Adding Sorts to an Isabelle Formalization of Superposition

Balazs Toth
balazs.toth@ifi.lmu.de
LMU München
Munich, Germany

Martin Desharnais-Schäfer
desharnais-schaefer@ifi.lmu.de
LMU München
Munich, Germany

Jasmin Blanchette
jasmin.blanchette@ifi.lmu.de
LMU München
Munich, Germany

Abstract

The superposition calculus has been formalized in Isabelle/HOL twice before but in both cases without a type system. Nowadays, modern superposition provers support types. We extend an existing Isabelle formalization of untyped superposition with simple monomorphic types, or sorts. This extension is straightforward on paper but surprisingly tricky to implement formally. We also use this opportunity to refactor the proof text to avoid quadruplicated definitions, lemmas, and proofs about terms, atoms, literals, and clauses. The extended formalization and its refactoring benefit from Isabelle's locales, structured Isar proofs, and Sledgehammer proof tool.

CCS Concepts: • **Computing methodologies** → **Theorem proving algorithms**.

Keywords: Superposition, verification, first-order logic, higher-order logic

ACM Reference Format:
Balazs Toth, Martin Desharnais-Schäfer, and Jasmin Blanchette. 2026. Adding Sorts to an Isabelle Formalization of Superposition. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779031.3779099

1 Introduction

There is an undeniable self-referential thrill to formalizing (automatic) theorem provers using (interactive) theorem provers. There have been many such formalization efforts, especially in the past decade. Notably, Bachmair and Ganzinger's superposition calculus [3], which is the basis of many modern first-order automatic provers, has been formalized at least twice in Isabelle/HOL, once by Peltier [19] and once by Waldmann, Tourret, and us [12].

However, there is a mismatch between the Isabelle formalizations and the logics implemented by modern provers such as E [21], SPASS [32], Vampire [17], and Zipperposition [11]. The formalizations, just like Bachmair and Ganzinger's original description of superposition, are for an untyped logic, whereas modern provers support types. A typing discipline enables different cardinalities for different domains.

In particular, types are useful to encode predicates. For example, the two-clause set $\{x \approx y, \neg \mathrm{p}\}$, where $\approx$ denotes equality, x and y are universally quantified term variables, and p is a nullary predicate symbol, is satisfiable in any interpretation equipped with a single-element domain, but its naive untyped encoding as $\{x \approx y, \mathrm{p} \not\approx \mathrm{t}\}$, where t encodes truth, is unsatisfiable. With a type system, we can assign different types to the variables and to the Boolean symbols and interpret them differently. Then, the encoding $\{x \approx y, \mathrm{p} \not\approx \mathrm{t}\}$ is satisfiable, like the original clause set. While types can be soundly and completely encoded in an untyped logic, native types are substantially more efficient than encodings [13, Table 9].

In the present paper, we extend our formalization [12] to support simple monomorphic types, or *sorts*. The formalization modularly separates the ground (i.e., variable-free) and nonground aspects of the calculus, using the *saturation framework* formulated by Waldmann et al. [29] and formalized by Tourret and Blanchette [27]. The development is closer to Bachmair and Ganzinger's proof than Peltier's; notably, it represents clauses as finite multisets of literals, and not as sets, a consequential difference.

Adding simple types to superposition on paper is generally considered straightforward, but there are many intricate details that arise when working formally. In particular, it is difficult to figure out where to add the typing predicates and environments, especially in conjunction with substitutions and most general unifiers. Other difficulties are connected to the use of the saturation framework.

In addition, we refactor the formal development to isolate reusable background theories about types, substitutions, orders, entailment, and clauses. We use these theories to lift definitions, lemmas, and proofs on terms to atoms, literals, and clauses, instead of having to quadruplicate the definitions, lemmas, and proofs.

Like the superposition calculus itself, our formalization takes various parameters that must fulfill specific assumptions. We use Isabelle's locale mechanism to capture these dependencies and facilitate reuse. To ensure that our definitions are not vacuous, we formally prove that all assumptions

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779099

can be fulfilled. In addition to locales, we rely on structured Isar proofs [33] and the Sledgehammer proof tool [18], which integrates superposition provers and satisfiability-modulo-theories solvers. Using superposition provers to prove their own metatheory gives our project a self-referential flavor.

Our development is part of the IsaFoL (Isabelle Formalization of Logic) effort [6] and is available in the *Archive of Formal Proofs* [14, 15, 25].

2 Background

Before we present our work, we first introduce a number of concepts and notations related to clausal first-order logic with equality, the superposition calculus, and the Isabelle/HOL proof assistant.

2.1 Clausal First-Order Logic with Equality

We consider a first-order logic with equality. A *term* is defined inductively as either a variable x or a function application $\mathsf{f}(t_1, \ldots, t_n)$ for a function symbol f and a (possibly empty) list of terms $t_1, \ldots, t_n$. The number of arguments n is called the arity of f. If $n = 0$, we write f instead of $\mathsf{f}()$.

An *atom* is an unordered pair of terms, typically written as an equation $t \approx t'$. A *literal* is either an atom $t \approx t'$ or a negated atom $t \not\approx t'$. A *clause* is a finite multiset $\{l_1, \ldots, l_n\}$ of literals, typically written as a disjunction $l_1 \vee \cdots \vee l_n$. The symbol $\bot$ denotes the empty clause (or empty disjunction), which is false. All variables in a clause are to be understood as implicitly universally quantified in that clause.

A *context* c is a term with one designated position that is to be filled by another term—in other words, a term with a hole. We use the syntax $c[t]$ to represent the term consisting of a subterm t in a context c. We write $\Box$ for the empty context, with $\Box[t] = t$.

Substitutions are total unary functions that let us replace variables with terms. We can apply a substitution σ to a syntactic entity X (e.g., a term or literal) by writing $X\sigma$. A substitution γ is a *grounding substitution*, or simply a *grounding*, for a syntactic entity X if $X\gamma$ is ground (i.e., if it does not contain variables). A substitution ρ is a *renaming substitution*, or simply a *renaming*, if it is injective and $x\rho$ is a variable for every variable x. The *composition* of two substitutions σ_1 and σ_2, denoted by $\sigma_1 \circ \sigma_2$, is a substitution that first applies σ_1 and then applies σ_2 (i.e., $x(\sigma_1 \circ \sigma_2) = x\sigma_1\sigma_2$). A substitution σ is a *unifier* for a set of terms $\mathcal{T}$ if its application makes all elements of the set equal (i.e., if $\forall t_1, t_2 \in \mathcal{T}.\ t_1\sigma = t_2\sigma$). A substitution μ is an *idempotent most general unifier* (IMGU) for a set of terms $\mathcal{T}$ if μ is a unifier for $\mathcal{T}$ and $\mu \circ \sigma = \sigma$ holds for every unifier σ for $\mathcal{T}$.

2.2 Superposition

Bachmair and Ganzinger's superposition calculus [2, 3] belongs to a class of proof techniques for automatic provers known as saturation calculi. A saturation prover takes a set of formulas, usually clauses, as input and processes it by performing two operations: First, it derives new formulas from the old ones and adds them to the set. Second, it deletes superfluous formulas from the set. This process is repeated until the prover either derives $\bot$ or reaches a state in which it is not required to add further formulas.

Abstractly, a saturation calculus can be defined by specifying two components: a set of inferences that defines the derivations of new formulas, and a redundancy criterion that describes which inferences are unnecessary and which formulas may be deleted from the set.

For the superposition calculus, the inferences are given by three inference rules called *superposition*, *equality factoring*, and *equality resolution*. To reduce the number of inferences that need to be computed during saturation, the inference rules are equipped with order restrictions. Moreover, there is a *selection function* that can override the order restrictions. These local restrictions are supplemented by a global redundancy criterion for inferences and clauses.

Example 1. Let $\mathbb{N}, \mathbb{Z}$ be types. Consider the first-order signature consisting of the function symbols $\mathsf{a}, \mathsf{b} : \mathbb{N}$ and $\mathsf{f} : \mathbb{N} \to \mathbb{Z}$, the order $\succ$ on terms induced by the precedence $\mathsf{f} \succ \mathsf{b} \succ \mathsf{a}$ on the function symbols, and the unsatisfiable clause set $\{x \approx \mathsf{a},\ \mathsf{f}(\mathsf{b}) \not\approx \mathsf{f}(\mathsf{a})\}$. A superposition prover would perform the following steps:

1. A superposition inference is possible from the first clause's left-hand side x into the second clause's b subterm, by unifying x and b. The inference replaces b by a in the second clause, resulting in the new clause $\mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(\mathsf{a})$.
2. An equality resolution inference notices that both sides of the new clause are equal and derives a contradiction ($\bot$).

Note that the type system makes it impossible to unify x with, e.g., $\mathsf{f}(\mathsf{b})$, since they have different types.

In their *Handbook of Automated Reasoning* chapter [4], Bachmair and Ganzinger gave a general account of components and properties of saturation calculi. Waldmann et al. [29] extended this saturation framework in various ways.

A calculus is called *dynamically refutationally complete* if for every set N_0 such that $N_0 \models \bot$ and every fair derivation $N_0, N_1, \ldots$, the formula $\bot$ belongs to some N_i. Directly proving the dynamic refutational completeness of a calculus is considered difficult, but it can be shown to be equivalent to proving static refutational completeness: A calculus is *statically refutationally complete* if for every saturated set N we have that $N \models \bot$ implies $\bot \in N$.

To prove static (and thus dynamic) refutational completeness, it is usually convenient to start by considering only the ground level of the calculus. Ground refutational completeness can then be lifted to refutational completeness at

the nonground level. This lifting is based on two grounding functions that map nonground formulas and nonground inferences to sets of ground instances.

Tourret and Blanchette [9, 26, 27] formalized the saturation framework in Isabelle and extended it. Together with Waldmann and Tourret, we used this Isabelle framework in our formalization [12] of the superposition calculus. The refutational completeness proof proceeded in two steps:

1. Given any set N_G of ground clauses that is saturated w.r.t. ground inferences and that does not contain $\bot$, construct a model of N_G.
2. If the set N of nonground clauses is saturated w.r.t. nonground inferences, then the set

$$N_G = \{C\gamma \mid C \in N \text{ and } C\gamma \text{ is ground}\}$$

of ground instances is saturated w.r.t. ground inferences. Hence, by step 1, there exists a model of N_G, which is also a model of N.

Step 1 incrementally constructs a confluent and terminating term rewriting system to produce a model of N_G. Step 2 amounts to showing that there exist nonground inferences corresponding to all nonredundant ground inferences of the calculus, via a result from the saturation framework. Moreover, the saturation framework helps overcome difficulties in relating the ground and nonground levels by allowing us to simultaneously lift a family of ground calculi to the nonground level.

2.3 Isabelle/HOL

In Isabelle, type variables are written with an apostrophe (e.g., $'a$), and type constructors are written in postfix notation (e.g., $'a\ list$). We write $\langle x_1, \ldots, x_n \rangle :: \tau_1 \times \cdots \times \tau_n$ for the n-tuple with components $x_1 :: \tau_1, \ldots, x_n :: \tau_n$. The empty tuple $\langle\rangle$ is the only value of type *unit*.

A *locale* consists of type variables, parameters that may depend on them, and assumptions that may refer to the type variables and parameters. Locales are a useful structuring mechanism that provides modularity. They allow us to declare parameters and assumptions once and reuse them in multiple related definitions and lemmas. When we later *interpret* a locale, we must supply concrete arguments for the type arguments and parameters and then discharge the proof obligations corresponding to the assumptions.

3 The Basic Definitions

Terms, atoms, literals, and clauses share many definitions, lemmas, and proofs at both the ground and the nonground levels. To avoid duplication, we develop background theories of generic definitions and lemmas that we then instantiate for our concrete use cases (e.g., ground or nonground clauses).

3.1 Typing Relations

Adding types to the formalization of superposition requires specifying a type system. Our specification builds on a typing relation of type $'expr \Rightarrow 'ty \Rightarrow bool$ that uniquely assigns types to expressions, where $'expr$ is the universe of expressions and $'ty$ is the universe of types.

Definition 2 (Typing Relation)**.** The typing locale specifies a right-unique typing relation.

locale typing =
  fixes welltyped :: $'expr \Rightarrow 'ty \Rightarrow bool$
  assumes
    $\forall e\ \tau\ \tau'$. welltyped $e\ \tau \longrightarrow$ welltyped $e\ \tau' \longrightarrow \tau = \tau'$

Often, the type is irrelevant, and it is enough to know whether an expression is well typed.

Definition 3. Given the typing relation welltyped :: $'expr \Rightarrow 'ty \Rightarrow bool$, the predicate is_welltyped :: $'expr \Rightarrow bool$ expresses that there exists a type such that an expression is well typed:

$\forall e$. is_welltyped $e \longleftrightarrow (\exists \tau$. welltyped $e\ \tau)$

3.2 Lifting Typing Relations

We need to reason not only about the well-typedness of terms but also that of atoms, literals, and clauses. We introduce a lifting mechanism that allows us to extend well-typedness properties step by step: from terms to atoms, then to literals, and finally to clauses. This avoids redundant definitions and ensures consistency. Abstractly, we lift from a subexpression of type $'sub$ to a complex expression of type $'expr$ that contains subexpressions of the same type. A complex expression type is specified by a function to_set :: $'expr \Rightarrow 'sub\ set$ that returns the subexpressions contained within a complex expression.

Definition 4 (Lifting a Typing Relation)**.** The typing_lifting locale lifts an interpretation of the typing locale for subexpressions to complex expressions. The result is an interpretation of the typing locale for the complex expressions. The locale parameters are a typing relation sub_welltyped :: $'sub \Rightarrow 'ty \Rightarrow bool$ for the subexpressions and a function to_set :: $'expr \Rightarrow 'sub\ set$. Then the locale defines a typing relation welltyped :: $'expr \Rightarrow unit \Rightarrow bool$ expressing that a complex expression is well typed if all its subexpressions have the same type:

$\forall e$. welltyped $e\ \langle\rangle \longleftrightarrow$
  $(\exists \tau.\ \forall sub \in$ to_set e. sub_welltyped $sub\ \tau)$

Note that the second parameter of welltyped in Definition 4 has type *unit*, since we do not assign types to atoms, literals, and clauses, but rather only check that the subexpressions are well typed.

For our use case, we first interpret the typing_lifting locale with terms as subexpressions and atoms as complex

expressions. The locale parameters are a typing relation that assigns types to terms and a function to_set such that $\forall t_1\ t_2.$ to_set $(t_1 \approx t_2) = \{t_1, t_2\}$. An atom is thus well typed if its two terms are well typed and have the same type. We then interpret the typing_lifting locale again, this time with atoms as subexpressions and literals as complex expressions. The locale parameters are the lifted typing relation, which assigns the type $\langle\rangle$ to all well-typed atoms, and a function to_set such that $\forall l.$ to_set $l =$ {atm_of l}, where the function atm_of :: *'a literal* ⇒ *'a* returns the atom of a literal. A literal is thus well typed if its atom is well typed. Finally, we interpret the typing_lifting locale one last time, with literals as subexpressions and clauses as complex expressions. The locale parameters are the lifted typing relation for literals and a function to_set that converts a multiset to a set. A clause is thus well typed if all of its literals are well typed.

We want to perform this lifting from terms to atoms, to literals, to clauses both at the ground and the nonground level. To avoid proof text duplication, we define a generic lifting for this.

Definition 5 (Lifting a Typing Relation from Terms to Clauses)**.** The clause_typing locale lifts an interpretation of the typing locale for terms to atoms, then to literals, and then to clauses. The results are interpretations of the typing locale for atoms, literals, and clauses in the namespaces atom, literal, and clause. The liftings are performed with the typing_lifting locale.

Definitions and lemmas of interpretations can be accessed using the dot notation (e.g., clause.is_welltyped).

4 The Ground Level

After establishing the type system infrastructure, we will first use it on the ground level of the calculus. We will define well-typedness for ground terms, lift it to atoms, literals, and clauses, and finally prove that the ground inferences preserve well-typedness.

On the ground level, terms are represented by the type *'f gterm*, where *'f* is the type of function symbols. Atoms are represented by the type *'f gatom*. Literals are represented by the type *'f gatom literal*. Clauses are represented by the type *'f gatom clause*.

A ground term consists of a function symbol applied to a (possibly empty) argument list. We say that it is well typed if each argument is well typed and has the type expected by the function symbol. The expected types are specified by a function-type environment of type *'f* ⇒ *nat* ⇒ (*'ty list*×*'ty*) *option*, where *'ty* is the universe of types. Given a function symbol and an arity, a function-type environment returns the domain (i.e., a list of the expected type of each argument) and the codomain of the function. Using this, we can define a typing relation.

Definition 6 (Well-Typed Ground Term)**.** Let $\mathcal{F}$ be a function-type environment. The predicate welltyped $\mathcal{F}$ :: *'f gterm* ⇒ *'ty* ⇒ *bool* expresses that a term is well typed and has the given type:

$$\frac{\mathcal{F}\ \mathsf{f}\ (\mathsf{length}\ ts) = \langle \tau s, \tau \rangle \qquad \mathsf{list\_all2}\ (\mathsf{welltyped}\ \mathcal{F})\ ts\ \tau s}{\mathsf{welltyped}\ \mathcal{F}\ (\mathsf{GFun}\ \mathsf{f}\ ts)\ \tau}$$

Above, the predicate list_all2 :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a list* ⇒ *'b list* ⇒ *bool* expresses that two given lists have the same length and that their elements are positionwise related by a given binary predicate. In Isabelle, we write GFun f *ts* for the ground term $\mathsf{f}(t_1, \ldots, t_n)$, where f is a function symbol and $ts = t_1, \ldots, t_n$ is an argument list.

The relation welltyped is a family of typing relations indexed by function-type environments. We interpret the typing locale for the typing relation welltyped $\mathcal{F}$ for an arbitrary function-type environment $\mathcal{F}$. Based on that, we interpret the clause_typing locale for the family of typing relations. This gives us families of typing relations for atoms, literals, and clauses, all of them indexed by function-type environments. For example, we get clause.is_welltyped, which we can use to state that the inference rules of the ground superposition calculus preserve well-typedness. We perform this interpretation in the context of locale ground_superposition_calculus, which we reuse unchanged [12, Section 4].

Theorem 7 (Ground Superposition Preserves Well-typedness)**.** *Let C, D, and E be ground clauses, and let $\mathcal{F}$ be a function-type environment. If D and E are both well typed, and C can be inferred by superposition, then C is also well typed:*

$$\begin{aligned}&\forall C\ D\ E\ \mathcal{F}.\ \mathsf{superposition}\ D\ E\ C \longrightarrow\\ &\quad \mathsf{clause.is\_welltyped}\ \mathcal{F}\ D \longrightarrow\\ &\quad \mathsf{clause.is\_welltyped}\ \mathcal{F}\ E \longrightarrow\\ &\quad \mathsf{clause.is\_welltyped}\ \mathcal{F}\ C\end{aligned}$$

The theorems for equality factoring and equality resolution are analogous. The theorems for soundness and refutational completeness remain unchanged.

5 The Nonground Level

Next, we will extend the definition of well-typedness to nonground terms and lift it to nonground atoms, literals, and clauses. Moreover, we will establish type-preserving substitutions and define and prove some related properties. To lift these properties from terms to atoms, literals, and clauses, we will create a lifting infrastructure based on monomorphic natural functors.

On the nonground level, terms are represented by the type $('f, 'v)$ *term*. In addition to the function symbols *'f*, terms are parameterized by *'v*, the type of term variables. We write Fun f *ts* for the nonground term $\mathsf{f}(t_1, \ldots, t_n)$, where f is a function symbol and $ts = t_1, \ldots, t_n$ is an argument list. We

write Var x for the nonground term x, where x is a variable. Atoms are represented by the type $('f, 'v)$ *atom* (synonymous with $('f, 'v)$ *term uprod*), literals are represented by the type $('f, 'v)$ *atom literal*, and clauses are represented by the type $('f, 'v)$ *atom clause*.

The typing relation for a nonground term is similar to Definition 6. However, it must also consider term variables. Accordingly, we extend the predicate by a variable-type environment $\mathcal{V} :: 'v \Rightarrow 'ty$ as parameter and a rule for term variables.

Definition 8 (Well-Typed Term)**.** Let $\mathcal{F}$ be a function-type environment and $\mathcal{V}$ be a variable-type environment. The predicate welltyped $\mathcal{F}\ \mathcal{V} :: ('f, 'v)\ term \Rightarrow 'ty \Rightarrow bool$ expresses that a term is well typed and has the given type:

$$\frac{\mathcal{F}\ \mathsf{f}\ (\mathsf{length}\ ts) = \langle \tau s, \tau \rangle \qquad \mathsf{list\_all2}\ (\mathsf{welltyped}\ \mathcal{F}\ \mathcal{V})\ ts\ \tau s}{\mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}\ (\mathsf{Fun}\ \mathsf{f}\ ts)\ \tau}$$

$$\frac{\mathcal{V}\ x = \tau}{\mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}\ (\mathsf{Var}\ x)\ \tau}$$

The welltyped predicate is a family of typing relations indexed by both function-type and variable-type environments.

We interpret the typing locale for the welltyped predicate and lift the predicate from terms to atoms, literals, and clauses by interpreting the clause_typing locale.

5.1 Functional Substitutions

The original formalization of untyped superposition [12] relies on a background theory [14] of abstract substitutions modeled as monoid actions. Abstract substitutions are specified by the substitution locale, which has the parameters subst :: $'expr \Rightarrow 's \Rightarrow 'expr$ and is_ground :: $'expr \Rightarrow bool$, among others. The subst function applies a substitution of type $'s$ to an expression of type $'expr$. Following Section 2.1, we write $e\sigma$ instead of subst $e\ \sigma$. The is_ground predicate is true for expressions that cannot be affected by any substitution. For example, is_ground is true for a term t if $t\sigma = t$ for all substitutions σ, which is equivalent to saying that t does not contain any variables.

We extend this background theory to substitutions that are functions mapping variables to base expressions of type $'base$. We call such a substitution of type $'v \Rightarrow 'base$ a *functional substitution*. Accordingly, the type of subst is specialized to $'expr \Rightarrow ('v \Rightarrow 'base) \Rightarrow 'expr$. In our use case, the base expressions are terms of type $('f, 'v)$ *term*. Substitution application for clauses has then for example the type subst :: $('f, 'v)$ *atom clause* $\Rightarrow ('v \Rightarrow ('f, 'v)$ *term*$) \Rightarrow ('f, 'v)$ *atom clause*.

Definition 9 (Functional Substitution)**.** Let vars :: $'expr \Rightarrow 'v$ *set* be a function that maps a given expression to its variables. The functional_substitution locale extends the substitution locale by fixing the type of the substitutions to $'v \Rightarrow 'base$ and by defining that an expression is ground if it does not contain variables (i.e., the vars function returns the empty set). It also specifies that the effect of a substitution on an expression is explained by the effects of the substitution on the expression's variables.

```
locale functional_substitution =
  substitution where
    subst = subst and is_ground = (λe. vars e = {})
  for
    subst :: 'expr ⇒ ('v ⇒ 'base) ⇒ 'expr and
    vars :: 'expr ⇒ 'v set +
  assumes
    ∀e σ σ'. (∀x ∈ vars e. σ x = σ' x) ⟶ eσ = eσ'
```

The specialization of abstract substitutions to functional substitutions enables many new definitions and lemmas on a generic level. It is one of the main parts of our refactoring aimed at avoiding proof text duplication. We can reuse the definitions and lemmas for terms, atoms, literals, clauses, and inferences.

While specifying a type system for the superposition calculus, we must bring substitutions and types together.

Definition 10 (Typed Functional Substitution)**.** The typed_functional_substitution locale combines the locales typing and functional_substitution, and extends the welltyped predicate by a variable-type environment as parameter.

In particular, we must specify substitutions that preserve the well-typedness of the expressions they are applied to. A functional substitution preserves an expression's type if each variable is replaced by a term of the same type.

Definition 11 (Type-Preserving Substitution)**.** Let welltyped $\mathcal{F}\ \mathcal{V} :: 'expr \Rightarrow 'ty \Rightarrow bool$ be a typing relation. The predicate type_preserving_on $\mathcal{F}\ \mathcal{V} :: 'v\ set \Rightarrow ('v \Rightarrow 'base) \Rightarrow bool$ expresses that a substitution preserves well-typedness for a given set of variables:

```
∀X σ. type_preserving_on F V X σ ⟷
  (∀x ∈ X. welltyped F V x (V x) ⟶
    welltyped F V (xσ) (V x))
```

Based on Definitions 10 and 11, we can define properties of our type system regarding substitutions on generic expressions and derive lemmas based on these properties. We present here three definitions and two lemmas.

Definition 12. A type system is *preserved by substitutions* if any substitution that preserves the types of the variables in an expression also preserves the type of the entire expression:

```
∀F V e σ τ. type_preserving_on F V (vars e) σ ⟶
  (welltyped F V (eσ) τ ⟷ welltyped F V e τ)
```

Definition 13. A type system is *preserved by renaming* if every expression has the same type before and after renaming w.r.t. two compatible variable-type environments:

$$\forall \mathcal{F}\ \mathcal{V}\ \mathcal{V}'\ e.\ (\forall x \in \mathsf{vars}\ e.\ \mathcal{V}'\ (x\rho) = \mathcal{V}\ x) \longrightarrow (\mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}'\ (e\rho)\ \tau \longleftrightarrow \mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}\ e\ \tau)$$

Definition 14. *All types have a witness* if, for every type, there exists a ground expression of that type:

$$\forall \mathcal{F}\ \mathcal{V}\ \tau.\ \exists e.\ \mathsf{is\_ground}\ e \wedge \mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}\ e\ \tau$$

Lemma 15 (Grounding Substitution Extension)**.** *Let e be an expression, $\mathcal{V}$ be a variable-type environment, and γ be a grounding substitution for e. If all types have a witness and γ is type-preserving w.r.t. $\mathcal{V}$ on the variables of e, then there exists a substitution γ' such that*

- *γ' is equal to γ on the variables of e,*
- *γ' is a grounding substitution for any expression, and*
- *γ' is type-preserving on the set of all variables w.r.t. $\mathcal{V}$.*

Lemma 16 (Type-Preserving IMGU)**.** *Let e and e' be expressions, $\mathcal{V}$ be a variable-type environment, and μ be an IMGU for e and e'. If μ is type-preserving w.r.t. $\mathcal{V}$ on the variables of e and e', then*

$$\forall \tau.\ \mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}\ e\ \tau \longleftrightarrow \mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}\ e'\ \tau$$

Using our infrastructure, we can now specify a type system for nonground terms. We want to globally assign types to functions, ensuring that the same function has a consistent type across different terms. To this end, we define a locale that fixes a function-type environment.

Definition 17 (Term Type System)**.** Let $\mathcal{F} :: 'f \Rightarrow nat \Rightarrow ('ty\ list \times 'ty)\ option$ be a function-type environment. The nonground_term_typing locale specifies that there exists a ground term for each type w.r.t. $\mathcal{F}$:

locale nonground_term_typing =
  fixes $\mathcal{F} :: 'f \Rightarrow nat \Rightarrow ('ty\ list \times 'ty)\ option$
  assumes $\forall \mathcal{V}\ \tau.\ \exists t.\ \mathsf{is\_ground}\ t \wedge \mathsf{welltyped}\ \mathcal{F}\ \mathcal{V}\ t\ \tau$

In the context of nonground_term_typing, we extend the interpretation term by interpreting the typed_functional_substitution locale for the typing relation welltyped $\mathcal{F}$. By fixing the function-type environment in welltyped $\mathcal{F}$, we ensure a consistent function-type environment in the entire specification.

Lemma 18. *The type system specified by the* nonground_term_typing *locale is preserved by substitutions and by renamings, and all its types have a witness.*

5.2 Infinite Variables per Type

When variables are renamed by renaming substitutions, we must ensure that fresh variables are available. In the context of a type system, we must also ensure that variables can be renamed without altering their types.

Definition 19. The predicate infinite_variables_per_type $:: ('v \Rightarrow 'ty) \Rightarrow bool$ expresses that in the image of a given variable-type environment, there exist infinitely many variables for every type:

$$\forall \mathcal{V}.\ \mathsf{infinite\_variables\_per\_type}\ \mathcal{V} \longleftrightarrow (\forall \tau \in \mathsf{range}\ \mathcal{V}.\ \mathsf{infinite}\ \{x.\ \mathcal{V}\ x = \tau\})$$

Above, the function range $:: ('a \Rightarrow 'b) \Rightarrow 'b\ set$ returns the image of a function.

Example 20. Suppose $'v$ is fixed to *string* and $'ty$ is fixed to $\{\mathbb{N}, \mathbb{Z}\}$. Consider the variable-type environment $\mathcal{V}$ defined such that $\forall x.\ \mathcal{V}\ x =$ (if $x =$ "n" then $\mathbb{N}$ else $\mathbb{Z}$). Then $\mathcal{V}$ does not satisfy the infinite_variables_per_type predicate, because we cannot rename the variable "n" to another variable of the same type. Now consider another variable-type environment $\mathcal{V}'$ defined such that $\forall x.\ \mathcal{V}'\ x =$ (if x starts with "n" then $\mathbb{N}$ else $\mathbb{Z}$). Then $\mathcal{V}'$ does satisfy the infinite_variables_per_type predicate, because there are infinitely many strings starting with "n", which means that we can always find a fresh variable of type $\mathbb{N}$ in $\mathcal{V}'$.

When we have multiple expressions with different variable-type environments, such as the premises of the nonground superposition inference rule, we will need to create a new variable-type environment that can handle all their types.

Lemma 21 (Merging of Variable-Type Environments)**.** *Let X and Y be disjoint finite sets of variables, and $\mathcal{V}_1$ and $\mathcal{V}_2$ be variable-type environments. If the universe of variables is infinite, then there exists a variable-type environment $\mathcal{V}_3$ such that* infinite_variables_per_type *$\mathcal{V}_3$ holds, $\mathcal{V}_3$ is equal to $\mathcal{V}_1$ on X, and $\mathcal{V}_3$ is equal to $\mathcal{V}_2$ on Y.*

Proof sketch. We define $\mathcal{V}_3$ as $\mathcal{V}_1$ on X and $\mathcal{V}_2$ on Y, which is possible since X and Y are disjoint. Let $C = \{\mathcal{V}_1\ x \mid x \in X\} \cup \{\mathcal{V}_2\ y \mid y \in Y\}$ be a finite set containing the types that $\mathcal{V}_1$ assigns to the variables in X and $\mathcal{V}_2$ assigns to the variables in Y. We construct $\mathcal{V}_3$ such that C is its image: From the assumption that the set of all variables is infinite, we can partition the set of variables excluding X and Y into finitely many infinite sets. For each type τ in C, we create such an infinite set that is mapped to τ by $\mathcal{V}_3$, guaranteeing infinitely many variables for each type in the image of $\mathcal{V}_3$. □

We can also use the infinite_variables_per_type predicate to reason about the existence of renaming and grounding substitutions.

Lemma 22. *Let e_1 and e_2 be expressions, $\mathcal{V}_1$ and $\mathcal{V}_2$ be variable-type environments, and γ_1 and γ_2 be type-preserving grounding substitutions for e_1 and e_2 w.r.t. $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively. If the* infinite_variables_per_type *predicate holds for $\mathcal{V}_1$ and the set of variables of e_2 is finite, then we can obtain renaming substitutions ρ_1 and ρ_2, and a grounding substitution γ such that*

- *ρ_1 and ρ_2 are type-preserving for e_1 and e_2 w.r.t. $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively,*
- *$e_1\rho_1$ and $e_2\rho_2$ have disjoint variables,*

- γ *is equal to* γ_1 *on the variables of* $e_1\rho_1$, *and*
- γ *is equal to* γ_2 *on the variables of* $e_2\rho_2$.

5.3 Functional Substitution Liftings

Interpreting the functional_substitution locale separately for terms, atoms, literals, and clauses already helps reduce duplicated proof text, since lemmas and definitions within functional_substitution can be shared. However, parameters such as the functions vars and subst must still be manually defined for each level, and their properties proved separately. These definitions and proofs are very similar. We can eliminate this redundancy by introducing a lifting mechanism similar to the approach described in Section 3.2. We lift a subexpression of type *'sub* to a complex expression of type *'expr*. As before, a complex expression is specified by a function to_set :: *'expr* ⇒ *'sub set* that returns the contained subexpressions. However, we need additional structure.

Definition 23. A monomorphic *natural functor* is equipped with the functions map :: $('a \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b$ and to_set :: $'b \Rightarrow 'a$ *set* that obey the following five laws:

1. $\forall b\ f\ g.\ \mathsf{map}\ f\ (\mathsf{map}\ g\ b) = \mathsf{map}\ (\lambda x.\ f\ (g\ x))\ b$;
2. $\forall b.\ \mathsf{map}\ (\lambda x.\ x)\ b = b$;
3. $\forall b\ f\ g.\ (\forall a \in \mathsf{to\_set}\ b.\ f\ a = g\ a) \longrightarrow \mathsf{map}\ f\ b = \mathsf{map}\ g\ b$;
4. $\forall b\ f.\ \mathsf{to\_set}\ (\mathsf{map}\ f\ b) = \mathsf{image}\ f\ (\mathsf{to\_set}\ b)$;
5. $\exists b.\ \mathsf{to\_set}\ b \neq \{\}$.

Above, the function image :: $('a \Rightarrow 'b) \Rightarrow 'a$ *set* ⇒ $'b$ *set* applies a given function to all elements of a given set.

This notion of a natural functor is inspired by Traytel et al. [28]. We specify a natural functor with the natural_functor locale. It has a map function that obeys the typical composition and identity functor laws (1 and 2). Mathematically, we describe a functor as a pair of the type of the functor and an action on functions between types, called map. The functor $\langle 'b, \mathsf{map}\rangle$ is monomorphic, meaning that it does not allow transformations between types, since Isabelle/HOL does not support higher-kinded types. The laws 3, 4, and 5 describe the relationship of the functions to_set and map such that to_set is a natural transformation between the functor $\langle 'b, \mathsf{map}\rangle$ and the monomorphic functor $\langle 'a\ set, \mathsf{image}\rangle$. We think of a natural functor $\langle 'b, \mathsf{map}, \mathsf{to\_set}\rangle$ as a complex structure containing elements of type *'a*.

Lemma 24. *The types 'a uprod, 'a literal, and 'a multiset together with their canonical* map *and* to_set *functions are natural functors.*

Based on natural functors, we can define a lifting for functional substitutions.

Definition 25. Let sub be the interpretation of functional_substitution with its parameters sub_vars :: *'sub* ⇒ *'v set* and sub_subst :: *'sub* ⇒ (*'v* ⇒ *'base*) ⇒ *'sub*, and the natural functor $\langle$*'expr*, map :: (*'sub* ⇒ *'sub*) ⇒ *'expr* ⇒ *'expr*, to_set :: *'expr* ⇒ *'sub set*$\rangle$. The functional_substitution_lifting locale then defines the functions vars :: *'expr* ⇒ *'v set* and subst :: *'expr* ⇒ (*'v* ⇒ *'base*) ⇒ *'expr*:

- $\forall e.\ \mathsf{vars}\ e = (\bigcup_{x \in \mathsf{to\_set}\ e} \mathsf{sub\_vars}\ x)$
- $\forall e\ \sigma.\ \mathsf{subst}\ e\ \sigma = \mathsf{map}\ (\lambda s.\ \mathsf{sub\_subst}\ s\ \sigma)\ e$

For the functional_substitution_lifting locale to actually be a lifting for functional substitutions, we need to prove that the lifted functions can interpret functional_substitution.

Lemma 26. *Let* vars :: *'expr* ⇒ *'v set and* subst :: *'expr* ⇒ (*'v* ⇒ *'base*) ⇒ *'expr be the functions defined by the* functional_substitution_lifting *locale. Then* vars *and* subst *can interpret the* functional_substitution *locale.*

We can also lift many properties of functional substitutions using this lifting. In particular, we can lift properties of typed functional substitutions.

Definition 27. The typed_functional_substitution_lifting locale combines the locales functional_substitution_lifting and typing_lifting.

Since the typed_functional_substitution locale combines the locales functional_substitution and typing, we can lift it using typed_functional_substitution_lifting. The lifting preserves many useful properties; we illustrate the preservation of two properties.

Lemma 28. *Let* sub *be an interpretation of the* typed_functional_substitution *locale.*

- *If the type system specified by* sub *is preserved by substitutions, then the type system lifted from* sub *by the* typed_functional_substitution_lifting *locale is also preserved by substitutions.*
- *If the type system specified by* sub *is preserved by renamings, then the type system lifted from* sub *by the* typed_functional_substitution_lifting *locale is also preserved by renamings.*

Finally, we extend the type system specification on nonground terms to nonground atoms, literals, and clauses using our lifting infrastructure.

Definition 29 (Nonground Typing)**.** The nonground_typing locale extends nonground_term_typing. Using the typed_functional_substitution_lifting locale, atom is an interpretation lifted from term, literal an interpretation lifted from atom, and clause an interpretation lifted from literal.

The proof obligations produced by the liftings in Definition 29 were discharged in Lemma 24.

5.4 The Nonground Calculus

Now, we will add a type system to the nonground superposition calculus and incrementally refine the inference rules so that they preserve well-typedness while maintaining the calculus's soundness and completeness.

The original untyped formalization defined the nonground calculus in the first_order_superposition_calculus locale [12, Section 5]. Based on this, we define our own locale superposition_calculus, which adds a function-type environment as a locale parameter and the locale assumption that every type is witnessed by a nullary function symbol. This parameter and this assumption are sufficient to extend the calculus to simple monomorphic types.

We use this opportunity to refactor the background theories regarding orders and entailment using a lifting infrastructure to avoid duplicated proof text. Moreover, we eliminate an assumption present in the original formalization—namely, the ground critical pairs theorem. This theorem was proved in the IsaFoR library [23]. For licensing reasons, we could not build on it, so we simply assumed that it holds. Later, the IsaFoR developers proved the nonground critical pairs theorem and released it under a more liberal license [24]. This allows us to adapt the nonground theorem to the ground case and remove the locale assumption.

Definition 30 (Nonground Superposition Calculus)**.** The superposition_calculus locale specifies the parameters and assumptions required by the superposition calculus and the proofs of its soundness and refutational completeness. In particular, it specifies a type system using the nonground_typing locale.

locale superposition_calculus =
  nonground_equality_order $\prec_t$ +
  nonground_selection_function select +
  nonground_typing $\mathcal{F}$
  for
    $\prec_t :: ('f, 'v)\ term \Rightarrow ('f, 'v)\ term \Rightarrow bool$ **and**
    select :: $('f, 'v)\ atom\ clause \Rightarrow$
      $('f, 'v)\ atom\ clause$ **and**
    $\mathcal{F} :: 'f \Rightarrow nat \Rightarrow ('ty\ list \times 'ty)\ option$ **and**
    tiebreakers :: $'f\ gatom\ clause \Rightarrow$
      $('f, 'v)\ atom\ clause \Rightarrow ('f, 'v)\ atom\ clause \Rightarrow$
      $bool$
  assumes
    infinite (UNIV :: $'v\ set$) **and**
    $\forall C_G$. wfp (tiebreakers C_G) $\wedge$ transp (tiebreakers C_G)

To define the calculus itself, we start from the untyped inference rules. However, these do not preserve well-typedness in their current form. Therefore, our objective is to revise the rules so they preserve well-typedness without losing soundness and refutational completeness. We start by recalling the inference rules of the untyped superposition calculus.

$$\frac{\overbrace{t_1 \approx t_1' \vee t_2 \approx t_2' \vee D'}^{D}}{\underbrace{(t_1' \not\approx t_2' \vee t_1 \approx t_2' \vee D')\mu}_{C}}\ \text{eq\_factoring } D\ C$$

Side conditions:

F1 select $D = \{\}$;
F2 μ is an IMGU of $\{t_1, t_2\}$;
F3 $(t_1 \approx t_1')\mu$ is maximal in $D\mu$;
F4 $t_1\mu \not\preceq_t t_1'\mu$.

$$\frac{\overbrace{t \not\approx t' \vee D'}^{D}}{\underbrace{D'\mu}_{C}}\ \text{eq\_resolution } D\ C$$

Side conditions:

R1 μ is an IMGU of $\{t, t'\}$;
R2 if select $D = \{\}$, then $t\mu \not\approx t'\mu$ is maximal in $D\mu$;
R3 if select $D \neq \{\}$, then $t\mu \not\approx t'\mu$ is maximal in (select $D)\mu$.

$$\frac{\overbrace{t_2 \approx t_2' \vee D'}^{D} \quad \overbrace{c[t_1] \bowtie t_1' \vee E'}^{E}}{\underbrace{((c\rho_1)[t_2'\rho_2] \bowtie t_1'\rho_1 \vee E'\rho_1 \vee D'\rho_2)\mu}_{C}}\ \text{superposition } D\ E\ C$$

Side conditions:

S1 $\bowtie \in \{\approx, \not\approx\}$;
S2 ρ_1 and ρ_2 are renamings;
S3 $E\rho_1$ and $D\rho_2$ are variable-disjoint;
S4 t_1 is not a variable;
S5 μ is an IMGU of $\{t_1\rho_1, t_2\rho_2\}$;
S6 $E\rho_1\mu \not\preceq_c D\rho_2\mu$;
S7 $t_2\rho_2\mu \not\preceq_t t_2'\rho_2\mu$;
S8 $(c[t_1])\rho_1\mu \not\preceq_t t_1'\rho_1\mu$;
S9 if $\bowtie = \approx$, then select $E = \{\}$ and $(c[t_1] \bowtie t_1')\rho_1\mu$ is strictly maximal in $E\rho_1\mu$;
S10 if $\bowtie = \not\approx$ and select $E = \{\}$, then $(c[t_1] \bowtie t_1')\rho_1\mu$ is maximal in $E\rho_1\mu$;
S11 if $\bowtie = \not\approx$ and select $E \neq \{\}$, then $(c[t_1] \bowtie t_1')\rho_1\mu$ is maximal in (select $E)\rho_1\mu$;
S12 select $D = \{\}$;
S13 $(t_2 \approx t_2')\rho_2\mu$ is strictly maximal in $D\rho_2\mu$.

5.5 Typed Clauses

Next to the global function-type environment $\mathcal{F}$, we introduce local variable-type environments, one per clause. This allows us to reuse variable names across different clauses, even with different types. For example, a clause C could contain a variable x of type $\mathbb{N}$, while another clause D independently could also contain a variable x of type $\mathbb{Z}$. Using a global variable-type environment would require a global invariant that enforces distinct variable names across all clauses. This would restrict the implementation of a saturation prover based on the calculus, and it would conflict with the superposition rule's local renaming strategy.

We define a typed clause as a pair consisting of a clause and an associated variable-type environment, represented as $('v \Rightarrow 'ty) \times ('f, 'v)$ *atom clause*, and abbreviate this type with $('f, 'v, 'ty)$ *typed_clause*. We update the inference rules accordingly to operate on typed clauses and keep the same side conditions.

$$\frac{\overbrace{t_1 \approx t_1' \vee t_2 \approx t_2' \vee D'}^{D}}{\underbrace{(t_1' \not\approx t_2' \vee t_1 \approx t_2' \vee D')\mu}_{C}}\ \text{eq\_factoring}\ \langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$$

$$\frac{\overbrace{t \not\approx t' \vee D'}^{D}}{\underbrace{D'\mu}_{C}}\ \text{eq\_resolution}\ \langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$$

$$\frac{\overbrace{t_2 \approx t_2' \vee D'}^{D} \quad \overbrace{c[t_1] \bowtie t_1' \vee E'}^{E}}{\underbrace{((c\rho_1)[t_2'\rho_2] \bowtie t_1'\rho_1 \vee E'\rho_1 \vee D'\rho_2)\mu}_{C}}\ \begin{array}{l}\text{superposition}\\ \langle D, \mathcal{V}_2\rangle\ \langle E, \mathcal{V}_1\rangle\\ \langle C, \mathcal{V}_3\rangle\end{array}$$

An easy way to ensure that the inference rules preserve well-typedness would be to add side conditions stating that the premises and conclusions need to be well-typed using the clause.is_welltyped predicate. But this approach would not reflect actual implementations in saturation provers, which improve runtime by performing type checking only as needed. To stay closer to the implementations, we instead add more fine-grained side conditions.

5.6 Type-Preserving IMGUs

All three inference rules rely on IMGUs to unify terms of their premises. To ensure that applying an IMGU does not alter the type of expressions, we introduce the following side conditions to the eq_factoring and eq_resolution rules, respectively:

F5 type_preserving_on $\mathcal{F}$ $\mathcal{V}$ (clause.vars D) μ;

R4 type_preserving_on $\mathcal{F}$ $\mathcal{V}$ (clause.vars D) μ.

The type_preserving_on predicate, as introduced in Definition 11, ensures that each IMGU only replaces variables with terms of the same type.

The superposition rule requires a similar new side condition, but we need to account for the presence of two premises:

S14 type_preserving_on $\mathcal{F}$ $\mathcal{V}_3$
(clause.vars $(E\rho_1)$ ∪ clause.vars $(D\rho_2)$) μ.

This condition guarantees that the IMGU preserves the types of the variables in the premises after they are renamed apart. Additionally, we use the variable-type environment $\mathcal{V}_3$, representing the type environment for expressions after renaming.

With this additional side condition, we can formally establish that the eq_factoring and eq_resolution rules preserve well-typedness.

Theorem 31. *Let C and D be clauses and $\mathcal{V}$ be a variable-type environment. If there exists an inference* eq_resolution $\langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$ *or* eq_factoring $\langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$*, then*

clause.is_welltyped $\mathcal{F}$ $\mathcal{V}$ D ⟶
clause.is_welltyped $\mathcal{F}$ $\mathcal{V}$ C

Proof sketch. Proof automation finds a proof by exploiting lifted simplification lemmas about the type system. In particular, it uses the preservation of the type system by substitutions, as established in Lemmas 18 and 28, and the property of IMGUs established in Lemma 16. □

5.7 Weakly Well-Typed Literals

In Theorem 31, we establish only that the inference rules produce well-typed conclusions when their premises are well typed. However, when performing rule elimination in proofs, for example in the soundness proof, we also need information about the well-typedness of the premises. Since we do not want to impose well-typedness of the premises as an explicit side condition on every inference rule, we instead introduce more selective side conditions that enforce well-typedness only where it is required.

Our initial approach was to strengthen the rules so that Theorem 31 would hold bidirectionally. Inspired by Lemma 16, we later identified a weaker property that is sufficient for our purposes and that requires only a single additional side condition. We refer to this property as *weak well-typedness*.

Definition 32. The predicate weakly_welltyped $\mathcal{F}$ :: $('v \Rightarrow 'ty) \Rightarrow ('f, 'v)$ *atom clause* ⇒ *bool* expresses that every literal in a clause consists of two terms with the same type or both are not well typed:

$\forall \mathcal{V}\ C$. weakly_welltyped $\mathcal{F}$ $\mathcal{V}$ C ⟷
($\forall t \bowtie t' \in C$. $\forall \tau$. welltyped $\mathcal{F}$ $\mathcal{V}$ t ⟷
welltyped $\mathcal{F}$ $\mathcal{V}$ t')

where $\bowtie \in \{\approx, \not\approx\}$.

We can prove that weakly well-typed conclusions can only originate from weakly well-typed premises.

Lemma 33. *Let D and C be clauses and $\mathcal{V}$ be a variable-type environment. If there exists an inference* eq_resolution $\langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$ *or* eq_factoring $\langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$*, then*

weakly_welltyped $\mathcal{F}$ $\mathcal{V}$ D ⟷ weakly_welltyped $\mathcal{F}$ $\mathcal{V}$ C

Proof sketch. Similar to the proof of Theorem 31. The implication from right to left relies on Lemma 16. □

5.8 Type-Preserving Renamings

The superposition rule relies both on IMGUs and on renaming substitutions. Specifically, there is one renaming substitution for each of the inference's two premises. Having two renamings instead of just one gives more flexibility in refinements, since we could use a heuristic to decide which of the two clauses should be renamed. To ensure that these substitutions preserve well-typedness, we add two side conditions:

S15 type_preserving_on $\mathcal{F}$ $\mathcal{V}_1$ (clause.vars E) ρ_1;
S16 type_preserving_on $\mathcal{F}$ $\mathcal{V}_2$ (clause.vars D) ρ_2.

These conditions guarantee that each renaming substitution ρ_i consistently replaces variables with other variables of the same type according to the type environment $\mathcal{V}_i$ associated with its corresponding premise.

Until now, the type environments $\mathcal{V}_1$ and $\mathcal{V}_2$ of the premises have been independent of type environment $\mathcal{V}_3$ of the conclusion. To ensure consistency between them, we add two more side conditions to the superposition rule:

S17 $\forall x \in$ clause.vars E. $\mathcal{V}_1\ x = \mathcal{V}_3\ (x\rho_1)$;
S18 $\forall x \in$ clause.vars D. $\mathcal{V}_2\ x = \mathcal{V}_3\ (x\rho_2)$.

These conditions guarantee that, after applying the renaming ρ_1, the variables of E have the same types in $\mathcal{V}_3$ as they had in $\mathcal{V}_1$ without renaming. Similarly, after applying ρ_2, the variables of D retain the same types in $\mathcal{V}_3$ as they had in $\mathcal{V}_2$ without renaming.

With these additional side conditions in place, the superposition rule derives well-typed conclusions from well-typed premises.

Theorem 34. *Let C, D, and E be clauses and $\mathcal{V}_1, \mathcal{V}_2$, and $\mathcal{V}_3$ be variable-type environments. If there exists an inference* superposition $\langle \mathcal{V}_2, D\rangle\ \langle \mathcal{V}_1, E\rangle\ \langle \mathcal{V}_3, C\rangle$*, then*

(clause.is_welltyped $\mathcal{F}$ $\mathcal{V}_2$ D ∧
clause.is_welltyped $\mathcal{F}$ $\mathcal{V}_1$ E) ⟶
clause.is_welltyped $\mathcal{F}$ $\mathcal{V}_3$ C

Proof sketch. We exploit, among other properties, the preservation of the type system specification by substitutions and renamings as shown in Lemmas 18 and 28 and the property of IMGUs established in Lemma 16. □

However, as with the other two inference rules (Lemma 33), we also want to guarantee that weakly well-typed conclusions only originate from weakly well-typed premises. To achieve this, we introduce a side condition expressing that the terms in the literal $t_2 \approx t_2'$ have the same type:

S19 $\forall \tau$. welltyped $\mathcal{F}$ $\mathcal{V}_2$ t_2 τ ⟷ welltyped $\mathcal{F}$ $\mathcal{V}_2$ t_2' τ.

Now, we can formally establish well-typedness preservation of the superposition rule, ensuring that only weakly well-typed clauses are involved in the derivation process.

Lemma 35. *Let C, D, and E be clauses and $\mathcal{V}_1, \mathcal{V}_2$, and $\mathcal{V}_3$ be variable-type environments. If there exists an inference* superposition $\langle \mathcal{V}_2, D\rangle\ \langle \mathcal{V}_1, E\rangle\ \langle \mathcal{V}_3, C\rangle$*, then*

(weakly_welltyped $\mathcal{F}$ $\mathcal{V}_2$ D ∧
weakly_welltyped $\mathcal{F}$ $\mathcal{V}_1$ E) ⟷
weakly_welltyped $\mathcal{F}$ $\mathcal{V}_3$ C

Proof sketch. Similar to the proof of Theorem 34. □

5.9 Regaining Completeness

The calculus now preserves well-typedness, but it is no longer complete because we cannot always guarantee the availability of fresh variables for every type. In the superposition rule, renaming variables apart is necessary to ensure that premises have distinct variable sets. In the untyped calculus, this was achieved by assuming that the set of variables is infinite. We retain this global assumption but must also guarantee that fresh variables are available for all types. The infinite_variables_per_type predicate (Definition 19) expresses this requirement.

When initially defining infinite_variables_per_type, we did not realize that it is sufficient to require infinitely many variables for each type in the image of the variable-type environment, rather than for every type. The original definition led to an additional assumption for the calculus, which stated that the cardinality of the set of all types must be bounded by the cardinality of the set of all variables. With our current definition, this assumption is no longer necessary.

Finally, we make the calculus complete again by adding two side conditions to the superposition rule, ensuring that we can create fresh variables for any type that appears in the premises:

S20 infinite_variables_per_type $\mathcal{V}_1$;
S21 infinite_variables_per_type $\mathcal{V}_2$.

We can now present the final inferences rules with all of their side conditions:

$$\frac{\overbrace{t_1 \approx t_1' \vee t_2 \approx t_2' \vee D'}^{D}}{\underbrace{(t_1' \not\approx t_2' \vee t_1 \approx t_2' \vee D')\mu}_{C}}\ \text{eq\_factoring}\ \langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$$

Side conditions:

F1 select $D = \{\}$;
F2 μ is an IMGU of $\{t_1, t_2\}$;
F3 $(t_1 \approx t_1')\mu$ is maximal in $D\mu$;
F4 $t_1\mu \not\preceq_t t_1'\mu$;
F5 type_preserving_on $\mathcal{F}$ $\mathcal{V}$ (clause.vars D) μ.

$$\frac{\overbrace{t \not\approx t' \vee D'}^{D}}{\underbrace{D'\mu}_{C}}\ \text{eq\_resolution}\ \langle \mathcal{V}, D\rangle\ \langle \mathcal{V}, C\rangle$$

Side conditions:

R1 μ is an IMGU of $\{t, t'\}$;
R2 if select $D = \{\}$, then $t\mu \not\approx t'\mu$ is maximal in $D\mu$;
R3 if select $D \neq \{\}$, then $t\mu \not\approx t'\mu$ is maximal in (select D)μ;
R4 type_preserving_on $\mathcal{F}$ $\mathcal{V}$ (clause.vars D) μ.

$$\frac{\overbrace{t_2 \approx t_2' \vee D'}^{D} \quad \overbrace{c[t_1] \bowtie t_1' \vee E'}^{E}}{\underbrace{((c\rho_1)[t_2'\rho_2] \bowtie t_1'\rho_1 \vee E'\rho_1 \vee D'\rho_2)\mu}_{C}}\ \begin{array}{l}\text{superposition}\\ \langle D, \mathcal{V}_2\rangle\ \langle E, \mathcal{V}_1\rangle\\ \langle C, \mathcal{V}_3\rangle\end{array}$$

Side conditions:

S1 $\bowtie \in \{\approx, \not\approx\}$;
S2 ρ_1 and ρ_2 are renamings;
S3 $E\rho_1$ and $D\rho_2$ are variable-disjoint;
S4 t_1 is not a variable;
S5 μ is an IMGU of $\{t_1\rho_1, t_2\rho_2\}$;
S6 $E\rho_1\mu \not\preceq_c D\rho_2\mu$;
S7 $t_2\rho_2\mu \not\preceq_t t_2'\rho_2\mu$;
S8 $(c[t_1])\rho_1\mu \not\preceq_t t_1'\rho_1\mu$;
S9 if $\bowtie = \approx$, then select $E = \{\}$ and $(c[t_1] \bowtie t_1')\rho_1\mu$ is strictly maximal in $E\rho_1\mu$;
S10 if $\bowtie = \not\approx$ and select $E = \{\}$, then $(c[t_1] \bowtie t_1')\rho_1\mu$ is maximal in $E\rho_1\mu$;
S11 if $\bowtie = \not\approx$ and select $E \neq \{\}$, then $(c[t_1] \bowtie t_1')\rho_1\mu$ is maximal in (select E)$\rho_1\mu$;
S12 select $D = \{\}$;
S13 $(t_2 \approx t_2')\rho_2\mu$ is strictly maximal in $D\rho_2\mu$;
S14 type_preserving_on $\mathcal{F}$ $\mathcal{V}_3$ (clause.vars $(E\rho_1) \cup$ clause.vars $(D\rho_2)$) μ;
S15 type_preserving_on $\mathcal{F}$ $\mathcal{V}_1$ (clause.vars E) ρ_1;
S16 type_preserving_on $\mathcal{F}$ $\mathcal{V}_2$ (clause.vars D) ρ_2;
S17 $\forall x \in$ clause.vars E. $\mathcal{V}_1\ x = \mathcal{V}_3\ (x\rho_1)$;
S18 $\forall x \in$ clause.vars D. $\mathcal{V}_2\ x = \mathcal{V}_3\ (x\rho_2)$;
S19 $\forall \tau$. welltyped $\mathcal{F}$ $\mathcal{V}_2$ t_2 $\tau \longleftrightarrow$ welltyped $\mathcal{F}$ $\mathcal{V}_2$ t_2' τ;
S20 infinite_variables_per_type $\mathcal{V}_1$;
S21 infinite_variables_per_type $\mathcal{V}_2$.

5.10 Lifting the Calculus

Before proving that the typed calculus is sound and complete, we must adapt the lifting of the ground calculus to the nonground level to accommodate the specified type system. The saturation framework's lifting_intersection locale [27, Section 3.3] allows us to lift a family of ground calculi. Before we can establish the lifting in the superposition_calculus locale, we must adapt the definitions of ground instances of clauses and inferences, and the nonground bottom elements, which are parameters of the lifting.

Definition 36. The function ground_instances :: $('f, 'v, 'ty)$ *typed_clause* $\Rightarrow$ $'f$ *gatom clause set* maps a given weakly well-typed clause to the set of its weakly well-typed ground instances:

$$\begin{aligned}&\forall \mathcal{V}\ C.\ \text{ground\_instances}\ \langle \mathcal{V}, C\rangle =\\ &\quad \{C\gamma \mid \text{clause.is\_ground}\ (C\gamma) \wedge\\ &\qquad \text{type\_preserving\_on}\ \mathcal{F}\ \mathcal{V}\ (\text{clause.vars}\ C)\ \gamma \wedge\\ &\qquad \text{infinite\_variables\_per\_type}\ \mathcal{V} \wedge\\ &\qquad \text{weakly\_welltyped}\ \mathcal{F}\ \mathcal{V}\ C\}\end{aligned}$$

Analogously, the $\text{ground\_instances}_{\text{Inf}}$ function maps well-typed inferences to the set of its well-typed ground instances. For inferences with multiple premises, $\text{ground\_instances}_{\text{Inf}}$ also renames the variables of the premises apart.

Definition 37. The set $\bot_F$ :: $('f, 'v, 'ty)$ *typed_clause set* contains all empty typed clauses whose variable-type environment maps infinitely many variables to each type in its range:

$$\bot_F = \{\langle \mathcal{V}, \bot\rangle \mid \text{infinite\_variables\_per\_type}\ \mathcal{V}\}$$

We consider empty typed clauses as bottom elements only if their variable-type environment satisfies the infinite_variables_per_type predicate. This maintains consistency with the definition of ground_instances, since otherwise, some bottom elements would have no ground instances. The saturation framework conveniently allows us to specify, instead of a single bottom element, a family of bottom elements. Without this, we would need to choose a single representative $\langle \mathcal{V}, \bot\rangle$ and use it systematically in the inferences.

Lemma 38. *The* superposition_calculus *locale lifts a family of ground calculi [12, Section 5] to the nonground level using the* lifting_intersection *locale.*

Proof sketch. We demonstrate two proof obligations that differ from the untyped calculus:

- *The set of bottom elements is nonempty.* We show that the set $\bot_F$ is nonempty by proving that there exists a variable-type environment that satisfies the infinite_variables_per_type predicate. This follows directly from Lemma 21.
- *Every bottom element has a ground instance.* Every bottom element in $\bot_F$ is the empty clause, which is ground and well typed. Additionally, every bottom element has a variable-type environment that satisfies the infinite_variables_per_type predicate. □

The lifting_intersection locale provides, among many other definitions and lemmas, a lifted definition of the entailment relation for nonground clause sets [27, Section 3.1]. We can use this definition to prove the soundness of the typed calculus.

Theorem 39 (Soundness). *Let ι be a nonground inference. The premises of ι entail the conclusion of ι.*

Proof sketch. Consider the case where ι = eq_resolution $\langle \mathcal{V}, D \rangle$ $\langle \mathcal{V}, C \rangle$, where C and D are nonground clauses and $\mathcal{V}$ is a variable-type environment. Then we must show that $\langle \mathcal{V}, D \rangle$ entails $\langle \mathcal{V}, C \rangle$. After unfolding multiple definitions, we arrive at a point where we have a valid interpretation $\mathcal{I}$ [12, Definition 4] and a ground clause $C_G \in$ ground_instances C. Furthermore, $\mathcal{I} \models D_G$ for all $D_G \in$ ground_instances D. From the definition of ground_instances (Definition 36), we observe that C is weakly well typed and that there exists a type-preserving grounding substitution γ such that $C_G = C\gamma$. From Lemma 33, we also know that D is weakly well typed. Applying Lemma 15, we obtain a type-preserving grounding substitution γ' for D such that $C_G = C\gamma = C\gamma'$. We then prove $D\gamma' \in$ ground_instances D, from which $\mathcal{I} \models D\gamma'$ follows. Thus, we can prove $\mathcal{I} \models C\gamma$ in the same way as in the untyped calculus [12, Section 5].

The proofs for the eq_factoring and superposition cases are analogous. However, for the superposition rule, we must also apply its additional side conditions S20 and S21, since they are required for the ground instances of the premises to be valid. □

We conclude our endeavor by proving the refutational completeness of the nonground calculus.

Theorem 40 (Refutational Completeness). *Let N be a set of typed clauses that is saturated w.r.t. the typed superposition calculus. If N entails $\bot_F$, then $\bot_F \in N$.*

Proof sketch. The lifting_intersection locale reduces our proof of static refutational completeness of the nonground calculus to two easier proof obligations:

1. *Every member of the ground calculus family is statically refutationally complete.* Since we did not alter the ground inferences, we can keep the proof unchanged [12, Theorem 14].
2. *The nonground inferences overapproximate all ground inferences of a member of the lifted ground calculus family.* We need to adapt the proof for the untyped calculus [12, Lemmas 23–25 and 27] to account for the type system. For example, the adapted inference lifting proofs require as additional assumptions that the premises are weakly well typed and exploit Lemmas 33 and 35 to show that the inferences preserve well-typedness. In the case of the superposition rule, we also require Lemma 15 to obtain type-preserving renaming and grounding substitutions for its two premises. Moreover, we use Lemma 21 to obtain the variable-type environment for the rule's conclusion. □

As a final sanity check, we instantiate the typed calculus with unit as the only type in the type system and show that it is equivalent to the original untyped calculus.

Our formalization, especially the refactoring, benefited from Isabelle's locale mechanism to capture dependencies and facilitate reuse. However, we identified two pain points related to locales. First, in most cases, we must explicitly provide the parameters for locale interpretations and cannot reuse those of dependent interpretations. This results in boilerplate code. Second, a deep hierarchy of locales can cause slow dependency resolution and increase verification times.

6 Related Work

Until the early 2010s, the superposition provers E [21], SPASS [32], and Vampire [20] supported only untyped first-order logic. Sorts were added to all three during that decade. With a typing discipline, it is possible to combine finite and infinite domains of discourse in a single problem, without resorting to encodings [7, 10]. In particular, sorts are especially useful for supporting infinite-domain theories such as arithmetic [16].

Going beyond monomorphic sorts, the Zipperposition [11] prover and recent versions of Vampire [5] support rank-1 polymorphism, and the experimental Pirate [30] supports rank-1 polymorphism extended with Isabelle-style type classes. Moreover, Leo-III [22], whose proof calculus is a variant of superposition, is another rank-1-polymorphic prover.

On the metatheory side, the most noteworthy results are probably the development of "soft sorts" (i.e., unary predicates treated specially by the calculus) by Weidenbach [31] and the extension of superposition with rank-1 polymorphism and type classes by Wand [30].

On the formalization side, Ahmed and Toth [1] formalized the ordered resolution calculus with simple monomorphic types based on our work. Moreover, during the development of our formalization, Yamada and Thiemann [34] independently produced a formalization of many-sorted first-order terms. Because our calculus is parameterized by a type system, we can instantiate it with their type system. Establishing the required assumptions (i.e., proving Lemma 18) for their type system requires only about 150 lines of proof text.

7 Conclusion

We extended an Isabelle/HOL formalization of the superposition calculus with simple monomorphic types, or sorts. We parameterized the calculus with a type system. Extending the ground calculus was straightforward, because the untyped inferences preserve well-typedness. At the nonground level, this is no longer the case. Inferences must compare some terms' types, ensure that variable renaming preserves sorts, and ensure that unification yields type-preserving substitutions.

As future work, we plan to extend the formalization to support rank-1 polymorphism in the style of TPTP TF1 [8]. Another avenue would be to refine the calculus to obtain a verified executable superposition prover.

Acknowledgments

We thank David Schrank for contributing proofs about the existence of type-preserving renamings. We thank Xavier Généreux, Massin Guerdi, Mark Summerfield, and the anonymous reviewers for suggesting textual improvements.

This research was cofunded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Adnan Mohammed Ahmed and Balazs Toth. 2025. Typed ordered resolution. *Archive of Formal Proofs* (2025). https://isa-afp.org/entries/Typed_Ordered_Resolution.html.

[2] Leo Bachmair and Harald Ganzinger. 1990. On restrictions of ordered paramodulation with simplification. In *CADE-10 (LNCS, Vol. 449)*, Mark E. Stickel (Ed.). Springer, 427–441. https://doi.org/10.1007/3-540-52885-7_105

[3] Leo Bachmair and Harald Ganzinger. 1994. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.* 4, 3 (1994), 217–247. https://doi.org/10.1093/logcom/4.3.217

[4] Leo Bachmair and Harald Ganzinger. 2001. Resolution theorem proving. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Vol. I. Elsevier and MIT Press, 19–99. https://doi.org/10.1016/b978-044450813-3/50004-7

[5] Ahmed Bhayat and Giles Reger. 2020. A polymorphic Vampire (short paper). In *IJCAR 2020, Part II (LNCS, Vol. 12167)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 361–368. https://doi.org/10.1007/978-3-030-51054-1_21

[6] Jasmin Christian Blanchette. 2019. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In *CPP 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 1–13. https://doi.org/10.1145/3293880.3294087

[7] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. 2016. Encoding monomorphic and polymorphic types. *Logical Methods in Computer Science* 12, 4 (2016). https://doi.org/10.2168/lmcs-12(4:13)2016

[8] Jasmin Christian Blanchette and Andrei Paskevich. 2013. TFF1: The TPTP typed first-order form with rank-1 polymorphism. In *CADE-24 (LNCS, Vol. 7898)*, Maria Paola Bonacina (Ed.). Springer, 414–420. https://doi.org/10.1007/978-3-642-38574-2_29

[9] Jasmin Christian Blanchette and Sophie Tourret. 2020. Extensions to the comprehensive framework for saturation theorem proving. *Archive of Formal Proofs* (2020). https://isa-afp.org/entries/Saturation_Framework_Extensions.html.

[10] Koen Claessen, Ann Lillieström, and Nicholas Smallbone. 2011. Sort it out with monotonicity: Translating between many-sorted and unsorted first-order logic. In *CADE-23 (LNCS, Vol. 6803)*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer, 207–221. https://doi.org/10.1007/978-3-642-22438-6_17

[11] Simon Cruanes. 2015. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond.* PhD thesis. École Polytechnique.

[12] Martin Desharnais, Balazs Toth, Uwe Waldmann, Jasmin Blanchette, and Sophie Tourret. 2024. A modular formalization of superposition in Isabelle/HOL. In *ITP 2024*, Yves Bertot, Temur Kutsia, and Michael Norrish (Eds.), Vol. 309. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 12:1–12:20. https://doi.org/10.4230/LIPICS.ITP.2024.12

[13] Martin Desharnais, Petar Vukmirović, Jasmin Blanchette, and Makarius Wenzel. 2022. Seventeen provers under the hammer. In *ITP 2022*, June Andronick and Leonardo de Moura (Eds.), Vol. 237. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 8:1–8:18. https://doi.org/10.4230/LIPICS.ITP.2022.8

[14] Martin Desharnais-Schäfer and Balazs Toth. 2024. Abstract substitution. *Archive of Formal Proofs* (2024). https://isa-afp.org/entries/Abstract_Substitution.html.

[15] Martin Desharnais-Schäfer and Balazs Toth. 2024. A modular formalization of superposition. *Archive of Formal Proofs* (2024). https://isa-afp.org/entries/Superposition_Calculus.html.

[16] Konstantin Korovin and Andrei Voronkov. 2007. Integrating linear arithmetic into superposition calculus. In *CSL 2007 (LNCS, Vol. 4646)*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer, 223–237. https://doi.org/10.1007/978-3-540-74915-8_19

[17] Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and Vampire. In *CAV 2013 (LNCS, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 1–35. https://doi.org/10.1007/978-3-642-39799-8_1

[18] 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 Series in Computing, Vol. 2)*, Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska (Eds.). EasyChair, 1–11. https://doi.org/10.29007/36dt

[19] Nicolas Peltier. 2016. A variant of the superposition calculus. *Archive of Formal Proofs* (2016). https://www.isa-afp.org/entries/SuperCalc.html.

[20] Alexandre Riazanov and Andrei Voronkov. 2002. The design and implementation of VAMPIRE. *AI Communications* 15, 2-3 (2002), 91–110.

[21] Stephan Schulz. 2002. E—a brainiac theorem prover. *AI Communications* 15, 2-3 (2002), 111–126.

[22] Alexander Steen and Christoph Benzmüller. 2018. The higher-order prover Leo-III. In *IJCAR 2018 (LNCS, Vol. 10900)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). Springer, 108–116. https://doi.org/10.1007/978-3-319-94205-6_8

[23] René Thiemann and Christian Sternagel. 2009. Certification of termination proofs using CeTA. In *TPHOLs 2009 (LNCS, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 452–468. https://doi.org/10.1007/978-3-642-03359-9_31

[24] René Thiemann, Christian Sternagel, Christina Kirk, Martin Avanzini, Bertram Felgenhauer, Julian Nagele, Thomas Sternagel, Sarah Winkler, and Akihisa Yamada. 2025. First-order rewriting. *Archive of Formal Proofs* (2025). https://isa-afp.org/entries/First_Order_Rewriting.html.

[25] Balazs Toth. 2025. First order clause. *Archive of Formal Proofs* (2025). https://isa-afp.org/entries/First_Order_Clause.html.

[26] Sophie Tourret. 2020. A comprehensive framework for saturation theorem proving. *Archive of Formal Proofs* (2020). https://www.isa-afp.org/entries/Saturation_Framework.html.

[27] Sophie Tourret and Jasmin Blanchette. 2021. A modular Isabelle framework for verifying saturation provers. In *CPP 2021*, Cătălin Hriţcu and Andrei Popescu (Eds.). ACM, 224–237. https://doi.org/10.1145/3437992.3439912

[28] Dmitry Traytel, Andrei Popescu, and Jasmin Christian Blanchette. 2012. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *LICS 2012*. IEEE Computer Society, 596–605. https://doi.org/10.1109/LICS.2012.75

[29] Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. 2022. A comprehensive framework for saturation theorem proving. *Journal of Automated Reasoning* 66, 4 (2022), 499–539. https://doi.org/10.1007/S10817-022-09621-7

[30] Daniel Wand. 2014. Polymorphic+typeclass superposition. In *PAAR-2014 (EPiC Series in Computing, Vol. 31)*, Stephan Schulz, Leonardo de Moura, and Boris Konev (Eds.). EasyChair, 105–119. https://doi.org/10.29007/8v2f

[31] Christoph Weidenbach. 2001. Combining superposition, sorts and splitting. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Vol. II. Elsevier and MIT Press, 1965–2013. https://doi.org/10.1016/B978-044450813-3/50029-1

[32] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS version 3.5. In *CADE-22 (LNCS, Vol. 5663)*, Renate A. Schmidt (Ed.). Springer, 140–145. https://doi.org/10.1007/978-3-642-02959-2_10

[33] Makarius Wenzel. 2007. Isabelle/Isar—a generic framework for human-readable proof documents. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, Roman Matuszewski and Anna Zalewska (Eds.). Studies in Logic, Grammar, and Rhetoric, Vol. 10(23). University of Białystok.

[34] Akihisa Yamada and René Thiemann. 2024. Sorted terms. *Archive of Formal Proofs* (2024). https://isa-afp.org/entries/Sorted_Terms.html.

Received 2025-09-09; accepted 2025-11-13