HAL open science

Superposition with Lambdas

Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, Uwe Waldmann

To cite this version:

Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, Uwe Waldmann. Superposition with Lambdas. CADE-27 - The 27th International Conference on Automated Deduction, Aug 2019, Natal, Brazil. pp.55-73, 10.1007/978-3-030-29436-6_4 . hal-02296038

HAL Id: hal-02296038

https://inria.hal.science/hal-02296038v1

Submitted on 24 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Superposition with Lambdas

Alexander Bentkamp[1(✉)], Jasmin Blanchette[1,2], Sophie Tourret[2], Petar Vukmirović[1], and Uwe Waldmann[2]

[1] Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
{a.bentkamp,j.c.blanchette,p.vukmirovic}@vu.nl
[2] Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
{jblanche,stourret,uwe}@mpi-inf.mpg.de

Abstract. We designed a superposition calculus for a clausal fragment of extensional polymorphic higher-order logic that includes anonymous functions but excludes Booleans. The inference rules work on $\beta\eta$-equivalence classes of λ-terms and rely on higher-order unification to achieve refutational completeness. We implemented the calculus in the Zipperposition prover and evaluated it on TPTP and Isabelle benchmarks. The results suggest that superposition is a suitable basis for higher-order reasoning.

1 Introduction

Superposition [5] is widely regarded as the calculus par excellence for reasoning about first-order logic with equality. To increase automation in proof assistants and other verification tools based on higher-order formalisms, we propose to generalize superposition to an extensional, polymorphic, clausal version of higher-order logic (also called simple type theory). Our ambition is to achieve a *graceful* extension, which coincides with standard superposition on first-order problems and smoothly scales up to arbitrary higher-order problems.

Bentkamp, Blanchette, Cruanes, and Waldmann [11] recently designed a family of superposition-like calculi for a λ-free fragment of higher-order logic, with currying and applied variables. We adapt their "extensional nonpurifying" calculus to also support λ-expressions (Section 3). Our calculus does not support first-class Booleans; it is conceived as the penultimate milestone towards a superposition calculus for full higher-order logic. If desired, Booleans can be encoded in our logic fragment using an uninterpreted type and uninterpreted "proxy" symbols corresponding to equality, the connectives, and the quantifiers.

Designing a higher-order superposition calculus poses three main challenges:

1. In first-order logic, superposition is parameterized by a ground-total simplification order $\succ$, but such orders do not exist for λ-terms considered equal up to β-conversion. The relations designed for proving termination of higher-order term rewriting systems, such as HORPO [39] and CPO [21], lack many of the desired properties (e.g., transitivity, stability under substitution).
2. Higher-order unification is undecidable and may give rise to an infinite set of incomparable unifiers. For example, the constraint $\mathsf{f}\,(y\,\mathsf{a}) \stackrel{?}{=} y\,(\mathsf{f}\,\mathsf{a})$ admits infinitely many independent solutions of the form $\{y \mapsto \lambda x.\, \mathsf{f}^n\, x\}$.

3. In first-order logic, to rewrite into a term s using an oriented equation $t \approx t'$, it suffices to find a subterm of s that is unifiable with t. In higher-order logic, this is insufficient. Consider superposition from $\mathsf{f}\,\mathsf{c} \approx \mathsf{a}$ into $y\,\mathsf{c} \not\approx y\,\mathsf{b}$. The left-hand sides can obviously be unified by $\{y \mapsto \mathsf{f}\}$, but the more general substitution $\{y \mapsto \lambda x.\, z\, x\, (\mathsf{f}\, x)\}$ also gives rise to a subterm $\mathsf{f}\,\mathsf{c}$ after β-reduction. The corresponding inference generates the clause $z\,\mathsf{c}\,\mathsf{a} \not\approx z\,\mathsf{b}\,(\mathsf{f}\,\mathsf{b})$.

To address the first challenge, we adopt η-short β-normal form to represent $\beta\eta$-equivalence classes of λ-terms. In the spirit of Jouannaud and Rubio's early joint work [38], we state requirements on the term order only for ground terms (i.e., closed monomorphic $\beta\eta$-equivalence classes); the nonground case is connected to the ground case via stability under substitution. Even on ground terms, it is impossible to obtain all desirable properties. We sacrifice compatibility with arguments (the property that $s' \succ s$ implies $s'\, t \succ s\, t$) and compensate for it with an *argument congruence* rule (ArgCong), as in Bentkamp et al. [11].

For the second challenge, we accept that there might be infinitely many incomparable unifiers and enumerate a complete set (including the notorious flex–flex pairs [36]), relying on heuristics to keep the combinatorial explosion under control. The saturation loop must also be adapted to interleave this enumeration with the theorem prover's other activities (Section 6). Despite its reputation for explosiveness, higher-order unification is a conceptual improvement over SK combinators, because it can often *compute* the right unifier. Consider the conjecture $\exists z.\, \forall x\, y.\, z\, x\, y \approx \mathsf{f}\, y\, x$. After negation, clausification, and skolemization, it becomes $z\, (\mathsf{sk_x}\, z)\, (\mathsf{sk_y}\, z) \not\approx \mathsf{f}\, (\mathsf{sk_y}\, z)\, (\mathsf{sk_x}\, z)$. Higher-order unification quickly computes the unique unifier: $\{z \mapsto \lambda x\, y.\, \mathsf{f}\, y\, x\}$. In contrast, an encoding approach based on combinators, similar to the one implemented in Sledgehammer [48], would blindly enumerate all possible SK terms for z until the right one, $\mathsf{S}(\mathsf{K}(\mathsf{Sf}))\mathsf{K}$, is found. Given the definitions $\mathsf{S}\, z\, y\, x \approx z\, x\, (y\, x)$ and $\mathsf{K}\, x\, y \approx x$, the E prover [55] in *auto* mode needs to perform 3756 inferences to derive the empty clause.

For the third challenge, when applying $t \approx t'$ to perform rewriting inside a higher-order term s, the idea is to encode an arbitrary context as a fresh higher-order variable z, unifying s with $z\, t$; the result is $(z\, t')\sigma$, for some unifier σ. This is performed by a dedicated *fluid subterm superposition* rule (FluidSup).

Functional extensionality (the property that $\forall x.\, y\, x \approx z\, x$ implies $y \approx z$) is also considered a challenge for higher-order reasoning [13], although similar difficulties arise with the first-order theories of sets and arrays [33]. Our approach is to add extensionality as an axiom and provide optional rules as optimizations (Section 5). With this axiom, our calculus is refutationally complete with respect to extensional Henkin semantics (Section 4). Detailed proofs are included in a technical report [10], together with more explanations, examples, and discussions.

We implemented the calculus in the Zipperposition prover [27] (Section 6). Our empirical evaluation includes benchmarks from the TPTP [59] and interactive verification problems exported from Isabelle/HOL [22] (Section 7). The results appear promising and suggest that an optimized implementation inside a competitive prover such as E [55], SPASS [64], or Vampire [44] would outperform existing higher-order automatic provers.

2 Logic

Our extensional polymorphic clausal higher-order logic is a restriction of full TPTP THF [15] to rank-1 polymorphism, as in TH1 [40]. In keeping with standard superposition, we consider only formulas in conjunctive normal form. Booleans can easily be axiomatized [10, Section 2.3]. We use Henkin semantics [14, 31, 34].

We fix a set Σ_{ty} of type constructors with arities and a set $\mathcal{V}_{\mathsf{ty}}$ of type variables. We require a binary function type constructor $\to \in \Sigma_{\mathsf{ty}}$ to be present. A type τ, υ is either a type variable $\alpha \in \mathcal{V}_{\mathsf{ty}}$ or has the form $\kappa(\bar{\tau}_n)$ for an n-ary type constructor $\kappa \in \Sigma_{\mathsf{ty}}$ and types $\bar{\tau}_n$. We use the notation $\bar{a}_n$ or $\bar{a}$ to stand for the tuple $(a_1, \ldots, a_n)$ or product $a_1 \times \cdots \times a_n$, where $n \geq 0$. We write κ for $\kappa()$ and $\tau \to \upsilon$ for $\to(\tau, \upsilon)$. A type declaration is an expression of the form $\Pi\bar{\alpha}_m.\, \tau$ (or simply τ if $m = 0$), where all type variables occurring in τ belong to $\bar{\alpha}_m$.

We fix a nonempty set Σ of (function) symbols $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{h}, \ldots$, with type declarations, written as $\mathsf{f} : \Pi\bar{\alpha}_m.\, \tau$ or f, and a set $\mathcal{V}$ of term variables with associated types, written as $x : \tau$ or x. The sets $(\Sigma_{\mathsf{ty}}, \mathcal{V}_{\mathsf{ty}}, \Sigma, \mathcal{V})$ form the signature. The set of *raw λ-terms* is defined inductively as follows. Every $x : \tau \in \mathcal{V}$ is a raw λ-term of type τ. If $\mathsf{f} : \Pi\bar{\alpha}_m.\, \tau \in \Sigma$ and $\bar{\upsilon}_m$ is a tuple of types, called *type arguments*, then $\mathsf{f}\langle\bar{\upsilon}_m\rangle$ (or simply f if $m = 0$) is a raw λ-term of type $\tau\{\bar{\alpha}_m \mapsto \bar{\upsilon}_m\}$. If $x : \tau$ and $t : \upsilon$, then the *λ-expression* $\lambda x.\, t$ is a raw λ-term of type $\tau \to \upsilon$. If $s : \tau \to \upsilon$ and $t : \tau$, then the *application* $s\, t$ is a raw λ-term of type υ.

The α-renaming rule is defined as $(\lambda x.\, t) \to_\alpha (\lambda y.\, t\{x \mapsto y\})$, where y does not occur free in t and is not captured by a λ in t. Raw λ-terms form equivalence classes modulo α-renaming, called *λ-terms*. A variable occurrence is *free* in a λ-term if it is not bound by a λ-expression. A λ-term is *ground* if it is built without using type variables and contains no free term variables. Using the spine notation [25], λ-terms can be decomposed in a unique way as a non-application *head* t applied to zero or more arguments: $t\, s_1 \ldots s_n$ or $t\, \bar{s}_n$ (abusing notation).

The β- and η-reduction rules are defined on λ-terms as $(\lambda x.t)\, u \to_\beta t\{x \mapsto u\}$ and $(\lambda x.tx) \to_\eta t$. For β, bound variables in t are renamed to avoid capture; for η, the variable x must not occur free in t. The λ-terms form equivalence classes modulo $\beta\eta$-reduction, called *βη-equivalence classes* or simply *terms*. When defining operations that need to analyze the structure of terms, we use the η-short β-normal form $t{\downarrow_{\beta\eta}}$, obtained by applying $\to_\beta$ and $\to_\eta$ exhaustively, as a representative of the equivalence class t. Many authors prefer the η-long β-normal form [36, 38, 47], but in a polymorphic setting it has the drawback that instantiating a type variable by a function type can lead to η-expansion. We reserve the letters s, t, u, v for terms and w, x, y, z for variables, and write $: \tau$ to indicate their type.

An equation $s \approx t$ is formally an unordered pair of terms s and t. A literal is an equation or a negated equation, written $\neg\, s \approx t$ or $s \not\approx t$. A clause $L_1 \vee \cdots \vee L_n$ is a finite multiset of literals L_j. The empty clause is written as $\bot$.

In general, a substitution $\{\bar{\alpha}_m, \bar{x}_n \mapsto \bar{\upsilon}_m, \bar{s}_n\}$, where each x_j has type τ_j and each s_j has type $\tau_j\{\bar{\alpha}_m \mapsto \bar{\upsilon}_m\}$, maps m type variables to m types and n term variables to n terms. The letters θ, ρ, σ are reserved for substitutions. Substitutions are lifted to terms and clauses in a capture-avoiding way. The composition $\rho\sigma$ applies ρ first: $t\rho\sigma = (t\rho)\sigma$. A *complete set of unifiers* on a set X

of variables for s and t is a set U of unifiers of s and t such that for every unifier ρ of s and t there exists a member $\sigma \in U$ and a substitution θ such that $x\sigma\theta = x\rho$ for all $x \in X$. We use $\mathrm{CSU}_X(s,t)$ to denote a fixed complete set of unifiers on X for s and t. The set X will consist of the free variables of the clauses in which s and t occur and will be left implicit.

3 The Calculus

Our superposition calculus for clausal higher-order logic is inspired by the λ-free *extensional nonpurifying* calculus described by Bentkamp et al. [11]. The text of this section is partly based on that paper (with Cruanes's permission). The central idea is that superposition inferences are restricted to unapplied subterms occurring in the "first-order outer skeleton" of the superterm—that is, outside λ-expressions and outside the arguments of applied variables. We call these "green subterms." Thus, an equation $\mathsf{g} \approx (\lambda x.\, \mathsf{f}\, x\, x)$ cannot be used directly to rewrite $\mathsf{g}\,\mathsf{a}$ to $\mathsf{f}\,\mathsf{a}\,\mathsf{a}$, because g is applied in $\mathsf{g}\,\mathsf{a}$. A separate inference rule, ArgCong, takes care of deriving $\mathsf{g}\, x \approx \mathsf{f}\, x\, x$, which can be oriented independently of its parent clause and used to rewrite $\mathsf{g}\,\mathsf{a}$ or $\mathsf{f}\,\mathsf{a}\,\mathsf{a}$.

A term (i.e., a $\beta\eta$-equivalence class) t is defined to be a *green subterm* of a term s if either $s = t$ or $s = \mathsf{f}\langle\bar{\tau}\rangle\, \bar{s}$ for some function symbol f, types $\bar{\tau}$ and terms $\bar{s}$, where t is a green subterm of s_i for some i. In $\mathsf{f}\,(\mathsf{g}\,\mathsf{a})\,(y\,\mathsf{b})\,(\lambda x.\, \mathsf{h}\,\mathsf{c}\,(\mathsf{g}\, x))$, the green subterms are a, $\mathsf{g}\,\mathsf{a}$, $y\,\mathsf{b}$, $\lambda x.\, \mathsf{h}\,\mathsf{c}\,(\mathsf{g}\, x)$, and the entire term. We write $t = s\langle u\rangle$ to express that u is a green subterm of t and call $s\langle\,\rangle$ a *green context*.

Another key notion is that of a "fluid" term. A subterm t of $s[t]$ is called *fluid* if (1) $t{\downarrow_{\beta\eta}}$ is of the form $y\,\bar{u}_n$, where y is not bound in $s[t]$ and $n \geq 1$, or (2) $t{\downarrow_{\beta\eta}}$ is a λ-expression and there exists a substitution σ such that $t\sigma{\downarrow_{\beta\eta}}$ is not a λ-expression (due to η-reduction). A necessary condition for case (2) is that $t{\downarrow_{\beta\eta}}$ contains an applied variable that is not bound in $s[t]$. Intuitively, fluid subterms are terms whose η-short β-normal form can change radically as a result of instantiation. For example, applying the substitution $\{z \mapsto (\lambda x.\, x)\}$ to the fluid term $\lambda x.\, y\,\mathsf{a}\,(z\, x)$ makes the λ-expression vanish: $(\lambda x.\, y\,\mathsf{a}\, x) = y\,\mathsf{a}$.

Term Order. The calculus is parameterized by a well-founded strict total order $\succ$ on ground terms satisfying the following properties:

- *green subterm property*: $t\langle s\rangle \succeq s$ (i.e., $t\langle s\rangle \succ s$ or $t\langle s\rangle = s$);
- *compatibility with green contexts*: $s' \succ s$ implies $t\langle s'\rangle \succ t\langle s\rangle$.

The literal and clause orders are defined as multiset extensions in the standard way [5]. Two properties that are not required are *compatibility with λ-expressions* ($s' \succ s$ implies $(\lambda x.\, s') \succ (\lambda x.\, s)$) and *compatibility with arguments* ($s' \succ s$ implies $s'\, t \succ s\, t$). The latter would even be inconsistent with totality. To see why, consider the symbols $\mathsf{c} \succ \mathsf{b} \succ \mathsf{a}$ and the terms $\lambda x.\, \mathsf{b}$ and $\lambda x.\, x$. Owing to totality, one of the terms must be larger than the other, say, $(\lambda x.\, \mathsf{b}) \succ (\lambda x.\, x)$. By compatibility with arguments, we get $(\lambda x.\, \mathsf{b})\,\mathsf{c} \succ (\lambda x.\, x)\,\mathsf{c}$, i.e., $\mathsf{b} \succ \mathsf{c}$, a contradiction. A similar line of reasoning applies if $(\lambda x.\, \mathsf{b}) \prec (\lambda x.\, x)$, using a instead of c.

For nonground terms, $\succ$ is extended to a strict partial order so that $t \succ s$ if and only if $t\theta \succ s\theta$ for all grounding substitutions θ. We also introduce a quasiorder $\succsim$ such that $t \succsim s$ if and only if $t\theta \succeq s\theta$ for all grounding substitutions θ, and similarly for literals and clauses. The quasiorder $\succsim$ is more precise than $\succeq$; for example, given $\mathsf{a}, \mathsf{b} : \iota$ with $\mathsf{b} \succ \mathsf{a}$, we can have $x\,\mathsf{b} \succsim x\,\mathsf{a}$ even though $x\,\mathsf{b} \not\succeq x\,\mathsf{a}$.

Our approach to derive a suitable order is to encode η-short β-normal forms into untyped λ-free higher-order terms and apply an order $\succ_{\mathsf{base}}$ such as the λ-free Knuth–Bendix order (KBO) [8], the λ-free lexicographic path order (LPO) [20], or the embedding path order (EPO) [9]. The encoding, denoted by $[\,]$, translates $\lambda x{:}\tau.\,t$ to $\mathsf{lam}\,[\tau]\,[t]$ and uses De Bruijn symbols db_i to represent bound variables x [24]. It replaces fluid terms t by fresh variables z_t and maps type arguments to term arguments; thus, $[\lambda x{:}\iota.\,\lambda y{:}\iota.\,x] = \mathsf{lam}\,\iota\,(\mathsf{lam}\,\iota\,(\mathsf{db}_1\,\iota))$ and $[\mathsf{f}\langle\iota\rangle(y\,\mathsf{a})] = \mathsf{f}\,\iota\,z_{y\,\mathsf{a}}$. We then define the *metaorder* $\succ_{\mathsf{meta}}$ induced by $\succ_{\mathsf{base}}$ in such a way that $t \succ_{\mathsf{meta}} s$ if and only if $[t] \succ_{\mathsf{base}} [s]$. The use of De Bruijn indices and the monolithic encoding of fluid terms ensure stability under α-renaming and under substitution.

The Inference Rules. The calculus is parameterized by a selection function, which maps each clause to a subclause consisting of negative literals. A literal $L\langle y\rangle$ must not be selected if $y\,\bar{u}_n$, with $n > 0$, is a $\succsim$-maximal term of the clause.

A literal L is (*strictly*) *eligible* in C if it is selected in C or if there are no selected literals in C and L is (strictly) maximal in C. A variable is *deep* in a clause C if it occurs inside a λ-expression or inside an argument of an applied variable; these cover all occurrences that may correspond to positions inside λ-expressions after applying a substitution.

We regard positive and negative superposition as two cases of a single rule

$$\frac{\overbrace{D' \vee t \approx t'}^{D} \qquad \overbrace{C' \vee [\neg]\, s\langle u\rangle \approx s'}^{C}}{(D' \vee C' \vee [\neg]\, s\langle t'\rangle \approx s')\sigma}\ \textsc{Sup}$$

with the following side conditions:

1. u is not a fluid subterm;
2. u is not a deep variable in C;
3. if u is a variable y, there must exist a grounding θ such that $t\sigma\theta \succ t'\sigma\theta$ and $C\sigma\theta \prec C\{y \mapsto t'\}\sigma\theta$;
4. $\sigma \in \mathrm{CSU}(t, u)$; 5. $t\sigma \not\precsim t'\sigma$; 6. $s\langle u\rangle\sigma \not\precsim s'\sigma$; 7. $C\sigma \not\precsim D\sigma$;
8. $(t \approx t')\sigma$ is strictly eligible in $D\sigma$;
9. $([\neg]\, s\langle u\rangle \approx s')\sigma$ is eligible in $C\sigma$, and strictly eligible if it is positive.

There are four main differences with the statement of the standard superposition rule: Contexts $s[\,]$ are replaced by green contexts $s\langle\,\rangle$. The standard condition $u \notin \mathcal{V}$ is generalized by conditions 2 and 3. Most general unifiers are replaced by complete sets of unifiers. And $\not\preceq$ is replaced by the more restrictive $\not\precsim$.

The second rule is a variant of Sup that focuses on fluid subterms occurring in green contexts. Its statement is

$$\frac{\overbrace{D' \vee t \approx t'}^{D} \qquad \overbrace{C' \vee [\neg]\, s\langle u\rangle \approx s'}^{C}}{(D' \vee C' \vee [\neg]\, s\langle z\, t'\rangle \approx s')\sigma}\ \textsc{FluidSup}$$

with the following side conditions, in addition to Sup's conditions 5 to 9:

1. u is either a deep variable in C or a fluid subterm;
2. z is a fresh variable; 3. $\sigma \in \mathrm{CSU}(z\, t, u)$; 4. $z\, t' \neq z\, t$.

The next two rules are almost identical to their standard counterparts:

$$\frac{C' \vee u \not\approx u'}{C'\sigma}\ \textsc{EqRes} \qquad\qquad \frac{C' \vee u' \approx v' \vee u \approx v}{(C' \vee v \not\approx v' \vee u \approx v')\sigma}\ \textsc{EqFact}$$

For EqRes: $\sigma \in \mathrm{CSU}(u, u')$ and $(u \not\approx u')\sigma$ is eligible in the premise. For EqFact: $\sigma \in \mathrm{CSU}(u, u')$, $u'\sigma \not\precsim v'\sigma$, $u\sigma \not\precsim v\sigma$, and $(u \approx v)\sigma$ is eligible in the premise.

Argument congruence, a higher-order concern, is embodied by the rule

$$\frac{C' \vee s \approx s'}{C'\sigma \vee (s\sigma)\, \bar{x}_n \approx (s'\sigma)\, \bar{x}_n}\ \textsc{ArgCong}$$

where σ is the most general type substitution that ensures well-typedness of the conclusion. In particular, if the result type of s is not a type variable, σ is the identity substitution; and if the result type is a type variable, it is instantiated with $\bar{\alpha}_n \to \beta$, where $\bar{\alpha}_n$ and β are fresh type variables, yielding infinitely many conclusions, one for each n. The literal $s\sigma \approx s'\sigma$ must be strictly eligible in $(C' \vee s \approx s')\sigma$, and $\bar{x}_n$ is a nonempty tuple of distinct fresh variables.

The rules are complemented by an axiom expressing functional extensionality:

$$y\, (\mathsf{diff}\langle \alpha, \beta\rangle\, y\, z) \not\approx z\, (\mathsf{diff}\langle \alpha, \beta\rangle\, y\, z) \vee y \approx z$$

The symbol $\mathsf{diff} : \Pi\alpha, \beta.\ (\alpha \to \beta) \to (\alpha \to \beta) \to \alpha$ is a Skolem symbol.

Rationale for the Rules. The calculus realizes the following division of labor: Sup and FluidSup are responsible for green subterms, which are outside λs, ArgCong indirectly gives access to the remaining positions outside λs, and the extensionality axiom takes care of subterms occurring inside λs.

Example 1. Applied variables give rise to subtle situations with no counterparts in first-order logic. Consider the clauses

$$\mathsf{f}\ \mathsf{a} \approx \mathsf{c} \qquad\qquad \mathsf{h}\ (y\ \mathsf{b})\ (y\ \mathsf{a}) \not\approx \mathsf{h}\ (\mathsf{g}\ (\mathsf{f}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{c})$$

where $\mathsf{f}\,\mathsf{a} \succ \mathsf{c}$. It is easy to see that the clause set is unsatisfiable, by grounding the second clause with $\theta = \{y \mapsto (\lambda x.\ \mathsf{g}\ (\mathsf{f}\ x))\}$. However, to mimic the superposition inference that can be performed at the ground level, it is necessary to superpose at an imaginary position *below* the applied variable y and yet *above* its argument a, namely, into the subterm $\mathsf{f}\ \mathsf{a}$ of $\mathsf{g}\ (\mathsf{f}\ \mathsf{a}) = (\lambda x.\ \mathsf{g}\ (\mathsf{f}\ x))\ \mathsf{a} = (y\ \mathsf{a})\theta$. FluidSup's z variable effectively transforms $\mathsf{f}\ \mathsf{a} \approx \mathsf{c}$ into $z\ (\mathsf{f}\ \mathsf{a}) \approx z\ \mathsf{c}$, whose left-hand side can be unified with $y\ \mathsf{a}$ by taking $\{y \mapsto (\lambda x.\ z\ (\mathsf{f}\ x))\}$. The resulting clause is $\mathsf{h}\ (z\ (\mathsf{f}\ \mathsf{b}))\ (z\ \mathsf{c}) \not\approx \mathsf{h}\ (\mathsf{g}\ (\mathsf{f}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{c})$, which has the right form for EqRes.

Example 2. Third-order clauses in which variables are applied to λ-expressions can be even more stupefying. The clause set

$$\mathsf{f}\,\mathsf{a} \approx \mathsf{c} \qquad\qquad \mathsf{h}\,(y\,(\lambda x.\,\mathsf{g}\,(\mathsf{f}\,x))\,\mathsf{a})\,y \not\approx \mathsf{h}\,(\mathsf{g}\,\mathsf{c})\,(\lambda w\,x.\,w\,x)$$

is unsatisfiable. To see this, apply $\theta = \{y \mapsto (\lambda w\,x.\,w\,x)\}$ to the second clause: $\mathsf{h}\,(\mathsf{g}\,(\mathsf{f}\,\mathsf{a}))\,(\lambda w\,x.\,w\,x) \not\approx \mathsf{h}\,(\mathsf{g}\,\mathsf{c})\,(\lambda w\,x.\,w\,x)$. Let $\mathsf{f}\,\mathsf{a} \succ \mathsf{c}$. A Sup inference is possible between the two ground clauses. But at the nonground level, the subterm $\mathsf{f}\,\mathsf{a}$ is not clearly localized: $\mathsf{g}(\mathsf{f}\mathsf{a}) = (\lambda x.\mathsf{g}(\mathsf{f}x))\mathsf{a} = (\lambda wx.wx)(\lambda x.\mathsf{g}(\mathsf{f}x))\mathsf{a} = (y(\lambda x.\mathsf{g}(\mathsf{f}x))\mathsf{a})\theta$. FluidSup can cope with this. One of the unifiers of $z\,(\mathsf{f}\,\mathsf{a})$ and $y\,(\lambda x.\,\mathsf{g}\,(\mathsf{f}\,x))\,\mathsf{a}$ will be $\{y \mapsto (\lambda w\,x.\,w\,x),\, z \mapsto \mathsf{g}\}$, yielding $\mathsf{h}\,(\mathsf{g}\,\mathsf{c})\,(\lambda w\,x.\,w\,x) \not\approx \mathsf{h}\,(\mathsf{g}\,\mathsf{c})\,(\lambda w\,x.\,w\,x)$.

Because it gives rise to flex–flex pairs (unification constraints where both sides are applied variables), FluidSup can be very prolific. The extensionality axiom is another prime source of flex–flex pairs.

Due to order restrictions and fairness, we cannot postpone solving flex–flex pairs indefinitely. Thus, we cannot use Huet's pre-unification procedure [36] and must instead choose a complete procedure such as Jensen and Pietrzykowski's [37] or Snyder and Gallier's [57]. On the positive side, optional inference rules can efficiently cover many cases where FluidSup or the extensionality axiom would otherwise be needed, and heuristics can help keep the explosion under control. Moreover, flex–flex pairs are not always as bad as their reputation; for example, $y\,\mathsf{a}\,\mathsf{b} \stackrel{?}{=} z\,\mathsf{c}\,\mathsf{d}$ admits a most general unifier: $\{y \mapsto (\lambda w\,x.\,y'\,w\,x\,\mathsf{c}\,\mathsf{d}),\, z \mapsto y'\,\mathsf{a}\,\mathsf{b}\}$.

The calculus is a graceful generalization of standard superposition, except for the extensionality axiom. From $\mathsf{g}\,x \approx \mathsf{f}\,x\,x$, the axiom can be used to derive clauses such as $(\lambda x.\,y\,x\,(\mathsf{g}\,x)) \approx (\lambda x.\,y\,x\,(\mathsf{f}\,x\,x))$, which are useless if the problem is first-order.

Redundancy Criterion. A redundant (or composite) clause is usually defined as a clause whose ground instances are entailed by smaller ($\prec$) ground instances of existing clauses. This would be too strong for our calculus; for example, it would make ArgCong inferences redundant. Our solution is to base the redundancy criterion on a weaker ground logic in which argument congruence and extensionality are not guaranteed to hold.

The weaker logic is defined via an encoding $\lfloor\ \rfloor$ of ground λ-terms into first-order terms. The $\lfloor\ \rfloor$ encoding indexes each symbol occurrence with its type arguments and argument count. Thus, $\lfloor \mathsf{f} \rfloor = \mathsf{f}_0$, $\lfloor \mathsf{f}\,\mathsf{a} \rfloor = \mathsf{f}_1(\mathsf{a}_0)$, and $\lfloor \mathsf{g}\langle\iota\rangle \rfloor = \mathsf{g}_0^\iota$. In addition, it conceals λs by replacing them with fresh symbols. These measures effectively disable argument congruence and extensionality. For example, the clause sets $\{\mathsf{g}_0 \approx \mathsf{f}_0,\, \mathsf{g}_1(\mathsf{a}_0) \not\approx \mathsf{f}_1(\mathsf{a}_0)\}$ and $\{\mathsf{b}_0 \approx \mathsf{a}_0,\, \mathsf{c}_0 \not\approx \mathsf{d}_0\}$ are satisfiable, even though $\{\mathsf{g} \approx \mathsf{f},\, \mathsf{g}\,\mathsf{a} \not\approx \mathsf{f}\,\mathsf{a}\}$ and $\{\mathsf{b} \approx \mathsf{a},\, (\lambda x.\,\mathsf{b}) \not\approx (\lambda x.\,\mathsf{a})\}$ are unsatisfiable.

Given a ground higher-order signature $(\Sigma_{\mathsf{ty}}, \{\}, \Sigma, \{\})$, we define a first-order signature $(\Sigma_{\mathsf{ty}}, \{\}, \Sigma^{\downarrow}, \{\})$ as follows. The type constructors Σ_{ty} are the same in both signatures, but $\to$ is uninterpreted in first-order logic. For each ground instance $\mathsf{f}\langle\bar{\upsilon}\rangle : \tau_1 \to \cdots \to \tau_n \to \tau$ of a symbol $\mathsf{f} \in \Sigma$, we introduce a first-order symbol $\mathsf{f}_j^{\bar{\upsilon}} \in \Sigma^{\downarrow}$ with argument types $\bar{\tau}_j$ and result type $\tau_{j+1} \to \cdots \to \tau_n \to \tau$, for

each j. Moreover, for each ground term $\lambda x.\, t$, we introduce a symbol $\lfloor \lambda x.\, t \rfloor \in \Sigma^{\downarrow}$ of the same type.

The $\lfloor\ \rfloor$ encoding is defined on ground η-short β-normal forms so that $\lambda x.\, t$ is mapped to the symbol $\lfloor \lambda x.\, t \rfloor$ and $\lfloor \mathsf{f}\langle \bar{\upsilon} \rangle\, \bar{s}_j \rfloor = \mathsf{f}_j^{\bar{\upsilon}}(\lfloor \bar{s}_j \rfloor)$ recursively. The encoding is extended to literals and clauses elementwise. Using the inverse mapping $\lceil\ \rceil$, the order $\succ$ can be transferred to the first-order level by defining $t \succ s$ as $\lceil t \rceil \succ \lceil s \rceil$. A crucial property of $\lfloor\ \rfloor$ is that green subterms of a term t correspond to first-order subterms of $\lfloor t \rfloor$. Thus, the subterms considered by Sup and FluidSup coincide with the the subterms exposed to the redundancy criterion.

In standard superposition, redundancy employs the entailment relation $\models$ on ground clauses. We define redundancy of higher-order clauses in the same way, but using $\models$ on the $\lfloor\ \rfloor$-encoded clauses. This definition gracefully generalizes the standard first-order notion of redundancy. Formally, a clause C is *redundant with respect to a set of clauses* N if for each ground instance $C\theta$, $\lfloor C\theta \rfloor$ is entailed by ground instances of clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ that are smaller than $\lfloor C\theta \rfloor$. Here, $\mathcal{G}_\Sigma(N)$ denotes the set of ground instances of clauses in N. We call N *saturated up to redundancy* if for each inference from clauses in N, its premise is redundant with respect to N or its conclusion is contained in N or redundant with respect to N.

The saturation procedures of superposition-based provers aggressively delete clauses that are strictly subsumed by other clauses. A clause C *subsumes* D if there exists a substitution σ such that $C\sigma \subseteq D$. A clause C *strictly subsumes* D if C subsumes D but D does not subsume C. For example, $x \approx \mathsf{c}$ strictly subsumes both $\mathsf{a} \approx \mathsf{c}$ and $\mathsf{b} \not\approx \mathsf{a} \lor x \approx \mathsf{c}$. The proof of refutational completeness of resolution and superposition provers relies on the well-foundedness of the strict subsumption relation [54, Section 7]. Unfortunately, this property does not hold for higher-order logic, where $\mathsf{f}\, x\, x \approx \mathsf{c}$ is strictly subsumed by $\mathsf{f}\, (x\, \mathsf{a})\, (x\, \mathsf{b}) \approx \mathsf{c}$, which is strictly subsumed by $\mathsf{f}\, (x\, \mathsf{a}\, \mathsf{a})\, (x\, \mathsf{b}\, \mathsf{b}') \approx \mathsf{c}$, and so on. Subsumption must be restricted to prevent such infinite chains—for example, by requiring that the subsumer is syntactically smaller than or of the same size as the subsumee.

4 Refutational Completeness

Besides soundness, the most important property of the higher-order superposition calculus introduced in Section 3 is refutational completeness:

Theorem 3. *Let $N \not\ni \bot$ be a clause set that is saturated up to redundancy and that contains the extensionality axiom. Then N has a Henkin model.*

The proof is adapted from Bentkamp et al. [11]. We present a brief outline in this section and point to our technical report [10] for the details. Let $N \not\ni \bot$ be a higher-order clause set saturated up to redundancy with respect to the inference rules and that contains the extensionality axiom. The proof proceeds in two steps:

1. Construct a model of the first-order grounded clause set $\lfloor \mathcal{G}_\Sigma(N) \rfloor$, where $\lfloor\ \rfloor$ is the encoding of ground terms used to define redundancy.

2. Lift this first-order model to a higher-order interpretation and show that it is a model of $\mathcal{G}_\Sigma(N)$ and hence of N.

The first step follows the same general idea as the completeness proof for standard superposition [5, 50, 63]. We construct a term rewriting system R_∞ and use it to define a candidate interpretation that equates all terms that share the same normal form with respect to R_∞. At this level, expressions $\lambda x.\, t$ are regarded as uninterpreted symbols $\lfloor \lambda x.\, t \rfloor$.

As in the standard proof, it is the set N, and not its grounding $\mathcal{G}_\Sigma(N)$, that is saturated. We must show that there exist nonground inferences corresponding to all necessary ground Sup, EqRes, and EqFact inferences. We face two specifically higher-order difficulties. First, in standard superposition, we can avoid Sup inferences into variables x by exploiting the order's compatibility with contexts: If $t' \prec t$, we have $C\{x \mapsto t'\} \prec C\{x \mapsto t\}$, which allows us to invoke the induction hypothesis at a key point in the argument to establish the truth of $C\{x \mapsto t'\}$. This technique fails for higher-order variables x that occur applied in C, because the order lacks compatibility with arguments. Hence, our Sup rule must perform some inferences into variables. The other difficulty also concerns applied variables. We must show that any necessary ground Sup inference into a position corresponding to a fluid term or a deep variable on the nonground level can be lifted to a FluidSup inference. This involves showing that the z variable in FluidSup can represent arbitrary contexts around a term t.

For the first-order model construction, $\beta\eta$-normalization is the proverbial dog that did not bark. At the ground level, the rules Sup, EqRes, and EqFact preserve η-short β-normal form, and so does first-order term rewriting. Thus, we can completely ignore $\rightarrow_\beta$ and $\rightarrow_\eta$. At the nonground level, β- and η-reduction can arise only through instantiation. This poses no difficulties thanks to the order's stability under substitution.

The second step of the completeness proof consists of constructing a higher-order interpretation and proving that it is a model of $\mathcal{G}_\Sigma(N)$, and hence of N. The difficulty is to show that the symbols representing λ-expressions behave like the λ-expressions they represent. This step relies on saturation with respect to the ArgCong rule—which connects a λ-expression with its value when applied to an argument x—and on the presence of the extensionality axiom.

5 Extensions

The calculus can be extended to make it more practical. The familiar simplification machinery can be adapted to higher-order terms by considering green contexts instead of arbitrary contexts. Optional inference rules provide lightweight alternatives to the extensionality axiom.

Two of the rules below are based on "orange subterms." A λ-term t is an *orange subterm* of a λ-term s if $s = t$; or if $s = \mathsf{f}\langle\bar{\tau}\rangle\, \bar{s}$ and t is an orange subterm of s_i for some i; or if $s = x\, \bar{s}$ and t is an orange subterm of s_i for some i; or if $s = (\lambda x.\, u)$ and t is an orange subterm of u. In $\mathsf{f}\,(\mathsf{g}\,\mathsf{a})\,(y\,\mathsf{b})\,(\lambda x.\, \mathsf{h}\,\mathsf{c}\,(\mathsf{g}\,x))$, the orange subterms include b, c, x, $\mathsf{g}\,x$, $\mathsf{h}\,\mathsf{c}\,(\mathsf{g}\,x)$, and all the green subterms. This

notion is lifted to $\beta\eta$-equivalence classes via representatives in η-short β-normal form. We write $t = s\langle\!\langle \bar{x}_n.\, u \rangle\!\rangle$ to indicate that u is an orange subterm of t, where $\bar{x}_n$ are the variables bound in the *orange context* around u.

Once a term $s\langle\!\langle \bar{x}_n.\, u \rangle\!\rangle$ has been introduced, we write $s\langle\!\langle \bar{x}_n.\, u' \rangle\!\rangle_\eta$ to denote the same context with a different subterm u' at that position. The η subscript is a reminder that u' is not necessarily an orange subterm of $s\langle\!\langle \bar{x}_n.\, u' \rangle\!\rangle_\eta$ due to potential applications of η-reduction. For example, if $s\langle\!\langle x.\, \mathsf{g}\, x\, x \rangle\!\rangle = (\lambda x.\, \mathsf{g}\, x\, x)$, then $s\langle\!\langle x.\, \mathsf{f}\, x \rangle\!\rangle_\eta = (\lambda x.\, \mathsf{f}\, x) = \mathsf{f}$.

Demodulation, which destructively rewrites using an equality $t \approx t'$, is available at green positions. A variant rewrites inside λ-expressions:

$$\frac{t \approx t' \quad C\langle s\langle\!\langle \bar{x}.\, t\sigma \rangle\!\rangle\rangle}{t \approx t' \quad C\langle s\langle\!\langle \bar{x}.\, t'\sigma \rangle\!\rangle_\eta\rangle \quad s\langle\!\langle \bar{x}.\, t\sigma \rangle\!\rangle \approx s\langle\!\langle \bar{x}.\, t'\sigma \rangle\!\rangle_\eta}\ \lambda\textsc{DemodExt}$$

where $s\langle\!\langle \bar{x}.\, t\sigma \rangle\!\rangle{\downarrow_{\beta\eta}}$ is a λ-expression or an applied variable. The term $t\sigma$ may refer to the bound variables $\bar{x}$. Side condition: The second premise is larger than ($\succ$) the second and third conclusion. This ensures that this premise is redundant with respect to these conclusions and may be removed. The double bar indicates that the conclusions collectively make the premises redundant and can replace them. An instance of the rule, where $\mathsf{g}\, z$ is rewritten to $\mathsf{f}\, z\, z$ under a λ, follows:

$$\frac{\mathsf{g}\, x \approx \mathsf{f}\, x\, x \quad \mathsf{k}\, (\lambda z.\, \mathsf{h}\, (\mathsf{g}\, z)) \approx \mathsf{c}}{\mathsf{g}\, x \approx \mathsf{f}\, x\, x \quad \mathsf{k}\, (\lambda z.\, \mathsf{h}\, (\mathsf{f}\, z\, z)) \approx \mathsf{c} \quad (\lambda z.\, \mathsf{h}\, (\mathsf{g}\, z)) \approx (\lambda z.\, \mathsf{h}\, (\mathsf{f}\, z\, z))}\ \lambda\textsc{DemodExt}$$

The next simplification rule can be used to prune arguments to variables that can be expressed as functions of the remaining arguments. For example, the clause $C[y\ \mathsf{a}\ \mathsf{b}\ (\mathsf{f}\ \mathsf{b}\ \mathsf{a}),\ y\ \mathsf{b}\ \mathsf{d}\ (\mathsf{f}\ \mathsf{d}\ \mathsf{b})]$, in which y occurs twice, can be simplified to $C[y'\ \mathsf{a}\ \mathsf{b},\ y'\ \mathsf{b}\ \mathsf{d}]$. The rule can also be used to remove the repeated arguments in $y\ \mathsf{b}\ \mathsf{b} \not\approx y\ \mathsf{a}\ \mathsf{a}$, the static argument a in $y\ \mathsf{a}\ \mathsf{c} \not\approx y\ \mathsf{a}\ \mathsf{b}$, and all four arguments in $y\ \mathsf{a}\ \mathsf{b} \not\approx z\ \mathsf{b}\ \mathsf{d}$. It is stated as

$$\frac{C}{C\{y \mapsto (\lambda \bar{x}_j.\, y'\ \bar{x}_{j-1})\}}\ \textsc{PruneArg}$$

where y' is a fresh variable, the minimum number k of arguments passed to any occurrence of y in the clause $C{\downarrow_{\beta\eta}}$ is at least j, and there exists a term t containing no variables bound in the clause such that $s_j = t\ \bar{s}_{j-1}\ s_{j+1} \ldots s_k$ for all terms of the form $y\ \bar{s}_k$ occurring in the clause. For example, clauses with a static argument correspond to the case $t := (\lambda \bar{x}_{j-1}\ x_{j+1} \ldots x_k.\, u)$, where u is the static argument (containing no variables bound in t) and j is its index in y's argument list.

Following the literature [33, 58], we provide a rule for negative extensionality:

$$\frac{C \vee s \not\approx s'}{C \vee s\ (\mathsf{sk}\langle \bar{\alpha}_m \rangle\ \bar{y}_n) \not\approx s'\ (\mathsf{sk}\langle \bar{\alpha}_m \rangle\ \bar{y}_n)}\ \textsc{NegExt}$$

where sk is a fresh Skolem symbol, $\bar{\alpha}_m, \bar{y}_n$ are the variables occurring free in the the literal $s \not\approx s'$, and $s \not\approx s'$ is eligible in the premise. Negative extensionality can also be applied as a simplification rule to all literals in the initial problem.

Superposition can be generalized to orange subterms as follows:

$$\frac{D' \vee t \approx t' \quad C' \vee [\neg]\, s\langle\!\langle \bar{x}.\, u \rangle\!\rangle \approx s'}{(D' \vee C' \vee [\neg]\, s\langle\!\langle \bar{x}.\, t' \rangle\!\rangle_\eta \approx s')\sigma\rho}\ \lambda\textsc{Sup}$$

Sup's side conditions apply. We also require that $\bar{x}\sigma = \bar{x}$ and that the variables $\bar{x}$ do not occur in $y\sigma$ for all variables y in u. Moreover, let $P_y = \{y\}$ for all type and term variables $y \notin \bar{x}$. For each i, let P_{x_i} be recursively defined as the union of all P_y such that y occurs free in the λ-expression that binds x_i in $s\langle\!\langle \bar{x}.\, u \rangle\!\rangle\sigma$ or that occurs free in the corresponding subterm of $s\langle\!\langle \bar{x}.\, t' \rangle\!\rangle_\eta\sigma$. The substitution ρ is defined as $\{x_i \mapsto \mathsf{sk}_i\langle\bar{\alpha}_i\rangle\, \bar{y}_i$ for each $i\}$, where $\bar{y}_i$ are the term variables in P_{x_i} and $\bar{\alpha}_i$ are the type variables in P_{x_i} and the type variables occurring in the type of the λ-expression binding x_i. The rule can be justified in terms of paramodulation and extensionality, with the Skolem terms standing for diff terms. An instance of the rule follows:

$$\frac{n \approx \mathsf{zero} \vee \mathsf{div}\ n\ n \approx \mathsf{one} \quad \mathsf{prod}\ K\ (\lambda k.\, \mathsf{div}\ (\mathsf{succ}\ k)\ (\mathsf{succ}\ k)) \not\approx \mathsf{one}}{\mathsf{succ}\ \mathsf{sk} \approx \mathsf{zero} \vee \mathsf{prod}\ K\ (\lambda k.\, \mathsf{one}) \not\approx \mathsf{one}}\ \lambda\textsc{Sup}$$

Intuitively, the term $\mathsf{prod}\ K\ (\lambda k.\, u)$ is intended to denote the product $\prod_{k \in K} u$, where k ranges over a finite set K of natural numbers.

6 Implementation

Zipperposition [26, 27] is an open source superposition prover written in OCaml.[1] Originally designed for polymorphic first-order logic (TF1 [19]), it was later extended with an incomplete higher-order mode based on pattern unification [49]. Bentkamp et al. [11] extended it further with a complete λ-free higher-order mode. As a prototype, we have now implemented a Boolean-free higher-order mode based on our calculus.

We use a metaorder induced by a λ-free KBO [8]. We currently use $\succeq$ as the nonstrict term order but could improve precision by employing a more precise computable approximation of $\succsim$.

Except for FluidSup, the core calculus rules already existed in Zipperposition in a similar form. To retrieve candidate right premises for FluidSup, we created an index of all fluid green subterms in the active clause set. Among the proposed higher-order optional rules, we implemented NegExt, λSup, a mildly incomplete variant of λDemodExt without the third conclusion, and a variant of the Prune-Arg rule that removes most functional dependencies that occur in practice.

For unification, we started with Jensen and Pietrzykowski's procedure [37]. The procedure is not ideal because it computes a nonminimal set of unifiers; for

[1] `https://github.com/c-cube/zipperposition`

example, given the flex–flex constraint $y\,\mathsf{a} \stackrel{?}{=} z\,\mathsf{b}$, it generates not only the most general unifier $\{y \mapsto (\lambda w.\ y'\,w\,\mathsf{b}),\ z \mapsto y'\,\mathsf{a}\}$ but also infinitely many superfluous unifiers. It is not clear whether Snyder and Gallier's procedure [57] would behave better. To support polymorphism, we extended Jensen and Pietrzykowski's projection rule to check type unifiability instead of equality and their iteration rule to consider the possibility that a type variable is instantiated with a function type. On the other hand, polymorphism allows us to avoid the enumeration of types in the iteration rule.

To interleave the unification with other computation, our unification procedure returns a possibly infinite stream of subsingletons (sets of cardinality 0 or 1) computed on demand. It can even cope with nonterminating unification problems that do not yield any unifiers, by representing them as an infinite stream of empty sets. We use this procedure for inference rules, keeping simpler pattern-style unification for simplification rules. The inference rules turn the possibly infinite streams of unifiers into possibly infinite streams of clauses—the conclusions of inferences. To consume these streams fairly while giving flexibility to heuristics, we designed a priority queue that associates a weight with each stream. This queue is used in the main given clause loop to store new streams resulting from inferences and to extract clauses, which are then moved to the passive clause set.

Based on informal experiments, we developed or tuned a few general heuristics of Zipperposition. Definition unfolding, in conjunction with β-reduction, transforms many higher-order TPTP problems into first-order problems. We also modified KBO's weight generation scheme to take symbol frequencies into account and modified other heuristics to prioritize clauses containing symbols present in the conjecture.

7 Evaluation

We evaluated our prototype implementation of the calculus in Zipperposition with other higher-order provers and with Zipperposition's modes for less expressive logics. All of the experiments presented in this section were performed on StarExec nodes equipped with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz. Provers were invoked with a CPU time limit of 300 s. The raw data are available online.[2]

We used both standard TPTP benchmarks [59] and Sledgehammer-generated benchmarks. From the TPTP, we selected all 709 TFF (monomorphic and polymorphic first-order) problems without arithmetic and all 597 TH0 (monomorphic higher-order) problems without first-class Booleans and arithmetic. We partitioned the TH0 problems into those containing no λs (TH0λf, 545 problems) and those containing λs (TH0λ, 52 problems). The Sledgehammer benchmarks, corresponding to Isabelle's Judgment Day suite [22], were regenerated to target Boolean-free higher-order logic. They comprise 5012 problems, divided in two groups based on the number of Isabelle facts (lemmas, definitions, etc.) selected for inclusion in each problem: either 256 (SH256) or 16 facts (SH16). Each group

[2] `http://matryoshka.gforge.inria.fr/pubs/lamsup_results.tgz`

	TFF	TH0λf	TH0λ	SH256-ll	SH16-ll	SH256-λ	SH16-λ
Leo-III	85	387	**42**	234	323	228	338
Satallax	–	400	**42**	495	371	516	384
Ehoh	–	396	–	**671**	397	–	–
FOZip	**238**	–	–	–	–	–	–
@+FOZip	194	398	–	495	389	–	–
λfreeZip	233	401	–	603	**401**	–	–
λZip-full	178	388	27	394	351	385	348
λZip-pragmatic	227	416	27	560	386	**567**	**387**
λZip-competitive	216	**418**	40	413	351	399	357
Leo-III-meta	252	438	44	706	412	688	416
Satallax-meta	–	427	42	491	372	513	385

Figure 1: Number of proved problems

is further divided into two subgroups based on the processing of λ-expressions: SH256-λ and SH16-λ preserve λ-expressions, whereas SH256-ll and SH16-ll encode them as λ-lifted supercombinators [48] to make the problems accessible to λ-free higher-order provers.

We chose Leo-III 1.3 and Satallax 3.3 as representatives of the state of the art. These are cooperative higher-order provers that can be set up to regularly invoke first-order provers as terminal proof procedures. Leo-III can be used on its own or as a metaprover (Leo-III-meta) with CVC4, E, and iProver as backends. Satallax can be used on its own or as a metaprover (Satallax-meta) with E. We also included Ehoh [62], the λ-free higher-order mode of E 2.3. For Zipperposition, we included its first-order and λ-free modes (FOZip and λfreeZip) as well as a mode that performs an applicative encoding [62, Section 2] before invoking the first-order mode (@+FOZip). We experimented with three variants of our calculus implementation. λZip-full is designed to be refutationally complete. λZip-pragmatic disables FluidSup and the extensionality axiom, and uses a lightweight higher-order unification algorithm instead of Jensen and Pietrzykowski's procedure. Finally, λZip-competitive is a variant of λZip-pragmatic that is further tuned for small problems requiring a substantial amount of higher-order reasoning.

A summary of our experiments is presented in Figure 1. To enhance readability, we highlight in bold the winning system for each column *excluding the metaprovers*. We observe that Leo-III-meta emerges as winner on all benchmark sets, but λZip-pragmatic and λZip-competitive compare very well with Leo-III and Satallax. In contrast, λZip-full cannot seem to keep its FluidSup rule and extensionality under control. More research into heuristics design appears necessary.

It is disappointing that on Sledgehammer problems (SH256 and SH16), we obtain better performance by using λfreeZip with λ-lifting than using λZip with native λs. On TH0λf problems, the situation is reversed. This seems to suggest that λ reasoning is rarely needed for Sledgehammer problems. Clearly, this is another area where research into heuristics design could be beneficial.

8 Discussion and Related Work

Bentkamp et al. [11] introduced four calculi for λ-free higher-order logic organized along two axes: *intensional* versus *extensional*, and *nonpurifying* versus *purifying*. The purifying calculi flatten the clauses containing applied variables, thereby eliminating the need for superposition into variables. As we extended their work to support λs, we found the purification approach problematic and quickly gave it up because it needs x to be smaller than $x\ t$, which is impossible to achieve with a term order on $\beta\eta$-equivalence classes. As for extensionality, it is the norm for higher-order unification [30] and is employed in the TPTP THF format [60] and in proof assistants such as HOL4, HOL Light, Isabelle/HOL, Lean, Nuprl, and PVS. Bentkamp et al. viewed their approach as "a stepping stone towards full higher-order logic." It already included a notion analogous to green subterms and an ArgCong rule, which help cope with the complications occasioned by β-reduction.

Our superposition calculus joins the family of proof systems for higher-order logic. Closely related are Andrews's higher-order resolution [1], Huet's constrained resolution [35], Jensen and Pietrzykowski's ω-resolution [37], Snyder's higher-order E-resolution [56], Benzmüller and Kohlhase's extensional higher-order resolution [13], and Benzmüller's higher-order unordered paramodulation and RUE resolution [12]. A noteworthy variant is Steen and Benzmüller's higher-order ordered paramodulation [58], whose order restrictions undermine refutational completeness but yield good empirical results. Other approaches are based on analytic tableaux [6,42,43,52], connections [2], sequents [46], and satisfiability modulo theories [7]. Andrews [3] and Benzmüller and Miller [14] provide excellent surveys.

The main advantage of our calculus is that it gracefully generalizes the highly successful first-order superposition rules without sacrificing refutational completeness. It also includes a powerful simplification rule, PruneArg, that could be useful in other provers. Among the drawbacks of our approach are the need to solve flex–flex pairs eagerly and the explosion caused by the extensionality axiom. We believe that this is a reasonable trade-off, especially for large problems with a substantial first-order component, such as those originating from proof assistants.

Our prototype λZipperposition joins the league of higher-order automatic theorem provers. We briefly list some of its rivals. TPS [4] is based on the connection method and expansion proofs. LEO [13] and Leo-II [16] implement variants of RUE resolution. Leo-III [58] is based on higher-order paramodulation. Satallax [23] implements a higher-order tableau calculus guided by a SAT solver. Leo-II, Leo-III, and recent versions of Satallax integrate first-order provers as terminal procedures. AgsyHOL [46] is based on a focused sequent calculus guided by narrowing. Finally, there is ongoing work by the developers of CVC4, veriT, and Vampire to extend their provers to higher-order logic [7, 17].

Half a century ago, Robinson [53] proposed to reduce higher-order logic to first-order logic via a translation. Tools such as Sledgehammer [51], Miz$\mathbb{AR}$ [61], HOLyHammer [41], and CoqHammer [28] have since popularized this approach. Such translations must eliminate the λ-expressions, typically using SKBCI combinators or λ-lifting [48], and encode typing information [18]. Most translations are implemented outside provers, but hybrid approaches are also possible [17, 29].

9 Conclusion

We presented a superposition calculus for a Boolean-free fragment of extensional polymorphic higher-order logic. With the notable exception of a functional extensionality axiom, it gracefully generalizes standard superposition. Our prototype prover Zipperposition shows promising results on TPTP and Isabelle benchmarks. In future work, we plan to pursue five main avenues of investigation.

We first plan to *extend the calculus to support Booleans and Hilbert choice.* Booleans are notoriously explosive. We want to experiment with both axiomatizations and native support in the calculus. Native support would likely take the form of a primitive substitution rule that enumerates predicate instantiations [2], delayed clausification rules [32], and rules for reasoning about Hilbert choice.

We want to investigate techniques to *curb the explosion caused by functional extensionality.* The extensionality axiom reintroduces the search space explosion that the calculus's order restrictions aim at avoiding.

We will also look into approaches to *curb the explosion caused by higher-order unification.* Our calculus suffers because it needs to solve flex–flex pairs. Existing procedures [37, 57] enumerate redundant unifiers. This can probably be avoided to some extent. It could also be interesting to investigate unification algorithms that would delay imitation/projection choices via special schematic variables, inspired by Libal's concise representation of regular unifiers [45].

We clearly need to *fine-tune and develop heuristics.* We expect heuristics to be a fruitful area for future research in higher-order reasoning. Proof assistants are an inexhaustible source of easy-looking benchmarks that are beyond the power of today's provers. Whereas "hard higher-order" may remain forever out of reach, there is a substantial "easy higher-order" fragment that awaits automation.

Finally, we plan to *implement the calculus in a state-of-the-art prover.* A suitable basis for an optimized implementation of our calculus would be Ehoh, the λ-free higher-order version of the E prover developed by Vukmirović et al. [62].

Acknowledgment. Simon Cruanes patiently explained Zipperposition's internals and allowed us to continue the development of his prover. Christoph Benzmüller and Alexander Steen shared insights and examples with us, guiding us through the literature and clarifying how the Leos work. Maria Paola Bonacina and Nicolas Peltier gave us some ideas on how to treat the extensionality axiom as a theory axiom, ideas we have yet to explore. Mathias Fleury helped us set up regression tests for Zipperposition. Ahmed Bhayat, Tomer Libal, and Enrico Tassi shared their insights on higher-order unification. Andrei Popescu and Dmitriy Traytel explained the terminology surrounding the λ-calculus. Haniel Barbosa, Daniel El Ouraoui, Pascal Fontaine, and Hans-Jörg Schurr were involved in many stimulating discussions. Christoph Weidenbach made this collaboration possible. Ahmed Bhayat, Mark Summerfield, and the anonymous reviewers suggested several textual improvements. We thank them all.

Bentkamp, Blanchette, and Vukmirović's 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). Bentkamp and Blanchette also benefited from the Netherlands Organization for Scientific Research (NWO) Incidental Financial Support scheme. Blanchette has received funding from the NWO under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

[1] Andrews, P.B.: Resolution in type theory. J. Symb. Log. 36(3), 414–432 (1971)

[2] Andrews, P.B.: On connections and higher-order logic. J. Autom. Reason. 5(3), 257–291 (1989)

[3] Andrews, P.B.: Classical type theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, pp. 965–1007. Elsevier and MIT Press (2001)

[4] Andrews, P.B., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., Xi, H.: TPS: A theorem-proving system for classical type theory. J. Autom. Reason. 16(3), 321–353 (1996)

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

[6] Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. J. Autom. Reason. 47(4), 451–479 (2011)

[7] Barbosa, H., Reynolds, A., Fontaine, P., Ouraoui, D.E., Tinelli, C.: Higher-order SMT solving (work in progress). In: Dimitrova, R., D'Silva, V. (eds.) SMT 2018 (2018)

[8] Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: A transfinite Knuth–Bendix order for lambda-free higher-order terms. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 432–453. Springer (2017)

[9] Bentkamp, A.: Formalization of the embedding path order for lambda-free higher-order terms. Archive of Formal Proofs (2018), `http://isa-afp.org/entries/Lambda_Free_EPO.html`

[10] Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas (technical report). Technical report (2019), `http://matryoshka.gforge.inria.fr/pubs/lamsup_report.pdf`

[11] Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 28–46. Springer (2018)

[12] Benzmüller, C.: Extensional higher-order paramodulation and RUE-resolution. In: Ganzinger, H. (ed.) CADE-16. LNCS, vol. 1632, pp. 399–413. Springer (1999)

[13] Benzmüller, C., Kohlhase, M.: Extensional higher-order resolution. In: Kirchner, C., Kirchner, H. (eds.) CADE-15. LNCS, vol. 1421, pp. 56–71. Springer (1998)

[14] Benzmüller, C., Miller, D.: Automation of higher-order logic. In: Siekmann, J.H. (ed.) Computational Logic, Handbook of the History of Logic, vol. 9, pp. 215–254. Elsevier (2014)

[15] Benzmüller, C., Paulson, L.C.: Multimodal and intuitionistic logics in simple type theory. Log. J. IGPL 18(6), 881–892 (2010)

[16] Benzmüller, C., Sultana, N., Paulson, L.C., Theiss, F.: The higher-order prover Leo-II. J. Autom. Reason. 55(4), 389–404 (2015)

[17] Bhayat, A., Reger, G.: Set of support for higher-order reasoning. In: Konev, B., Urban, J., Rümmer, P. (eds.) PAAR-2018. CEUR Workshop Proceedings, vol. 2162, pp. 2–16. CEUR-WS.org (2018)

[18] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Log. Meth. Comput. Sci. 12(4) (2016)

[19] Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) CADE-24. LNCS, vol. 7898, pp. 414–420. Springer (2013)

[20] Blanchette, J.C., Waldmann, U., Wand, D.: A lambda-free higher-order recursive path order. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 461–479. Springer (2017)
[21] Blanqui, F., Jouannaud, J.P., Rubio, A.: The computability path ordering. Log. Meth. Comput. Sci. 11(4) (2015)
[22] Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer (2010)
[23] Brown, C.E.: Satallax: An automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 111–117. Springer (2012)
[24] de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. Indag. Math 75(5), 381–392 (1972)
[25] Cervesato, I., Pfenning, F.: A linear spine calculus. J. Log. Comput. 13(5), 639–688 (2003)
[26] Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. Ph.D. thesis, École polytechnique (2015)
[27] Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS, vol. 10483, pp. 172–188. Springer (2017)
[28] Czajka, Ł., Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory (2018)
[29] Dougherty, D.J.: Higher-order unification via combinators. Theor. Comput. Sci. 114(2), 273–298 (1993)
[30] Dowek, G.: Higher-order unification and matching. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, pp. 1009–1062. Elsevier and MIT Press (2001)
[31] Fitting, M.: Types, Tableaus, and Gödel's God. Kluwer (2002)
[32] Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. Information and Computation 199(1–2), 3–23 (2005)
[33] Gupta, A., Kovács, L., Kragl, B., Voronkov, A.: Extensional crisis and proving identity. In: Cassez, F., Raskin, J. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 185–200. Springer (2014)
[34] Henkin, L.: Completeness in the theory of types. J. Symb. Log. 15(2), 81–91 (1950)
[35] Huet, G.P.: A mechanization of type theory. In: Nilsson, N.J. (ed.) IJCAI-73. pp. 139–146. William Kaufmann (1973)
[36] Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. 1(1), 27–57 (1975)
[37] Jensen, D.C., Pietrzykowski, T.: Mechanizing ω-order type theory through unification. Theor. Comput. Sci. 3(2), 123–171 (1976)
[38] Jouannaud, J.P., Rubio, A.: Rewrite orderings for higher-order terms in eta-long beta-normal form and recursive path ordering. Theor. Comput. Sci. 208(1–2), 33–58 (1998)
[39] Jouannaud, J.P., Rubio, A.: Polymorphic higher-order recursive path orderings. J. ACM 54(1), 2:1–2:48 (2007)
[40] Kaliszyk, C., Sutcliffe, G., Rabe, F.: TH1: The TPTP typed higher-order form with rank-1 polymorphism. In: Fontaine, P., Schulz, S., Urban, J. (eds.) PAAR-2016. CEUR Workshop Proceedings, vol. 1635, pp. 41–55. CEUR-WS.org (2016)
[41] Kaliszyk, C., Urban, J.: HOL(y)Hammer: Online ATP service for HOL Light. Math. Comput. Sci. 9(1), 5–22 (2015)
[42] Kohlhase, M.: Higher-order tableaux. In: Baumgartner, P., Hähnle, R., Posegga, J. (eds.) TABLEAUX '95. LNCS, vol. 918, pp. 294–309. Springer (1995)

[43] Konrad, K.: HOT: A concurrent automated theorem prover based on higher-order tableaux. In: Grundy, J., Newey, M.C. (eds.) TPHOLs '98. LNCS, vol. 1479, pp. 245–261. Springer (1998)

[44] Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer (2013)

[45] Libal, T.: Regular patterns in second-order unification. In: Felty, A.P., Middeldorp, A. (eds.) CADE-25. LNCS, vol. 9195, pp. 557–571. Springer (2015)

[46] Lindblad, F.: A focused sequent calculus for higher-order logic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 61–75. Springer (2014)

[47] Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theor. Comput. Sci. 192(1), 3–29 (1998)

[48] Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reason. 40(1), 35–60 (2008)

[49] Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. Log. Comput. 1(4), 497–536 (1991)

[50] Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, pp. 371–443. Elsevier and MIT Press (2001)

[51] Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) IWIL-2010. EPiC, vol. 2, pp. 1–11. EasyChair (2012)

[52] Robinson, J.: Mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 151–170. Edinburgh University Press (1969)

[53] Robinson, J.: A note on mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 5, pp. 121–135. Edinburgh University Press (1970)

[54] Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalizing Bachmair and Ganzinger's ordered resolution prover. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 89–107. Springer (2018)

[55] Schulz, S.: System description: E 1.8. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) LPAR-19. LNCS, vol. 8312, pp. 735–743. Springer (2013)

[56] Snyder, W.: Higher order E-unification. In: Stickel, M.E. (ed.) CADE-10. LNCS, vol. 449, pp. 573–587. Springer (1990)

[57] Snyder, W., Gallier, J.H.: Higher-order unification revisited: Complete sets of transformations. J. Symb. Comput. 8(1/2), 101–140 (1989)

[58] Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 108–116. Springer (2018)

[59] Sutcliffe, G.: The TPTP problem library and associated infrastructure—from CNF to TH0, TPTP v6.4.0. J. Autom. Reason. 59(4), 483–502 (2017)

[60] Sutcliffe, G., Benzmüller, C., Brown, C.E., Theiss, F.: Progress in the development of automated theorem proving for higher-order logic. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 116–130. Springer (2009)

[61] Urban, J., Rudnicki, P., Sutcliffe, G.: ATP and presentation service for Mizar formalizations. J. Autom. Reason. 50(2), 229–241 (2013)

[62] Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. pp. 192–210. LNCS, Springer (2019)

[63] Waldmann, U.: Automated reasoning II. Lecture notes, Max-Planck-Institut für Informatik (2016), http://resources.mpi-inf.mpg.de/departments/rg1/teaching/autrea2-ss16/script-current.pdf

[64] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 140–145. Springer (2009)