Complete and Efficient Higher-Order Reasoning via Lambda-Superposition

Alexander Bentkamp, Heinrich-Heine-Universität Düsseldorf, Germany
Jasmin Blanchette, Ludwig-Maximilians-Universität München, Germany
Visa Nummelin, Vrije Universiteit Amsterdam, the Netherlands
Sophie Tourret, Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
Uwe Waldmann, Max-Planck-Institut für Informatik, Saarland Informatics Campus, Germany

Superposition is a highly successful proof calculus for reasoning about first-order logic with equality. We present λ-superposition, which extends superposition to higher-order logic. Its design goals include soundness, completeness, efficiency, and gracefulness with respect to standard first-order superposition. The calculus is implemented in two automatic theorem provers: E and Zipperposition. These provers regularly win trophies at the CADE ATP System Competition, confirming the calculus's applicability. This paper is a summary of research that took place between 2017 and 2022.

1. INTRODUCTION

Higher-order logic (also called simple type theory) [Church 1940; Gordon and Melham 1993; Andrews 2002] is a rich logic that generalizes classical first-order logic to allow functions as first-class objects. The term language is that of the simply typed λ-calculus. Terms are considered syntactically equal modulo α-, β-, and η-conversion—for example, $(\lambda x.\ x)\ \mathsf{c}$ is syntactically equal to c. A formula or proposition is simply a term of Boolean type. Some formulations of higher-order logic allow polymorphism. To aid readability, binary connectives and other operators are normally written in infix syntax, and the universal and existential quantifiers are written using traditional notations. Thus, we write $p \land q$ and $\forall x.\ p\ x$ rather than $\land\ p\ q$ and $\forall\ (\lambda x.\ p\ x)$.

Higher-order logic is widely used as the basis of proof assistants such as HOL4 [Slind and Norrish 2008], HOL Light [Harrison 2009], and Isabelle/HOL [Nipkow et al. 2002]. It is also a fragment of classical dependent type theory, as found for example in Lean [de Moura and Ullrich 2021]. With its native notion of syntactic binding, it is a natural language for expressing a wide range of mathematics.

Consider the following mathematical statements:

$$\left(\sum_{i=1}^{n} i^2 + 2i + 1\right) = \left(\sum_{i=1}^{n} i^2\right) + \left(\sum_{i=1}^{n} 2i\right) + \left(\sum_{i=1}^{n} 1\right)$$
$$f(n) + g(n) \in \mathrm{O}(h(n)) \ \Rightarrow\ f(k(n)) + g(k(n)) \in \mathrm{O}(h(k(n)))$$

Notice that the i's are bound by the big sum operators in the first statement and the n's are bound by the O's in the second statement. The statements can be encoded in higher-order logic as the following formulas, using λ-abstractions to bind variables locally:

$$\mathsf{sum}\ 1\ n\ (\lambda i.\ i \mathbin{\hat{}} 2 + 2 * i + 1) = \mathsf{sum}\ 1\ n\ (\lambda i.\ i \mathbin{\hat{}} 2) + \mathsf{sum}\ 1\ n\ (\lambda i.\ 2 * i) + \mathsf{sum}\ 1\ n\ (\lambda i.\ 1)$$
$$(\lambda n.\ f\ n + g\ n) \in \mathsf{O}\ (\lambda n.\ h\ n) \ \Rightarrow\ (\lambda n.\ f\ (k\ n) + g\ (k\ n)) \in \mathsf{O}\ (\lambda n.\ h\ (k\ n))$$

With suitable definitions and lemmas for the symbols they contain, these formulas are provable.

Proving higher-order formulas automatically is interesting because it increases the productivity of users of proof assistants. One way to prove higher-order formulas, implemented in tools such as Sledgehammer [Paulson and Blanchette 2012], is to encode them in first-order logic—for example, using SKBCI combinators [Turner 1979]. For the two formulas above, this would yield the following:

$$\begin{gathered}\mathsf{sum}\ 1\ n\ (\mathsf{C}\ (\mathsf{B}\ (+)\ (\mathsf{S}\ (\mathsf{B}\ (+)\ (\mathsf{C}\ (\hat{}\,)\ 2))\ ((*)\ 2)))\ 1) \\ = \mathsf{sum}\ 1\ n\ (\mathsf{C}\ (\hat{}\,)\ 2) + \mathsf{sum}\ 1\ n\ ((*)\ 2) + \mathsf{sum}\ 1\ n\ (\mathsf{K}\ 1) \\ \mathsf{S}\ (\mathsf{B}\ (+)\ f)\ g \in \mathsf{O}\ h \Rightarrow \mathsf{S}\ (\mathsf{B}\ (+)\ (\mathsf{B}\ f\ k))\ (\mathsf{B}\ g\ k) \in O\ (\mathsf{B}\ h\ k)\end{gathered}$$

In addition, the definitions of the SKBCI combinators would have to be included in the problem:

$$\mathsf{S}\ x\ y\ z = x\ z\ (y\ z) \qquad \mathsf{K}\ x\ y = x \qquad \mathsf{B}\ x\ y\ z = x\ (y\ z) \qquad \mathsf{C}\ x\ y\ z = x\ z\ y \qquad \mathsf{I}\ x = x$$

To see why the translation works, we can replace S by its closed-term definition $\lambda x\ y\ z.\ x\ z\ (y\ z)$ in the encoded formulas above, and similarly for the other combinators. Up to β-conversion, we obtain the original formulas.

Finally, curried applications should be replaced by a distinguished symbol app—for example, $\mathsf{I}\ x = x$ would become $\mathsf{app}(\mathsf{I}, x) = x$. The result is a first-order problem, which can be given to first-order provers.

In practice, the encoding approach underperforms on genuinely higher-order formulas. In the presence of extraneous axioms (which often cannot be avoided), first-order provers time out on the two examples above. Facing such failures, the user of a proof assistant would have to carry out a manual proof, wasting time.

The obvious alternative to the clumsy encoding of higher-order logic using combinators and app is to go native. The last few years have seen the rise of a new generation of efficient automatic theorem provers for higher-order logic: cvc5 [Barbosa et al. 2019], E [Vukmirović et al. 2023], Leo-III [Steen and Benzmüller 2021], Vampire [Bhayat and Reger 2020a], veriT [Barbosa et al. 2019], and Zipperposition [Vukmirović et al. 2022]. Like their predecessors, these provers can reason natively about higher-order constructs such as λ-abstractions and currying. But unlike their predecessors, they are based on proof calculi modeled after the most successful approaches from the world of first-order logic: superposition [Bachmair and Ganzinger 1994] and satisfiability modulo theories (SMT) [Nieuwenhuis et al. 2006].

In this paper, we present one of these native higher-order proof calculi: λ-superposition [Bentkamp et al. 2023b]. It is a variant of standard superposition and underlies E and Zipperposition, to which we contributed.

Our first design goal has been to extend standard superposition in a *graceful* fashion, following the zero-overhead principle: "What you don't use, you don't pay for" [Stroustrup 1995]. The extension should behave, as much as possible, like standard superposition on formulas belonging to the first-order fragment of higher-order logic, and scale well in the presence of higher-order constructs. We wanted to start from a position of strength, to benefit from the decades of research that had led to superposition.

Other design goals were *soundness* and *refutational completeness*. When designing a calculus, we care about its completeness because a complete calculus is likely to prove more formulas than an incomplete one. We also care because the completeness proof gives invaluable information that guides the development of an efficient calculus. In the same way that superposition is refutationally complete with respect to standard first-order models, λ-superposition is refutationally complete with respect to general (Henkin) models. It is not, however, complete with respect to standard models—by Gödel's incompleteness theorem, this would be too much to ask of a sound calculus for higher-order logic.

A fourth design goal was *efficiency*. Our calculus performs well in practice, both on examples originating from proof assistants [Desharnais et al. 2022; Vukmirović et al. 2023] and at the CADE ATP System Competition (CASC), where E and Zipperposition collectively won six trophies since 2020.

This article is structured as follows. Section 2 presents the standard superposition calculus, after which λ-superposition is modeled. Section 3 presents the completeness proof of standard superposition. Section 4 introduces λ-superposition. Section 5 presents its completeness proof. Section 6 surveys competing approaches. Section 7 reviews the CASC results. Section 8 concludes the article.

This article summarizes our research on λ-superposition. It follows another piece, published in the *Communications of the ACM* [Bentkamp et al. 2023a], which had a similar objective but addressed a wider audience. The present article omits some preliminaries and delves deeper into the metatheory of superposition and λ-superposition, with the emphasis on the completeness proofs.

2. THE SUPERPOSITION CALCULUS

The superposition calculus [Bachmair and Ganzinger 1994] works not directly on formulas of first-order logic but on clauses. The clauses are obtained by putting the problem axioms and the negated conjecture in clausal normal form [Nonnengart and Weidenbach 2001]. From these clauses, the calculus attempts to derive $\bot$, denoting falsehood. A successful refutation amounts to a proof of the original (unnegated) conjecture.

Clauses are defined as follows. Given two first-order terms s, t, the equality $s = t$, viewed as an unordered pair, is an atom. Equality is the only predicate in superposition; other predicates can be encoded as functions, writing $\mathsf{p}(\bar{t}) = \mathsf{true}$ instead of $\mathsf{p}(\bar{t})$. Next, $s = t$ and its negation $s \neq t$ (also viewed as an unordered pair) are literals. A clause $L_1 \vee \cdots \vee L_n$ is then a finite multiset of literals. If $n = 0$, we get the empty clause $\bot$. Notice that by commutativity of the elements of unordered pairs and multisets, the clauses $\mathsf{a} = \mathsf{b} \vee \mathsf{c} = \mathsf{d}$ and $\mathsf{d} = \mathsf{c} \vee \mathsf{b} = \mathsf{a}$ are equal.

Consider the problem consisting of the two axioms $\forall x.\ \mathsf{g}(x) = \mathsf{f}(x)$ and $\mathsf{g}(\mathsf{a}) = 0$ and the conjecture $\mathsf{f}(\mathsf{a}) = 0$. After clausification, the problem consists of three clauses:

$$\mathsf{g}(x) = \mathsf{f}(x) \qquad\qquad \mathsf{g}(\mathsf{a}) = 0 \qquad\qquad \mathsf{f}(\mathsf{a}) \neq 0$$

Once the problem is in clausal form, the prover can attempt to *saturate* it—that is, to perform all possible inferences from the current clause set and add their conclusions to the set. The set keeps on growing until $\bot$ is derived, or until no more inferences are possible. If the clause set is satisfiable, the prover might also run forever, never deriving $\bot$.

The superposition calculus consists of inference rules that are performed in this saturation process. A simplistic version of the calculus is presented below:

$$\frac{D' \vee t = t' \qquad C' \vee s[t] \doteq s'}{D' \vee C' \vee s[t'] \doteq s'}\,\textsc{Sup} \qquad\qquad \frac{C' \vee u \neq u}{C'}\,\textsc{ERes}$$

The premises are displayed above the horizontal bar and the conclusion below. The conclusion is generated only if all premises belong to the current clause set. Both occurrences of the symbol $\doteq$ simultaneously denote either $=$ or $\neq$. The notation $s[\ ]$ stands for a context around a subterm. The context may be empty.

This simplistic presentation ignores four features. First, only ground (i.e., variable-free) clauses are considered, but superposition in general supports clauses with variables. We will come back to this. Second, superposition is parameterized by a term order that restricts the calculus, leading to a smaller search space. We will also come back to this. Third, superposition includes a selection mechanism that works in tandem with the term order to restrict inferences. We will ignore it to simplify the presentation. Fourth, superposition has a third rule, which is rarely needed in practice but nonetheless necessary for completeness. We will also ignore it.

The SUP rule is the main engine of superposition, hence its name. If we for a moment ignore C and D, it uses an equation $t = t'$ to rewrite some term s that contains t, replacing t by t'. The extra literals in C and D can be viewed as conditions on the respective (dis)equations. For example, $C \lor t = t'$ can be read as $\neg C \Rightarrow t = t'$. The literals from C and D must be added to the conclusion; otherwise, the rule would not be sound—the conclusion would not be entailed by the premises.

The other main rule is ERES (equality resolution). It simply eliminates a trivially false literal $u \neq u$ from a clause.

Example 2.1. Let us try to saturate the unsatisfiable clause set $\{\mathsf{g}(\mathsf{a}) = \mathsf{f}(\mathsf{a}), \mathsf{g}(\mathsf{a}) = 0, \mathsf{f}(\mathsf{a}) \neq 0\}$. First, we can apply the SUP rule with the second and first clauses, in that order, as premises, generating the conclusion $0 = \mathsf{f}(\mathsf{a})$. A second SUP inference is now possible, with this new clause as the left premise and $\mathsf{f}(\mathsf{a}) \neq 0$ as the right premise, and $0 \neq 0$ as the conclusion. At this point, ERES can be applied to derive $\bot$, and we stop the saturation process.

Example 2.2. The simplistic version of the calculus is very explosive, as this example will show. Consider the clause set $\{\mathsf{e} = \mathsf{d},\ \mathsf{d} = \mathsf{c},\ \mathsf{c} = \mathsf{b},\ \mathsf{b} = \mathsf{a},\ \mathsf{f}(\mathsf{e}) \neq \mathsf{f}(\mathsf{a})\}$. The clauses we are interested in generating are of the form $\mathsf{f}(s) \neq \mathsf{f}(s)$, because given any such clause we can apply ERES to derive $\bot$. The shortest chains of SUP reasoning that leads to $\mathsf{f}(s) \neq \mathsf{f}(s)$ requires four steps. For example:

— From $\mathsf{e} = \mathsf{d}$ and $\mathsf{f}(\mathsf{e}) \neq \mathsf{f}(\mathsf{a})$, derive $\mathsf{f}(\mathsf{d}) \neq \mathsf{f}(\mathsf{a})$.
— From $\mathsf{d} = \mathsf{c}$ and $\mathsf{f}(\mathsf{d}) \neq \mathsf{f}(\mathsf{a})$, derive $\mathsf{f}(\mathsf{c}) \neq \mathsf{f}(\mathsf{a})$.
— From $\mathsf{c} = \mathsf{b}$ and $\mathsf{f}(\mathsf{c}) \neq \mathsf{f}(\mathsf{a})$, derive $\mathsf{f}(\mathsf{b}) \neq \mathsf{f}(\mathsf{a})$.
— From $\mathsf{b} = \mathsf{a}$ and $\mathsf{f}(\mathsf{b}) \neq \mathsf{f}(\mathsf{a})$, derive $\mathsf{f}(\mathsf{a}) \neq \mathsf{f}(\mathsf{a})$.

But if we are not careful, we might end up deriving 5^2 $(= 25)$ clauses of the form $\mathsf{f}(s) \neq \mathsf{f}(s')$. This kind of freedom can lead the prover to explore an excessively large search space.

The explosion of Example 2.2 can be averted in the calculus without relying on heuristics of the prover. The idea is to fix a total well-founded order $\prec$ on all ground terms, the *term order*, and to restrict the calculus so that it focuses on the largest terms; for example, the SUP rule is applicable only if t is the larger side of the largest literal of the first premise, and similarly for $s[t]$ in the second premise. Typical choices for the term order are the Knuth–Bendix order [Knuth and Bendix 1970] and the lexicographic path order [Kamin and Lévy 1980].

Although not needed in theory, the term order is crucial to obtain good performance. When faced with a clause $L_1 \lor \cdots \lor L_n$, where $L_1 \prec \cdots \prec L_n$, the prover first focuses on L_n, trying to eliminate it using SUP and ERES. Then it proceeds to the next largest literal. If it gets stuck while trying to eliminate L_n, the other literals will never be visited—a considerable gain.

The situation is analogous to when we want to apply a lemma of the form "If H_1 and ... and H_n, then F" in a pen-and-paper proof. To use the lemma, we need to show the hypotheses $H_1, \ldots, H_n$. Without loss of generality, we can fix the order $H_1 \prec \cdots \prec H_n$ and first try to show H_n, then H_{n-1}, and so on until H_1. If we fail at showing H_n, there is no point in continuing with H_{n-1}; we will never be able to use F anyway.

So far, we have assumed that clauses are ground, but superposition supports variables. Like in mathematics, variables are understood to be $\forall$-quantified. Thus, assuming that our signature contains the nullary functions (i.e., constants) a and b, the unary function f, and no other functions, a clause such as $C(x)$, where x is a variable, represents, in the spirit of Herbrand interpretations, the infinite set of ground clauses $C(\mathsf{a})$, $C(\mathsf{b})$, $C(\mathsf{f}(\mathsf{a}))$, $C(\mathsf{f}(\mathsf{b}))$, $C(\mathsf{f}(\mathsf{f}(\mathsf{a})))$, $C(\mathsf{f}(\mathsf{f}(\mathsf{b})))$, $\ldots$, where x is instantiated with all possible terms that can be built using the signature. (This process assumes that the signature contains at least one nullary symbol. We can extend the signature if needed.)

Suppose that we have the two clauses $C(y) \vee \mathsf{f}(\mathsf{a}, y) = 0$ and $D(x) \vee \mathsf{f}(x, \mathsf{b}) \neq 0$, where $C(y)$ denotes a clause that depends on y and $D(x)$ a clause that depends on x. The terms $\mathsf{f}(\mathsf{a}, y)$ and $\mathsf{f}(x, \mathsf{b})$ are not syntactically identical, but they can be made the same by taking $x := \mathsf{a}$ and $y := \mathsf{b}$—i.e., by unifying [Robinson 1965] the two terms. After instantiating the variables, we get the two clauses

$$C(\mathsf{b}) \vee \mathsf{f}(\mathsf{a}, \mathsf{b}) = 0 \qquad\qquad D(\mathsf{a}) \vee \mathsf{f}(\mathsf{a}, \mathsf{b}) \neq 0$$

From these clauses, a SUP inference derives $C(\mathsf{b}) \vee D(\mathsf{a}) \vee 0 \neq 0$.

In first-order logic, if two expressions are unifiable, then there exists a variable assignment that captures that fact in the most general manner possible, up to the naming of variables. This variable assignment is called the *most general unifier*. We can use it to compute the substitution to use in the conclusion of the superposition inference rules:

$$\frac{D' \vee t = t' \qquad C' \vee s[u] \doteq s'}{(D' \vee C' \vee s[t'] \doteq s')\sigma}\,\textsc{Sup} \qquad\qquad \frac{C' \vee u \neq u'}{C'\sigma}\,\textsc{ERes}$$

In SUP, σ is the most general unifier of t and u. In ERES, σ is the most general unifier of u and u'.

Example 2.3. Consider the clause set $\{\mathsf{g}(x) = \mathsf{f}(x), \mathsf{g}(\mathsf{a}) = 0, \mathsf{f}(\mathsf{a}) \neq 0\}$. This example is similar to Example 2.1, but this time the first clause contains a variable. Let us try to derive $\bot$. First, we can apply the SUP rule with the second and first clauses, in that order, as premises, generating the conclusion $0 = \mathsf{f}(\mathsf{a})$. This unifies $\mathsf{g}(x)$ and $\mathsf{g}(\mathsf{a})$. The rest of the saturation is as in Example 2.1.

Example 2.4. We will use superposition to prove the following lemma:

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

Conversion into clausal normal form produces the clause set

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

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

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

In each clause, underlining identifies the larger sides of the largest literals according to the Knuth–Bendix order with a weight of 1 for each symbol and the reverse alphabetical order as the precedence. If we take $x := \mathsf{pi}$ in clause (1), the underlined term in (1) matches a subterm of the underlined term in clause (3). Thus, we can apply SUP using as premises clauses (1) and (3) to generate

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

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

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

Now that the larger literal of (4) has been eliminated, we can work on the remaining literal. Multiple SUP inferences are possible: between (5) and (2), between (5) and (3), or between (5) and (4). The first one generates

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

Finally, an ERES inference on clause (6) yields $\bot$, thereby proving the original lemma.

When saturating, a prover might identify clauses that are useless, either because they are tautologies or because they are entailed by other, smaller clauses. These clauses are called *redundant* and can be deleted at any point during the saturation process. For example, $\mathsf{f}(x) = 0 \lor \mathsf{f}(x) = 1$ is redundant with respect to the more informative clause $\mathsf{f}(x) = 0$. Formally, we write $\mathsf{f}(x) = 0 \lor \mathsf{f}(x) = 1 \in \mathrm{Red}(\{\mathsf{f}(x) = 0\})$. In general, for a clause set N, the set $\mathrm{Red}(N)$ consists of all the clauses that are redundant with respect to N. Removing redundant clauses is vital for performance in practice.

There is also an analogous notion of redundant inference. An inference is called *redundant* with respect to a clause set N if its conclusion is already in N or redundant with respect to N. Thus, performing an inference (i.e., adding its conclusion to N) is a sure way to make it redundant.

3. COMPLETENESS OF SUPERPOSITION

An important property of the superposition calculus is that it is refutationally complete [Bachmair and Ganzinger 1994]. Intuitively, this property tells us that the prover will always find a proof if there is one. Alternatively, focusing just on what happens after negating the conjecture and clausification, refutational completeness means that the calculus will always derive the empty clause if the initial clause set is unsatisfiable. Formally:

Definition 3.1. A clause C is inductively defined to be *derivable* by a (clausal) inference system Γ from a clause set N, written $N \vdash_\Gamma C$, if $C \in N$ or if there exists a Γ-inference with conclusion C and premises that are derivable by Γ from N.

An inference system Γ is *refutationally complete* if for all clause sets N, we have that $N \models \bot$ implies $N \vdash_\Gamma \bot$.

However, given the possibility that the prover may remove redundant clauses during a derivation, this simple definition does not accurately reflect what might happen in an implementation. That is why we prefer to represent the state of the prover over time as a sequence of sets of clauses $N_0, N_1, \ldots$. At each step, $N_{i+1} \setminus N_i$ are the clauses that the prover newly derived, and $N_i \setminus N_{i+1}$ are the clauses that the prover decided to remove. We require that the removed clauses are redundant—i.e., $N_i \setminus N_{i+1} \subseteq \mathrm{Red}(N_{i+1})$. Moreover, we want the prover to eventually perform all possible Γ-inferences except for redundant inferences. Formally, we can enforce this by requiring that the conclusions of all Γ-inferences from the limit inferior $\bigcup_i \bigcap_{j \geq i} N_j$ are contained in $\bigcup_i N_i \cup \mathrm{Red}(N_i)$. A sequence $N_0, N_1, \ldots$ that fulfills these requirements is called a *fair derivation*. In other words, in a fair derivation, every possible inference is eventually performed or otherwise made redundant. Based on this notion, we can define a notion of refutational completeness that allows for deletion of redundant clauses:

Definition 3.2. We call (Γ, Red) *dynamically refutationally complete* if for every fair derivation $N_0, N_1, \ldots$ with $N_0 \models \bot$, we have $\bot \in N_i$ for some i.

Completeness as per Definition 3.1 corresponds to the special case of Definition 3.2 with the trivial redundancy criterion, which deletes no clauses.

Under reasonable assumptions about the redundancy criterion, we can show that dynamic refutational completeness is equivalent to the following property:

Definition 3.3. A clause set N is *saturated up to redundancy* if every inference from N is redundant with respect to N. An inference system with redundancy (Γ, Red) is *statically refutationally complete* if all sets N of formulas saturated up to redundancy with $\bot \notin N$ are satisfiable.

To prove that superposition is statically refutationally complete, we fix a set N with $\bot \notin N$ and assume that it is saturated up to redundancy. We must then show that this set is satisfiable by constructing a model of N.

We first construct an interpretation that we will show to be model of the set of ground clauses

$$\mathrm{G}(N) = \{C\theta \mid C \in N \text{ and } \theta \text{ is a substitution such that } C\theta \text{ is ground}\}$$

Considering these ground clauses first is simpler because we can define a total order on ground clauses—the clause order—based on the term order used by superposition. The construction naturally only works if $\bot \notin N$. Iterating over the clauses in $\mathrm{G}(N)$ following this clause order, we collect a set of equalities and define an interpretation such that two terms are equal in the interpretation if and only if one can rewrite the terms into each other using the collected set of equalities. Concretely, iterating over each clause $C \in \mathrm{G}(N)$ in clause order, we collect the equality $s = t$ if

— C is of the form $C' \vee s = t$ where $s = t$ is the largest literal in C;
— C is false in the interpretation resulting from the previously collected equalities;
— C' will remain false when we add the equality $s = t$; and
— the larger sides of the equalities collected so far are not subterms of the larger side of $s = t$.

Otherwise, we do not collect any equality and proceed with the next clause.

For example, let $N = \{\mathsf{a} = \mathsf{b}, \mathsf{a} \neq \mathsf{b} \vee \mathsf{c} = \mathsf{d}\}$ where $\mathsf{a} = \mathsf{b} \prec \mathsf{a} \neq \mathsf{b} \prec \mathsf{c} = \mathsf{d}$. All clauses are ground, so $\mathrm{G}(N) = N$. The smallest clause is $\mathsf{a} = \mathsf{b}$. In the absense of any collected equalities, it is currently false and also fulfills the other conditions above. Thus, we collect the equality $\mathsf{a} = \mathsf{b}$. The next clause is $\mathsf{a} \neq \mathsf{b} \vee \mathsf{c} = \mathsf{d}$. It is also false in the current interpretation resulting from the collected equality $\mathsf{a} = \mathsf{b}$. Since the other conditions are fulfilled as well, we collect the equality $\mathsf{c} = \mathsf{d}$. The two collected equalities now yield a model of N.

For a different example, let $N = \{\mathsf{a} = \mathsf{b}, \mathsf{a} \neq \mathsf{b} \vee \mathsf{c} = \mathsf{d}, \mathsf{a} \neq \mathsf{b} \vee \mathsf{c} \neq \mathsf{d}\}$. The procedure collects the equalities $\mathsf{a} = \mathsf{b}$ and $\mathsf{c} = \mathsf{d}$ as before. The new clause $\mathsf{a} \neq \mathsf{b} \vee \mathsf{c} \neq \mathsf{d}$, however, is not of the form $C' \vee s = t$ and we do not collect an equality. The resulting interpretation based on the equalities $\mathsf{a} = \mathsf{b}$ and $\mathsf{c} = \mathsf{d}$ is not a model of this last clause. The construction fails in this regard when the set N is not saturated up to redundancy. Here, N is not saturated because a SUP inference between the second and the third clause is possible:

$$\frac{\mathsf{a} \neq \mathsf{b} \vee \mathsf{c} = \mathsf{d} \qquad \mathsf{a} \neq \mathsf{b} \vee \mathsf{c} \neq \mathsf{d}}{\mathsf{a} \neq \mathsf{b} \vee \mathsf{c} = \mathsf{c}}\ \textsc{Sup}$$

Based on this idea, by well-founded induction with respect to the clause order, we can show that the constructed interpretation is indeed a model of $\mathrm{G}(N)$. The inductive argument is roughly as follows: As our induction hypothesis, we assume that the constructed interpretation is a model of all clauses in $\mathrm{G}(N)$ that are smaller than a given clause $C\theta \in \mathrm{G}(N)$. To show that the interpretation is also a model of $C\theta$, we assume that it is not a model of $C\theta$ for a proof by contradiction. Using this assumption, we can show that there exists an inference from C, and by saturation up to redundancy, this inference must be redundant. So its conclusion must be in N or redundant with respect to N. Moreover, we can show that a ground instance of this conclusion is smaller than $C\theta$ and thus, by the induction hypothesis, this ground instance of the conclusion is true in the constructed interpretation. We can then argue that this contradicts the assumption that $C\theta$ is false in the constructed interpretation.

Finally, we can show that the constructed model of $\mathrm{G}(N)$ is also a model of N, proving static refutational completeness. The proof is straightforward because a clause is equivalent to its groundings, due to the special structure of the constructed model.

4. LAMBDA-SUPERPOSITION

In our venture to generalize superposition to higher-order logic, we encountered various obstacles. The challenge was to overcome them without compromising our design goals of *gracefully* generalizing the first-order calculus, proving the higher-order calculus *sound and refutationally complete*, and designing a calculus that can be implemented *efficiently*.

4.1. Term Order

In standard superposition, the term order must fulfill certain properties for the completeness proof to work. Two of them are:

— totality on ground terms: for two distinct ground terms, we have $u \prec v$ or $u \succ v$;
— compatibility with contexts: $u \succ v$ implies $s[u] \succ s[v]$.

The first property is required to construct the total order on ground clauses used for the induction argument, and the second property is used to show that the conclusion of a ground superposition inference is smaller than its right premise, which we need to apply the induction hypothesis.

In higher-order logic, considering terms modulo β-conversion and defining ground terms as terms without free variables, these properties cannot be fulfilled: To see why, let $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c}$ be constants. Consider the terms $\lambda x.\ \mathsf{b}$ and $\lambda x.\ x$. By totality on ground terms, we must have $\lambda x.\ \mathsf{b} \prec \lambda x.\ x$ or $\lambda x.\ \mathsf{b} \succ \lambda x.\ x$. In the first case, by compatibility with contexts, using the context $\ldots\ \mathsf{a}$, we have $\mathsf{b} \equiv (\lambda x.\ \mathsf{b})\ \mathsf{a} \prec (\lambda x.\ x)\ \mathsf{a} \equiv \mathsf{a}$, in contradiction to $\mathsf{a} \prec \mathsf{b}$. In the second case, by compatibility with contexts, using the context $\ldots\ \mathsf{c}$, we have $\mathsf{b} \equiv (\lambda x.\ \mathsf{b})\ \mathsf{c} \succ (\lambda x.\ x)\ \mathsf{c} \equiv \mathsf{c}$, in contradiction to $\mathsf{b} \prec \mathsf{c}$.

We can overcome this issue by noticing that it only occurs with contexts that would not be considered valid contexts in first-order logic, such as $\ldots\ \mathsf{a}$ and $\ldots\ \mathsf{c}$. We call contexts that would be considered valid in first-order logic *green contexts*:

Definition 4.1. We define *green subterms* inductively as follows: A term t is a green subterm of t. If u is a green subterm of t_i, then u is a green subterm of $\mathsf{f}\ t_1\ \cdots\ t_n$ for a symbol f and terms $t_1, \ldots, t_n$. We write $s\langle u\rangle$ for a term s with a green subterm u. A *green context* is a context around a green subterm.

If we require compatibility only with green contexts instead of all higher-order contexts, we can construct appropriate term orders quite easily. However, it can then happen that for instance $\mathsf{g} \succ \mathsf{f}$ and $\mathsf{g}\ \mathsf{a} \prec \mathsf{f}\ \mathsf{a}$. A naive generalization of the first-order SUP rule would then allow for inferences such as

$$\frac{\mathsf{g} = \mathsf{f} \qquad \mathsf{g}\ \mathsf{a} \neq \mathsf{f}\ \mathsf{a}}{\mathsf{f}\ \mathsf{a} \neq \mathsf{f}\ \mathsf{a}}\,\textsc{Sup}$$

We seemingly rewrote larger into smaller terms because $\mathsf{g} \succ \mathsf{f}$, but still the conclusion is larger than the right premise because $\mathsf{g}\ \mathsf{a} \prec \mathsf{f}\ \mathsf{a}$. The attempt to use such an inference in the completeness proof fails because the conclusion is too large to allow us to invoke the induction hypothesis.

The solution is to restrict the SUP rule to green contexts as well. This is the ground version of the SUP rule of our calculus:

$$\frac{D' \vee t = t' \qquad C' \vee s\langle t\rangle \doteq s'}{C' \vee D' \vee s\langle t'\rangle \doteq s'}\,\textsc{Sup}$$

This ensures that for all ground SUP inferences, the conclusion is smaller than the right premise. However, by restricting SUP to green contexts, we lose the ability to derive the empty clause from $\mathsf{g} = \mathsf{f}$ and $\mathsf{g}\ \mathsf{a} \neq \mathsf{f}\ \mathsf{a}$. We add the following rule to compensate:

$$\frac{C' \vee s = t}{C' \vee s\ x = t\ x}\,\textsc{ArgCong}$$

where s and t have a functional type and x is a fresh variable. From $\mathsf{g} = \mathsf{f}$, ARGCONG derives $\mathsf{g}\ x = \mathsf{f}\ x$. The SUP rule is then applicable to derive $\mathsf{g}\ \mathsf{a} \neq \mathsf{g}\ \mathsf{a}$, from which ERES derives $\bot$.

Seemingly, we have not gained much because ARGCONG also has the flaw that the conclusion is often larger than the premise. But with ARGCONG, we can divide the completeness proof in two steps. We first consider a clause set, in which the occurrences of each function constant f have been replaced with new constants $\mathsf{f}_0, \mathsf{f}_1, \mathsf{f}_2, \ldots$, depending on how many arguments the specific occurrence has. We can construct a model of this clause set via the usual induction with respect to the clause order without making use of ARGCONG. Only in a second step, outside of the induction with respect to the clause order, we use ARGCONG to show that the interpretations of different constants $\mathsf{f}_0, \mathsf{f}_1, \mathsf{f}_2, \ldots$ can be consolidated.

4.2. Unification

In first-order logic, Robinson's unification algorithm offers an efficient procedure to compute the most general unifier of two terms. In higher-order logic, unification is a substantially harder problem.

First, higher-order unification is undecidable. No unification algorithm can decide for all term pairs whether they are unifiable. There will always be pairs of nonunifiable terms that a given procedure cannot determine to be nonunifiable. Instead, the procedure will simply not terminate on these term pairs.

Second, most general unifiers of unifiable higher-order terms do not always exist. In general, we must consider sets of unifiers to capture all possible unifiers. We call these sets *complete sets of unifiers*. Complete sets of unifiers may even need to be infinite. For example, given the unification problem $z\ (\mathsf{f}\ \mathsf{a}) = \mathsf{f}\ (z\ \mathsf{a})$, there are infinitely many unifiers that do not subsume each other: $\{z \mapsto \lambda x.\ x\}$, $\{z \mapsto \lambda x.\ \mathsf{f}\ x\}$, $\{z \mapsto \lambda x.\ \mathsf{f}\ (\mathsf{f}\ x)\}$, $\ldots$. A higher-order unification procedure will therefore need to enumerate an infinite set of unifiers in some cases, another reason why the procedure may not terminate.

To deal with the possible nontermination of unification, we must restructure the prover architecture to be able to interleave computation of unifiers and computation of inferences fairly. Our unification procedure regularly interrupts the computation of unifiers to allow the main program to focus on other computations before resuming the computation of unifiers.

Given two terms t and u, we denote the complete set of unifiers that our unification procedure eventually enumerates by $\mathrm{CSU}(t, u)$. The nonground versions of our higher-order rules are

$$\frac{D' \vee t = t' \qquad C' \vee s\langle u\rangle \doteq s'}{(C' \vee D' \vee s\langle t'\rangle \doteq s')\sigma}\,\textsc{Sup} \qquad\qquad \frac{C' \vee u \neq u'}{C'\sigma}\,\textsc{ERes}$$

where $\sigma \in \mathrm{CSU}(t, u)$ for SUP and $\sigma \in \mathrm{CSU}(u, u')$ for ERES, plus various other conditions—e.g., regarding the order.

4.3. Booleans

In first-order logic, computing the clausal normal form is straightforward because first-order logic distinguishes between terms and formulas and formulas cannot occur within terms. For instance, an expression such as $\mathsf{f}(\mathsf{a} = \mathsf{b})$ is not permissible in most treatments of first-order logic.

By contrast, in higher-order logic, Boolean terms may occur anywhere, including under λ-abstractions. This makes it impossible to eliminate all Boolean terms in preprocessing. Instead, we integrate the computation of the clausal normal form into the calculus by adding dedicated inference rules.

For example, the following inference rules transform equalities occurring within terms:

$$\frac{C\langle s = t\rangle}{C\langle\bot\rangle \vee s = t}\,\textsc{EqHoist} \qquad\qquad \frac{C\langle s = t\rangle}{(C\langle\top\rangle)\sigma}\,\textsc{BoolRw}$$

where, for BOOLRW, $\sigma \in \mathrm{CSU}(s, t)$. We can impose order restrictions on these rules to reduce the number of inferences that must be computed.

To achieve a graceful generalization of standard superposition, we add rules that apply adaptations of first-order clausification rules on Boolean connectives and quantifiers that occur on the outside, without being nested inside other terms. Applying these clausification rules destructively can be justified to be refutationally complete by our redundancy criterion.

4.4. Example

To demonstrate how λ-superposition operates in practice, consider the following example from the introduction:

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

To prove it, we will need to use the fact that the big sum operator distributes over addition:

$$\left(\textstyle\sum_{i=m}^{n} f(i) + g(i)\right) = \left(\textstyle\sum_{i=m}^{n} f(i)\right) + \left(\textstyle\sum_{i=m}^{n} g(i)\right)$$

We negate our conjecture and express the two statements as clauses in higher-order logic:

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

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

Again, underlining identifies the larger sides of the largest literals, here using a higher-order variant of the Knuth–Bendix order [Becker et al. 2017] with a weight of 1 for each symbol.

Using the substitution $\{f \mapsto \lambda i.\ i \mathbin{\hat{}} 2,\ g \mapsto \lambda i.\ 2 * i,\ m \mapsto 1\}$, we can apply a SUP inference of (8) into the subterm $\mathsf{sum}\ 1\ n\ (\lambda i.\ i \mathbin{\hat{}} 2) + \mathsf{sum}\ 1\ n\ (\lambda i.\ 2 * i)$ of (7), yielding

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

Then, using the substitution $\{f \mapsto \lambda i.\ i \mathbin{\hat{}} 2 + 2 * i,\ g \mapsto \lambda i.\ 1,\ m \mapsto 1\}$, we can apply a SUP inference of (8) into the right-hand side of (9), yielding

$$\underline{\mathsf{sum}\ 1\ n\ (\lambda i.\ i \mathbin{\hat{}} 2 + 2 * i + 1)} \neq \underline{\mathsf{sum}\ 1\ n\ (\lambda i.\ i \mathbin{\hat{}} 2 + 2 * i + 1)} \qquad (10)$$

Finally, by ERES, we obtain the empty clause $\bot$.

5. COMPLETENESS OF LAMBDA-SUPERPOSITION

The λ-superposition calculus is sound and refutationally complete. However, there are caveats with both of these claims.

Regarding soundness, the caveat is that clausification, in particular Skolemization, extends the signature with new symbols, and thus we cannot speak about soundness when integrating clausification into our calculus. We can, however, define our logic in a way that all Skolem symbols we might ever need are already in the signature and have the correct interpretation. With respect to such a logic, we can then prove the calculus sound.

Regarding completeness, the caveat is that completeness depends on the semantics of higher-order logic. With standard semantics, higher-order logic is strong enough to formalize arithmetic and by Gödel's first incompleteness theorem there exist unprovable valid statements that even λ-superposition cannot prove. But if we use general (Henkin) semantics instead, a semantics that allows for nonstandard models of the function space, higher-order logic can no longer formalize arithmetic and Gödel's first incompleteness theorem does not apply. The good news is that the semantics do not matter when we only care about what is provable. Stating that our calculus is refutationally complete simply means that we can prove the same statements as traditional proof systems such as the ones implemented in proof assistants.

More precisely, we prove static refutational compeleteness with respect to a polymorphic variant of Church's simple type theory [Church 1940] with Hilbert choice and functional and Boolean extensionality, but without the axiom of infinity. The proof is sketched below. We fix a set N of higher-order clauses with $\bot \notin N$ and assume that it is saturated up to redundancy. We must then show that this set is satisfiable by constructing a model of N.

Grounding. Like in the first-order proof, we consider the set $\mathrm{G}(N)$ of ground instances of N.

Encoding. We then encode these ground higher-order clauses into first-order clauses. For instance, we encode the higher-order term $\mathsf{f}\ (\lambda x.\, x)\ (\mathsf{g}\ \mathsf{f})$ as the first-order term $\mathsf{f}_2(\mathsf{lam}_{\lambda x.\, x}, \mathsf{g}_1(\mathsf{f}_0))$, where $\mathsf{lam}_{\lambda x.\, x}$, f_2, f_0 and g_1 are first-order function symbols representing the entire λ-expression $\lambda x.\, x$ and the functions f and g with different numbers of arguments. We denote this first-order encoding of the ground clauses by $\mathcal{F}(\mathrm{G}(N))$. The encoding is vaguely related to the one we saw in the introduction, but it is only needed for the proof, not in the practical implementation of a prover.

Model construction. Now we would like to apply the first-order completeness result to obtain a model of $\mathcal{F}(\mathrm{G}(N))$. However, the encoding $\mathcal{F}$ does not remove Boolean terms. Therefore, we need to extend first-order superposition and its completeness proof with support for Booleans. Then we can show that the clause set $\mathcal{F}(\mathrm{G}(N))$ is saturated up to redundancy with respect to the resulting calculus and that $\bot \notin \mathcal{F}(\mathrm{G}(N))$. As a result, we obtain a first-order model for $\mathcal{F}(\mathrm{G}(N))$.

Lifting to higher-order logic. We transform the first-order model into a higher-order model of $\mathrm{G}(N)$ using saturation up to redundancy with respect to some of the higher-order rules such as ARGCONG.

Lifting to variables. Finally, we show that this model of $\mathrm{G}(N)$ is also a model of N.

6. COMPETING APPROACHES

Since the 1960s, a wide range of approaches have been proposed to prove higher-order formulas automatically. We briefly review the main ones.

Resolution. The resolution calculus is a precursor of superposition, without built-in support for equality. Like λ-superposition, higher-order resolution [Huet 1973] crucially relies on higher-order unification [Huet 1975], but it can postpone solving so-called flex–flex pairs, an important optimization. Equality reasoning was added in the Leo series of provers: LEO [Benzmüller and Kohlhase 1998], LEO-II [Benzmüller et al. 2015], and Leo-III [Steen and Benzmüller 2018]. Leo-III also features a term order. Its calculus, higher-order paramodulation, is similar to an incomplete version of λ-superposition.

Superposition. Beside λ-superposition and higher-order paramodulation, another superposition-based approach, combinatory superposition, consists of using SKBCI combinators to represent λ-expressions inside a modified superposition calculus. This is the route taken by the Vampire prover [Bhayat and Reger 2020b]. Recently, this approach has been enhanced to compute unifiers lazily [Bhayat et al. 2023].

Tableaux. Tableau calculi analyze the formula systematically, building a tree that keeps track of the remaining cases to prove. Higher-order tableau calculi have been proposed multiple times [Robinson 1969; Kohlhase 1995; Konrad 1998; Backes and Brown 2011]. A major strength of tableaux is that they can work directly on unclausified formulas. However, they can end up repeating work in separate branches of the tree, and they tend to be weaker than superposition at equality reasoning. Tableaux form the basis of the Satallax prover [Brown 2012]. A related approach, focused sequent calculus, is implemented in the agsyHOL prover [Lindblad 2014].

Matings. Another related approach relies on matings, or connections. The TPS prover [Andrews et al. 1996] implements this approach. The idea is to translate formulas to matrices and look for paths with certain properties in these matrices. Once the prover successfully closes a path, a contradiction is found. The main benefit of the approach is ease of implementation. Unfortunately, it does not scale very well, especially in the presence of extraneous axioms.

Satisfiability Modulo Theories (SMT). SMT solvers have been very successful in a wide range of applications. Some solvers support full first-order logic, including quantifiers. Among these, cvc5 and veriT have been partly extended to higher-order logic [Barbosa et al. 2019]. The strength of SMT solvers is their support for decidable theories. Their main weakness might be their handling of quantifiers.

7. COMPETITION RESULTS

Although refutational completeness is a desirable property, in practice what matters more is performance on actual problems. One way to assess this is to run several provers on the same problems. This is done every year as part of CASC, a competition that takes place at CADE (Conference on Automated Deduction) and IJCAR (International Joint Conference on Automated Reasoning).

Every year, the competition includes a higher-order theorems (THF) division featuring automatic theorem provers for higher-order logic. The problem set consists of 500 problems from the TPTP (Thousands of Problems for Theorem Provers) library [Sutcliffe 2017], chosen to be neither too easy nor too difficult. Among the provers, E and Zipperposition implement λ-superposition, but so does the newcomer Duper as well.

	2019	2020	2021	2022	2023
CVC4	268	194	–	–	–
cvc5	–	–	239	282	258
Duper	–	–	–	–	36
E	–	–	300	419	407
Lash	–	–	–	280	208
LEO-II	179	112	95	174	58
Leo-III	359	287	357	336	302
Satallax	**418**	319	–	329	268
Vampire	304	299	386	367	**452**
Zipperposition	356	**424**	**467**	**456**	440
Number of problems	500	500	500	500	500

Fig. 1. CASC results in the THF division per year

Figure 1 presents the results (in number of problems solved) in that competition from 2019 to 2023. For each year, the winning entry is shown in bold. Satallax has been winning for most of the 2010s, including in 2019. It was dethroned by Zipperposition in 2020, which in turn was dethroned in 2023 by Vampire. Both Zipperposition and Vampire implement variants of superposition. Based on a private communication with Ahmed Bhayat, the main developer of Vampire's higher-order support, we believe that the following factors explain Vampire's recent victory: (1) Vampire now performs lazy unification [Bhayat et al. 2023]; (2) it has been extensively tuned using machine learning; and (3) it is generally more optimized than Zipperposition.

	2021	2022	2023
cvc5	310	595	362
Duper	–	–	51
E	655	**655**	**467**
Lash	–	–	219
Leo-III	499	–	125
Satallax	–	–	278
Vampire	626	629	454
Zipperposition	**675**	654	462
Number of problems	720	720	1000

Fig. 2. CASC results in the Sledgehammer theorems division per year

Next to the THF division, CASC has also included, for the last three years, a Sledgehammer theorems (SLH) division whose problems were exported from the Isabelle/HOL [Nipkow et al. 2002] proof assistant using the Sledgehammer tool [Paulson and Blanchette 2012]. Figure 2 presents these results. The SLH problems tend to contain more axioms than those in THF but typically require less advanced higher-order reasoning. In this setting, Zipperposition finished first in 2021, whereas E won in 2022 and 2023. (Officially, Zipperposition enters as a so-called demonstration system, because the set of its developers intersects with the set of Sledgehammer problem providers. Thus, E won the first-place trophy three years in a row.)

8. CONCLUSION

We presented λ-superposition and its predecessor, superposition, focusing on the main metatheoretical result: refutational completeness. The CASC results have confirmed λ-superposition as a leading approach to reason about problems in higher-order logic. Good results were also obtained on Isabelle/HOL benchmarks, suggesting that native higher-order reasoning can be useful also in the context of interactive theorem proving.

In ongoing work, we are developing a variant of λ-superposition that performs unification more lazily, replacing full unification by unification up to flex–flex pairs [Huet 1975]. Other avenues for future work include extensively fine-tuning E and Zipperposition on benchmark suites and improving Zipperposition's support for polymorphism.

ACKNOWLEDGMENTS

We thank Christel Baier for suggesting to write this article and Marvin Brieger for suggesting many textual improvements. We also thank Simon Cruanes and Petar Vukmirović for participating in the research.

The research 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).

REFERENCES

Peter B. Andrews. 2002. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof* (second ed.). Applied Logic, Vol. 27. Springer.

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.

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

Julian Backes and Chad E. Brown. 2011. Analytic Tableaux for Higher-Order Logic with Choice. *J. Autom. Reason.* 47, 4 (2011), 451–479.

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)*, Pascal Fontaine (Ed.), Vol. 11716. Springer, 35–54.

Heiko Becker, Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. 2017. A Transfinite Knuth–Bendix Order for Lambda-Free Higher-Order Terms. In *CADE-26 (LNCS)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, 432–453.

Alexander Bentkamp, Jasmin Blanchette, Visa Nummelin, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. 2023a. Mechanical Mathematicians. *Commun. ACM* 66, 4 (2023), 80–90.

Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirović. 2023b. Superposition for Higher-Order Logic. *J. Autom. Reason.* 67, 1 (2023), 10. DOI:http://dx.doi.org/10.1007/s10817-022-09649-9

Christoph Benzmüller and Michael Kohlhase. 1998. Extensional Higher-Order Resolution. In *CADE-15 (LNCS)*, Claude Kirchner and Hélène Kirchner (Eds.), Vol. 1421. Springer, 56–71.

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.

Ahmed Bhayat, Michael Rawson, and Johannes Schoisswohl. 2023. Superposition with Delayed Unification. (2023).

Ahmed Bhayat and Giles Reger. 2020a. A Combinator-Based Superposition Calculus for Higher-Order Logic. In *IJCAR 2020 (1) (LNCS)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.), Vol. 12166. Springer, 278–296.

Ahmed Bhayat and Giles Reger. 2020b. A Combinator-Based Superposition Calculus for Higher-Order Logic. In *IJCAR 2020, Part I (LNCS)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.), Vol. 12166. Springer, 278–296.

Chad E. Brown. 2012. Satallax: An Automatic Higher-Order Prover. In *IJCAR 2012 (LNCS)*, Bernhard Gramlich, Dale Miller, and Uli Sattler (Eds.), Vol. 7364. Springer, 111–117.

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5, 2 (1940), 56–68.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE-28 (LNCS)*, André Platzer and Geoff Sutcliffe (Eds.), Vol. 12699. Springer, 625–635.

Martin Desharnais, Petar Vukmirović, Jasmin Blanchette, and Makarius Wenzel. 2022. Seventeen provers under the hammer. In *ITP 2022 (LIPIcs)*, June Andronick and Leonardo de Moura (Eds.), Vol. 237. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 8:1–8:18.

M. J. C. Gordon and T. F. Melham (Eds.). 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

John Harrison. 2009. HOL Light: An Overview. In *TPHOLs 2009 (LNCS)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 60–66.

Gérard P. Huet. 1973. A Mechanization of Type Theory. In *IJCAI-73*, Nils J. Nilsson (Ed.). William Kaufmann, 139–146.

Gérard P. Huet. 1975. A Unification Algorithm for Typed lambda-Calculus. *Theor. Comput. Sci.* 1, 1 (1975), 27–57.

Samuel Kamin and Jean-Jacques Lévy. 1980. *Two generalizations of the recursive path ordering*. Unpublished manuscript. University of Illinois.

D. E. Knuth and P. B. Bendix. 1970. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, J. Leech (Ed.). Pergamon Press, 263–297.

Michael Kohlhase. 1995. Higher-Order Tableaux. In *TABLEAUX '95 (LNCS)*, Peter Baumgartner, Reiner Hähnle, and Joachim Posegga (Eds.), Vol. 918. Springer, 294–309.

Karsten Konrad. 1998. HOT: A Concurrent Automated Theorem Prover Based on Higher-Order Tableaux. In *TPHOLs '98 (LNCS)*, Jim Grundy and Malcolm C. Newey (Eds.), Vol. 1479. Springer, 245–261.

Fredrik Lindblad. 2014. A Focused Sequent Calculus for Higher-Order Logic. In *IJCAR 2014 (LNCS)*, Stéphane Demri, Deepak Kapur, and Christoph Weidenbach (Eds.), Vol. 8562. Springer, 61–75.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to $DPLL(T)$. *J. ACM* 53, 6 (2006), 937–977.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer.

Andreas Nonnengart and Christoph Weidenbach. 2001. Computing Small Clause Normal Forms. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Vol. I. Elsevier, 335–367.

Lawrence C. Paulson and Jasmin Christian Blanchette. 2012. Three Years of Experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In *IWIL-2010 (EPiC)*, Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska (Eds.), Vol. 2. EasyChair, 1–11.

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

J. A. Robinson. 1969. Mechanizing higher order logic. In *Machine Intelligence*, B. Meltzer and D. Michie (Eds.). Vol. 4. Edinburgh University Press, 151–170.

Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *TPHOLs 2008 (LNCS)*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.), Vol. 5170. Springer, 28–32.

Alexander Steen and Christoph Benzmüller. 2018. The Higher-Order Prover Leo-III. In *IJCAR 2018 (LNCS)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.), Vol. 10900. Springer, 108–116.

Alexander Steen and Christoph Benzmüller. 2021. Extensional Higher-Order Paramodulation in Leo-III. *J. Autom. Reason.* 65, 6 (2021), 775–807.

Bjarne Stroustrup. 1995. *The Design and Evolution of C++*. Addison-Wesley.

Geoff Sutcliffe. 2017. The TPTP Problem Library and Associated Infrastructure—From CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.* 59, 4 (2017), 483–502.

D. A. Turner. 1979. Another Algorithm for Bracket Abstraction. *J. Symb. Log.* 44, 2 (1979), 267–270.

Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. 2022. Making Higher-Order Superposition Work. *J. Autom. Reason.* 66, 4 (2022), 541–564.

Petar Vukmirović, Jasmin Blanchette, and Stephan Schulz. 2023. Extending a High-Performance Prover to Higher-Order Logic. In *TACAS 2023 (LNCS)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.), Vol. 13994. Springer, 111–129.