EPiC Series in Computing

Volume 94, 2023, Pages 36–47

Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

EPiC Computing

Refining Unification with Abstraction

Ahmed Bhayat[2], Konstantin Korovin[2], Laura Kovács[1], and Johannes Schoisswohl[1]

[1] TU Wien, Vienna, Austria
[2] University of Manchester, Manchester, UK

Abstract

Automated reasoning with theories and quantifiers is a common demand in formal methods. A major challenge that arises in this respect comes with rewriting/simplifying terms that are equal with respect to a background first-order theory $\mathcal{T}$, as equality reasoning in this context requires unification modulo $\mathcal{T}$. We introduce a refined algorithm for unification with abstraction in $\mathcal{T}$, allowing for a fine-grained control of equality constraints and substitutions introduced by standard unification with abstraction approaches. We experimentally show the benefit of our approach within first-order linear rational arithmetic.

1 Introduction

The two most prominent approaches supporting automated reasoning with theories and quantifiers are SMT solving [9, 13, 4] and saturation-based first-order proving [27, 19, 12, 25]. While SMT solvers provide strong theory reasoning, the strength of first-order proving comes with complete quantifier reasoning. *In this paper, we focus on the latter and aim at strengthening first-order proving with built-in theory reasoning, complementing efficient SMT solving.*

State-of-the-art. Most saturation-based provers implement the superposition calculus for first-order logic with equality [2, 20]. This calculus heavily relies on unification algorithms for processing quantified formulas. For two terms s, t, unification computes the most general substitution $\sigma = \mathbf{mgu}(s, t)$ such that $s\sigma = t\sigma$. The $\mathbf{mgu}(s, t)$ is used to apply the inference rules of the superposition calculus to formulas containing s, t. Extending first-order proving with reasoning modulo a background theory $\mathcal{T}$ requires extending the superposition calculus with *unification modulo* $\mathcal{T}$; that is, finding substitutions σ such that $\mathcal{T} \models s\sigma \approx t\sigma$. However, unlike the uniqueness of $\mathbf{mgu}(s, t)$ in standard first-order logic, most common theories $\mathcal{T}$, such as linear rational arithmetic $\mathbb{Q}$, do not admit a single most general unifier modulo $\mathcal{T}$, but yield a complete set of unifiers modulo $\mathcal{T}$, in short $\mathbf{csu}_{\mathcal{T}}(s, t)$. Moreover, $\mathbf{csu}_{\mathcal{T}}(s, t)$ can be very large, shown, for example, to be doubly exponential in the case of $\mathcal{T}$ with associativity and commutativity (**AC**) [11] or even infinite in the case of higher-order unification [17].

In order to bypass such inefficiencies of unification modulo $\mathcal{T}$ and effectively handle the large set of unifiers $\mathbf{csu}_{\mathcal{T}}(s, t)$, *unification with abstraction (UWA)* has been introduced in [22]. UWA applies usual unification up to the point where two terms s, t might have more than one unifier modulo $\mathcal{T}$; then, instead of computing the entire $\mathbf{csu}_{\mathcal{T}}(s, t)$, UWA introduces constraints $s_i \not\approx t_i$

that capture $\mathbf{csu}_{\mathcal{T}}(s,t)$. Arguably, UWA is a lazy form of full abstraction, with the latter being shown to be complete for some calculi in [26]. Recently in [18], we extended UWA to support refutationally complete reasoning using superposition within linear rational arithmetic.

The two main advantages of UWA over unification modulo $\mathcal{T}$ are the following. First, UWA comes with uniqueness: there is always one single abstracting unifier computed by UWA instead of large sets $\mathbf{csu}_{\mathcal{T}}(s,t)$. Second, UWA brings simplicity: we do not need complex checks of unifiability modulo $\mathcal{T}$, but we can introduce an over-approximating constraints and let the inference system reason with this constraint.

Example 1. *Consider unifying the terms $s = f(x+y)$ and $t = f(t_1 + \cdots + t_n)$ modulo a theory with* $\mathbf{AC}$*. Unification modulo* $\mathbf{AC}$ *introduces the exponential set of unifiers* $\mathbf{csu}_{\mathcal{T}}(s,t) = \big\{\{x \mapsto \sum_{i\in I} t_i, y \mapsto \sum_{i\notin I} t_i\} \mid I \subseteq \{1,\dots,n\}\big\}$*, whereas UWA returns the single (abstracting unifier) constraint $x + y \not\approx t_1 + \ldots + t_n$.*

UWA Benefits and Limitations. Unlike full abstraction, UWA in [18] controls the application of abstracting unifiers using a so-called *abstraction predicate* **canAbstract**. That is, UWA skips abstraction in cases where terms can never be equal in the background theory $\mathcal{T}$. For example, UWA avoids unifying $f(a)$ and $f(a+b)$, whereas full abstraction introduces a constraint $a \not\approx a + b$. Yet, despite an abstraction predicate providing more fine grained control than full abstraction, UWA still lacks efficiencies, as illustrated next.

Example 2. *Consider the application of the factoring rule to clause $P(4 + x) \vee P(3x)$, for removing duplicate literals. When computing an abstracting unifier using [18], we introduce a constraint $3x \not\approx 4+x$ deriving the clause $P(4+x) \vee 3x \not\approx 4+x$. In the ordered resolution setting, the literal $P(4 + x)$ is still maximal and hence will be prioritized and resolved with all literals $\neg P(t)$, growing the search space with unnecessary consequences, even though there is only one substitution making $3x$ and $4 + x$ equal, namely $x \mapsto 2$.*

Our Contributions. To improve UWA, in this paper we refine the approach of [18] in order to allow immediately computing substitutions in cases where this can be done in a "cheap" manner. To do so, we replace the abstraction predicate **canAbstract** used in [18, 22] by a so-called abstraction oracle **abstr** that gives more generic information about the unifiability of two terms (Sect. 3). With such an abstraction oracle within Example 2, our refined UWA does not introduce the constraint $3x \not\approx 4 + x$, but computes the substitution $\{x \mapsto 2\}$. Our refined UWA with abstraction oracles also supports, for example, finding the substitution $\{x \mapsto a, y \mapsto a, z \mapsto b\}$ when unifying $2f(a) + g(z)$ and $f(y) + f(x) + g(b)$, or failing unification for the terms $g(x + a, f(x + 1))$ and $g(a, f(0))$; such and similar cases cannot be handled by the abstraction predicates of [18, 22].

The first main contribution of this paper is that, given an abstraction oracle fulfilling certain conditions (Def. 4), we prove that the abstracting unifier computed by our refined UWA algorithm (Alg. 1) ensures soundness and refutational completeness. As our second main contribution, we complement [18] with a thorough experimental analysis. To this end, we implement our refined UWA approach in the Vampire prover [19] and showcase that UWA with our abstraction oracle brings improvements upon [18] (Sect. 4).

2 Preliminaries

We assume familiarity with multi-sorted first-order logic with equality and saturation-based theorem proving. For details we refer to [2].

Syntax. We consider a signature with function symbols $\mathbf{F}$, predicate symbols $\mathbf{P}$, variables $\mathbf{V}$, and sorts $\mathbf{S}$. For a term $t = f(t_1, \ldots, t_n)$ we write $\mathbf{sym}(t)$ for f, and call t_i its term arguments. In the same way we define arguments and $\mathbf{sym}$ for literals. By $\approx$, and $\not\approx$ we denote the positive, and negative equality predicate. We distinguish a sort $\tau_{\mathbb{Q}}$, the sort of rationals, with a binary function symbol $+$, unary function symbols k for $k \in \mathbb{Q}$, and a constant symbol 1. We call $k \in \mathbb{Q}$ numeral multiplications, where $k(t)$ has the intended semantics of multiplying t by k. We omit parenthesis for numeral multiplications, and write $-$ for -1, and k for $k(1)$. Further we write the symbol $+$ in infix notation and omit parenthesis, and do not distinguish between terms where the arguments of $+$ are permuted. We call a term t atomic if it is a variable or if $\mathbf{sym}(t) \notin \mathbb{Q} \cup \{+\}$. We call a term $\mathbb{Q}$-normalized if all its subterms are $\mathbb{Q}$-normalized; if it is of sort $\tau_{\mathbb{Q}}$, then it is of the form $k_1 t_1 + \cdots k_n t_n$ such that all for all i, $k_i \neq 0$, and for $j \neq i$, we have that $t_i \neq t_j$ and all summands t_i are atomic and sorted with respect to some arbitrary but fixed total ordering on terms. For a $\mathbb{Q}$-normalized term $k_1 t_1 + \cdots + k_n t_n$ we call t_i its top level terms. For sets of pairs of terms S, we write $S^{\approx}$ for $\bigwedge_{\langle s,t\rangle \in S} s \approx t$, and $S^{\not\approx}$ for $\bigvee_{\langle s,t\rangle \in S} s \not\approx t$. We use s, t for terms, f, g for function symbols, x,y,z for variables, P, Q for predicates, L for literals, j, k for numeral multiplications, and C, D for clauses, all possibly with indices. We write $\trianglelefteq$ for the subterm relation, and $\triangleleft$ for the strict subterm relation.

Semantics. Let ϕ be a formula, Φ be a set of formulas, and $\mathcal{I}$ an interpretation. We write $\mathcal{I} \models \phi$ for $\mathcal{I}$ being a model of ϕ. We write $\Phi \models \phi$, if every model of Φ is a model of ϕ. A theory $\mathcal{T}$ is a set of formulas, which we associate with the class of all models of $\mathcal{T}$. We write $s \equiv_{\mathcal{T}} t$ for $\mathcal{T} \models s \approx t$, and leave away $\mathcal{T}$ if it is clear in the context. A $f \in \mathbf{F} \cup \mathbf{P}$ is called uninterpreted wrt. a theory $\mathcal{T}$ if whenever $\mathcal{T} \models f(s_1 \ldots s_n) \approx f(t_1 \ldots t_n)$, then $\mathcal{T} \models s_1 \approx t_1 \wedge \ldots \wedge s_n \approx t_n$. We call an f interpreted if it is not uninterpreted. We say t occurs as uninterpreted (similarly interpreted) argument in a term/literal/clause L iff $f(t_1 \ldots t_n) \trianglelefteq L$, and $t_i = t$ for some i, and f is uninterpreted. We say s occurs in an uninterpreted position of t, if $s \trianglelefteq t$, and every function symbol on the path to s in t is uninterpreted.

Substitutions. We write $\{x_1 \mapsto t_1 \ldots x_n \mapsto t_n\}$ for a substitution σ such that $\forall i.\sigma(x_i) = t_i$. We extend substitutions to be applied to terms, literals, and clauses in the standard way. A substitution θ is called a grounding of a term/literal/clause t if $t\theta$ is ground. We usually use σ, μ for substitutions, and θ for groundings. For a theory $\mathcal{T}$ we write $\sigma \equiv_{\mathcal{T}} \sigma'$ to denote that for every $x \in \mathbf{V}$ we have $x\sigma \equiv_{\mathcal{T}} x\sigma'$; we call σ a $\mathcal{T}$-unifier of s, t, if $\mathcal{T} \models s\sigma \approx t\sigma$. If $\mathcal{T} = \emptyset$, we call σ a syntactic unifier. A complete set of unifiers $\mathbf{csu}_{\mathcal{T}}(s, t)$ is a set of $\mathcal{T}$-unifiers of s and t such that for every unifier σ there is a μ and a $\sigma' \in \mathbf{csu}_{\mathcal{T}}(s, t)$ such that $\sigma \equiv_{\mathcal{T}} \sigma'\mu$. We say two terms are $\mathcal{T}$-unifiable if $\mathbf{csu}_{\mathcal{T}}(s, t) \neq \emptyset$. For syntactic unification there is a unique most general unifier $\mathbf{mgu}(s, t)$ for unifiable terms s, t. The terms s, t are trivially unifiable if either of them is a variable and not subterm of the other one.

Term Orderings. A term ordering $\prec$ is a relation on terms such that (i) $\prec$ is stable under substitutions; that is $s \prec t$ implies $s\sigma \prec t\sigma$ for any σ; (ii) $\prec$ is a total, well-founded ordering on ground terms. A term ordering $\prec$ is compatible with a theory $\mathcal{T}$, or simply $\mathcal{T}$-compatible, if for all terms s, s', t, t' we have that $s \equiv_{\mathcal{T}} s'$, $t \equiv_{\mathcal{T}} t'$ and $s \prec t$ implies $s' \prec t'$. We write $s \preceq t$ to denote $s \prec t$ or $s = t$. Literal and clause orderings are defined in the same manner. We say a $\mathcal{T}$-compatible literal/clause ordering has the uninterpreted argument property if $s \not\approx t \prec L$ whenever $s \equiv_{\mathcal{T}} t$ and s or t occurs as uninterpreted argument in L.

3 Refined Unification with Abstraction

3.1 Abstacting Unifiers

In contrast to unification modulo $\mathcal{T}$, UWA does not compute sets of substitutions $\mathbf{csu}_{\mathcal{T}}(s,t)$ but derives so-called abstracting unifiers $\mathbf{uwa}(s,t)$ as defined below.

Definition 1 (Abstracting Unifier)**.** *A function* **uwa** *that maps two terms either to* $\perp$ *or to a pair* $\langle\sigma,\mathcal{C}\rangle$*, where* σ *is a substitution and* $\mathcal{C}$ *is a clause, is called an* abstracting unifier*.*

Intuitively, abstracting unifiers ensure that, if none of the constraints $\mathcal{C}$ is violated, then $s\sigma \equiv_{\mathcal{T}} t\sigma$. For the terms $g(a+b)$ and $g(y+x+z)$, potential abstracting unifiers are $\langle\{x \mapsto a, y \mapsto b, z \mapsto 0\}, \emptyset\rangle$, $\langle\{z \mapsto 0\}, \{y+x \not\approx a+b\}\rangle$ and $\langle\emptyset, \{y+x+z \not\approx a+b\}\rangle$.

Recall that unification modulo $\mathcal{T}$ uses a *complete* sets of $\mathcal{T}$-unifiers. Similarly for UWA, we impose the following properties over abstracting unifiers for ensuring sound and complete reasoning with them.

Definition 2. *Let* **uwa** *be an abstracting unifier and* $s,t \in \mathbf{T}$*. Consider an arbitrary grounding* θ*. If* $\mathbf{uwa}(s,t) = \langle\sigma,\mathcal{C}\rangle$*, we define* **uwa** *to be*

- $\mathcal{T}$-sound *iff* $\mathcal{T} \models (s \approx t)\sigma \vee \mathcal{C}$*;*
- $\mathcal{T}$-general *iff* $\mathcal{T} \models s\theta \approx t\theta \Rightarrow \exists\theta'.\sigma\theta' \equiv_{\mathcal{T}} \theta$*;*
- $\mathcal{T}$-minimal *iff* $\mathcal{T} \models (s \approx t)\sigma\theta \Rightarrow \mathcal{T} \vDash \neg\mathcal{C}\theta$*;*
- subterm-founded *with respect to the clause ordering* $\prec$ *iff whenever* $s\theta \equiv_{\mathcal{T}} t\theta$ *and* s,t *occur as uninterpreted argument in some literals* L_s, L_t *respectively, it holds that* $\mathcal{C}\theta \prec L_s\theta$ *or* $\mathcal{C}\theta \prec L_t\theta$*.*

Further, **uwa** *is* $\mathcal{T}$-complete *if, for all* $s,t \in \mathbf{T}$ *with* $\mathbf{uwa}(s,t) = \perp$*, we have* $\mathbf{csu}_{\mathcal{T}}(s,t) = \emptyset$*.*

We remark that $\mathcal{T}$-generality ensures that a substitution σ introduced by $\mathbf{uwa}(s,t)$ can be turned into any ground $\mathcal{T}$-unifier of s,t. $\mathcal{T}$-minimality guarantees that an inference with $\mathbf{uwa}(s,t)$ is equivalent to an inference using unification modulo $\mathcal{T}$. Subterm-foundedness is necessary to keep the calculus reductive, and $\mathcal{T}$-completeness ensures that **uwa** returns an abstracting unifier when s,t are unifiable. As such, an abstracting unifier satisfying the properties of Def. 2 can be used to replace unification modulo $\mathcal{T}$ in compatible calculi [18]. For example, the resolution factoring rule

$$\frac{C \vee P \vee P'}{(C \vee P)\sigma} \text{ where } \sigma \in \mathbf{csu}_{\mathcal{T}}(P,P') \quad \text{becomes} \quad \frac{C \vee P \vee P'}{(C \vee P)\sigma \vee \mathcal{C}} \text{ where } \langle\sigma,\mathcal{C}\rangle = \mathbf{uwa}(P,P')$$

.

Note that with unification modulo $\mathcal{T}$, the factoring rule is applied with every unifier $\sigma \in \mathbf{csu}_{\mathcal{T}}(P,P')$. In contrast, when using UWA, we apply factoring for *only one* unique abstracting unifier $\langle\sigma,\mathcal{C}\rangle = \mathbf{uwa}(P,P')$ and additionally introduce constraint literals $\mathcal{C}$, which are usually negative equality literals.

3.2 UWA with Abstraction Oracles

In [18], abstracting unifiers are computed using an *abstraction predicate*, denoted as **canAbstract**, allowing to only introduce a constraint $\mathcal{C}$ whenever two terms s,t unify in the background theory $\mathcal{T}$. We now refine this approach of [18] by using an *abstraction oracle* instead of an abstraction predicate. Our *abstraction oracle* acts as a tailored theory solver that either (i) solves the unification problem in simple cases or (ii) introduces constraints $\mathcal{C}$ if it fails to solve the problem or there is more than one **mgu**.

```
fn uwa(s,t)
    unproc ← {s ∼ t}; σ ← ∅; C ← ∅;
    while unproc ≠ ∅
        s′ ∼ t′ ← unproc.pop()σ;
        if s′ ∼ t′ ∈ {x ∼ u, u ∼ x} for x ∈ V, !occurs(x, u)
            σ ← σ ∪ {x ↦ u};
        else
            match abstr(s′, t′)
                case ⊥:  return ⊥;
                case ⟨unif, constr⟩:
                    C.push(constr);
                    unproc.push(unif);
                case undefined:
                    if s′ = f(s_1 … s_n), t′ = f(t_1 … t_n)
                        unproc.push({s_1 ∼ t_1 … s_n ∼ t_n})
                    else
                        return ⊥
    return abstracting unifier ⟨σ, (C^≉)σ⟩;
```

Algorithm 1: Refined UWA with the abstraction oracle **abstr** over terms s, t

Definition 3 (Abstraction Oracle)**.** *An* abstraction oracle *is a partial function* **abstr** *that maps terms to either* $\bot$ *or to a pair* $\langle \mathsf{unif}, \mathsf{constr}\rangle$*, where both* unif *and* constr *are sets of pairs of terms. For pairs* $\langle s, t\rangle \in \mathsf{unif} \cup \mathsf{constr}$ *we also write* $s \sim t$*.*

Alg. 1 summarizes our refined UWA algorithm using an abstraction oracle **abstr** to compute an abstracting unifier. Alg. 1 modifies standard unification algorithms [24], as follows. Upon failing to bind a variable x to a term, Alg. 1 queries the abstraction oracle **abstr** to proceed with the unification of s', t'. If $\mathbf{abstr}(s, t)$ is undefined, Alg. 1 proceeds as in [24]. Otherwise, Alg. 1 either returns early failing unification, or uses further information on which subterms to continue unification with (by adding unif to *unproc*) and which constraints to introduce (by adding constr to $\mathcal{C}$). We note that Alg. 1 computes the so-called *triangle form* [15] of the actual unifier σ, which means that the union in the first branch is equivalent to composition, but can be implemented as a constant-time operation. The following properties allow us to ensure in Lem. 2 that Alg. 1 computes an abstracting unifier fulfilling the properties in Def. 2.

Definition 4. *Let* **abstr** *be an abstraction oracle and* $s, t \in \mathbf{T}$ *not trivially unifiable. Consider a* $\mathcal{T}$*-compatible clause ordering* $\prec$*, a literal* L *and an arbitrary ground substitution* θ*. We define* **abstr** *to be*

- $\mathcal{T}$-sound *iff* $\begin{cases} \mathbf{abstr}(s,t) = \langle \mathsf{unif}, \mathsf{constr}\rangle & \Rightarrow \mathcal{T} \models (\mathsf{unif} \cup \mathsf{constr})^{\approx} \rightarrow (s \approx t) \\ \mathbf{abstr}(s,t) = \bot & \Rightarrow \forall \theta. \mathcal{T} \not\models (s \approx t)\theta \end{cases}$
- $\mathcal{T}$-minimal *iff* $\mathbf{abstr}(s,t) = \langle \mathsf{unif}, \mathsf{constr}\rangle \Rightarrow \forall \theta. \Big(s\theta \equiv_{\mathcal{T}} t\theta \Rightarrow \mathcal{T} \models (\mathsf{constr} \cup \mathsf{unif})^{\approx}\theta \Big)$
- $\prec$-founded *iff* $\mathbf{abstr}(s,t) = \langle \mathsf{unif}, \mathsf{constr}\rangle \Rightarrow \forall \theta. \Big(s\theta \equiv_{\mathcal{T}} t\theta \Rightarrow (\mathsf{unif} \cup \mathsf{constr})^{\not\approx}\theta \preceq (s \not\approx t)\theta \Big)$
- terminating *iff there is a well-founded relation* $\ll_{\mathbf{abstr}}$ *such that for all* s, t *we have*

$$\begin{pmatrix} \mathbf{abstr}(s,t) = \langle \mathsf{unif}, \mathsf{constr}\rangle \\ \&\ s' \sim t' \in \mathsf{unif} \end{pmatrix} \Rightarrow \begin{pmatrix} s' \sim t' \ll_{\mathbf{abstr}} s \sim t,\ \mathbf{abstr}(s', t') \textit{ is undefined} \\ \textit{or } s' \textit{ and } t' \textit{ are trivally unifiable} \end{pmatrix}$$

- captures $\mathcal{T}$ *iff whenever* $\mathbf{abstr}(s,t)$ *is undefined, then either* $\mathbf{csu}_{\mathcal{T}}(s,t) = \emptyset$ *or* $\mathbf{sym}(s) = \mathbf{sym}(t)$ *is uninterpreted.*

We remark that the property of $\prec$-founded ensures that Alg. 1 does not introduce constraints that are bigger than the initial terms s, t, implying that the computed abstracting unifier is subterm-founded. Termination of **abstr** is necessary for Alg. 1 to terminate; otherwise, for example, we may define define $\mathbf{abstr}(s,t) = \langle\{s-1 \sim t+1\}, \emptyset\rangle$, which in turn yields an infinite loop in Alg. 1. Further, **abstr** needs to capture $\mathcal{T}$ so that the abstration oracle handles all terms that cannot be treated uninterpreted.

Let us now show that given an abstraction oracle fulfilling all the properties for Def. 5, the abstracting unifier computed by Alg. 1 will fulfil all the desired properties from Def. 2 needed to lift a compatible inference system. In order to do this we first will need a set of invariants our algorithm fulfils, given the relevant properties of the abstraction oracle. The invariants will then entail the desired properties of the computed abstracting unifier. Figure 1 illustrates how the properties of the abstraction oracle, the invariants of the algorithm and the properties of the abstracting unifier are related.

Lemma 1. *Consider the following invariants.*

(I1) $\mathcal{T} \models (\mathsf{unproc} \cup \mathcal{C})^{\approx}\sigma \rightarrow (s \approx t)\sigma$

(I2) $\forall\theta.\big(s\sigma\theta \equiv_{\mathcal{T}} t\sigma\theta \Rightarrow (\mathsf{unproc} \cup \mathcal{C})^{\not\approx}\sigma\theta \preceq (s \not\approx t)\sigma\theta\big)$

(I3) $\forall\theta.\big(s\sigma\theta \equiv_{\mathcal{T}} t\sigma\theta \Rightarrow \mathcal{T} \models (\mathsf{unproc} \cup \mathcal{C})^{\approx}\sigma\theta\big)$

(I4) $\forall\theta.(s\theta \equiv_{\mathcal{T}} t\theta \Rightarrow \exists\rho.\theta \equiv_{\mathcal{T}} \sigma\rho)$

If **abstr** *is sound, the loop in Algorithm 1 fulfils the invariant (I1), it $\prec$-founded (I2) is fulfilled, and if is $\mathcal{T}$-minimal and captures $\mathcal{T}$, the invariants (I3), and (I4) hold.*

Proof. It can easily be seen that all invariants hold at the start of the loop. Note that for none of the invariants we have to check the case where $\mathbf{abstr}(s',t') = \bot$, or the else branch after the case where $\mathbf{abstr}(s',t')$ is undefined, as both of these branches end with an early return, hence the end of the loop is not reached.

(I1) After the first if branch the invariant holds as $x \approx u \rightarrow \phi[x]$ is equivalent to $\phi[u]$ for any first-order formula ϕ.

After the case branch where $\mathbf{abstr}(s',t') = \langle\mathsf{unif}, \mathsf{constr}\rangle$, the invariant is holds, as **abstr** is sound, which means that unif, and constr together imply $s' \approx t'$, which was in $\mathsf{unproc}\sigma$ at the start of the loop. When $\mathbf{abstr}(s',t')$ is undefined, and $\mathbf{sym}(s') = \mathbf{sym}(t') = f$, the invariant holds due to function congruence.

(I2) In the case-branch where $\mathbf{abstr}(s',t') = \langle\mathsf{unif}, \mathsf{constr}\rangle$, as **abstr** is $\prec$-founded, the invariant will by definition hold after the loop. In all other cases, it is obvious to see that the invariant holds at the end of the loop.

(I3) Let's first consider the first if-branch. Let θ be arbitrary, such that $\mathcal{T} \models (s \approx t)\sigma\theta$. As the invariant holds at the start of the loop, we know that $\mathcal{T} \models (x \approx u)\theta$. Further, again due to the invariant we know that $\mathcal{T} \models (\mathsf{unproc} \cup \mathcal{C})^{\approx}\sigma\theta$ at the end of the branch, from which we can conclude $\mathcal{T} \models (\mathsf{unproc} \cup \mathcal{C})^{\approx}\sigma\{x \mapsto u\}\theta$ holds, which means the invariant holds after the branch.

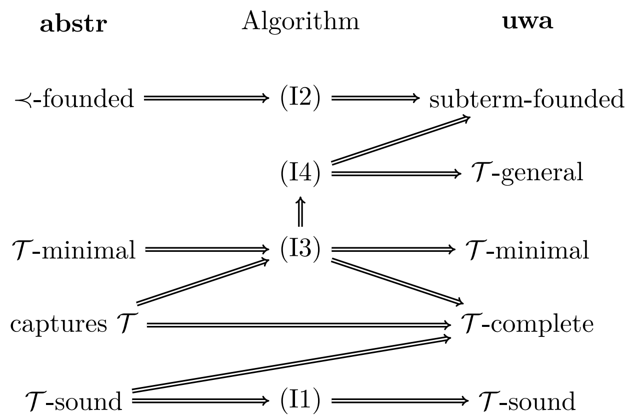

Figure 1: Implication structure between properties of **canAbstract**, and the abstracting unifier computed by Algorithm 1.

In the case branch where $\langle \mathsf{unif}, \mathsf{constr} \rangle = \mathbf{abstr}(s', t')$, the invariant obviously holds due to $\mathcal{T}$-minimality of **abstr**.

In the case where $\mathbf{abstr}(s', t')$ is undefined, and $\mathbf{sym}(s') = \mathbf{sym}(t') = f$, we know that as **abstr** captures $\mathcal{T}$, that either t is uninterpreted, or s', and t' do not unify. In case they are uninterpreted, this means by definition of uninterpreted functions that the invariant will hold after the loop iteration. In case s' and t' do not unify, this further means by using the invariant contrapositively, that s, and t do not unify, which means that there is no θ such that $\mathcal{T} \models s\theta \approx t\theta$, hence the property will hold after the loop iteration as well.

(I4) Let θ be arbitrary such that $\mathcal{T} \models (s \approx t)\sigma\theta$ Again we consider the if-branch first. Due to invariant (I3), we know that $\mathcal{T} \models (x \approx u)\theta$. This further means that $\{x \mapsto u\}\theta \equiv_{\mathcal{T}} \theta$, hence the invariant must hold after this branch. In all other branhes σ does not change which means that the invariant is preserved.

□

Lemma 2 (Soundness & Completeness)**.** *Let* **abstr** *be an abstraction oracle and* **uwa** *an abstracting unifier computed by Alg. 1. Let ≺ be a $\mathcal{T}$-compatible clause ordering. If* **abstr** *is $\mathcal{T}$-sound then* **uwa** *is sound. Further, if* **abstr** *is $\mathcal{T}$-sound, $\mathcal{T}$-minimal, ≺-founded, terminating and captures $\mathcal{T}$, then* **uwa** *is subterm-founded, minimal, general and complete.*

Proof. Note that if the algorithm returns $\langle \sigma, \mathcal{C} \rangle$, then unproc is empty. Therefore it is easy to see that (I1) implies, $\mathcal{T}$-soundness, (I2) and (I4) together imply subterm-foundedness, (I3) implies $\mathcal{T}$-minimality, and (I4) implies generality.

Further let's consider the cases where the algorithm returns $\perp$. In the first early return, where $\mathbf{abstr}(s', t') = \perp$, we know by soundness of **abstr** that $\mathbf{csu}_{\mathcal{T}}(s', t') = \emptyset$, which means by (I3) that $\mathbf{csu}_{\mathcal{T}}(s, t)$ must be empty as well. In the second early return we know as **abstr** captures $\mathcal{T}$, that $\mathbf{csu}_{\mathcal{T}}(s', t')$ is empty, which again means due to (I3) that $\mathbf{csu}_{\mathcal{T}}(s, t)$ is empty as well.

□

Similarly to [18], Lem. 2 implies that Alg. 1 can be used to lift a **uwa**-compatible calculi in a sound and complete way whenever the abstraction oracle **abstr** satisfies the properties of Def. 4. In Sect. 3.3 we give a concrete instance of such an **abstr** oracle (Def. 5), used further in our experiments (Sect. 4).

3.3 An Abstraction Oracle for Refined UWA in $\mathbb{Q}$

We now present a concrete instance of Alg. 1 for the theory $\mathbb{Q}$ of linear rational arithmetic. For this theory, the ALASCA calculus of [18] proposes unification modulo $\mathcal{A}^{\mathbb{Q}}_{eq}$, using a partial axiomatisation of arithmetic equalities and the QKBO ordering. Our abstraction oracle $\mathbf{abstr}_{\mathbb{Q}}$ for this calculus is given in Def. 5.

Definition 5. *Let $l, r \in \mathbf{T}$. If $\mathbf{sym}(l) = \mathbf{sym}(r)$ is interpreted, $\mathbf{abstr}_{\mathbb{Q}}(l,r)$ is undefined. If l and r are not of rational sort, $\mathbf{abstr}_{\mathbb{Q}}$ is only defined if $\langle x, u\rangle \in \{\langle l,r\rangle, \langle r,l\rangle\}$ such that $x \in \mathbf{V}$, $x \triangleleft u$, and x does not occur in an uninterpreted positions of u; then, $\mathbf{abstr}_{\mathbb{Q}}(x, t[x]) = \langle \emptyset, x \sim t[x]\rangle$. Otherwise, let $t = \sum k_i t_i$ be a $\mathbb{Q}$-normalized form of $l - r$ and $\mathbf{abstr}_{\mathbb{Q}}(l,r) = \mathbf{abstr}_{\mathbb{Q}}(t)$ where*

$$\mathbf{abstr}_{\mathbb{Q}}(t) = \begin{cases} \langle \{x \sim \frac{1}{k}t'\}, \emptyset\rangle & \textit{if there is an } x \in \mathbf{V} \textit{ s.t. } t = kx + t', \textit{ and } t' \textit{ does not contain } x \quad (\alpha_1) \\ \bot & \textit{else if there is an } x \in \mathbf{V}, \textit{ s.t. } t = kx + t'[x], \textit{ and } t' \textit{ contains } x \textit{ in an uninterpreted position} \quad (\alpha_2) \\ \langle \emptyset, \{x \sim \frac{1}{k}t'\}\rangle & \textit{else if there is an } x \in \mathbf{V} \textit{ such that } t = kx + t'[x] \quad (\alpha_3) \\ \bot & \textit{else if } \bot = \mathbf{abstr}_{\mathbb{Q}}(\mathit{split}_f(t)) \textit{ for some } f \in \mathbf{F} \quad (\beta_1) \\ \langle \bigcup_{f\in\mathbf{F}} \mathsf{unif}_f, \bigcup_{f\in\mathbf{F}} \mathsf{constr}_f\rangle & \textit{else where } \langle \mathsf{unif}_f, \mathsf{constr}_f\rangle = \mathbf{abstr}_{\mathbb{Q}}(\mathit{split}_f(t)). \quad (\beta_2) \\ \bot & \textit{else if } \sum k_i \neq 0, \textit{ and } t = \sum k_i t_i. \quad (\gamma_1) \\ \langle \{t_1 \sim t_2\}, \emptyset\rangle & \textit{else if } t = kt_1 + -kt_2 \textit{ for atomic } t_1, t_2 \quad (\gamma_2) \\ \langle \emptyset, \{t \sim 0\}\rangle & \textit{else} \quad (\omega) \end{cases}$$

$$\textit{with} \quad \mathit{split}_f(\sum k_i t_i) = \sum_{\mathbf{sym}(t_i)=f} k_i t_i$$

Let us discuss the various cases of Def. 5. As noted in [21], constraints introduced by UWA during proof search are often of the form $kx + s \not\approx t$, where $x \not\trianglelefteq s$ or $x \not\trianglelefteq t$; for these constraints there is one unique most general solution, as defined in case α_1 of Def. 5. There are however cases in $\mathcal{A}^{\mathbb{Q}}_{eq}$ where $x \triangleleft t$ but still x, t can unify. An example for this are the terms x and $f(f(0) - x)$, which unify with $\{x \mapsto f(0)\}$. As computing all such constraints of $\mathcal{A}^{\mathbb{Q}}_{eq}$ is challenging, in α_3 we introduce a constraint in cases where x may be cancelled out. For cases when x cannot be cancelled out, x, t cannot be equal in $\mathbb{Q}$; hence, returning $\bot$ in α_2 is sound.

After cases $\alpha_1, \alpha_2, \alpha_3$, we are left with unifying terms without top level variables. An example of such terms is given by $f(x) + g(t)$ and $g(y) + f(y)$. Unifying such and similar terms is guided (split) by the top level symbols of the respective terms, as different uninterpreted functions cannot be unified. This is handled by $split_f$ in the cases $\beta_1 - \beta_2$.

Going further, unifying terms that are sums with the same top symbol is handled in $\gamma_1 - \gamma_2$ of Def. 5. Consider for example $2f(a)$ and $f(y)$, which cannot unify as the respective interpreted coefficients are not equal. This case is handled more generally in γ_1. Case γ_2 handles the special unification setting of two atomic summands, making sure we continue unifying instead of introducing a constraint. Finally, if all afore discussed cases of Def. 5 fail, a constraint is introduced in case ω.

We conclude this section by noting that, as argued above $\mathbf{abstr}_{\mathbb{Q}}$ is clearly $\mathcal{A}^{\mathbb{Q}}_{eq}$-sound. In order to see that $\mathbf{abstr}_{\mathbb{Q}}$ is minimal, we need have a look at all cases where we do not return $\bot$. All of them, except for β_2 we have $\langle \mathsf{unif}, \mathsf{constr} \rangle = \mathbf{abstr}_{\mathbb{Q}}(l, r)$ with $\mathsf{unif} \cup \mathsf{constr} = \{l' \sim r'\}$, where $l \approx r$ is equivalent to $l' \approx r'$ modulo $\mathcal{A}^{\mathbb{Q}}_{eq}$; hence in these cases minimality holds. As argued above distinct uninterpreted funcitons cannot be unified, hence the splitting in β_2 also preserves minimality. Moreover, $\mathbf{abstr}_{\mathbb{Q}}$ is terminating, as for all $s \sim t$ introduced in unif we have that $\mathbf{abstr}_{\mathbb{Q}}$ is undefined or s, t are trivially unifiable. In addition, $\mathbf{abstr}_{\mathbb{Q}}$ captures $\mathcal{A}^{\mathbb{Q}}_{eq}$, as $\mathbf{abstr}_{\mathbb{Q}}$ is defined for variables and interpreted functions. Finally, $\mathbf{abstr}_{\mathbb{Q}}$ is $\prec_{\text{QKBO}}$-founded, as literals in $(\mathsf{unif} \cup \mathsf{constr})^{\not\approx}$ are either equivalent to the original literal or their summands are subsets of the summands of the original terms (and hence smaller). Hence, Alg. 1 with $\mathbf{abstr}_{\mathbb{Q}}$ computes an abstracting unifier, fulfilling all properties form Def. 2. By Lem. 2, Alg. 1 replaces unification modulo $\mathcal{T}$ in a complete way.

4 Implementation and Experiments

We implemented our refined UWA approach for the theory $\mathbb{Q}$ of linear rational arithmetic in Vampire [19]. We used Alg. 1 with the abstraction oracle $\mathbf{abstr}_{\mathbb{Q}}$ to extend the Alasca reasoning of [18] in Vampire[1].

Implementation Details. Note that efficiency of Alg. 1 depends on the order how terms in *unproc* are processed. We identified three classes of constraints $\mathcal{C}$ we want to avoid introducing, which are illustrated in Figure 2 and discussed next.

(i) $\top$*-constraints.* These constraints are over terms s, t that are equal in $\mathbb{Q}$; hence, $\top$-constraints are redundant and can be dropped during *unproc*.

(ii) $\bot$*-constraints.* These constraints express $s \not\approx t$ for terms s, t that are not unifiable in $\mathbb{Q}$. Introducing such constraints over-approximates the set of ground unifiers, which is sound but inefficient. Unlike $\top$-constraints, $\bot$-constraints cannot be simplified during saturation.

(iii) weak substitutions. We avoid introducing constraints $s \not\approx t$ for which there is a unique **mgu** μ modulo $\mathcal{T}$ that can be cheaply computed, as such constraints may result in deriving more consequences of unified clauses than necessary.

We post-process every abstracting unifier $\langle \sigma, \mathcal{C} \rangle$, by fixed-point iterating the computation of **uwa** as long as there is a change in the result; that is until either $\mathbf{uwa}(s,t) = \bot$ or $\langle \sigma, \mathcal{C} \rangle = \mathbf{uwa}(s,t)$ and, for all constraints $s \not\approx t \in \mathcal{C}$, we have that $\langle \sigma, \mathcal{C} \rangle \equiv_{\mathcal{T}} \mathbf{uwa}(s,t)$. As $\mathbf{abstr}_{\mathbb{Q}}$ is $\prec$-founded, such fixed-point iteration terminates for well-founded $\prec$.

Experimental Setup. We revisit the setup of [18] and use the following benchmarks: (i) LRA, NRA and UFLRA, consiting of all SMT-LIB examples [5] that include real arithmetic, but no other theories; (ii) SH, containing the benchmarks of [10], with selecting only those involving real arithmetic and no other theories; (iii) Triangular and Limit, representing mathematical

[1]our implementation is publicly available at https://github.com/vprover/vampire/tree/alasca-new-uwa

Issue	Example	left-to-right	right-to-left
$\top$-constraints	$g(f(y)+f(x),x,y)$	$\sigma=\{x\mapsto a, y\mapsto b\}$	$\sigma=\{x\mapsto a, y\mapsto b\}$
	$g(f(a)+f(b),a,b)$	$\mathcal{C}=\{f(a)+f(b)\not\approx f(b)+f(a)\}$	$\mathcal{C}=\emptyset$
$\perp$-constraints	$g(f(x)+f(y),x,y)$	$\sigma=\{x\mapsto a, y\mapsto c\}$	$\perp$
	$g(f(a)+f(b),a,c)$	$\mathcal{C}=\{f(a)+f(b)\not\approx f(a)+f(c)\}$	
weak substitutions	$g(f(x)+f(y),x)$	$\sigma=\{x\mapsto a\}$	$\sigma=\{x\mapsto a, y\mapsto b\}$
	$g(f(b)+f(a),a)$	$\mathcal{C}=\{f(a)+f(y)\not\approx f(b)+f(a)\}$	$\mathcal{C}=\emptyset$

Figure 2: Different results obtained by Alg. 1 depending on the traversal order of arguments. The second column gives example terms being unified, the last two columns show different results depending on the traversal order.

Benchmarks (#)	Alasca$^*_2$	Alasca	Cvc5	Vampire	Yices	UltElim	SmtInt	veriT	solved
total (6374)	**5753**	5744	5626	5585	5531	5218	828	465	5988
LRA (1722)	1581	1572	1401	1396	**1722**	1469	623	89	1722
NRA (3814)	3800	3800	3804	3803	**3809**	3669	0	0	3812
UFLRA (10)	**10**	**10**	**10**	**10**	0	0	**10**	**10**	10
Triangular (34)	**24**	**24**	10	13	0	0	0	6	25
Limit (280)	**100**	**100**	90	81	0	80	0	90	100
SH (514)	238	238	**311**	282	0	0	195	270	319

Figure 3: Overall experimental results.

properties from [18]. As in SMT-COMP 2022, our experiments were carried out on the StarExec Iowa cluster, with a timeout of 20 minutes and 4 cores[2].

Experimental Results. We compared our work to all solvers from the arithmetic division of SMT-COMP 2022, namely: Cvc5 [3], Vampire [23], Yices [14], UltElim [6], SmtInt [16], and veriT [1]. We also compared our work against Alasca [18]. Figure 3 summarizes our results, showing that our new and optimized implementation (Alasca$^+_2$) performs overall best.

We also tested our refined UWA implementation with different options. The first setting is Alasca$^+_0$, where we use an abstraction oracle $\mathbf{abstr}_0$ that behaves like a simple **canAbstract** predicate from [22]; that is, $\mathbf{abstr}_0(s,t) = \langle\emptyset, \langle s,t\rangle\rangle$ if either $\mathbf{sym}(s)$, or $\mathbf{sym}(t)$ is interpreted or one of the two terms is a variable contained in the other one that might cancel out (see Def. 5), and is undefined otherwise. The second configuration is Alasca$^+_1$, which uses an abstraction oracle that behaves like a smarter abstraction predicate, defined $\mathbf{abstr}_1(s,t) = \langle\emptyset, \{s \sim t\}\rangle$ iff $\mathbf{abstr}_{\mathbb{Q}}(s,t) = \langle\mathsf{constr}, \mathsf{unif}\rangle$, for some $\mathsf{unif}, \mathsf{constr}$. The third one is Alasca$^+_2$ which uses $\mathbf{abstr}_{\mathbb{Q}}$ straight away. For each of these configurations Alasca^+_X ($X \in \{0,1,2\}$), we also considered a respective Alasca$^*_X$ configuration that additionally uses our fixed-point iteration described above. Figure 4 shows that each refinement of Alg. 1 gives gradually better performances, with the fixed-point iteration of Alg. 1 using $\mathbf{abstr}_{\mathbb{Q}}$ performing overall the best.

5 Conclusions

We refined unification with abstraction with abstraction oracles. We prove soundness, and completeness of such unification with background theories $\mathcal{T}$ and experimentally demonstrate the gains of our work within linear rational arithmetic. We plan to further expand our work to capture function extensionality axioms by abstraction [8, 7], tackling higher-order unification.

[2] Results, solvers, and benchmarks can be publicly accessed on `https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=536083`.

Benchmarks (#)	ALASCA$_0^+$	ALASCA$_0^*$	ALASCA$_1^+$	ALASCA$_1^*$	ALASCA$_2^+$	ALASCA$_2^*$	solved
total (6374)	5739	5739	5746	5746	5752	**5753**	5755
LRA (1722)	1575	1576	1580	1580	**1581**	**1581**	1581
NRA (3814)	3799	3799	3799	3799	**3800**	**3800**	3800
UFLRA (10)	**10**	**10**	**10**	**10**	**10**	**10**	10
TRIANGULAR (34)	22	22	23	**24**	**24**	**24**	24
LIMIT (280)	**100**	**100**	**100**	**100**	**100**	**100**	100
SH (514)	233	232	234	233	237	**238**	240

Figure 4: Experimental results comparing various configurations of Alg. 1.

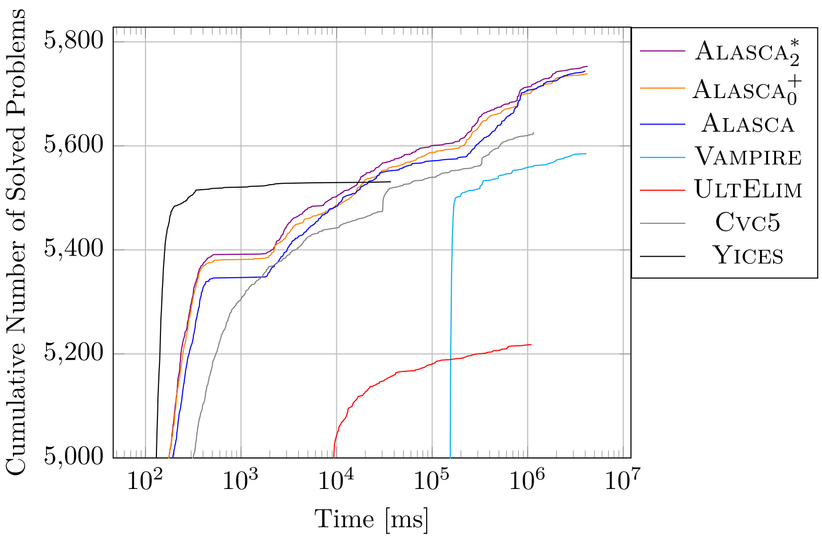

Figure 5: Cumulative number of solved problems by time, for each solvers. The lines stop at the last problem solved by each solver

Acknowledgements. This work was partially supported by the ERC Consolidator Grant ARTIST 101002685, the EPSRC grant EP/V000497/1, the TU Wien Doctoral College SecInt, and the FWF special research project SFB SpyCoDe F8504.

References

[1] Bruno Andreotti, Haniel Barbosa, Pascal Fontaine, and Hans-Jörg Schurr. veriT at SMT-COMP 2022. `https://smt-comp.github.io/2022/system-descriptions/veriT.pdf`, 2022.

[2] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001.

[3] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. CVC5 at the SMT Competition 2022. `https://smt-comp.github.io/2022/system-descriptions/cvc5.pdf`, 2022.

[4] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS*, pages 415–442, 2022.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[6] Max Barth, Daniel Dietsch, Matthias Heizmann, and Andreas Podelski. Ultimate Eliminator at SMT-COMP 2022. https://smt-comp.github.io/2022/system-descriptions/UltimateEliminator%2BMathSAT.pdf, 2022.

[7] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with Lambdas. *Journal of Automated Reasoning*, 65(7):893–940, 2021.

[8] Ahmed Bhayat and Giles Reger. A Combinator-Based Superposition Calculus for Higher-Order Logic. In *IJCAR*, pages 278–296, 2020.

[9] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[10] Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen Provers Under the Hammer. In *ITP*, pages 8:1–8:18, 2022.

[11] Eric Domenjoud. A Technical Note on AC-Unification. The Number of Minimal Unifiers of the Equation $\alpha x_1 + \cdots + \alpha x_p \doteq_{AC} \beta y_1 + \cdots + \beta y_q$. *Journal of Automated Reasoning*, 8(1):39–44, 1992.

[12] André Duarte and Konstantin Korovin. Implementing Superposition in iProver (System Description). In *IJCAR*, pages 388–397, 2020.

[13] Bruno Dutertre. Yices 2.2. In *CAV*, pages 737–744, 2014.

[14] Stéphane Graham-Lengrand. Yices-QS 2022, an Extension of Yices for Quantified Satisfiability. https://smt-comp.github.io/2022/system-descriptions/YicesQS.pdf, 2022.

[15] Krystof Hoder and Andrei Voronkov. Comparing Unification Algorithms in First-Order Theorem Proving. In *KI*, pages 435–443, 2009.

[16] Jochen Hoenicke and Tanja Schindler. SMTInterpol with Resolution Proofs. https://smt-comp.github.io/2022/system-descriptions/smtinterpol.pdf, 2022.

[17] Gérard P. Huet. A Unification Algorithm for Typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[18] Konstantin Korovin, Laura Kovacs, Giles Reger, Johannes Schoisswohl, and Andrei Voronkov. ALASCA: Reasoning in Quantified Linear Arithmetic. In *TACAS*, 2023. To appear.

[19] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.

[20] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. Elsevier and MIT Press, 2001.

[21] Giles Reger, Johannes Schoisswohl, and Andrei Voronkov. Making Theory Reasoning Simpler. In *TACAS*, pages 164–180, 2021.

[22] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with Abstraction and Theory Instantiation in Saturation-Based Reasoning. In *TACAS*, pages 3–22. Springer, 2018.

[23] Giles Reger, Martin Suda, Andrei Voronkov, Laura Kovács, Ahmed Bhayat, Bernhard Gleiss, Marton Hajdu, Petra Hozzova, Jakob Rath Evgeny Kotelnikov, Michael Rawson, Martin Riener, Simon Robillard, and Johannes Schoisswohl. Vampire 4.7-SMT System Description. https://smt-comp.github.io/2022/system-descriptions/Vampire.pdf, 2022.

[24] John Alan Robinson. A Machine-Oriented Logic based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[25] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, Higher, Stronger: E 2.3. In *CADE*, pages 495–507.

[26] Uwe Waldmann. Superposition for Divisible Torsion-Free Abelian Groups. In *CADE*, pages 144–159, 1998.

[27] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In *CADE*, pages 140–145, 2009.