascal Fontaine (Ed.). Springer, 35–54.
[5] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirović. 2021. Superposition for full higher-order logic. In *CADE-28 (LNCS, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 396–412.
[6] Christoph Benzmüller and Michael Kohlhase. 1998. Extensional higher-order resolution. In *CADE-15 (LNCS, Vol. 1421)*, Claude Kirchner and Hélène Kirchner (Eds.). Springer, 56–71.
[7] Christoph Benzmüller, Nik Sultana, Lawrence C. Paulson, and Frank Theiss. 2015. The higher-order prover Leo-II. *J. Autom. Reason.* 55, 4 (2015), 389–404.

[2] http://www.tptp.org/CASC/28/WWWFiles/DivisionSummary1.html
[3] https://matryoshka-project.github.io/pubs/seventeen.pdf

[8] Ahmed Bhayat and Giles Reger. 2020. A combinator-based superposition calculus for higher-order logic. In *IJCAR 2020, Part I (LNCS, Vol. 12166)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 278–296.

[9] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *J. Formaliz. Reason.* 9, 1 (2016), 101–148.

[10] Chad E. Brown. 2012. Satallax: An automatic higher-order prover. In *IJCAR 2012 (LNCS, Vol. 7364)*, Bernhard Gramlich, Dale Miller, and Uli Sattler (Eds.). Springer, 111–117.

[11] Gérard P. Huet. 1973. A mechanization of type theory. In *IJCAI-73*, Nils J. Nilsson (Ed.). William Kaufmann, 139–146.

[12] Gérard P. Huet. 1975. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.* 1, 1 (1975), 27–57.

[13] Fredrik Lindblad. 2014. A focused sequent calculus for higher-order logic. In *IJCAR 2014 (LNCS, Vol. 8562)*, Stéphane Demri, Deepak Kapur, and Christoph Weidenbach (Eds.). Springer, 61–75.

[14] Sharad Malik and Lintao Zhang. 2009. Boolean satisfiability: From theoretical hardness to practical success. *Commun. ACM* 52, 8 (2009), 76–82.

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

[16] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. 2019. Faster, higher, stronger: E 2.3. In *CADE-27 (LNCS, Vol. 11716)*, Pascal Fontaine (Ed.). Springer, 495–507.

[17] Alexander Steen and Christoph Benzmüller. 2018. The higher-order prover Leo-III. In *IJCAR 2018 (LNCS, Vol. 10900)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). Springer, 108–116.

[18] Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. 2013. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic* 11, 1 (2013), 91–102.

[19] Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. 2021. Making higher-order superposition work. In *CADE-28 (LNCS, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 415–432.