Polymorphic+Typeclass Superposition

Daniel Wand

Polymorphic+Typeclass Superposition

Daniel Wand

Max-Planck-Institut für Informatik, Saarbruecken, DE.
dwand@mpi-inf.mpg.de
http://www.mpi-inf.mpg.de/~dwand/paar14/

Abstract

We present an extension of superposition that natively handles a polymorphic type system extended with type classes, thus eliminating the need for type encodings when used by an interactive theorem prover like Isabelle/HOL. We describe syntax, typing rules, semantics, the polymorphic superposition calculus and an evaluation on a problem set that is generated from Isabelle/HOL theories. Our evaluation shows that native polymorphic+typeclass performance compares favorably to monomorphisation, a highly efficient but incomplete way of dealing with polymorphism.

1 Introduction

Polymorphism is a prevalent feature in interactive theorem proving and functional programming. There is extensive research in encoding polymorphism into untyped or monomorphic form (e.g. [5, 9, 10]), in an effort to be able to use ATPs on polymorphic problems. Despite of this there are few automated theorem provers (ATPs) capable of natively supporting polymorphism. We are only aware of the SMT solver Alt-Ergo[8].

Superposition and SMT based provers, even without supporting polymorphism, are successful in discharging goals arising in interactive theorem proving; [4] reports combined success rates of 60% on a set of Isabelle/HOL theories. In this abstract we report on work to bring rank-1 polymorphism extended with type classes[15] to superposition. We used TFF1[6]'s polymorphic semantic as a starting point for ours, and extended it with type classes. Our prototype implementation uses an extended version of SPASS's input syntax, which allows utilizing additional annotations described in [7].

2 Declarations

Notation We use f for function symbols, p for predicate symbols, t, s for terms, ϕ for formulas, u for term variables, σ for substitutions, τ for types, κ for type constructors, α for type variables and c for type classes, C for sets of type classes, n for type arity and m for term arity. We write $t[s]_p$ to denote a term, which is t with the subterm at position p of t replaced by s.

Signature A polymorphic+typeclass first-order language signature is a tuple $\Sigma = (\mathcal{F},\mathcal{P},\mathcal{T},\mathcal{TC})$. There must be exactly one declaration per function and predicate symbol.

- $\mathcal{F}$ The elements f : $\forall\alpha_1 : C_1 \ldots \alpha_n : C_n.\tau_1 \ldots \tau_m \to \tau \in \mathcal{F}$ are function declarations. We use the syntax $f< \tau_{f_1}, \ldots, \tau_{f_n} >(t_1, \ldots, t_m)$, to write a function term, denoting the appropriate instantiations of the α_i by the corresponding type argument τ_{f_i} and that of τ_i by the term argument t_i. With the declaration, the return type τ is derivable from this representation.
- $\mathcal{P}$ The elements p : $\forall\alpha_1 : C_1 \ldots \alpha_n : C_n.\tau_1 \ldots \tau_m \to o_{\mathcal{B}} \in \mathcal{P}$ are predicate declarations. Similar to functions we write $p< \tau_{p_1}, \ldots, \tau_{p_n} >(t_1, \ldots, t_m)$.

$\mathcal{T}$ The elements $\forall\alpha_1 : C_1 \ldots \alpha_n : C_n \ . \ \kappa(\alpha_i, \ldots) : c \in \mathcal{T}$ are type declarations.
There must be at most one declaration per type constructor κ and c and at least one per κ. κ must always have the same arity. In order to allow κs, that are not part of any (custom) type class, there is the special type class $\emptyset$ that can be used in the required declaration and represents no type class. All κ are part of this type class.

$\mathcal{TC}$ The elements $c_1 \subseteq c_2 \in \mathcal{TC}$ are subclass declarations.

Type terms Let $\mathcal{X}_\tau$ be a given countably infinite set of type variables. All type terms are recursively defined as:

- Every type variable in $\mathcal{X}_\tau$ is a type term.
- For all $\forall\alpha_1 : C_1 \ldots \alpha_n : C_n \ . \ \kappa(\alpha_i, \ldots) : c \in \mathcal{T}$ let $\tau_1,\ldots,\tau_n$ be type terms then $\kappa(\tau_1,\ldots,\tau_n)$ is also a type term.

Terms Let $\mathcal{X}_t$ be a given countably infinite set of term variables. All, possibly ill-typed, terms are recursively defined as:

- Every term variable in $\mathcal{X}_t$ is a term.
- For all f : $\forall\alpha_1 : C_1 \ldots \alpha_n : C_n.\tau_1 \ldots \tau_m \to \tau \in \mathcal{F}$ let $t_1,\ldots,t_m$ be terms and $\tau_1,\ldots,\tau_n$ be type terms then f$< \tau_1,\ldots,\tau_n >($ $t_1,\ldots,t_m$ $)$ is also a term.

Formulas All, possibly ill-typed, formulas are recursively defined as:

1. For all p : $\forall\alpha_1 : C_1 \ldots \alpha_n : C_n.\tau_1 \ldots \tau_m \to o_\mathcal{B} \in \mathcal{P}$ let $t_1,\ldots,t_m$ be terms and $\tau_1,\ldots,\tau_n$ be type terms, then p$< \tau_1,\ldots,\tau_n >($ $t_1,\ldots,t_m$ $)$ is a (atomic) formula.
2. Let t_1, t_2 be terms, then $t_1 \approx t_2$ is a (atomic) formula.
3. Let ϕ_1, ϕ_2 be formulas, then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \Rightarrow \phi_2$, $\phi_1 \Leftrightarrow \phi_2$ and $\neg\phi_1$ are also formulas.
4. Let ϕ be a formula, u a fresh term variable and τ a type term, then $\forall u{:}\tau.\phi$ and $\exists u{:}\tau.\phi$ are also formulas.
5. Let ϕ be a formula, α a fresh type variable and C an optional type class constraint (a set of type classes written as $c_1\&\ldots\&c_n$), then $\forall_{\text{types}}\alpha{:}C.\ \phi$ is also a formula.

Note that both quantifiers in 4. range over term variables. The quantifier in 5. ranges over type variables and its existential equivalent is not well typed, since existential types are not supported by this formalization.

3 Syntax

We extended the monomorphic SPASS syntax to support our polymorphic+typeclass calculus. We show here one simple example file and make the complete specification available on the abstracts webpage. Every problem file must begin with the following line, where "example" can be replaced by an other string:

```
begin_problem(example).
```

Continuing with the preamble where additional information may be entered inbetween $*$ and $*$.

```
list_of_descriptions.
name({**}).
author({**}).
```

```
status(unknown).
description({**}).
end_of_list.
```

Next, the used symbols must be defined with their arity. Where a single number describes the term arity and where number1+number2 describes the type arity to be number1 and the term arity to be number2.

```
list_of_symbols.
functions [(nil, 1+0), (cons, 1+2), (zero, 0), (s, 1), (plus, 2)].
predicates [(p1, 1+1),(p2, 2)].
types [(list, 1), nat].
classes [ordered,superordered].
end_of_list.
```

Now the declarations described in the previous section follow:

```
list_of_declarations.
function(nil, [A], list(A)).
function(cons,[A], (A, list(A)) list(A)).
function(zero, nat).
function(s, (nat) nat).
function(plus, (nat, nat) nat).
predicate(p1,[A:ordered], A,A).
predicate(p2, nat, nat).
type(nat, ordered).
type([A:superordered], list(A), superordered).
subclass(ordered, superordered).
end_of_list.
```

The set of axioms:

```
list_of_formulae(axioms).
formula(forall([X:nat],equal(plus(zero,X),X)), plus_zero).
formula(forall([X:nat,Y:nat],equal:lr(plus(s(X),Y),s(plus(X,Y)))), plus_s).
formula(forall_types([A],forall([X:A,Y:A],p1<A>(X,Y))), p1).
end_of_list.
```

And the conjecture:

```
list_of_formulae(conjectures).
formula(forall([X:nat],equal(plus(zero,X),X)), conjecture).
end_of_list.
```

4 Typing Rules

Let $\gamma_{\mathcal{C}}^{\mathcal{T}}$ represent a typing context, with $\mathcal{T}$ as mapping from term variables to types or to a special value denoting "undefined" and $\mathcal{C}$ as mapping from type variables to a sets of type classes or to "undefined". All well-typedness assertions are implicitly relative to a given signature.

Type Classes Type classes represent sets of types. A set of type classes corresponds to the intersection of the sets the individual type classes represent.

$$\frac{}{\mathcal{C} \vdash \tau{:}\emptyset}\ ^{\text{(empty)}} \qquad \frac{}{\mathcal{C} \vdash \alpha{:}\mathcal{C}(\alpha)}\ ^{\text{(type var)}}$$

$$\frac{\forall \alpha_1 : \mathrm{C}_1 \ldots \alpha_\mathrm{n} : \mathrm{C}_\mathrm{n}.\ \kappa(\alpha_\mathrm{i}, \ldots) : \mathrm{c} \in \mathcal{T} \quad \mathcal{C} \vdash \kappa(\alpha_i, \ldots)\sigma{:}C \quad \mathcal{C} \vdash \alpha_1\sigma{:}C_1 \quad \ldots \quad \mathcal{C} \vdash \alpha_m\sigma{:}C_m}{\mathcal{C} \vdash \kappa(\alpha_i, \ldots)\sigma{:}C \uplus \{c\}}\ ^{(\kappa \text{ of c})}$$

$$\frac{\mathrm{c}_1 \subseteq \mathrm{c}_2 \in \mathcal{TC} \quad \mathcal{C} \vdash \tau{:}C \uplus \{c_1\}}{\mathcal{C} \vdash \tau\sigma{:}C \uplus \{c_2\}}\ ^{\text{(subclass)}}$$

Terms A term is well-typed, for a given $\gamma_\mathcal{C}^\mathcal{T}$, if and only if a type can be derived by the following rules:

$$\frac{}{\gamma_\mathcal{C}^\mathcal{T} \vdash u : \mathcal{T}(u)}\ ^{\text{(term var)}}$$

$$\frac{\mathrm{f} : \forall \alpha_1 : \mathrm{C}_1 \ldots \alpha_\mathrm{n} : \mathrm{C}_\mathrm{n}.\tau_1 \ldots \tau_\mathrm{m} \to \tau \in \mathcal{F} \quad \gamma_\mathcal{C}^\mathcal{T} \vdash t_1 : \tau_1\sigma \quad \ldots \quad \gamma_\mathcal{C}^\mathcal{T} \vdash t_n : \tau_n\sigma}{\gamma_\mathcal{C}^\mathcal{T} \vdash f < \alpha_1\sigma, \ldots, \alpha_m\sigma > (t_1, \ldots, t_n) : \tau\sigma}\ ^{\text{(function)}}$$

and for all $i : 1..n.\ \mathcal{C} \vdash \alpha_i\sigma \mathbf{:} C_i$

Formulas Without loss of generality this section assumes that all variables are named uniquely. It further assumes that $o_\mathcal{B}$, o are types which do not match any type in the signature. We use two different boolean types to fix the quantifier for types ($\forall_{\text{types}}$) to appear only on the very top position in formulas.

$$\frac{\gamma_\mathcal{C}^\mathcal{T} \vdash s : \tau \quad \gamma_\mathcal{C}^\mathcal{T} \vdash t : \tau}{\gamma_\mathcal{C}^\mathcal{T} \vdash s \approx t : o_\mathcal{B}}\ ^{(\approx)} \qquad \frac{\gamma_\mathcal{C}^\mathcal{T} \vdash \phi_1 : o_\mathcal{B} \quad \gamma_\mathcal{C}^\mathcal{T} \vdash \phi_2 : o_\mathcal{B}}{\gamma_\mathcal{C}^\mathcal{T} \vdash \phi_1\ \#\ \phi_2 : o_\mathcal{B}}\ ^{(\#\in\{\wedge,\vee,\Rightarrow,\Leftrightarrow\})} \qquad \frac{\gamma_\mathcal{C}^\mathcal{T} \vdash \phi : o_\mathcal{B}}{\gamma_\mathcal{C}^\mathcal{T} \vdash \neg\phi : o_\mathcal{B}}\ ^{(\neg)}$$

$$\frac{\mathrm{p} : \forall \alpha_1 : \mathrm{C}_1 \ldots \alpha_\mathrm{n} : \mathrm{C}_\mathrm{n}.\ \tau_1 \ldots \tau_\mathrm{m} \to \mathrm{o}_\mathcal{B} \in \mathcal{P} \quad \gamma_\mathcal{C}^\mathcal{T} \vdash t_1 : \tau_1\sigma \quad \ldots \quad \gamma_\mathcal{C}^\mathcal{T} \vdash t_n : \tau_n\sigma}{\gamma_\mathcal{C}^\mathcal{T} \vdash p < \alpha_1\sigma, \ldots, \alpha_m\sigma > (t_1, \ldots, t_n) : o_\mathcal{B}}\ ^{\text{(predicate)}}$$

and for all $i : 1..m.\ \mathcal{C} \vdash \alpha_i\sigma \mathbf{:} C_i$

$$\frac{\gamma_\mathcal{C}^{\mathcal{T}[u\mapsto\tau]} \vdash \phi : o_\mathcal{B}}{\gamma_\mathcal{C}^\mathcal{T} \vdash \#u : \tau.\ \phi : o_\mathcal{B}}\ ^{(\#\in\{\forall,\exists\})} \qquad \frac{\gamma_\mathcal{C}^\mathcal{T} \vdash \phi : o_\mathcal{B}}{\gamma_\mathcal{C}^\mathcal{T} \vdash \phi : o}\ ^{\text{(top)}} \qquad \frac{\gamma_{\mathcal{C}[\alpha\mapsto C]}^\mathcal{T} \vdash \phi : o}{\gamma_\mathcal{C}^\mathcal{T} \vdash \forall_{\text{types}}\alpha : C.\ \phi : o}\ ^{(\forall_{\text{types}})}$$

We call a formula ϕ well-typed if and only if one can derive starting from

$$\gamma_\mathcal{C}^\mathcal{T} \vdash \phi : o$$

a proof tree by applying the above rules with an empty $\gamma_\mathcal{C}^\mathcal{T}$.

5 Semantic

Structure Given a polymorphic+typeclass first-order language signature the polymorphic+typeclass first-order language structure is a tuple $\Sigma = (\mathcal{U},\mathcal{D},\mathcal{I})$:

- $\mathcal{U}$ The (non-empty) universe.
- $\mathcal{D}$ The (non-empty) domains representing (non-empty) types (subsets of $\mathcal{U}$).
- $\mathcal{I}$ The interpretation function

We denote the interpretation of a well typed {formula,term,type,typeclass} t as $[\![t]\!]^{\mathcal{I}}_{\theta,\xi}$[1]. Where θ is a mapping from type variables to elements of $\mathcal{D}$ and ξ is a mapping from term variables to elements of $\mathcal{U}$. We write $\theta[\alpha \mapsto \tau]$ (respectively $\xi[u \mapsto e]$) to represent a new mapping which is identical to θ (ξ) but maps α to τ (u to e). To interpret a (non-sub) formula, θ and ξ are set to be empty.

Types	$[\![\alpha]\!]^{\mathcal{I}}_{\theta}$	::=	$\theta(\alpha)$
	$[\![\kappa(\tau_1,\ldots,\tau_n)]\!]^{\mathcal{I}}_{\theta}$	::=	$\kappa^{\mathcal{I}}([\![\tau_1]\!]^{\mathcal{I}}_{\theta},\ldots,[\![\tau_n]\!]^{\mathcal{I}}_{\theta})$
Term	$[\![u]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$\xi(u)$
	$[\![f<\tau_1,\ldots,\tau_n>(t_1,\ldots,t_m)]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$f^{\mathcal{I}}<[\![\tau_1]\!]^{\mathcal{I}}_{\theta},\ldots,[\![\tau_n]\!]^{\mathcal{I}}_{\theta}>([\![t_1]\!]^{\mathcal{I}}_{\theta,\xi},\ldots,[\![t_m]\!]^{\mathcal{I}}_{\theta,\xi})$
Formulas	$[\![p<\tau_1,\ldots,\tau_n>(t_1,\ldots,t_m)]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$p^{\mathcal{I}}<[\![\tau_1]\!]^{\mathcal{I}}_{\theta},\ldots,[\![\tau_n]\!]^{\mathcal{I}}_{\theta}>([\![t_1]\!]^{\mathcal{I}}_{\theta,\xi},\ldots,[\![t_m]\!]^{\mathcal{I}}_{\theta,\xi})$
	$[\![s \approx t]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$[\![s]\!]^{\mathcal{I}}_{\theta,\xi} \approx^{\mathcal{I}} [\![t]\!]^{\mathcal{I}}_{\theta,\xi}$
	$[\![s \wedge t]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$[\![s]\!]^{\mathcal{I}}_{\theta,\xi} \wedge^{\mathcal{I}} [\![t]\!]^{\mathcal{I}}_{\theta,\xi}$
	$[\![s \vee t]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$[\![s]\!]^{\mathcal{I}}_{\theta,\xi} \vee^{\mathcal{I}} [\![t]\!]^{\mathcal{I}}_{\theta,\xi}$
	$[\![s \Rightarrow t]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$[\![s]\!]^{\mathcal{I}}_{\theta,\xi} \Rightarrow^{\mathcal{I}} [\![t]\!]^{\mathcal{I}}_{\theta,\xi}$
	$[\![s \Leftrightarrow t]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$[\![s]\!]^{\mathcal{I}}_{\theta,\xi} \Leftrightarrow^{\mathcal{I}} [\![t]\!]^{\mathcal{I}}_{\theta,\xi}$
	$[\![\neg t]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$\neg^{\mathcal{I}}[\![t]\!]^{\mathcal{I}}_{\theta,\xi}$
	$[\![\forall u:\tau.\psi]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$\forall^{\mathcal{I}} e \in [\![\tau]\!]^{\mathcal{I}}_{\theta}.[\![\psi]\!]^{\mathcal{I}}_{\theta,\xi[u\to e]}$
	$[\![\exists u:\tau.\psi]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$\exists^{\mathcal{I}} e \in [\![\tau]\!]^{\mathcal{I}}_{\theta}.[\![\psi]\!]^{\mathcal{I}}_{\theta,\xi[u\to e]}$
	$[\![\forall_{\text{types}}\alpha:C.\psi]\!]^{\mathcal{I}}_{\theta,\xi}$	::=	$\forall^{\mathcal{I}}_{\text{types}}\tau \in [\![C]\!]^{\mathcal{I}}.[\![\psi]\!]^{\mathcal{I}}_{\theta[\alpha\mapsto\tau],\xi}$
Type Class	$[\![\emptyset]\!]^{\mathcal{I}}$	::=	$\mathcal{D}$
	$[\![C]\!]^{\mathcal{I}}$	::=	$\bigcap_{c\in C}[\![c]\!]^{\mathcal{I}}$
	$[\![c]\!]^{\mathcal{I}}$	::=	least fix point of rules 1 and 2

Type Class Fix point

1. For all declarations $\forall\alpha_1{:}C_1,\ldots,\alpha_n{:}C_n.\tau{:}c \in \mathcal{T}$ require that all valid instantiations of τ's type variables are also of τ's type class c:

   $(\bigwedge_{1\le i\le n} \tau_i \in [\![C_i]\!]^{\mathcal{I}}) \Rightarrow (\ldots, \alpha_i \mapsto \tau_i, \ldots)(\tau) \in [\![\mathrm{c}]\!]^{\mathcal{I}}$

2. For all declarations $c_1 \subseteq c_2 \in \mathcal{TC}$ require that if a type term τ is element of the subclass c_1 then τ is also element of the superclass c_2.

   $\tau \in [\![c_1]\!]^{\mathcal{I}} \Rightarrow \tau \in [\![c_2]\!]^{\mathcal{I}}$

[1] ξ is not needed for interpretation of types. θ, ξ are not needed for interpretation of type classes.

Following from Tarski's fix point theorem the rules 1 and 2 have a least fix point.

6 Typed Superposition

Preprocessing Initially formulas are transformed into Clause Normal Form[2] (CNF). This is unaffected by the type system, except for skolemization. Skolemization has to add corresponding type declarations into $\mathcal{T}$. The necessary declarations are obvious, the existential variables' type becomes the Skolem function's return type and its argument types are the types of the universal variables, whose scope includes the existential variable.
Since we only have universally quantified variables in CNF, we omit all quantifiers and annotate variables(u) with their type(τ) and write $u\{\tau\}$ to express that u is of type τ. Functions and predicates[2] are represented by their return type and their argument terms and 'true' types arguments. In detail:

$$f < \tau_{\alpha_1}, \ldots, \tau_{\alpha_n} > (t_1, \ldots, t_m)$$

is represented by

$$f\{\tau_r, \tau_{\alpha_i}, \ldots\}(t_1, \ldots, t_m)$$

where $\forall \alpha_1 : C_1 \ldots \alpha_n : C_n . \tau_1 \ldots \tau_m \to \tau \in \mathcal{F}$ is f's declaration. τ_r is defined to be τ instantiated by the substitution σ such that for all j in 1..n: $\alpha_j\sigma$ is τ_{a_j} and for all j in 1..m: the type of t_j is $\tau_j\sigma$. The τ_{α_i} are of those i whose α_i does not occur in τ and not in any j in 1..m: τ_j (and are ordered increasing in i). This is useful to e.g. have polymorphic unary predicates. Clearly both representation, together with $\mathcal{F}$ or $\mathcal{P}$, provide the same information.

The advantage of the second form is that it requires no lookup of $\mathcal{F}$ and $\mathcal{P}$ to perform unification between variables and functions.

Substitutions Let σ be a mapping $X \to T_\Sigma(X)$ which maps term variables to terms and type variables to type terms. We write substitutions as $(u_1 \mapsto t_1, \ldots, \alpha_1 \mapsto \tau_1 \ldots)$ and instead of $\sigma(x)$ we also write $x\sigma$. We define $\sigma[u_1 \mapsto t_1]$ to be t_1 for u_1 and for all other variables u to be $u\sigma$ (correspondingly for α). We define the application of σ on terms as follows:

$$f\{\tau\}(t_1, \ldots, t_m)\sigma ::= f\{\tau\sigma\}(t_1\sigma, \ldots, t_m\sigma)$$

and on type terms as:

$$\kappa(\tau_1, \ldots, \tau_n)\sigma ::= \kappa(\tau_1\sigma, \ldots, \tau_n\sigma)$$

We call a substitution σ *grounding* for a given term t if tσ contains no variables; a *unifier* of the terms t_1 and t_2 if $t_1\sigma \approx t_2\sigma$; *more general* than σ_2 if for all terms t there exists σ_1 such that $t\sigma\sigma_1 \approx t\sigma_2$ and a *most general unifier* if it is more general than all other unifiers.

Unification We now give the unification rules for terms; starting with non-variable terms: Note that preprocessing has simplified unification for functions and predicates.

1	$f\{\tau_l\}(t_{l_1}, \ldots, t_{l_m}) \doteq f\{\tau_r\}(t_{r_1}, \ldots, t_{r_m})$, E	$\Rightarrow$	$\tau_l \doteq \tau_r, t_{l_1} \doteq t_{r_1}, \ldots, t_{l_m} \doteq t_{r_m}$,E
2	$p(t_{l_1}, \ldots, t_{l_m}) \doteq p(t_{r_1}, \ldots, t_{r_m})$, E	$\Rightarrow$	$t_{l_1} \doteq t_{r_1}, \ldots, t_{l_m} \doteq t_{r_m}$,E
3	$\kappa(\tau_{1_l}, \ldots, \tau_{n_l}) \doteq \kappa(\tau_{1_r}, \ldots, \tau_{n_r})$, E	$\Rightarrow$	$\tau_l \doteq \tau_{l_1} \doteq \tau_{r_1}, \ldots, t_{l_n} \doteq t_{r_n}$,E

for term variables:

[2]We use the same transformation for predicates, but omit the return type.

4	$t \doteq u$, E	$\Rightarrow$	$u \doteq t$,E	t non-variable
5	$u\{\tau_u\} \doteq t\{\tau_t\}$, E	$\Rightarrow$	$u\{\tau_u\} = t\{\tau_t\}, \tau_u \doteq \tau_t, (u\{\tau_u\} \mapsto t\{\tau_t\})$(E)	$u \notin \text{vars}(t)$
6	$u_l\{\tau_l\} \doteq u_r\{\tau_r\}$, E	$\Rightarrow$	$u_l\{\tau_l\} = u_r\{\tau_r\}, \tau_l \doteq \tau_r, (u_l\{\tau_l\} \mapsto u_r\{\tau_r\})$(E)	

for type variables:

7	$\alpha_1\{C_1\} \doteq \alpha_2\{C_2\}$, E	$\Rightarrow$	$C_3 = C_1 \cup C_2, \alpha_1\{C_1\} = \alpha_3\{C_3\}, \alpha_2\{C_1\} = \alpha_3\{C_3\},$ $(\alpha_1\{C_1\} \mapsto \alpha_3\{C_3\}, \alpha_2\{C_1\} \mapsto \alpha_3\{C_3\})(E)$	α_3 fresh C_3 pop.
8	$\alpha_1\{C\} \doteq \tau$, E	$\Rightarrow$	$\alpha_1\{C\} = \tau\sigma, (\sigma[\alpha_1\{C\} \mapsto \tau\sigma])$(E)	σ $\text{mgs}_{\tau,C}$

Where **vars**(t) is the set of all variables that are t or subterms of t. A variable is **fresh** if it was not previously used anywhere. C **pop.** tests if C is populated, meaning that it has at least one type constructor which can construct all type classes $c \in C$. A substitution σ is the most general substitution (**mgs**$_{\tau,C}$) of τ and C if for all σ_2 exists a σ_3 such that if $\tau\sigma_2$ is a member of C (according to $\mathcal{T}$) then $\sigma_2 = \sigma\sigma_3$. This has no effect on ground type terms and will only restrict type variables' type classes constraint, since restricting a type variable is more general than instantiating it[3].

Two terms t and s are unifiable if and only if $t \doteq s$ can be transformed by the above rules into a set of equations not containing $\doteq$. The most general unifier[4] can be derived from the resulting equations.

Typed Knuth-Bendix Order The Knuth-Bendix Order (KBO) is a widely used order in superposition based provers. Based on the presentation in [1, p. 124] we extend the standard KBO to our typed setting, we claim that it is still a simplification order:
Let w_f be a function that maps function symbols to positive numbers, w its extension to terms and $>_f$ be an ordering on function symbols. We define s $\succ_{KBO}$ t to hold if and only if:

1. For all term variables u: $|s|_u \geq |t|_u$ and $w(s) > w(t)$, or
2. For all term variables u: $|s|_u \geq |t|_u$ and $w(s) = w(t)$, and one of the following:
   (a) There exists a unary function symbol f, a variable u and a positive integer n such that $s = (f\{\tau\})^n(u)$ and $t = u$
   (b) There exists functions symbols f, g such that $f >_f g$ and $s = f\{\tau\}(s_1, \ldots, s_n)$ and $t = g\{\tau\}(t_1, \ldots, t_n)$
   (c) There exists a function symbol f and an index i, $1 \leq i \leq n$, such that $s = f\{\tau\}(s_1, \ldots, s_n)$, $t = f\{\tau\}(t_1, \ldots, t_n)$ and $s_1 =_{Term} t_1, \ldots, s_{i-1} =_{Term} t_{i-1}$ and $s_i \succ_{KBO} t_i$
   (d) $s \succ_\tau t$

And the following definitions:

- $\tau_s \succ_\tau \tau_t$ iff there exists a σ such that $\tau_s\sigma \neq \tau_s$ and $\tau_s\sigma = \tau_t$
- $s \succ_\tau t$ iff there exists a function symbol f such that $s = f\{\tau_s\}(s_1, \ldots, s_n)$,
  $t = f\{\tau_t\}(t_1, \ldots, t_n)$, $s_1 =_{Term} t_1, \ldots, s_n =_{Term} t_n$, $s_1 \succeq_\tau t_1, \ldots, s_n \succeq_\tau t_n$
  and
  $\tau_s \succ_\tau \tau_t$ or ($\tau_s = \tau_t$ and there exists an index i, $1 \leq$ i $\leq$ m, such that $s_i \succ_\tau t_i$)
- $=_{Term}$ to be equality but only considering non-type (sub)terms.

[3] Except in the corner case, where restricting the type variable and instantiating it, represents the same type terms.

[4] The most general unifier is unique, which we will not show here.

In its presented form, $\succ_{KBO}$ allows ordering of terms with same type and additionally if the non-type terms are equal, more generally typed terms are smaller than stricter typed terms. One could further relax 2a, 2b and 2c to more general types, similar to $\succ_\tau$, but might have to also consider subterms' type terms.

Inferences' Side Conditions The following side conditions are used in the inference rules below. Let $\prec$ be a fixed simplification order[1].

1. σ is the most general unifier of s and s_2
2. σ is the most general unifier of s and s'
3. s_2 is not a term variable
4. $t\sigma \not\preceq t'\sigma$
5. $s\sigma \not\preceq s'\sigma$
6. $(t \approx t')\sigma$ *strictly maximal* in $(D' \vee t \approx t')\sigma$, nothing selected
7. $(s \approx s')\sigma$ *strictly maximal* in $(C' \vee s \approx s')\sigma$, nothing selected
8. $((s \not\approx s')\sigma$ *maximal* in $(C' \vee s \approx s')\sigma$, nothing selected$) \vee s \not\approx s'$ selected
9. $D' \vee t \approx t' \not\succeq C' \vee s[s_2]_p \ \ldots \ s'$
10. $(s \approx t)\sigma$ maximal in $(C' \vee s' \approx t' \vee s \approx t)\sigma$, nothing selected

Inferences We take the superposition calculus of [3] and sketch that it still refutationally complete in our typed setting.

1. Positive Superposition (PSup) with side conditions 1, 3, 4, 5, 6, 7 and 9

$$\frac{D' \vee t \approx t' \quad C' \vee s[s_2]_p \approx s'}{(D' \vee C' \vee s[t']_p \approx s')\sigma} \ (PSup)$$

2. Negative Superposition (NSup) with side conditions 1, 3, 4, 5, 6, 8 and 9

$$\frac{D' \vee t \approx t' \quad C' \vee s[s_2]_p \not\approx s'}{(D' \vee C' \vee s[t']_p \not\approx s')\sigma} \ (NSup)$$

3. Equality Resolution (ER) with side conditions 2 and 8

$$\frac{C' \vee s \not\approx s'}{C'\sigma} \ (ER)$$

4. Equality Factoring (EF) with side conditions 2, 10, $s'\sigma \not\preceq t'\sigma$ and $s\sigma \not\preceq t\sigma$,

$$\frac{C' \vee s' \approx t' \vee s \approx t}{(C' \vee s' \approx t' \vee t \not\approx t')\sigma} \ (EF)$$

Refutational Completeness Sketch In order to show refutational completeness of typed superposition, two main theorems have to be proven. First the inference rules have to be shown well-typedness preserving. This is already sufficient to prove the ground version refutational complete. Then new versions of the lifting lemmas have to be shown. The remaining completeness proof is essentially unchanged.

Lemma 1 Well typedness is determined only by the well typedness of the argument and the typing rule of the construct.

Lemma 2 For all well typed terms t_1, t_2, t_3 and substitutions σ, if σ is the most general unifier of t_2 and t_3 then $t_1\sigma$ is well typed.

Lemma 3 If $t \approx t'$ then the type of t and t' is identical.

Lemma 4 For each inference if the premises are well typed so is the conclusion.

Proof Sketch

Equality Resolution	:	By *Lemma 2*.
Equality Factoring	:	By *Lemma 3* $s\sigma$, $t\sigma$ and $t'\sigma$ all have the common type τ. Thus the equations from the conclusion are also well-typed. $C'\sigma$ is well typed by *Lemma 2*.
Superposition	:	By *Lemma 3* $t\sigma$ and $t'\sigma$ have the common type τ. Thus by *Lemma 1* they can be used interchangeably. $C'\sigma$ and $D'\sigma$ are well typed by *Lemma 2*.

Lemma 5 If there exists a σ such that $t_1\sigma \approx t_2\sigma$ then there exists a (unique up to renaming) most general unifier.

The completeness of typed ground superposition now follows from the completeness proof for untyped superposition given in [3]. To show the lifting, we need to show for each inference that if σ is a grounding substitution, then every inference from $C\sigma$[5] is an ground instance of an inference of C. After that the model construction lemma and the remaining completeness proof can be reused without further changes.

Redundancy Criteria Since we reused most of the original completeness proof also the redundancy criteria holds. Thus all commonly used reductions can be also used in the typed superposition calculus.

7 Implementation

We have implemented a prototype of the above calculus from scratch in Scala. Unification is implemented with the help of unification contexts, with separate contexts for term variables and type variables. Additionally we implemented several of the reductions described in [14]; namely we have implemented Subsumption Deletion, Trivial Literal Elimination, Condensation, Unit Rewriting and Assignment Equality Deletion, as well as simple forms of Merging Replacement Resolution (MRR) and Tautology Elimination. We implement sharing, by keeping term and type variable numbers separate (and normalizing them), the number of distinct type terms is usually low. We use a worked-off/usable style main loop and fully reduce both sets. Indexing is implemented as a combination of a naive implementation of Substitution Trees[11] and Feature Vector Indexing[13].

Overall the implementation is reasonable, but not expected to be remotely competitive to an efficient C implementation featuring more advanced reduction techniques, such as full MRR

[5] And $D\sigma$ for superposition, where C and D do not share variables

and Splitting. A web interface is available from the abstract's web page.

7.1 Specifics of the Polymorphic+Typeclass implementation

In this section we describe the implementation necessary to specifically support Polymorphic+Typeclass superposition. We were able to keep the implementation of the inferences and reductions completely without any type-system related code, since the generic interfaces for indexing, unification and substitution take care of the type-system handling.

Start up We have extended the syntax, where the changes to the untyped settings are restricted to the additional information to be parsed. We also save all declarations, which are used for type-class unification and for assertions that ensure well-typedness throughout our prototype. CNF generation is also mostly unchanged. Only skolemization has to create adequate type declarations, which can be easily derived from the generated term.

Datastructures We use algebraic datastructures to encode literals, terms and type terms in the processed form described in the beginning of section 6. Clauses are unchanged by the type-system.

Assertions Our prototype is a relatively new piece of software which we use to experiment with. To ensure that the implementation does what we expect from it we employ Scala's assertions. We employ over 300 distinct assertions, checking various properties such as: Well typed-ness, correctness of unifiers, assumptions on term sharing. We also ensure more complex properties such as that clauses are not lost from our Worked Off / Usable sets by tracking all reductions and inferences and tracing them to the input clause set.

Sharing Sharing can be easily extended to also share type-terms. We use a hash map based sharing and literals, terms and type terms are are only created by the sharing.

Indexing For indexing we treat typed terms as untyped terms, the indexing "pretends" that type terms are just normal first-order terms. This yields all valid unifications and some unifications which are not well-typed, in particular since the indexing ignores type-classes. Therefore all unifications are verified before they are used. For Feature Vector indexing, the original untyped variant can also be used, and types can be treated as an additional feature. We have not noticed any further changes to be required. We are thus able to reuse existing datastructures almost unchanged, by adding post indexing unification checks, which verify the well-typedness.

Unification Unification is where most of the type-system related changes can be contained to. It is a straight forward implementation of the previously described unification algorithm. We have also implemented it to check/generate instantiations and renamings.

8 Evaluation

Setup We have evaluated our prototype on problems generated by Sledgehammer[4] from Isabelle/HOL theories, of areas as diverse as the fundamental theorem of algebra, the com-

pleteness of Hoare logic and the type soundness of a subset of Java[6]. The 579[7] problems were tested on 4 different type encodings, at 4 different fact sizes (16, 64, 256 and 1024 axioms) with 2 hours of runtime on one HyperThread of an 2xIntel Xeon E5620 with a JVM heap size of 1 GB.

The type encodings are:

Poly Tags	Encoding polymorphism into untyped first-order with the help of special typing function symbols (see [5, p. 9]). According to [7, p. 11,12] this is a more efficient encoding than guard predicates for polymorphism.
Monomorph	An incomplete encoding of polymorphism by picking relevant monomorphic instances (see [9]). According to [7, p. 11,12] currently the most efficient encoding.
Poly Native	Using native polymorphism but encoding type classes as predicates.
Type Classes	Using native polymorphism with its type classes.

Comparison to SPASS For reference we have compared the performance of our prototype at 256 axioms and Mono and Poly Tags encodings with the performance of a SPASS version specifically optimized for Isabelle/HOL[7]. SPASS is significantly faster than our prototype. Our tool can derive 75% (208 vs. 277) of SPASS's *Monomorph* proof count and 74% (162 vs. 219) of its *Poly Tags* proof count. Since our tool exhibits similar relative performance on the encodings that SPASS supports, we believe that our results will also carry over to efficient implementations.

Impact of the axioms The fewer axioms a problem has, the easier it is, since fewer axioms mean fewer clauses and thus fewer possible inferences. It is often possible to quickly find saturations for 16 axioms, but seldom for the bigger axiom sizes. The more axioms a problem has, the more likely it is that the relevant axioms for the proof are included. These observations are usually exploited by running a portfolio of different settings (slices). E.g. as the first slice with 16 axioms and a short timeout, followed by a larger axiom set since with longer timeout. Our evaluation will focus on a single slice, because here we are only interested in how well the different type encodings perform. In particular how well a type encoding handles additional axioms, e.g. how well it scales.

Scalability of type encodings Figure 1 shows the success rate and scalability of the tested encodings. Where the X axis denotes the number of axioms, and the Y axis the percentage of problems proved, in a single run of up to 2 hours. For all encodings, the most successful axiom set size tested is 256 axioms. The *Poly Tags* encoding performs worst. For all axiom sizes it solves fewer problems then the native encodings. Its performance also deteriorates the most when increasing the axiom set from 256 axioms to 1024, resulting in a drop of the success rate of 8 percentage points. The *Monomorph* and the *Type Classes* encoding both only suffer a drop of 3 percentage points.

[6] We evaluated on Arr, FFT, FTA, Hoa, Jin, Lam and NS see [4] for details

[7] Our original set had 694 problems, but 115 had to be excluded because they were missing typing information.

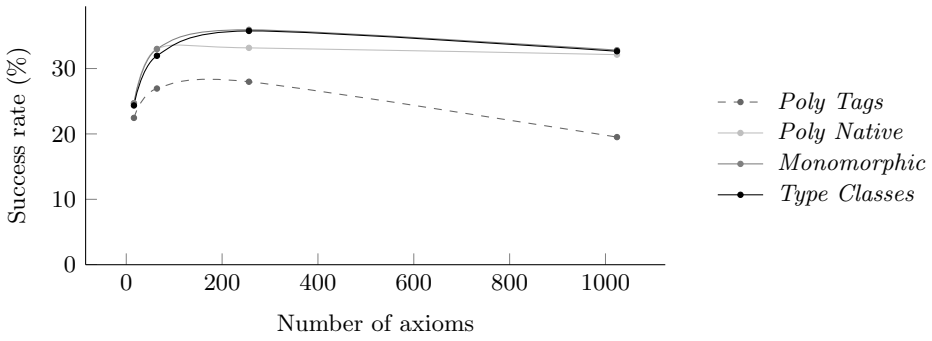


Figure 1: Success rate and scalability of each type encoding

Overall the *Monomorph* encoding performs best. It is closely followed by the *Type Classes* encoding which at worst (64 axioms) solves 96.8% of the problems that *Monomorph* solves. *Type Classes* and *Monomorph* are almost identical on the other axiom set sizes.

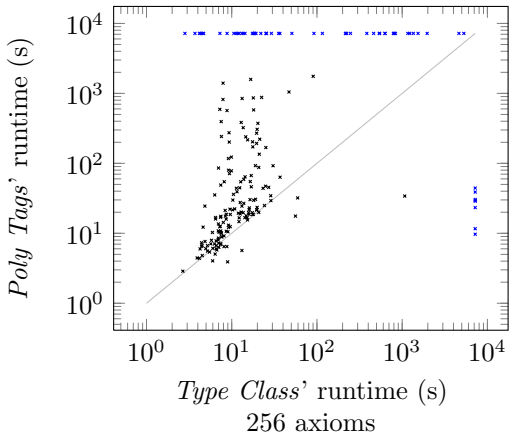
8.1 Detailed Evaluation

As can be seen in Fig. 1 the most successful axiom set size, for all encodings, is 256 axioms. Thus we will now focus on this to provide a more detailed evaluation of the encoding.
The following table provides the number of proofs an the average time it took to find them:

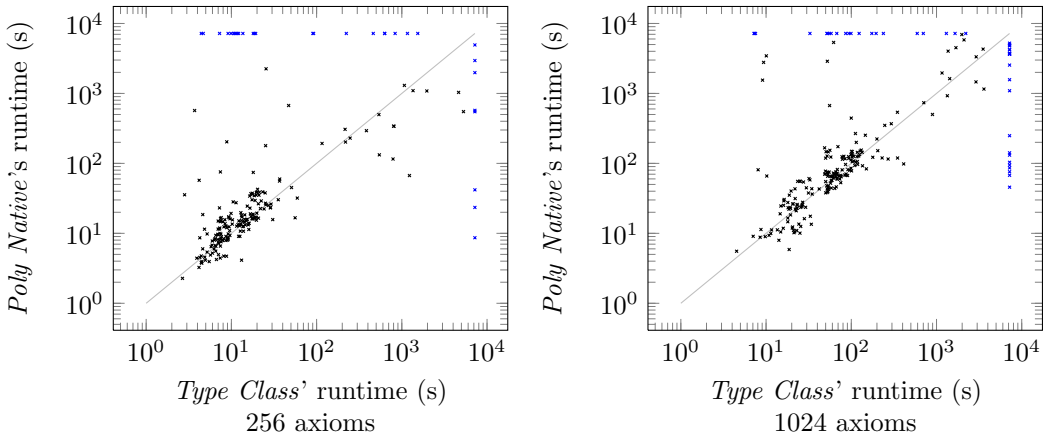
Encoding	Proofs	Average Time
Monomorph	208	384s
Type Classes	207	137s
Poly Native	192	133s
Poly Tags	162	115s

Poly Tags has the fewest proofs and even though it has the lowest average runtime it is actually the slowest if we compare it to the other encodings only on the problems both solve. Its solutions overlap with *Type classes* on 154 proofs. On those *Poly Tags*' average runtime is ∅119s, whereas *Type Classes*' average runtime is only ∅20s. The other encodings are also faster, the overlap with *Monomorph* is also 154 proofs, where *Poly Tags*' average runtime is ∅118s and *Monomorph*'s average runtime is ∅80s. The overlap with *Monomorph* is also 154 proofs, where *Poly Tags*' average runtime is ∅108s and *Poly Native*'s average runtime is ∅29s.

Below we show a scatter plot comparing the runtime of *Poly Tags* and *Type Classes*. The X axis is the logarithmic runtime of *Type Classes* and the Y axis the logarithmic runtime of *Poly Tags*. The results cluster in the upper-left half, again showing that *Type Classes* is faster.

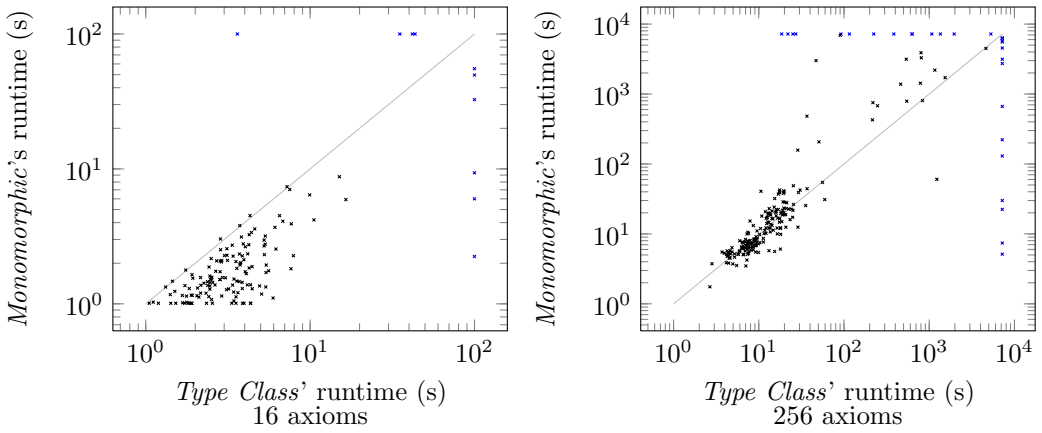


Poly Native has, similarly to *Poly Tags*, fewer proofs and a lower average runtime than *Type Classes* and *Monomorph*. *Poly Native* is actually faster, but less successful. Its solutions overlap with *Type classes* on 184 proofs. On those, *Poly Native*'s average runtime is ∅78s, whereas *Type Classes*' average runtime is ∅122s. The overlap with *Monomorph* is on 179 proofs, where *Poly Native*'s average runtime is ∅102s and *Monomorph*'s average runtime is ∅239s. Below we show again a scatter plot, comparing the runtime of *Poly Native* and *Type Classes*. Note that the average runtime is dominated by the long running proofs in the upper right corner. For the faster proofs (up to 100s runtime) *Type Classes* is actually slightly faster. We believe that this shows that on some problems the possible advantages of native type classes do not compensate their overhead during unification. The advantages start to show on the larger (e.g. 1024) axiom sizes. Here *Type Classes*' average runtime is ∅195s and *Poly Native*'s average runtime is ∅248s on 193 proofs they both find.



Monomorph* and *Type Classes find a similar amount of proofs with *Type Classes* having a lower average total runtime. They overlap on 193 proofs. On those *Type Classes* has an average runtime of ∅85s and *Monomorph* an average runtime of ∅198s. It is also faster on axiom set sizes of 1024. For an axiom set size of 16, *Monomorph* is almost uniformly faster. The overlap for 16 axioms is 140 proofs, where *Monomorph* has an average runtime of ∅2s and

Type Classes of ∅4s. We believe that in an efficient implementation both most of the runtimes would be well below 1s (for a size of 16 axioms).



9 Conclusion

We have presented a typed form of superposition with rank-1 polymorphism and type classes. We have given its semantic; adapted substitution, unification and the commonly used KBO ordering to our new setting. We have also presented an initial preprocessing step for functions and predicates, that simplifies unification and lets us easily reuse existing indexing techniques. We have implemented the calculus in a prototype and first results are encouraging. Even though polymorphic encodings solve a harder problem, our evaluation shows, that they are competitive with the monomorphic encoding, which itself is a highly efficient encoding. With increasing number of axioms, the native polymorphic+typeclass runtime behavior performs best. We plan to use the typed calculus as a basis to bring further features of interactive theorem proving to the superposition calculus.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[2] M. Baaz, U. Egly, and A. Leitsch. Normal form transformations. In Robinson and Voronkov [12], pages 273–333.

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

[4] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with smt solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013.

[5] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *LNCS*, pages 493–507. Springer, 2013.

[6] J. C. Blanchette and A. Paskevich. Tff1: The tptp typed first-order form with rank-1 polymorphism. In M. P. Bonacina, editor, *CADE*, volume 7898 of *LNCS*, pages 414–420. Springer, 2013.

[7] J. C. Blanchette, A. Popescu, D. Wand, and C. Weidenbach. More spass with isabelle. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 345–360. Springer Berlin Heidelberg, 2012.

[8] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.

[9] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *LNCS*, pages 87–102. Springer, 2011.

[10] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 263–278. Springer, 2007.

[11] P. Graf. Substitution tree indexing. In J. Hsiang, editor, *Rewriting Techniques and Applications*, volume 914 of *LNCS*, pages 117–131. Springer Berlin Heidelberg, 1995.

[12] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[13] S. Schulz. Simple and efficient clause subsumption with feature vector indexing. In M. Bonacina and M. Stickel, editors, *Automated Reasoning and Mathematics*, volume 7788 of *LNCS*, pages 45–67. Springer Berlin Heidelberg, 2013.

[14] C. Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [12], pages 1965–2013.

[15] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. P. Felty, editors, *TPHOLs*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997.