REST: Integrating Term Rewriting with Program Verification

Zachary Grannan
University of British Columbia, Vancouver, Canada

Niki Vazou
IMDEA Software Institute, Madrid, Spain

Eva Darulova[1]
Uppsala University, Sweden

Alexander J. Summers
University of British Columbia, Vancouver, Canada

Abstract

We introduce `REST`, a novel term rewriting technique for theorem proving that uses online termination checking and can be integrated with existing program verifiers. `REST` enables flexible but terminating term rewriting for theorem proving by: (1) exploiting newly-introduced term orderings that are more permissive than standard rewrite simplification orderings; (2) dynamically and iteratively selecting orderings based on the path of rewrites taken so far; and (3) integrating external oracles that allow steps that cannot be justified with rewrite rules. Our `REST` approach is designed around an easily implementable core algorithm, parameterizable by choices of term orderings and their implementations; in this way our approach can be easily integrated into existing tools. We implemented `REST` as a Haskell library and incorporated it into Liquid Haskell's evaluation strategy, extending Liquid Haskell with rewriting rules. We evaluated our `REST` implementation by comparing it against both existing rewriting techniques and E-matching (as used in most SMT solvers) and by showing that it can be used to supplant manual lemma application in many existing Liquid Haskell proofs.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases term rewriting, program verification, theorem proving

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.13

Related Version *Extended Version*: `https://arxiv.org/abs/2202.05872` [26]

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.12`

Funding This work was supported by the Juan de la Cierva grants IJC2019-041599-I, the HaCrypt ONR project N00014-19-1-2292, and the ERC starting grant CRETE (101039196). We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Acknowledgements We thank Jonathan Chan, Eric Conlon, Rui Ge, Paulette Koronkevich and the anonymous reviewers for their helpful and constructive feedback.

1 Introduction

For all disjoint sets s_0 and s_1, the identity $(s_0 \cup s_1) \cap s_0 = s_0$ can be proven in many ways. Informally accepting this property is easy, but a machine-checked formal proof may require the instantiation of multiple set theoretic axioms. Analogously, further proofs relying on this

[*] This work was partly done while the author was at MPI-SWS.

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 13; pp. 13:1–13:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

identity may themselves need to apply it as a previously-proven lemma. For example, proving functional correctness of any program that relies on a set data structure typically requires the instantiation of set-related lemmas. Manual instantiation of such universally quantified equalities is tedious, and the burden becomes substantial for more complex proofs: a proof author needs to identify exactly which equalities to instantiate and with which arguments; in the context of program verification, a wide variety of such lemmas are typically available. Given this need, most program verifiers provide some automated technique or heuristics for instantiating universally quantified equalities.

For the wide range of practical program verifiers that are built upon SMT solvers (e.g., [35, 23, 51, 39, 47, 45]), quantified equalities can naturally be expressed in the SMT solver's logic. However, relying solely on such solvers' E-matching techniques [19] for quantifier instantiation (as the majority of these verifiers do) can lead to both non-termination and incompletenesses that may be unpredictable [34] and challenging to diagnose [7]. Theory and techniques for proving termination and completeness of encodings using E-matching for equality reasoning is relatively unexplored [21]; the inherent treatment of terms modulo equalities makes standard term orderings based on term structure unsound.

A classical alternative approach to automating equality reasoning is *term rewriting* [28], which can be used to encode lemma properties as (directed) rewrite rules, matching terms against the existing set of rules to identify potential rewrites; the termination of these systems is a well-studied problem [16]. Although SMT solvers often perform rewriting as an internal simplification step, verifiers built on top typically cannot access or customize these rules, e.g., to add previously-proved lemmas as rewrite rules. By contrast, many mainstream proof assistants (e.g., `Coq` [11], `Isabelle/HOL` [40], `Lean` [5]) provide automated, customizable term rewriting tactics. However, the rewriting functionalities of mainstream proof assistants either do not ensure the termination of rewriting (potentially resulting in divergence, for example Isabelle) or enforce termination checks that are overly restrictive in general, potentially rejecting necessary rewrite steps (for example, Lean).

In this paper, we present *`REST` (REwriting and Selecting Termination orderings)*: a novel technique that equips program verifiers with automatic lemma application facilities via term rewriting, enabling equational reasoning with complementary strengths to E-matching-based techniques. While term rewriting in general does not guarantee termination, our technique weaves together three key technical ingredients to automatically generate and explore guaranteed-terminating restrictions of a given rewriting system while typically retaining the rewrites needed in practice: (1) `REST` compares terms using well-quasi-orderings derived from (strict) simplification orderings; thereby facilitating common and important rules such as commutativity and associativity properties. (2) `REST` simultaneously considers an entire family of term orderings; selecting the appropriate term ordering to justify rewrite steps *during term rewriting itself.* (3) `REST` allows integration of an *external oracle* that generates additional steps outside of the term rewriting system. This allows the incorporation of reasoning steps awkward or impossible to justify via rewriting rules, all without compromising the termination and relative completeness guarantees of our overall technique.

Contributions and Overview. We make the following contributions:

1. We design and present a new approach (`REST`) for applying term rewriting rules and simultaneously selecting appropriate term orderings to permit as many rewriting steps as possible while guaranteeing termination (Sec. 3).
2. We introduce ordering constraint algebras, an abstraction for reasoning effectively about multiple (and possibly infinitely many) term orderings simultaneously (Sec. 4).

Name	*Formula*
idem-union	$X \cup X \;=\; X$
idem-inter	$X \cap X \;=\; X$
empty-union	$X \cup \emptyset \;=\; X$
empty-inter	$X \cap \emptyset \;=\; \emptyset$
commut-union	$X \cup Y \;=\; Y \cup X$
symm-inter	$X \cap Y \;=\; Y \cap X$
distrib-union	$(X \cup Y) \cap Z \;=\; (X \cap Z) \cup (Y \cap Z)$
distrib-inter	$(X \cap Y) \cup Z \;=\; (X \cup Z) \cap (Y \cup Z)$
assoc-union	$X \cup (Y \cup Z) \;=\; (X \cup Y) \cup Z$

Figure 1 Set identities used for examples in this section. Variables X, Y, Z are implicitly quantified. We write the binary functions $\cup, \cap$ infix; along with (nullary) $\emptyset$ these are fixed function symbols.

3. We introduce and formalize recursive path quasi-orderings (RPQOs) derived from the well-known recursive path ordering [15] (Sec. 4.1.2). RPQOs are more permissive than classical RPOs, and so let us prove more properties.
4. We formalize and prove key results for our technique: soundness, relative completeness, and termination (Sec. 5).
5. We implement `REST` as a stand-alone library, and integrate the `REST` library into Liquid Haskell to facilitate automatic lemma instantiation (Sec. 6).
6. We evaluate `REST` by comparison to other term rewriting tactics and E-matching-based axiomatization, and show that `REST` can simplify equational reasoning proofs (Sec. 7).

We discuss related work in Sec. 8; we begin (Sec. 2) by identifying five key challenges of a reliable and automatic integration of term rewriting into a program verification tool.

2 Five Challenges for Automating Term Rewriting

In this section, we describe *five key challenges* that naturally arise when term rewriting is used for program verification and outline how `REST` is designed to address them. To illustrate the challenges, we use simple verification goals that involve uninterpreted functions and the set operators ($\emptyset$, $\cup$, $\cap$) that satisfy the standard properties of Figure 1. The variables x, y, z are implicitly quantified[2] in these rules. In formalizations of set theory, such properties may be assumed as (quantified) axioms, or proven as lemmas and then used in future proofs.

Term rewriting systems (defined formally in Sec. 5) are a standard approach for formally expressing and applying equational reasoning (rewriting terms via known identities). A term rewriting system consists of a finite set of *rewrite rules*, each consisting of a pair of a *source term* and a *target term*, representing that terms matching a rule's source can be replaced by corresponding terms matching its target. For example, the rewrite rule $X \cup \emptyset \to X$ can replace set unions of some set X and the empty set with the corresponding set X. Rewrite rules are applied to a term t by identifying some subterm of t which is equal to a rule's source under some substitution of the source's free variables (here, X, but not constants such as $\emptyset$); the subterm is then replaced with the correspondingly substituted target term. This rewriting step *induces an equality* between the original and new terms. For instance, the example rewrite rule above can be used to rewrite a term $f(s_0 \cup \emptyset)$ into $f(s_0)$, inducing an equality between the two.

[2] over sets; we omit explicit types in such formulas, whose type-checking is standard.

Rewrite rules classically come with two restrictions: the free variables of the target must all occur in the source and the source must not be a single variable. This precludes rewrite rules which invent terms, such as $\emptyset \to X \cap \emptyset$, and those that trivially lead to infinite derivations. Under these restrictions, the first four identities induce rewrite rules from left-to-right (which we denote by e.g., *idem-inter*$\rightarrow$), while the remaining induce rewrite rules in both directions (e.g., *assoc-union*$\rightarrow$ vs. *assoc-union*$\leftarrow$).

Next, we present a simple proof obligation taken from [36] in the style of equational reasoning (*calculational proofs*) supported in the Dafny program verifier [35].

▶ **Example 1.** We aim to prove, for two sets s_0 and s_1 and some unary function f on sets, that, if the sets are disjoint (that is, $s_1 \cap s_0 = \emptyset$), then $f((s_0 \cup s_1) \cap s_0) = f(s_0)$.

$$\begin{array}{llll} \textit{Equational Proof:} \quad f((s_0 \cup s_1) \cap s_0) & = & f((s_0 \cap s_0) \cup (s_1 \cap s_0)) & \textit{(distrib-union}\rightarrow\textit{)} \\ & = & f(s_0 \cup (s_1 \cap s_0)) & \textit{(idem-inter}\rightarrow\textit{)} \\ & = & f(s_0 \cup \emptyset) & \textit{(disjointness ass.}\rightarrow\textit{)} \\ & = & f(s_0) & \textit{(empty-union}\rightarrow\textit{)} \end{array}$$

(Possible Term Ordering, as explained shortly: RPO instance with $\cap > \cup$)

This manual proof closely follows the user annotations employed in the corresponding Dafny proof [36]; the application of the function f serves only to illustrate equational reasoning on subterms. Every step of the proof could be explained by term rewriting, hinting at the possibility of an *automated* proof in which term rewriting is used to solve such proof obligations. In particular, taking the term rewriting system naturally induced by the set identities of Figure 1 *along with* the assumed equality expressing disjointness of s_0 and s_1 results in a term rewriting system in which the four proof steps are all valid rewriting steps.

In the remainder of the section, we consider what it would take to make term rewriting effective for reliably automating such verification tasks. Perhaps unsurprisingly, there are multiple problems with the simplistic approach outlined so far. The first and most serious is that term rewriting systems in general *do not guarantee termination*; a proof search may continue indefinitely by repeatedly applying rewrite rules. For example, the rules *distrib-union* and *distrib-inter* can lead to an infinite derivation $(s_0 \cup s_1) \cap s_2 \to (s_0 \cap s_2) \cup (s_1 \cap s_2) \to (s_0 \cup (s_1 \cap s_2)) \cap (s_2 \cup (s_1 \cap s_2)) \to \dots$

Challenge 1: Unrestricted term rewriting systems do not guarantee termination.

To ensure termination (as proved in Theorem 22) `REST` follows the classical approach of restricting a term-rewriting system to a variant in which sequences of term rewrites (*rewrite paths*) are allowed only if each consecutive pair of terms is *ordered* according to some term ordering which rules out infinite paths.

For example, *Recursive path orderings* (RPOs) [15] define well-founded orders $>_{\mathcal{T}}$ on terms $\mathcal{T}$ based on an underlying well-founded strict partial order $>$ on *function symbols*. Intuitively, such orderings use $>$ to order terms with different top-level function symbols, combined with the properties of a *simplification order* [14] (e.g., compatibility with the subterm relation). Different choices of the underlying $>$ parameter yield different RPO instances that order different pairs of terms; in particular, potentially allowing or disallowing certain rewrite paths.

In Example 1, an RPO based on a partial order where $\cap > \cup$ and $\cap > \emptyset$ permits all the rewriting steps, that is, the left-hand-side of each equation is greater than the right-hand-side.

Sadly, this ordering will not permit the rewriting steps required by our next example.

▶ **Example 2.** We aim to prove, for two sets s_0 and s_1 and some unary function f on sets, that, if s_1 is a subset of s_0 (that is, $s_0 \cup s_1 = s_0$), then $f((s_0 \cap s_1) \cup s_0) = f(s_0)$.

$$\begin{array}{llcll} \textit{Equational Proof:} & f((s_0 \cap s_1) \cup s_0) & = & f((s_0 \cup s_0) \cap (s_1 \cup s_0)) & \textit{(distrib-inter}\rightarrow\textit{)} \\ & & = & f(s_0 \cap (s_1 \cup s_0)) & \textit{(idem-union}\rightarrow\textit{)} \\ & & = & f(s_0 \cap (s_0 \cup s_1)) & \textit{(commut-union}\rightarrow\textit{)} \\ & & = & f(s_0 \cap s_0) & \textit{(subset ass.}\rightarrow\textit{)} \\ & & = & f(s_0) & \textit{(idem-inter}\rightarrow\textit{)} \end{array}$$

(Possible Term Ordering: RPQO instance, explained shortly, with $\cup > \cap$)

An RPO based on an ordering where $\cap > \cup$ (as required by Example 1) will not permit the first step of this proof (since the RPO ordering first compares the top level function symbols). Instead, this step requires an RPO based on an ordering where $\cup > \cap$. To accept *both* this proof step *and* the Example 1 we need different restrictions of the rewrite rules for different proofs; in particular, different rewrite paths may be ordered according to RPOs that are based on different function orderings.

To generalize this problem we will call RPOs a term ordering *family* that is *parametric* with respect to the underlying function ordering. Thus, a concrete RPO term ordering (called an *instance* of the family) is obtained after the parametric function ordering is instantiated. With this terminology, the next challenge can be stated as follows:

Challenge 2: Different proofs require different term orderings within a family.

Note that enumerating all term orderings in a term ordering family is typically impractical (this set is often very large and may be infinite). To address this challenge, `REST` uses a novel algebraic structure (Sec. 4.2) to allow for an abstract representation of sets of term orderings with which one can efficiently check whether any instance of a chosen term ordering family can orient the necessary rewrite steps to complete a proof.

Going back to Example 2, the RPO instance with $\cup > \cap$ will permit all the steps, apart from the commutativity axiom expressed by *(commut-union→)*. To permit this step we need an ordering for which $t_1 \cup t_2 >_{\mathcal{T}} t_2 \cup t_1$. But for RPO instances, as well as for many other term orderings, the terms $t_1 \cup t_2$ and $t_2 \cup t_1$ are equivalent and thus cannot be oriented; associativity axioms are also similarly challenging. Since many proofs require such properties, it is important in practice for rewriting to support them.

Challenge 3: Strict orderings restrict commutativity and associativity steps.

To address this challenge `REST` relaxes the strictness constraint by requiring the chosen term ordering family to consist (only) of *thin well-quasi-orderings* (Def. 5). Intuitively, such orderings permit rewriting to terms which are *equal* according to the ordering, but such equivalence classes of terms must be finite. In Sec. 4 we show how to lift well-known families of term orderings to more-permissive families of thin well-quasi-orders. In particular, we show how to lift RPOs to a particularly powerful family of term orderings that we call *recursive path quasi-orderings (RPQOs)* (Def. 10), whose instances allow us to accept Example 2.

Despite the permissiveness of RPQOs, there remain some rewrite derivations that will be rejected by all term orderings in the RPQO family. For example, consider the following proof that set union is monotonic with respect to the subset relation:

▶ **Example 3.** We aim to prove, for sets s_0, s_1, and s_2, that, if s_1 is a subset of s_0 (that is, $s_0 \cup s_1 = s_0$), then $(s_2 \cup s_1) \cup (s_2 \cup s_0) = s_2 \cup s_0$.

$$
\begin{array}{llll}
\textit{Equational Proof:} \quad (s_2 \cup s_1) \cup (s_2 \cup s_0) & = & s_2 \cup (s_1 \cup (s_2 \cup s_0)) & \textit{(assoc-union}\leftarrow\textit{)} \\
 & = & s_2 \cup ((s_1 \cup s_2) \cup s_0) & \textit{(assoc-union}\rightarrow\textit{)} \\
 & = & s_2 \cup ((s_2 \cup s_1) \cup s_0) & \textit{(commut-union}\rightarrow\textit{)} \\
 & = & s_2 \cup (s_2 \cup (s_1 \cup s_0)) & \textit{(assoc-union}\leftarrow\textit{)} \\
 & = & s_2 \cup (s_2 \cup (s_0 \cup s_1)) & \textit{(commut-union}\rightarrow\textit{)} \\
 & = & s_2 \cup (s_2 \cup s_0) & \textit{(subset ass.}\rightarrow\textit{)} \\
 & = & (s_2 \cup s_2) \cup s_0 & \textit{(assoc-union}\rightarrow\textit{)} \\
 & = & s_2 \cup s_0 & \textit{(idem-union}\rightarrow\textit{)}
\end{array}
$$

(Possible Term Ordering: any KBQO instance)

The above rewrite rule steps cannot be oriented by any RPQO, but are trivially oriented by a quasi-ordering that is based on the syntactic size of the term, e.g., a quasi-ordering based on the well-known Knuth-Bendix family of term orderings [31]. Yet, a Knuth-Bendix quasi-ordering (KBQO, defined in Sec. 4) cannot be used on our previous two examples; fixing even a single choice of term ordering *family* would still be too restrictive in general.

Challenge 4: Some proofs require different families of term orderings.

To address this challenge, REST (Sec. 3.2) is defined parametrically in the choice and representation of a term ordering family.

Finally, although equational reasoning is powerful enough for these examples, general verification problems usually require reasoning beyond the scope of simple rewriting. For example, simply altering Example 1 to express the disjointness hypothesis instead via cardinality as $|s_0 \cap s_1| = 0$ means that, to achieve a similar proof, reasoning within the theory of sets is necessary to deduce that this hypothesis implies the equality needed for the proof; this is beyond the abilities of term rewriting.

Challenge 5: Program verification needs proof steps not expressible by rewriting.

To address this challenge, our REST approach allows the integration of an external oracle that can generate equalities not justifiable by term rewriting (Sec. 3.3).

3 The REST Approach

We develop REST to tackle the above five challenges and integrate a flexible, expressive, and guaranteed-terminating term rewriting system with a verification tool. REST consists of an interface that defines term orderings and an algorithm that explores the terminating rewrite paths. In Sec. 3.1 we describe the representation of term orderings in REST and how they address Challenges 2 and 4. In Sec. 3.2 we describe the REST algorithm that is parametric to these orderings and Sec. 3.3 describes the integration with external oracles (Challenge 5).

3.1 Representation of Term Orderings in REST

Rather than considering individual term orderings, REST operates on indexed sets (families) of term orderings (whose instances must all be thin well-quasi-orderings [Def. 5]).

▶ **Definition 4** (Term Ordering Family). *A* term ordering family Γ *is a set of thin well-quasi-orderings on terms, indexed by some parameters* P*. An* instance *of the family is a term ordering obtained by a particular instantiation of* P*.*

For example, the recursive path ordering is defined parametrically with respect to a precedence on function symbols, and therefore defines a term ordering family indexed by this choice of function symbol ordering.

A core concern of `REST` is determining whether any instance of a given term ordering family can orient a rewrite path. However, term ordering families cannot directly compare terms; doing so requires choosing an ordering inside the family. The root of Challenge 2 is that choosing an ordering in advance is too restrictive: different orderings are necessary to complete different proofs. The idea behind `REST`'s search algorithm is to address this challenge by simultaneously considering all orderings in the family when considering rewrite paths and continuing the path so long as it can be oriented by *any* ordering.

To demonstrate the technique, we show how `REST`'s approach can be derived from a naïve algorithm. The purpose of the algorithm is to determine if any ordering in a family Γ can orient a path $t_1 \to \ldots \to t_n$; i.e., if there is a $>_{\mathcal{T}} \in \Gamma$ such that $t_1 >_{\mathcal{T}} \ldots >_{\mathcal{T}} t_n$.

`orients` : (Set $O \times$ List $\mathcal{T}$) $\to$ *Bool*		`orients` : ($OCA \times$ List $\mathcal{T}$)) $\to$ *Bool*
`orients`$(\Gamma, ts) =$		`orients`$(\langle \top, \mathit{refine}, \mathit{sat} \rangle, ts) =$
$os := \Gamma$;	(1)	$c := \top$;
for $i \in 1$ **to** $\lvert ts \rvert - 1$ {		**for** $i \in 1$ **to** $\lvert ts \rvert - 1$ {
$os := \{>_{\mathcal{T}} \in os \mid ts_i >_{\mathcal{T}} ts_{i+1}\}$;	(2)	$c := \mathit{refine}(c, ts_i, ts_{i+1})$;
if $(os = \emptyset)$	(3)	**if** (**not**$(\mathit{sat}(c))$)
return *false*;		**return** *false*;
}		}
return *true*;		**return** *true*;

Figure 2 Two algorithms that determine if an ordering in the term ordering family Γ can orient a path of terms ts. **Left** presents the naïve, exhaustive algorithm. **Right** is using the ordering constraint algebra $\langle \top, \mathit{refine}, \mathit{sat} \rangle$ that returns true iff an ordering in Γ can orient ts without explicitly constructing any term orderings. $O$is the type of a term ordering.

The naïve algorithm is depicted on the left of Figure 2. The naïve algorithm works iteratively, computing the set of orderings os that can orient an increasingly-long path, short-circuiting if the set becomes empty. The algorithm enumerates each ordering in Γ and compares terms with each ordering (potentially multiple times). Unfortunately, this enumeration is not practical: some term ordering families have infinite or prohibitively large numbers of instances. `REST` avoids these issues by allowing the set of term orderings to be abstracted via a structure called an Ordering Constraint Algebra (OCA, Def. 14 of Sec. 4.2).

An OCA for a term ordering family Γ consists of a type C along with four parameters $\gamma : C \to \mathcal{P}(\Gamma)$, $\top : C$, $\mathit{refine} : C \to \mathcal{T} \to \mathcal{T} \to C$, and $\mathit{sat} : C \to \mathit{Bool}$. C is a type whose elements *represent* subsets of Γ. The function γ is the *concretisation function* of the OCA, not needed programmatically but instead defining the *meaning* of elements of C in terms of the subsets of the term ordering family they represent. The remaining three functions correspond to the operations on sets of term orderings used in lines (1), (2), and (3) of the naïve algorithm. $\top$ represents the set of all term orderings in Γ, $\mathit{refine}(c, t, u)$ filters the set of orderings represented by c to include only those where $t >_{\mathcal{T}} u$, and $\mathit{sat}(c)$ is a predicate that returns true if the set of orderings represented by c is nonempty. Figure 2 on the right shows

$$
\begin{array}{|l|}
\hline
\mathsf{REST} : (OCA \times \mathcal{R} \times \mathcal{T} \ \times \ (\mathcal{T} \to \text{Set } \mathcal{T})) \to \text{Set } \mathcal{T} \\
\hline
\mathsf{REST}(\langle \top, \mathit{refine}, \mathit{sat} \rangle, R, t_0, \mathcal{E}) = \\
\quad o := \emptyset; \\
\quad p := [([t_0], \top)]; \\
\quad \textbf{while } (p \text{ is not empty})\{ \\
\quad\quad \textbf{pop}(ts, c) \text{ from } p; \\
\quad\quad t := \text{last } ts; \\
\quad\quad o := o \cup \{t\}; \\
\quad\quad \textbf{foreach } (t' \mathit{such\ that}\ t' \notin ts \ \wedge \ (t \to_R t' \ \vee \ t' \in \mathcal{E}(t)))\{ \\
\quad\quad\quad \textbf{if } (t' \in \mathcal{E}(t) \vee (t \to_R t' \wedge \mathit{sat}(\mathit{refine}(c, t, t'))))\{ \\
\quad\quad\quad\quad \textbf{push } (ts +\!\!+ [t'], \mathit{refine}(c, t, t')) \textbf{ to } p \\
\quad\quad\quad \} \\
\quad\quad \} \\
\quad \} \\
\quad \textbf{return } o; \\
\hline
\end{array}
$$

Figure 3 The REST algorithm.

how the ordering constraint algebra can be used to perform an equivalent computation to the naïve algorithm, without explicitly instantiating sets of term orderings. The OCA plays a role similar to abstract interpretation in a program analysis, where C is an abstraction over sets of term orderings, and the results of the abstract operations on C correspond to their concrete equivalents. Namely, we have $\gamma(\top) = \Gamma$, $\gamma(\mathit{refine}(c, t_l, t_r)) = \{\succcurlyeq \ | \ \succcurlyeq \in \gamma(c) \ \wedge \ t_l \succcurlyeq t_r\}$, and $\mathit{sat}(c) \ \Leftrightarrow \ \gamma(c) \neq \emptyset$.

The ordering constraint algebra enables three main advantages compared to direct computation with sets of term orderings:

1. The number of term orderings can be very large, or even infinite, thus making enumeration of the entire set intractable.
2. An OCA can provide efficient implementations for *refine* and *sat* by exploiting properties of the term ordering family. Comparing terms using the constituent term orderings requires repeating the comparison for each ordering, despite the fact that most orderings will differ in ways that are irrelevant for the comparison.
3. The OCA does not impose any requirements on the type of C or the implementation of $\top$, *refine*, and *sat*. For example, an OCA can use $\top$ and *refine* to construct logical formulas, with *sat* using an external solver to check their satisfiability. Alternatively, it could define C to be sets of term orderings that are reasoned about explicitly, and implement $\top$, *refine*, and *sat* as the operations of the naïve algorithm.

We now describe how the REST algorithm uses the OCA to explore rewrite paths.

3.2 The REST Algorithm

Figure 3 presents the REST algorithm. The algorithm takes four parameters. The first parameter is an OCA $\langle \top, \mathit{refine}, \mathit{sat} \rangle$, as discussed above. The algorithm's second parameter, R, is a finite set of term rewriting rules (not required to be terminating); for example, we could pass the oriented rewrite rules corresponding to Figure 1. The third parameter t_0 is the term from which term rewrites are sought. The final parameter $\mathcal{E}$ acts as an external oracle, generating additional rewrite steps that need *not* follow from the term rewriting rules R. To simplify the explanation, we will initially assume that $\mathcal{E} = \lambda t.\emptyset$, i.e., this parameter

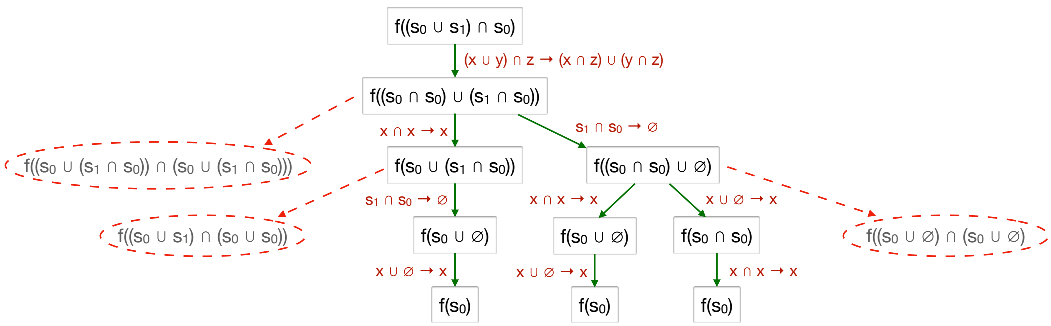

Figure 4 A visualization of `REST` running on the term from Example 1. Each path through the tree shown represents a rewrite path uncovered by our algorithm; the edge labels show the rewrite rule applied. The red dotted lines indicate rewrite steps rejected by `REST`.

has no effect. Our algorithm produces a set of terms, each of which are reachable by *some* rewrite path beginning from t_0, and for which *some* ordering allows the rewrite path. The algorithm addresses Challenge 1 (termination; Theorem 22) because every path must be finite: no ordering could orient an infinite path.

`REST` operates in worklist fashion, storing in p a list of pairs (ts, c) where ts is a non-empty list of terms representing a rewrite path already explored (the head of which is always t_0) and c tracks the ordering constraints of the path so far. The set o records the output terms (initially empty): all terms discovered equal to t_0 via the rewriting paths explored.

While there are still rewrite paths to be extended, i.e., p is not empty, a tuple (ts, c) is popped from p. `REST` puts t, i.e., the last term of the path, into the set of output terms o and considers all terms t' that are: (a) not *already* in the path and (b) reachable by a single rewrite step of R (or returned by the function $\mathcal{E}$ explained later). The crucial decision of whether or not to extend a rewrite path with the additional step $t \to t'$ is handled in the **if** check of `REST`. This check is to guarantee termination: the *sat* check enforces that we only add rewrite steps which would leave the extended path still justifiable by *some* term ordering.

Figure 4 visualizes the rewrite paths explored by our algorithm for a run corresponding to the problem from Example 1, using the OCA for the recursive path quasi-ordering (Sec. 4.2)[3]. The manual proof in Example 1 corresponds to the right-most path in this tree; the other paths apply the same reasoning steps in different orders. In our implementation, we optimize the algorithm to avoid re-exploring the same term multiple times unless this could lead to further rewrites being discovered (cf. Sec. 6).

The arrow from the root of the tree to its child corresponds to the first rewrite `REST` applies: $f((s_0 \cup s_1) \cap s_0) \to f((s_0 \cap s_0) \cup (s_1 \cap s_0))$. This rewrite step can only be oriented by RPQOs with precedence $\cap > \cup$; therefore applying this rewrite constrains the set of RPQOs that `REST` must consider in subsequent applications. For example, the rewrite to the left child of $f((s_0 \cap s_0) \cup (s_1 \cap s_0))$ can only be oriented by RPQOs with precedence $\cup > \cap$. Since no RPQO can have both $\cap > \cup$ and $\cup > \cap$, no RPQO can orient the entire path from the root; `REST` must therefore reject the rewrite. On the other hand, the rewrite to the right child can be oriented by any RPQO where $s_0 > \emptyset$, $s_1 > \emptyset$, or $\cap > \emptyset$. The path from the root can thus continue down the right-hand side, as there are RPQOs that satisfy both $\cap > \cup$ and the other conditions. The subsequent rewrites down the right-hand side do not impose any new constraints on the ordering: $f((s_0 \cap s_0) \cup \emptyset) >_{\mathcal{T}} f(s_0 \cap s_0) >_{\mathcal{T}} f(s_0)$ in all RPQOs.

[3] We omit the commutativity rules from this run, just to keep the diagram easy to visualize, but our implementation handles the example easily with or without them.

Similarly, REST will prove Example 2 but will reject Example 3 when the input OCA represents RPQO orderings. As shown in our benchmarks (Table 2 of Sec. 7), Example 3 is solved by REST with an OCA for the Knuth-Bendix term ordering family.

3.3 Integrating an External Oracle

Finally, to tackle Challenge 5, we turn to the (so far ignored) third parameter of the algorithm, the external oracle $\mathcal{E}$. In the example variant presented at the end of Sec. 2, such a function might supply the rewrite step $s_0 \cap s_1 \to \emptyset$ by analysis of the logical assumption $|s_0 \cap s_1| = 0$, which goes beyond term-rewriting. More generally, any external solver capable of producing rewrite steps (equal terms) can be connected to our algorithm via $\mathcal{E}$. In our implementation in Liquid Haskell, we use the pre-existing *Proof by Logical Evaluation (PLE)* technique [52], which complements rewriting with the expansion of program function definitions, under certain checks made via SMT solving. Our only requirements on the oracle $\mathcal{E}$ are that it is bounded (finitely-branching) and strongly normalizing (cf. Sec. 5).

Our algorithm therefore flexibly allows the interleaving of term rewriting steps and external oracle steps; we avoid the potential for this interaction to cause non-termination by conditioning any further rewriting steps on the fact that the entire path (including the steps inserted by the oracle) can be oriented by at least one candidate term ordering.

The combination of our interface for defining term orderings via ordering constraint algebras, a search algorithm that effectively explores all rewrites enabled by the orderings, and the flexible possibility of combination with external solvers via the oracle parameter makes REST very adaptable and powerful in practice.

4 Well-Quasi-Orderings and the Ordering Constraint Algebra

Term orderings are typically defined as *strict well-founded* orderings; this requirement ensures that rewriting will obtain a normal form. However, as mentioned in Challenge 3, the restriction to strict orderings limits what can be achieved with rewriting. In this section we describe the derivation of well-quasi-orderings from strict orderings (Sec. 4.1) and introduce Knuth-Bendix quasi-orderings (Sec. 4.1.1) and recursive path quasi-orderings (Sec. 4.1.2), two novel term ordering families respectively based on the classical recursive path and Knuth-Bendix orderings. In addition, we formally introduce ordering constraint algebras (Sec. 4.2) and use them to develop an efficient ordering constraint algebra for RPQOs.

4.1 Well-Quasi-Orderings

We define well-quasi-orderings in the standard way.

▶ **Definition 5** (Well-Quasi-Orderings). *A relation $\geqslant$ is a quasi-order if it is reflexive and transitive. Given elements t and u in S, we say $t \approx u$ if $t \geqslant u$ and $u \geqslant t$. A quasi-order $\geqslant$ is also characterized as:*

1. *WQO, when for all infinite chains $x_1, x_2, \ldots$ there exists an $i, j, i < j$ such that $x_j \geqslant x_i$,*
2. *thin, when forall $t \in S$, the set $\{u \in S \mid t \approx u\}$ is finite, and*
3. *total, when for all $t, u \in S$ either $t \geqslant s$ or $s \geqslant t$.*

Well-quasi-orderings are not required to be antisymmetric, however the corresponding strict part of the ordering must be well-founded. Hence, a WQO derives a strict ordering over equivalence classes of terms; REST also requires that these equivalence classes are finite (i.e., the ordering is thin). With this requirement, REST guarantees termination by exploring only duplicate-free paths. Many simplification orderings can be converted into more permissive WQOs. Intuitively, given an ordering $>_o$ its quasi-ordering derivation also accepts equal terms, so we denote it as $\geqslant_o$. We next present two such derivations.

4.1.1 Knuth-Bendix Quasi-Orderings (KBQO)

The Knuth-Bendix ordering [31] is a well-known simplification ordering used in the Knuth-Bendix completion procedure. Here, we present a simplified version of the ordering, used by REST that is using ordering to only compare ground terms.

▶ **Definition 6.** *A weight function w is a function $\mathcal{F} \to \mathbb{N}$, where $w(f) > 0$ for all nullary functions symbols, and $w(f) = 0$ for at most one unary function symbol. w is compatible with a quasi-ordering $\geqslant_\mathcal{F}$ on $\mathcal{F}$ if, for any unary function f such that $w(f) > 0$, we have $f >_\mathcal{F} g$ for all g. $w(t)$ denotes the weight of a term t, such that $w(f(t_1, \ldots, t_n)) = w(f) + \sum_{1 \leqslant i \leqslant n} w(t_i)$.*

▶ **Definition 7** (Knuth-Bendix ordering (KBO) on ground terms)**.** *The Knuth-Bendix Ordering $>_{kbo}$ for a given weight function w and compatible precedence order $\geqslant_\mathcal{F}$ is defined as $f(t_1, \ldots, t_m) = t >_{kbo} u = g(u_1, \ldots, u_n)$ iff $w(t) \geqslant w(u)$, and 1) $w(t) > w(u)$, or 2) $f >_\mathcal{F} g$, or 3) $f \geqslant_\mathcal{F} g$ and $(t_1, \ldots, t_m) >_{kbolex} (u_1, \ldots, u_n)$. Where $>_{kbolex}$ performs a lexicographic comparison using $>_{kbo}$ as the underlying ordering.*

Intuitively, KBO compares terms by their weights, using $\geqslant_\mathcal{F}$ and the lexicographic comparison as "tie-breakers" for cases when terms have equal weights. However, as $\geqslant$ is already a well-quasi-ordering on $\mathbb{N}$, we can derive a more general ordering by removing these tie-breakers and the need for a precedence ordering at all.

▶ **Definition 8** (Knuth-Bendix Quasi-ordering (KBQO))**.** *Given a weight function w, the Knuth-Bendix quasi-ordering $\geqslant_{kbo}$ is defined as $t \geqslant_{kbo} u$ iff $w(t) \geqslant w(u)$.*

The resulting quasi-ordering is simpler to implement and more permissive: $t >_{kbo} u$ implies $t \geqslant_{kbo} u$; and also enables arbitrary associativity and commutativity axioms as rewrite rules, since it only considers the weights of the function symbols and no structural components of the term. One caveat is that REST operates on well-quasi-ordering that are thin (Def. 5), so it can only consider KBQOs with $w(f) > 0$ for all unary function symbols f.

However, the fact that KBO and KBQO largely ignore the structure of the term in their comparison has a corresponding downside: it is not possible to orient distributivity axioms, or many other axioms that increase the number of symbols in a term. Therefore, we have found that a WQO derived from the recursive path ordering [15] to be more useful in practice.

4.1.2 Recursive Path Quasi-Orderings (RPQO)

In this section, we define a particular family of orderings designed to be typically useful for term-rewriting via REST. Our family of orderings is a novel extension of the classical notion of RPO, designed to also be more compatible with symmetrical rules such as commutativity and associativity (cf. Challenge 3, Sec. 2).

Like the classical RPO notions, our *recursive path quasi-ordering* (RPQO) is defined in three layers, derived from an underlying ordering on function symbols:

- The input ordering $\succcurlyeq_\mathcal{F}$ can be any quasi-ordering over $\mathcal{F}$.

- The corresponding *multiset quasi-ordering* $\succcurlyeq_{M(X)}$ lifts an ordering $\succcurlyeq_X$ over X to an ordering $\succcurlyeq_{M(X)}$ over multisets of X. Intuitively $T \succcurlyeq_{M(X)} U$ when U can be obtained from T by replacing zero or more elements in T with the same number of equal (with respect to $\succcurlyeq_X$) elements, and replacing zero or more elements in T with a finite number of smaller ones (Def. 9).
- Finally, the corresponding *recursive path quasi-ordering* $\succcurlyeq_{rpo}$ is an ordering over terms. Intuitively $f(ts) \succcurlyeq_{rpo} g(us)$ uses $\succcurlyeq_{\mathcal{F}}$ to compare the function symbols f and g and the corresponding $\succcurlyeq_{M(rpo)}$ to compare the argument sets ts and us (Def. 10).

Below we provide the formal definitions of the multiset quasi-ordering and recursive path quasi-ordering respectively generalized from the multiset ordering of [18] and the recursive path ordering [15] to operate on quasi-orderings. For all the three orderings, we write $x_l < x_r \doteq x_l \not\succcurlyeq x_r$ and $x_l > x_r \doteq x_l \succcurlyeq x_r \wedge x_r \not\succcurlyeq x_l$.

▶ **Definition 9** (Multiset Ordering). *Given a ordering $\succcurlyeq_X$ over a set X, the derived multiset ordering $\succcurlyeq_{M(X)}$ over finite multisets of X is defined as $T \succcurlyeq_{M(X)} U$ iff: 1) $U = \emptyset$, or 2) $t \in T \wedge u \in U \wedge t \approx u \wedge (T - t) \succcurlyeq_{M(X)} (U - u)$, or 3) $t \in T \wedge (T - t) \succcurlyeq_{M(X)} (U \setminus \{u \in U \mid u <_X t\})$.*

▶ **Definition 10** (Recursive Path Quasi-Ordering). *Given a basic ordering $\succcurlyeq_{\mathcal{F}}$, the recursive path quasi-ordering (RPQO) is the ordering $\succcurlyeq_{rpo}$ over $\mathcal{T}$ defined as follows: $f(t_1, \ldots, t_m) \succcurlyeq_{rpo} g(u_1, \ldots, u_n)$ iff: 1) $f >_{\mathcal{F}} g$ and $\{f(t_1, \ldots, t_m)\} >_{M(rpo)} \{u_1, \ldots, u_n\}$, or 2) $g >_{\mathcal{F}} f$ and $\{t_1, \ldots, t_m\} \succcurlyeq_{M(rpo)} \{g(u_1, \ldots, u_n)\}$, or 3) $f \approx g$ and $\{t_1, \ldots, t_m\} \succcurlyeq_{M(rpo)} \{u_1, \ldots, u_n\}$.*

▶ **Example 11.** As a first example, any RPQO $\succcurlyeq_{\mathcal{T}}$ used to restrict term rewriting will accept the rule $X + Y \to Y + X$, since $X + Y \succcurlyeq_{\mathcal{T}} Y + X$ always holds. Since the top level function symbol is the same $+ \approx +$, by Def. 10 (3) we need to show $\{X, Y\} \succcurlyeq_{M(rpo)} \{Y, X\}$. By Def. 9 (2) (choosing both t and u to be X), we can reduce this to $\{Y\} \succcurlyeq_{M(rpo)} \{Y\}$; the same step applied to y reduces this to showing $\emptyset \succcurlyeq_{M(rpo)} \emptyset$ which follows directly from Def. 9 (1).

From this example, we can see that both $X + Y \succcurlyeq_{rpo} Y + X$ and $Y + X \succcurlyeq_{rpo} X + Y$ hold, in this case independently of the choice of input ordering $\succcurlyeq_{\mathcal{F}}$ on function symbols. In our next example, the choice of input ordering makes a difference.

▶ **Example 12.** As a next example, we compare the terms $s(X) + Y$ and $s(X + Y)$. Now that the outer function symbols are *not* equal, the order relies on the ordering between $+$ and s. Let's assume that $+ >_{\mathcal{F}} s$. Now to get $s(x) + y \succcurlyeq_{rpo} s(X + Y)$, the first case of Definition 10 further requires $\{s(X)+Y\} >_{M(rpo)} \{X+Y\}$, which holds if $s(X)+y >_{rpo} X+Y$. The outermost symbol for both expressions is $+$, so we must check the multiset ordering: $\{s(X), Y\} >_{M(rpo)} \{X, Y\}$, which holds because by case splitting on the relation between s and X, we can show that $s(X)$ is always greater than X. In short, if $+ >_{\mathcal{F}} s$, then $s(X) + Y \succcurlyeq_{rpo} s(X + Y)$.

Developing on our *RPQO* notion (Def. 10), we consider the set of *all* such orderings that are generated by any total, well-quasi-ordering over the operators. We prove that such term orderings satisfy the termination requirements of Theorem 22. Concretely:

▶ **Theorem 13.** *If $\succcurlyeq_{\mathcal{F}}$ is a total, well-quasi-ordering, then 1) $\succcurlyeq_{rpo}$ is a well-quasi-ordering, 2) $\succcurlyeq_{rpo}$ is thin, and 3) $\succcurlyeq_{rpo}$ is thin well-founded.*

4.2 Ordering Constraint Algebras

Ordering constraint algebras play a crucial role in the REST algorithm (Sec. 3.2), by enabling the algorithm to simultaneously consider an entire family of term orderings during the exploration of rewrite paths. In this section, we provide a formal definition for ordering constraint algebras and describe the construction of an algebra for the RPQO.

▶ **Definition 14** (Ordering Constraint Algebra). *An Ordering Constraint Algebra (OCA) over the set of terms T and term ordering family Γ, is a tuple $\mathcal{A}_{(T,\Gamma)} \doteq \langle C, \gamma, \top, \mathit{refine}, \mathit{sat}\rangle$, where:*
1. *C, the constraint language, can be any non-empty set. Elements of C are called constraints, and are ranged over by c.*
2. *γ, the concretization function of $\mathcal{A}_{(T,\Gamma)}$, is a function from elements of C to subsets of Γ.*
3. *$\top$, the top constraint, is a distinguished constant from C, satisfying $\gamma(\top) = \Gamma$.*
4. *refine, the refinement function, is a function $C \to T \to T \to C$, satisfying (for all c, t_l, t_r) $\gamma(\mathit{refine}(c, t_l, t_r)) = \{\succcurlyeq \;\mid\; \succcurlyeq \in \gamma(c) \;\wedge\; t_l \succcurlyeq t_r\}$.*
5. *sat, the satisfiability function, is a function $C \to \mathit{Bool}$, satisfying (for all c) $\mathit{sat}(c) = \mathit{true} \;\Leftrightarrow\; \gamma(c) \neq \emptyset$.*

The functions $\top$, *refine*, and *sat* are all called from our REST algorithm (Figure 3) and must be implemented as (terminating) functions to implement REST. Specifically, REST instantiates the initial path with constraints $c = \top$. When a path can be extended via a rewrite application $t_l \to_R t_r$, REST refines the prior path constraints c to $c' \doteq \mathit{refine}(c, t_l, t_r)$. Then, the new term is added to the path only if the new constraints are satisfiable ($\mathit{sat}(c')$ holds); that is, if c' admits an ordering that orients the generated path. The function γ need *not* be implemented in practice; it is purely a mathematical concept that gives semantics to the algebra.

Given terms T and a finite term ordering family Γ, a trivial OCA is obtained by letting $C = \mathcal{P}(\Gamma)$, and making γ the identity function; straightforward corresponding elements $\top$, *refine*, and *sat* can be directly read off from the constraints in the definition above.

However, for efficiency reasons (or in order to support potentially infinite sets of orderings, which our theory allows), tracking these sets symbolically via some suitably chosen constraint language can be preferable. For example, consider lexicographic orderings on pairs of constants, represented by a set T of terms of the form $p(q_1, q_2)$ for a fixed function symbol p and q_1, q_2 chosen from some finite set of constant symbols Q. We choose the term ordering family $\Gamma = \{\succcurlyeq_{lex(\succcurlyeq)} \mid \succcurlyeq \textit{ is a total order on } Q\}$ writing $\succcurlyeq_{lex(\succcurlyeq)}$ to mean the corresponding lexicographic ordering on $p(q_1, q_2)$ terms generated from an ordering $\succcurlyeq$ on Q.

A possible OCA over these T and Γ can be defined by choosing the constraint language C to be *formulas*: conjunctions and disjunctions of atomic constraints of the forms $q_1 > q_2$ and $q_1 = q_2$ prescribing conditions on the underlying orderings on Q. The concretization γ is given by $\gamma(c) = \{\succcurlyeq_{lex(\succcurlyeq)} \mid \succcurlyeq \;\textit{ satisfies } c\}$, i.e., a constraint maps to all lexicographic orders generated from orderings of Q that satisfy the constraints described by c, defined in the natural way. We define $\top$ to be e.g., $q = q$ for some $q \in Q$. A satisfiability function *sat* can be implemented by checking the satisfiability of c as a formula. Finally, by inverting the standard definition of lexicographic ordering, we define:

$$\mathit{refine}(c, p(q_1, q_2), p(r_1, r_2)) = c \wedge (q_1 > r_1 \vee (q_1 = r_1 \wedge q_2 > r_2))$$

Using this example algebra, suppose that REST explores two potential rewrite steps $p(a_1, a_2) \to p(b_1, a_2) \to p(a_1, a_1)$. Starting from the initial constraint $c_0 = \top$, the constraint for the first step $c_1 \doteq \mathit{refine}(c_0, p(a_1, a_2), p(b_1, a_2)) = a_1 > b_1 \vee (a_1 = b_1 \wedge a_2 > a_2)$ is satisfiable, e.g., for any total order for which $a_1 > b_1$. However, considering the subsequent step, the refined constraint $c_2 \doteq \mathit{refine}(c_1, p(b_1, a_2), p(a_1, a_1))$, computed as $c_2 = c_1 \wedge (a_2 >$

$a_2 \vee (a_2 = a_2 \wedge b_1 > a_1))$ is no longer satisfiable. Note that this allows us to conclude that there is no lexicographic ordering allowing this sequence of two steps, even without explicitly constructing any orderings.

We now describe an OCA for RPQOs (Sec. 4.1.2), based on a compact representation of sets of these orderings.

An Ordering Constraint Algebra for $\succcurlyeq_{rpo}$

The OCA for RPQOs enables their usage in REST's proof search. One simple but computationally intractable approach would be to enumerate the entire set of RPQOs that orient a path; continuing the path so long as the set is not empty. This has two drawbacks. First, the number of RPQOs grows at an extremely fast rate with respect to the number of function symbols; for example there are $6,942$ RPQOs describing five function symbols, and $209,527$ over six [29]. Second, most of these orderings differ in ways that are not relevant to the comparisons made by REST.

Instead, we define a language to succinctly describe the set of candidate RPQOs, by calculating the minimal constraints that would ensure orientation of the path of terms; REST continues so long as there is some RPQO that satisfies the constraints. Crucially the satisfiability check can be performed effectively using an SMT solver without actually instantiating any orderings.

Before formally describing the language, we begin with some examples, showing how the ordering constraints could be constructed to guide the termination check of REST.

▶ **Example 15** (Satisfiability of Ordering Constraints)**.** Consider the following rewrite path given by the rules $r_1 \doteq f(g(X), Y) \to g(f(X, X))$ and $r_2 \doteq f(X, X) \to f(k, X)$:

$$f(g(h), k) \to_{r_1} g(f(h, h)) \to_{r_2} g(f(k, h))$$

To perform the first rewrite REST has to ensure that there exists an RPQO $\succcurlyeq_{rpo}$ such that $f(g(h), k) \succcurlyeq_{rpo} g(f(h, h))$. Following from Definition 10, we obtain three possibilities:

1. $f >_{\mathcal{F}} g$ and $\{f(g(h), k)\} >_{M(rpo)} \{f(h, h)\}$, or
2. $g >_{\mathcal{F}} f$ and $\{g(h), k\} \succcurlyeq_{M(rpo)} \{g(f(h, h))\}$, or
3. $f \approx g$ and $\{g(h), k\} \succcurlyeq_{M(rpo)} \{f(h, h)\}$.

We can further simplify these using the definition of the multiset quasi-ordering (Def. 9). Concretely, the multiset comparison of (1) always holds, while the multiset comparisons of (2) and (3) reduce to $k >_{\mathcal{F}} f \wedge k >_{\mathcal{F}} g \wedge k >_{\mathcal{F}} h$. Thus, we can define the exact constraints c_0 on $\succcurlyeq_{rpo}$ to satisfy $f(g(h), k) \succcurlyeq_{rpo} g(f(h, h))$ as

$$c_0 \doteq f >_{\mathcal{F}} g \vee (k >_{\mathcal{F}} f \wedge k >_{\mathcal{F}} g \wedge k >_{\mathcal{F}} h)$$

Since there exist many quasi-orderings satisfying this formula (trivially, the one containing the single relation $f >_{\mathcal{F}} g$), the first rewrite is satisfiable.

Similarly, for the second rewrite, the comparison $g(f(z, z)) \succcurlyeq_{rpo} g(f(k, z))$ entails the constraints $c_1 \doteq z \succcurlyeq_{\mathcal{F}} k$. To perform this second rewrite the conjunction of c_0 and c_1 must be satisfiable. Since the second disjunct of c_0 contradicts c_1, the resulting constraints $f >_{\mathcal{F}} g \wedge z \succcurlyeq_{\mathcal{F}} k$ is satisfiable by an RPQO, thus the path is satisfiable.

▶ **Example 16** (Unsatisfiable Ordering Constraint)**.** As a second example, consider the rewrite rules $r_1 \doteq f(x) \to g(s(x))$ and $r_2 \doteq g(s(x)) \to f(h(x))$. These rewrite rules can clearly cause divergence, as applying rule r_1 followed by r_2 will enable a subsequent application of r_1 to a larger term. Now let's examine how our ordering constraint algebra can show the unsatisfiability of the diverging path:

$$f(z) \to_{r_1} g(s(z)) \not\to_{r_2} f(h(z))$$

$f(z) \succcurlyeq_{rpo} g(s(z))$ requires $c_0 \doteq f > g \wedge f > s$ which is satisfiable, but $g(s(z)) \succcurlyeq_{rpo} f(h(z))$ requires $c_1 \doteq (g \geqslant f \wedge g \geqslant h) \vee (g \geqslant f \wedge s \geqslant h) \vee (s > f \wedge s > h)$, which, although satisfiable, conflicts with c_0. Since no $RPQO$ can satisfy both c_0 and c_1, the rewrite path is unsatisfiable.

Having primed intuition through the examples, we now present a way to compute such constraints. First, it is clear that we can define an RPQO based on the precedence over symbols $\mathcal{F}$. Therefore, we define our language of constraints to include the standard logical operators as well as atoms representing the relations between elements of $\mathcal{F}$, as:

$$C_{\mathcal{F}} \doteq f >_{\mathcal{F}} g \mid f \approx g \mid C_{\mathcal{F}} \wedge C_{\mathcal{F}} \mid C_{\mathcal{F}} \vee C_{\mathcal{F}} \mid \top \mid \bot$$

Next, we lift our definition of $RPQO$ and the multiset quasi-ordering to derive functions: $\mathit{rpo} : \mathcal{T} \to \mathcal{T} \to C_{\mathcal{F}}$, and $\mathit{mul} : (\mathcal{T} \to \mathcal{T} \to C_{\mathcal{F}}) \to M(\mathcal{T}) \to M(\mathcal{T}) \to C_{\mathcal{F}}$. rpo is derived by a straightforward translation of Def. 10:

$$\begin{aligned} \mathit{rpo}(f(t_1,\ldots,t_m), g(u_1,\ldots,u_n)) = \quad & f >_{\mathcal{F}} g \quad \wedge \quad \mathit{mul'}(\mathit{rpo}, \{f(t_1,\ldots,t_m)\}, \{u_1,\ldots,u_n\}) \vee \\ & g >_{\mathcal{F}} f \quad \wedge \quad \mathit{mul}(\mathit{rpo}, \{t_1,\ldots,t_m\}, \{g(u_1,\ldots,u_n)\}) \vee \\ & f \approx g \quad \wedge \quad \mathit{mul}(\mathit{rpo}, \{t_1,\ldots,t_m\}, \{u_1,\ldots,u_n\}) \end{aligned}$$

where $\mathit{mul'}$ is the strict multiset comparison: $\mathit{mul'}(f,T,U) = \mathit{mul}(f,T,U) \ \wedge \ \neg\mathit{mul}(f,U,T)$. $\neg : C_{\mathcal{F}} \to C_{\mathcal{F}}$ inverts the constraints, with $\neg(f >_{\mathcal{F}} g) = f \approx g \vee g >_{\mathcal{F}} f$ and $\neg(f \approx g) = f >_{\mathcal{F}} g \vee g >_{\mathcal{F}} f$; the other cases are defined in the typical way.

The definition for mul is more complex. Recall that $T \succcurlyeq_{M(X)} U$ when U can be obtained from T by replacing zero or more elements in T with the same number of equal (with respect to $\succcurlyeq_X$) elements, and by replacing zero or more elements in T with a finite number of smaller ones. Therefore each justification for $\{t_1,\ldots,t_m\} \succcurlyeq_{M(X)} \{u_1,\ldots,u_n\}$ can be represented by a bipartite graph with nodes labeled $t_1,\ldots,t_m$ and $u_1,\ldots,u_n$, such that:

1. Each node u_i has exactly one incoming edge from some node t_j.
2. If a node t_i has exactly one outgoing edge, it is labeled either `GT` or `EQ`.
3. If a node t_i has more than one outgoing edge, it is labeled `GT`.

$\mathit{mul}(f, \{t_1,\ldots,t_m\}, \{u_1,\ldots,u_n\})$ generates all such graphs: for each graph converts each labeled edge $(t, u, \texttt{EQ})$ to the formula $f(t,u) \wedge f(u,t)$, each edge $(t,u,\texttt{GT})$ to the formula $f(t,u) \wedge \neg f(u,t)$, and finally joins the formulas for the graph via a conjunction. The resulting constraint is defined to be the disjunction of the formulas generated from all such graphs.

Having defined the lifting of recursive path quasi-orderings to the language of constraints, we define our ordering constraint algebra $\mathcal{A}_{(\mathcal{T},\Gamma)}$ as the tuple $\langle C_{\mathcal{F}}, \top, \mathit{refine}, \gamma, \mathit{sat}\rangle$ where:

- $\mathit{refine}(c,t,u) = c \wedge \mathit{rpo}(t,u)$,
- Γ is the set of all RPQOs,
- $\gamma(c)$ is the set of RPQOs derived from the underlying quasi-orders $\succcurlyeq_{\mathcal{F}}$ that satisfy c, and
- $\mathit{sat}(c) = \mathit{true}$ if and only if there exists a quasi-order $\succcurlyeq_{\mathcal{F}}$ satisfying c.

That $\mathcal{A}_{(\mathcal{T},\Gamma)}$ is an OCA, i.e., satisfies the requirements of Def. 14, follows by construction. Namely, the function $\mathit{rpo}(t,u)$ produces constraints c such that, for any RPQO $\succcurlyeq_{rpo}$, $t \succcurlyeq_{rpo} u$ if and only if its underlying ordering $\succcurlyeq_{\mathcal{F}}$ satisfies c.

Having shown that using RPQOs as a term ordering is useful for theorem proving, satisfies the necessary properties for `REST`, and admits an efficient ordering constraint algebra, we continue our formal work by stating and proving the metaproperties of `REST`.

5 REST Metaproperties: Soundness, Completeness, and Termination

We now present the correctness, completeness, and termination of the REST algorithm defined in Figure 3. Here we only state the formal results; the detailed proofs can be found in [26]. Our formalism of rewriting is standard; based on that of Klop [30] (details in our extended version [26]). For a set of rewrite rules R, we $v \to_R w$ iff $v \to_r w$ for some $r \in R$. For oracle functions (from terms to sets of terms) $\mathcal{E}$, we write $t \to_{\mathcal{E}} t'$ *iff* $t' \in \mathcal{E}(t)$. We write $t \to_{R+\mathcal{E}} t'$ if $t \to_R t'$ or $t \to_{\mathcal{E}} t'$. For a relation $\to$ we write $\to^*$ for its reflexive, transitive closure. A *path* $t_1, \ldots, t_n$ is an indexed list of terms. A binary relation $\succcurlyeq$ *orients* a path $t_1, \ldots, t_n$ if $\forall i, 1 \leq i < n, t_i \ \succcurlyeq \ t_{i+1}$.

Soundness of REST means that any term of the output ($u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$) can be derived from the original input term by some combination of term rewriting steps from R and steps via the oracle function $\mathcal{E}$ (in other words, $t_0 \to^*_{R+\mathcal{E}} u$).

▶ **Theorem 17** (Soundness). *For all R, u, and t_0, if $u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$, then $t_0 \to^*_{R+\mathcal{E}} u$.*

Completeness of REST would naïvely be that, for any terms t_0 and u, if $t_0 \to^*_{R+\mathcal{E}} u$ then u is in our output ($u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$). This result doesn't hold by design, since REST explores only paths permitted by at least one instance of its input term ordering family. We prove this *relative* completeness result in two stages. First (Theorem 18), we show that completeness always holds if all steps only involve the external oracle. Second (Theorem 19), we prove relative completeness of REST with respect to the provided term ordering family.

▶ **Theorem 18** (Completeness w.r.t. $\mathcal{E}$). *For all R, u, t_0, if $t_0 \to^*_{\mathcal{E}} u$, then $u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$.*

▶ **Theorem 19** (Relative Completeness). *For all R, u, and t_0, if $t_0 \to^*_{R+\mathcal{E}} u$ and there exists an ordering $\succcurlyeq \in \gamma(\top)$ that orients the path justifying $t_0 \to^*_{R+\mathcal{E}} u$, then $u \in \mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$.*

Termination of REST requires conditions on the external oracle $\mathcal{E}$ and the ordering constraint algebra $\mathcal{A}$. Next, we formally define these requirements and state termination of REST.

▶ **Definition 20** (Well-Founded $\mathcal{A}$). *For ordering constraint algebras $\mathcal{A} = \langle C, \top, \mathit{refine}, \mathit{sat}, \gamma\rangle$, for $c, c' \in C$, we say c' strictly refines c (denoted $c' \sqsubset_{\mathcal{A}} c$) if $c' = \mathit{refine}(c, t, u)$ for some terms t and u, and $\gamma(c') \subset \gamma(c)$. Then, we say $\mathcal{A}$ is well-founded if $\sqsubset_{\mathcal{A}}$ is.*

Down every path explored by REST, the tracked constraint is only ever refined; well-foundedness of $\mathcal{A}$ guarantees that finitely many such refinements can be strict.

We note that if the OCA describes a finite set of orderings, then it is trivially well-founded: $\subset$ is well-founded on finite sets. For example, the ordering constraint algebra for RPQOs (Sec. 4.2) is well-founded when the set of functions symbols $\mathcal{F}$ is finite, as there are a only a finite number of possible RPQOs over a finite set of function symbols.

▶ **Definition 21** (Normalizing & Bounded $\mathcal{E}$). *A relation $t_l \to t_r$ is normalizing if it does not admit an infinite path and bounded if for each t_l it only admits finite t_r.*

Note that any deterministic, terminating external oracle is both normalizing and bounded.

▶ **Theorem 22** (Termination). *For any finite set of rewriting rules R, if: 1) $\to_{\mathcal{E}}$ is normalizing and bounded, 2) $\mathcal{A}$ is well-founded, and 3) the refine and sat functions of $\mathcal{A}$ are decidable (implemented to always-terminate), then, for all terms t_0, $\mathsf{REST}(\mathcal{A}, R, t_0, \mathcal{E})$ terminates.*

```
{-@ example1 :: s0 : Set → { s1 : Set | IsDisjoint s0 s1 } → f : (Set → a) → { f ((s0
      \/ s1) /\ s0)  = f s0 } @-}
example1 :: Set → Set → (Set → a) → Unit
example1 s0 s1 f =
       f ((s0 \/ s1) /\ s0)         ? distribUnion s0 s1 s0
   === f ((s0 /\ s0) \/ (s1 /\ s0)) ? idemInter s0
   === f (s0 \/ (s1 /\ s0))         ? symmInter s1 s0
   === f (s0 \/ (s0 /\ s1))         -- Disjoint
   === f (s0 \/ emptySet)           ? emptyUnion s0
   === f s0
   *** QED
```

Figure 5 Liquid Haskell version of the proof from Example 1.

6 Implementation of REST

We implemented REST as a standalone library, of 2337 lines of Haskell code (Sec. 6.1) and we integrated this library into Liquid Haskell [51] (Sec. 6.2) to automate *lemma* applications.

6.1 The REST Library

Our REST library is available on Hackage [25] and can be used directly by other Haskell projects. The library is designed modularly; for example, a client of the library can decide to use REST only for comparing terms via an OCA, without also using the proof search algorithm of Sec. 3.2. In addition, our library has a small code footprint and can be used with or without external solvers, making it ideal for integration into existing program analysis tools.

Furthermore, we include in the library built-in helper utilities for encoding and solving constraints on term orderings. Although the library enables integration of arbitrary solvers; it provides several built-in solvers for constraints on finite WQOs and also provides an interface for solving constraints with external SMT solvers. These utilities comprise the majority of the code in the REST library (1369 out of the 2337 lines).

Our implementation defines the OCA interface of Sec. 4.2 and provides three built-in instances for RPQOs, LPQOs (derived from the Lexicographic path ordering), and KBQOs (Sec. 4.1.1). The helper utilities included in the library enable a concise implementation of these OCAs: the three OCA implementations consist of 200 lines of code in total.

To facilitate debugging and evaluation of OCAs, the library also provides an executable that visualizes the rewrite paths that REST explores when using the OCA to compute the rewrite paths from a given term. For example, Figure 4 was produced using this functionality.

6.2 Integration of REST in Liquid Haskell

We used REST to automate lemma application in Liquid Haskell. Here we start with a brief overview of Liquid Haskell (Sec. 6.2.1), then present how REST is used to automate lemma instantiations (Sec. 6.2.2) and how it interacts with Liquid Haskell's automation (Sec. 6.2.3).

6.2.1 Liquid Haskell and Program Lemmas

Liquid Haskell performs program verification via *refinement types* for Haskell; function types can be annotated with refinements that capture logical/value constraints about the function's parameters, return value and their relation. For example, the function `example1` in Figure 5 ports the set example of Example 1 to Liquid Haskell, without any use of REST. User-defined lemmas amount to nothing more than additional program functions, whose refinement types express the logical requirements of the lemma. The first line of the figure is special comment syntax used in Liquid Haskell to introduce refinement types; it expresses that the first parameter `s0` is unconstrained, while the second `s1` is refined in terms of `s0`: it must be some value such that `IsDisjoint s0 s1` holds. The refinement type on the (unit) return value expresses the proof goal; the body of the function provides the proof of this lemma. The proof is written in equational style; the `?` annotations specify lemmas used to justify proof steps [50]. The penultimate step requires no lemma; the verifier can discharge it based on the refinement on the `s1` parameter.

6.2.2 REST for Automatic Lemma Application in Liquid Haskell

We apply `REST` to automate the application of equality lemmas in the context of Liquid Haskell. The basic idea is to extract a set of rewrite rules from a set of refinement-typed functions, each of which must have a refinement type signature of the following shape:

```
{-@ rrule :: x₁:t₁ → ... → xₙ:tₙ → {v:() | eₗ = eᵣ } @-}
```

In particular, the equality $e_l = e_r$ refinement of the (unit) return value generates potential rewrite rules to feed to `REST`, in both directions. Let $FV(e)$ be the free variables of e, if $FV(e_r) \subseteq FV(e_l)$ and $e_l \notin \{x_1, \ldots, x_n\}$ then $e_l \to e_r$ is generated as a rewrite rule. Symmetrically, if $FV(e_l) \subseteq FV(e_r)$ and $e_r \notin \{x_1, \ldots, x_n\}$ then $e_r \to e_l$ is generated as a rewrite rule. These rewrite rules are fed to `REST` along with the current terms we are trying to equate in the proof goal; any rewrites performed by `REST` are fed back to the context of the verifier as assumed equalities.

`REST` is using Liquid Haskell to ensure that the rewrite rules are correct. The body of `rrule` provides an proof (machine-checked by Liquid Haskell) that the equality $e_l = e_r$ holds. Such proofs can themselves use `REST`'s rewrites, but mutual dependencies are not permitted, e.g., if $\texttt{rrule}_1$ is proved using $\texttt{rrule}_2$, then $\texttt{rrule}_2$'s proof cannot use $\texttt{rrule}_1$.

Selective Activation of Lemmas: Local and Global Rewrite Rules. In our Liquid Haskell extension, the user can activate a rewrite rule globally or locally, using the `rewrite` and `rewriteWith` pragmas, *resp.*. For example, with the below annotations

```
  {-@ rewrite global @-}   {-@ rewriteWith theorem [local] @-}
```

the rule `global` will be active when verifying every function in the current Haskell module, while the rule `local` is used only when verifying `theorem`.

Lemma Automation. Using our implementation, the same Example 1 proven manually in Figure 5 can be alternatively proven (with all relevant rewrite rules in scope) as follows:

```
{-@ example1 :: s0 : Set → { s1 : Set | IsDisjoint s0 s1 } →  f : (Set → a)
             → { f ((s0 \/ s1) /\ s0) = f s0 } @-}
example1 s0 s1 _ = ()
```

The proof is fully automatic: all required lemmas are handled by `REST`. Integrating `REST` into Liquid Haskell required around 500 lines of code, mainly for surface syntax.

6.2.3 Mutual PLE and REST Interaction

Liquid Haskell includes the *Proof by Logical Evaluation* (PLE) [52] tactic that automatically expands terminating functions. PLE expands function calls into single cases of their (possibly conditional) bodies exactly when the SMT can prove that a unique case definitely applies. In our implementation, PLE plays the role of its external oracle (cf. Sec. 3). Since PLE is proven terminating [52], the termination of this collaboration is also guaranteed (cf. Sec. 5).

Table 1 Comparison of `REST` with existing theorem provers. `LH+` is Liquid Haskell with rewriting. The potential outcomes are ✓, followed by the runtime, when the property is proved; **loop** when no answer is returned after 300 sec; and **fail** when the property cannot be proven. `Isa+` is `Isabelle/HOL` with `Sledgehammer`.

Property	LH+	Coq	Agda	Lean	Isabelle	Zeno	Isa+
`Diverge`	✓0.62s	loop	loop	fail	loop	✓0.47s	✓7.58s
`Plus AC`	✓1.13s	loop	loop	fail	fail	✓0.54s	✓4.30s
`Congruence`	✓0.69s	✓0.22s	✓26.10s	✓0.36s	✓3.86s	fail	✓4.39s

PLE takes as input a set $\mathcal{F}$ of (provably) terminating, user-defined function definitions that it iteratively evaluates. Meanwhile, `REST` is provided with the rewrite rules extracted from in-scope lemmas in the program (cf. Sec. 6.2.2); these two techniques can then generate paths of equal terms including steps justified by each technique. For example, consider the following simple lemma `countPosExtra`, stating that the number of strictly positive values in `xs ++ [y]` is the number in `xs`, provided that `y <= 0`, and a lemma stating that *countPos* of two lists appended gives the same result if their orders are swapped.

```
{-@ lm :: xs : [Int] → ys : [Int] → { countPos (xs ++ ys) = countPos (ys ++ xs) } @-}

{-@ rewriteWith countPosExtra [lm] @-}
{-@ countPosExtra :: xs : [Int] → {y : Int | y <= 0 } →
                     { countPos (xs ++ [y]) = countPos xs } @-}
countPosExtra :: [Int] → Int → ()
countPosExtra _ _ = () -- proof is fully automatic!
```

The proof requires rewriting `countPos(xs ++ [y])` first via lemma `lm` (by `REST`), expanding the definition of `++` twice (via PLE) to give `countPos(y:xs)`, and finally one more PLE step evaluating `countPos`, using the logical fact that `y` is not positive. Note that the first step requires applying an external lemma (out of scope for PLE) and the last requires SMT reasoning not expressible by term rewriting. The two techniques together allow for a fully automatic proof.

7 Evaluation

Our evaluation seeks to answer three research questions:

§ 7.1: How does `REST` compare to existing rewriting tactics?
§ 7.2: How does `REST` compare to E-matching based axiomatization?
§ 7.3: Does `REST` simplify equational proofs?

We evaluate `REST` using the Liquid Haskell implementation described in Sec. 6. In Sec. 7.1, we compare our implementation's rewriting functionality with that of other theorem provers, with respect to the challenges mentioned in Sec. 2. In Sec. 7.2, we compare against Dafny [35] by porting Dafny's calculational proofs to Liquid Haskell. Finally, in Sec. 7.3, we port proofs from various sources into Liquid Haskell both with and without rewriting, and compare the performance and complexity of the resulting proofs.

7.1 Comparison with Other Theorem Provers

To compare `REST` with the rewriting functionality of other theorem provers, we developed three examples to test the five challenges described in Sec. 2 and compare our implementation to that of other solvers. We chose to evaluate against `Agda` [41], `Coq` [11], `Lean` [5], `Isabelle/HOL` [40], and `Zeno` [46], as they are widely known theorem provers that either support a rewrite tactic, or use rewriting internally. `Agda`, `Lean`, and `Isabelle/HOL` allow user-defined rewrites. In `Lean` and `Isabelle/HOL`, the tactic for applying rewrite rules multiple times is called `simp`; for simplification. `Agda`, `Coq`, and `Isabelle/HOL`'s implementation of rewriting can diverge for nonterminating rewrite systems [11, 1, 40]. On the other hand, `Lean` enforces termination, at least to some degree, by ensuring that associative and commutative operators can only be applied according to a well-founded ordering [4]. `Zeno` [46] does not allow for user-defined rewrite rules, rather it generates rewrites internally based on user-provided axioms. `Sledgehammer` [38, 44, 43] is a powerful tactic supported by `Isabelle/HOL` that (on top of the built-in rewriting) dispatches proof obligations to various external provers and succeeds when any of the external provers succeed; this tactic operates under a built-in (customizable) timeout.

1. `Diverge` tests how the prover handles the challenges 1 and 5: restricting the rewrite system to ensure termination and integrating external oracle steps. This example encodes a single (terminating) rewrite rule $f(X) \to g(s(s(X)))$ and terminating, mutually recursive definitions for f and g. However, the combination of the rules and function expansions can cause divergence. The proof follows directly from the function definitions.
2. `Plus AC` tests the challenges 2 and 3 by encoding a task that requires a permissive term ordering. This example encodes `p`, `q`, and `r`, user-defined natural numbers, and requires that expressions such as `(p + q) + r` can be rewritten into different groupings such as `(r + q) + p`, via associativity and commutativity rules.
3. `Congruence` is an additional test to ensure that the implementation of the rewrite system is permissive enough to generate the expected result. This test evaluates a basic expected property, that the expressions $f(g(t))$ and $f(g'(t))$ can be proved equal if there exists a rewrite rule of the form $g(X) \to g'(X)$.

We present our results in Table 1. As expected, `Coq`, `Agda`, and `Isabelle/HOL` diverge on the first example, as they do not ensure termination of rewriting. `Lean` does not diverge, but it also fails to prove the theorem. Unsurprisingly, the commutativity axiom of `Plus AC` causes theorem provers that don't ensure termination of rewriting to loop. Although `Lean` ensures termination, it does not generate the necessary rewrite application in every case, because it orients associative-commutative rewriting applications according to a fixed order. With the exception of `Zeno`, all of the theorem provers tested were able to prove the necessary theorem for the final example. Our implementation succeeds on these three examples by implementing a permissive termination check based on non-strict orderings.

The only two tools that proved all three examples are our implementation and Isabelle's Sledgehammer. The latter combines many techniques which go beyond term rewriting. Nonetheless, our novel approach provides a clear and general formal basis for incorporation with a wide variety of verifiers and reasoning techniques (due to its generic definition and formal requirements) and comes with strong formal guarantees for such combinations. In particular, `REST` guarantees termination and relative completeness, which Sledgehammer (via its timeout mechanism) does not.

7.2 Comparison with E-matching

To evaluate `REST` against the E-matching based approach, we compared with Dafny [35], a state-of-the-art program verifier. Dafny supports equational reasoning via calculational proofs [36] and calculation with user-defined functions [2]. We ported the calculational proofs of [36] to Liquid Haskell, using rewriting to automatically instantiate the necessary axioms.

7.2.1 List Involution

Figure 6 shows an example taken directly from Dafny [36], proving that the reverse operation on lists is an involution, i.e., $\forall \text{xs}.\mathit{reverse(reverse(xs))} = \mathit{xs}$. In this example, both Liquid Haskell and Dafny operate on inductively defined lists with user-defined functions $++$ and `reverse`. The original Dafny proof goes through via the combination of a manual application of a lemma called `ReverseAppendDistrib` (stating that for all lists xs and ys, $reverse(xs ++ ys) = reverse(ys) ++ reverse(xs)$) and induction on the size of the list.

Using `REST`'s term rewriting, Liquid Haskell is able to simplify the proof, with PLE expanding the function definitions for `reverse` and `append`, and `REST` applying the necessary equality `reverse (reverse xs ++ [x]) = reverse [x] ++ reverse (reverse xs)`.

In Dafny, a similar simplification of the calculational proof is not possible; the proof fails if the manual equality steps are simply removed. We experimented further and found that the lemma `ReverseAppendDistrib` can be alternatively encoded as a user-defined axiom which, by itself, does not appear to cause trouble for E-matching, and with this change alone the proof succeeds without the need for this single lemma call. On the other hand, the equalities must still be mentioned for the calculational proof to succeed. Perhaps surprisingly, removing these intermediate equality steps caused Dafny to stall[4]; analysis with the Axiom Profiler [7] indicated the presence of a (rather complex) matching loop involving the axiom `ReverseAppendDistrib` in combination with axioms internally generated by the verifier itself. This illustrates that achieving further automation of such E-matching-based proofs is not straightforward, and can easily lead to performance difficulties due to matching loops which can be hard to predict and understand, even in this state-of-the-art verifier. By contrast, `REST` can automatically provide the necessary equality steps without risking divergence.

7.2.2 Set Properties

Figure 7 shows the Dafny and Liquid Haskell proofs for the implication $s_0 \cap s_1 = \emptyset \implies f((s_0 \cup s_1) \cap s_0) = f(s_0)$. Dafny uses a calculational proof to show the equality $(s_0 \cup s_1) \cap s_0 = s_0$, seemingly by applying distributivity. In fact, the distributivity aspect is not relevant to the proof; rather, the set equality in the proof syntax causes Dafny to instantiate the set extensionality axiom discharging the proof. It is for this reason that Dafny requires an extra proof step to prove $f((s_0 \cup s_1) \cap s_0) = f(s_0)$, as this term does not include an equality on sets, but rather on applications of f. Dafny's set axiomatization does not include the distributivity axiom, as such an axiom could easily lead to matching loops.

`REST`'s termination property allows arbitrary lemmas to be encoded as rewrite rules; in this case rewriting with the distributivity lemma can complete the proof.

In conclusion, we have shown that `REST`'s rewriting can be used as an alternative to E-matching based axiomatization. Furthermore, the termination guarantee of `REST` enables axioms that may give rise to matching loops to, instead, be encoded as rewrite rules.

7.3 Simplification of Equational Proofs

Finally, we evaluate how `REST` can simplify equational proofs. We chose to include the set example from [36] (described in Sec. 7.2.2), data structure proofs from [50], examples from the Liquid Haskell test suite, as well as our own case study. We developed each example in Liquid Haskell both with and without rewriting, and compared the timing and proof

[4] We include this version in the Appendix of our extended paper [26].

```
lemma LemmaReverseTwice(xs: List)
    ensures reverse(reverse(xs)) == xs;
{
  match xs {
    case Nil =>
    case Cons(x, xrest) =>
      calc {
        reverse(reverse(xs));
        reverse(append(reverse(xrest), Cons(x, Nil)));
        { ReverseAppendDistrib(reverse(xrest), Cons(x, Nil)); }
        append(reverse(Cons(x, Nil)), reverse(reverse(xrest)));
        { LemmaReverseTwice(xrest); }
        append(reverse(Cons(x, Nil)), xrest);
        append(Cons(x, Nil), xrest);
        xs;
      }
  }
}
```

(a) Calculation-style proof in Dafny, from [36].

```
{-@ involutionP :: xs:[a] → {reverse (reverse xs) == xs } @-}
{-@ rewriteWith involutionP [distributivityP] @-}
involutionP []     = ()
involutionP (x:xs) = involutionP xs
```

(b) An equivalent proof implemented in Liquid Haskell extended with `REST`.

Figure 6 List Involution proofs in Liquid Haskell and Dafny.

complexity. Each proof using rewriting was evaluated using each different ordering constraint algebras built-in to our Haskell `REST` library. The proofs in [50] were selected because they require induction, expansion of user-defined functions, and equational reasoning steps to prove properties about trees and lists. The examples from the Liquid Haskell test suite were taken to evaluate the rewriting across a range of representative proofs.

Our `DSL` case study evaluates the performance of our implementation using a larger set of rewrite rules, by verifying optimizations for a simple programming language, containing statements (i.e., print, sequence, branches, repeats and no-ops) and expressions (i.e., constants, variables, arithmetic and boolean expressions) using 23 rewrite rules. Our rewriting technique can prove the kind of equivalences used in techniques such as supercompilation [8, 54, 48], by encoding the basic equality axioms as rewrite rules and using them to prove more complicated theorems. A full list of the axioms and proved theorems are available in our extended version [26]. We note that we encoded arithmetic operations as uninterpreted SMT functions, so that the built-in arithmetic theory of the SMT does not aid proof automation.

We present our results in Table 2. By using rewriting, we were able to eliminate all but two of the non-inductive axiom instantiations, while maintaining a reasonable verification time. As expected, no ordering constraint algebra was able to complete all the proofs using rewriting; however, each proof could be verified with at least one of them.

The test cases `LH-FingerTree` and `LH-MapReduce` required manual axiom instantiations because the structure of the term did not match the rewrite rule for the axiom. `LH-MapReduce`, requires proving the identity `op (f (take n is)) (mapReduce n f op (drop n is)) = f is`. An inductive lemma application generates the background equality `mapReduce n f op (drop n is) = f (drop n is)`, and a rewrite matching the term `op (f (take n is)) (f (drop n is))` must be instantiated to complete the proof. However, since the background equality

```
lemma Proof<a>(s0: set<int>, s1: set<int>, f: set<int> → a)
   requires s0 * s1 == {}
   ensures f((s0 + s1) * s0) == f(s0) {
     calc {
       (s0 + s1) * s0; (s0 * s0) + (s1 * s0);
       s0;
     }
   }
```

(a) Proof in Dafny using built-in set axiomatization.

```
{-@ assume unionEmpty     :: ma : Set → {v : () | ma \/ emptySet = ma } @-}
{-@ assume intersectComm  :: ma : Set → mb : Set → {v : () | ma /\ mb = mb /\ ma } @-}
{-@ assume intersectSelf  :: s0 : Set → { s0 /\ s0 = s0 } @-}
{-@ assume unionIntersect :: s0 : Set → s1 : Set → s2 : Set
                          → { (s0 \/ s1) /\ s2 = (s0 /\ s2) \/ (s1 /\ s2) } @-}
{-@ rwDisjoint :: s0 : Set → {s1 : Set | IsDisjoint s0 s1} → { s0 /\ s1 = emptySet } @-}

{-@ example1 :: s0 : Set → { s1 : Set | IsDisjoint s0 s1 } →  f : (Set → a) →
                { f ((s0 \/ s1) /\ s0) = f s0 } @-}
example1 s0 s1 _ = ()
```

(b) An equivalent proof implemented in Liquid Haskell, with a user-defined axiomatization of sets.

Figure 7 Set Proofs in Liquid Haskell and Dafny.

is neither a rewrite rule nor an evaluation step, the necessary term `op (f (take n is)) (f (drop n is))` never appears. Therefore, it is necessary to manually instantiate the lemma. As future work, a limited form of E-matching [12] could address this issue in the general case.

In conclusion, we've shown that extending Liquid Haskell to use `REST` enables rewriting functionality not subsumed by existing theorem provers, that `REST` is effective for axiom instantiation, and that `REST` can simplify equational proofs.

8 Related Work

Theorem Provers & Rewriting. Term rewriting is an effective technique to automate theorem proving [27] supported by most standard theorem provers. § 7.1 compares, by examples, our technique with `Coq`, `Agda`, `Lean`, and `Isabelle/HOL`. In short, our approach is different because it uses user-specified rewrite rules to derive, in a terminating way, equalities that strengthen the SMT-decidable verification conditions required for program verification.

SMT Verification & Rewriting. Our rewrite rules could be encoded in SMT solvers as universally quantified equations and instantiated using *E-matching* [12], i.e., a common algorithm for quantifier instantiation. Without careful choice of user-specified *triggers*, E-matching can lead to hard-to-predict an unstable performance, including non-termination due to axioms generating new instantiations indefinitely in a *matching loop*. [34] refers to this unpredictable behavior of E-matching as the "the butterfly effect" and partially addresses it by detecting formulas that could give rise to simple matching loops. However, as we show in Sec. 7.2.1, guaranteeing termination in general remains subtle, fundamentally due to the fact that every equality generates a (potentially-infinite) equivalence class of terms available in the solver's search. Our approach circumvents unpredictability by using the terminating `REST` algorithm to instantiate the rewrite rules outside of the SMT solver.

Table 2 Results from simplification of proofs with rewriting. **Set-Dafny** is the set example from[36], **Set-Mono** describes a similar property. **List** and **Tree** are equational proofs from [50]. **DSL** is the program equivalence case study. The remaining proofs are from the Liquid Haskell test suite folder `tests/pos`, excluding those using only inductive or mutually inductive lemmas. **Orig.** is the number of non-inductive lemma applications in the original proof. **Cut** is the number of lemma applications that were removed by rewriting; where **Cut** is the same as **Orig.**, all non-inductive lemma applications have been removed. **Rules** is the number of axioms encoded as rewrite rules. **Time (Orig.)** is verification time in seconds for the original proof. **LPQO** and **KBQO** are OCAs derived from the Lexicographic Path Ordering and Knuth-Bendix ordering respectively, and **Fuel** is an OCA allowing up to 5 rewrite applications per proof goal.

Name	Orig.	Cut	Rules	Time				
				Orig.	RPQO	LPQO	KBQO	Fuel
Set-Dafny	4	4	5	1.11s	✓1.15s	✓1.19s	✗1.13s	✓1.22s
Set-Mono	7	7	4	1.16s	✗1.40s	✗1.41s	✓1.47s	✓1.60s
List	3	3	3	2.46s	✓3.17s	✗4.21s	✗2.24s	✓3.54s
Tree	3	3	3	1.61s	✓2.64s	✓3.40s	✓3.08s	✓3.12s
DSL	43	43	23	2.89s	✓5.46s	✗3.85s	✗4.19s	✓6.54s
LH-FingerTree	2	1	1	5.55s	✓5.60s	✓5.57s	✓5.64s	✓5.95s
LH-T1013	1	1	1	1.11s	✓1.06s	✓1.00s	✓1.02s	✓1.06s
LH-T1025	2	2	2	1.03s	✓1.05s	✓1.08s	✓1.07s	✓1.13s
LH-T1548	1	1	1	1.45s	✓1.33s	✓1.38s	✓1.32s	✓1.45s
LH-T1660	1	1	1	1.09s	✓1.12s	✓1.12s	✓1.12s	✓1.20s
LH-MapReduce	4	3	2	14.38s	✓29.50s	✓518.91s	✓28.49s	✗Timeout

Z3 [13] and `CVC4` [6] are state-of-the-art SMT solvers; both support theory-specific rewrite rules internally. Recent work [42] enables user-provided rewrite rules to be added to `CVC4`. However, using the SMT solver as a rewrite engine offers little control over rewrite rule instantiation, which is necessary for ensuring termination.

Rewriting in Haskell. Haskell itself has used various notions of rewriting for program verification. GHC supports the `RULES` pragma with which the user can specify unchecked, quantified expression equalities that are used at compile time for program optimization. [10] proposes Inspection Testing as a way to check such rewrite rules using runtime execution and metaprogramming, while [22] prove rewrite rules via metaprogramming and user-provided hints. In a work closely related to ours, Zeno [46] is using rewriting, induction, and further heuristics to provide lemma discovery and fully automatic proof generation of inductive properties. Unlike our approach, Zeno's syntax is restricted (e.g., it does not allow for existentials) and it does not allow for user-provided hints when automation fails. HALO [53] enables Haskell verification by converting Haskell into logic and using an SMT solver to verify user-defined formulas. However, this approach relies on SMT quantifiers to encode user functions, thus the solver can diverge and verification becomes unpredictable.

Termination of Rewriting and Runtime Termination Checking. Early work on proving termination of rewriting using simplification orderings is described in [15]. More recent work involves dependency pairs [3] and applying the size-change termination principle [33] in the context of rewriting [49]. Tools like AProVE [24] and NaTT [56] can statically prove the termination of rewriting. In contrast, `REST` is not focused on statically proving termination of rewriting; rather it uses a well-founded ordering to ensure termination at runtime. This approach enables integration of arbitrary external oracles to produce rewrite applications, as a static analysis is not possible in principle. Furthermore, our approach enables nonterminating rewriting systems to be useful: REST will still apply certain rewrite rules to satisfy a proof obligation, even if the rewrite rules themselves cannot be statically shown to terminate.

We used a well-quasi-ordering [32] because it enables rewriting to terms that are not strictly decreasing in a simplification ordering. WQOs are commonly used in online termination checking [37], especially for program optimization techniques such as supercompilation [9].

Equality Saturation. In our implementation, REST passes equalities to the SMT environment, ultimately used for *equality saturation* via an E-graph data structure [20]. Equality saturation has also been used for supercompilation[48]. REST does not currently exploit equality saturation (unless indirectly via its oracle). However, as future work we might explore local usage of efficient E-graph implementations. (e.g., [55]) for caching the equivalence classes generated via rewrite applications.

Associative-Commutative Rewriting. Traditionally, enforcing a strict ordering on terms prevents the application of rewrite rules for associativity or commutativity (AC); this problem motivates REST's use of well-quasi orders. However, another solution is to omit the rules and instead perform the substitution step of rewriting modulo AC. Termination of the resulting system can be proved using an AC ordering [17]; the requirement is that the ordering respects AC: for all terms t' AC-equivalent to t and u' AC-equivalent to u, $t > u$ implies $t' > u'$.

REST's use of well-quasi-orderings enables AC axioms to be encoded as rewrite rules, guaranteeing completeness if the AC-equivalence class of a term is a subset of the equivalence class induced by the ordering. This is a significant practical benefit as it does not require REST to identify AC symbols and treat them differently for unification.

However, treating AC axioms as rewrite rules can lead to an explosion in the number of terms obtained via rewriting. As future work, it could be possible to extend REST to support AC rewriting and unification in order to reduce the number of explicitly instantiated terms.

9 Conclusion

We presented REST, a novel approach to rewriting that uses an online termination check that simultaneously considers entire families of term orderings via Ordering Constraint Algebras. We defined our algebra on well-quasi orderings that are more permissive than standard simplification orderings and demonstrated how to derive well-quasi orderings from well-known simplification orderings. We proved correctness, relative completeness, and (online) termination of REST and implemented it as a small Haskell library suitable for integration with existing verification tools. To evaluate REST we integrated our implementation with Liquid Haskell and showed that the resulting system compares well with existing rewriting techniques and can substantially simplify equational proofs.

References

1 Agda Developers. *The Agda Language Reference, version 2.6.1*, 2020. Available electronically at `https://agda.readthedocs.io/en/v2.6.1/language/index.html`.

2 Nada Amin, K Rustan M Leino, and Tiark Rompf. Computing with an smt solver. In *International Conference on Tests and Proofs*, pages 20–35. Springer, 2014.

3 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1):133–178, April 2000. doi:10.1016/S0304-3975(99)00207-8.

4 Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean, Release 3.20.0*, September 2020. p 73. URL: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

5 Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich. *The Lean Reference Manual, Release 3.3.0*, 2018. URL: https://leanprover.github.io/reference/lean_reference.pdf.

6 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah. URL: http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf.

7 N. Becker, P. Müller, and A. J. Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2019*, LNCS, pages 99–116. Springer-Verlag, 2019.

8 Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. *SIGPLAN Not.*, 45(11):135–146, September 2010. doi:10.1145/2088456.1863540.

9 Maximilian Bolingbroke, Simon Peyton Jones, and Dimitrios Vytiniotis. Termination combinators forever. In *Proceedings of the 4th ACM symposium on Haskell*, pages 23–34, 2011.

10 Joachim Breitner. A promise checked is a promise kept: inspection testing. In Nicolas Wu, editor, *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 14–25. ACM, 2018. doi:10.1145/3242744.3242748.

11 The Coq Development Team. *The Coq Reference Manual, version 8.11.2*, 2020. Available electronically at http://coq.inria.fr/refman.

12 Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

13 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

14 Nachum Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979. doi:10.1016/0020-0190(79)90071-1.

15 Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical computer science*, 17(3):279–301, 1982.

16 Nachum Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 3(1-2):69–115, 1987.

17 Nachum Dershowitz, Jieh Hsiang, N Alan Josephson, and David A Plaisted. Associative-commutative rewriting. In *IJCAI*, pages 940–944, 1983.

18 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 188–202, Berlin, Heidelberg, 1979. Springer. doi:10.1007/3-540-09510-1_15.

19 David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005. doi:10.1145/1066100.1066102.

20 David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

21 Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Adding decision procedures to smt solvers using axioms with triggers. *Journal of Automated Reasoning*, 56(4):387–457, 2016.

22 Andrew Farmer, Neil Sculthorpe, and Andy Gill. Reasoning with the hermit: Tool support for equational reasoning on ghc core programs. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 23–34, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2804302.2804303`.

23 Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

24 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

25 Zachary Grannan. rest-rewrite: Rewriting library with online termination checking, 2022. URL: `https://hackage.haskell.org/package/rest-rewrite`.

26 Zachary Grannan, Niki Vazou, Eva Darulova, and Alexander J. Summers. Rest: Integrating term rewriting with program verification (extended version), 2022. `arXiv:2202.05872`.

27 Jieh Hsiang, Hélène Kirchner, Pierre Lescanne, and Michaël Rusinowitch. The term rewriting approach to automated theorem proving. *The Journal of Logic Programming*, 14(1):71–99, October 1992. `doi:10.1016/0743-1066(92)90047-7`.

28 Gerard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 30–45, USA, 1977. IEEE Computer Society. `doi:10.1109/SFCS.1977.9`.

29 OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2022. A000798: Number of different quasi-orders (or topologies, or transitive digraphs) with n labeled elements. URL: `https://oeis.org/A000798`.

30 J. W. Klop. *Term Rewriting Systems*, pages 1–116. Oxford University Press, Inc., USA, 1993.

31 Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

32 Joseph B Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory, Series A*, 13(3):297–305, November 1972. `doi:10.1016/0097-3165(72)90063-5`.

33 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 81–92, New York, NY, USA, 2001. Association for Computing Machinery. `doi:10.1145/360204.360210`.

34 K. R. M. Leino and Clément Pit-Claudel. Trigger Selection Strategies to Stabilize Program Verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 361–381, Cham, 2016. Springer International Publishing. `doi:10.1007/978-3-319-41528-4_20`.

35 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

36 K Rustan M Leino and Nadia Polikarpova. Verified calculations. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 170–190. Springer, 2013.

37 Michael Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566, pages 379–403. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. Series Title: Lecture Notes in Computer Science. `doi:10.1007/3-540-36377-7_17`.

38 Jia Meng and Lawrence C Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

39 P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, 2016.

40 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2020.

41 Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2008. Springer-Verlag.

42 Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. Syntax-guided rewrite rule enumeration for smt solvers. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 279–297, Cham, 2019. Springer International Publishing.

43 Lawrence C Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In *International Conference on Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.

44 Lawrence C Paulsson and Jasmin C Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *Proceedings of the 8th International Workshop on the Implementation of Logics (IWIL-2010), Yogyakarta, Indonesia. EPiC*, volume 2, 2012.

45 Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-c: a software analysis perspective. *Formal Aspects of Computing*, 27, October 2012. doi:10.1007/s00165-014-0326-7.

46 William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–421, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

47 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016. URL: https://www.fstar-lang.org/papers/mumon/.

48 Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480915.

49 René Thiemann and Jürgen Giesl. Size-Change Termination for Term Rewriting. In *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706, pages 264–278, March 2007. doi:10.1007/3-540-44881-0_19.

50 Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: equational reasoning in liquid Haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, pages 132–144, St. Louis, MO, USA, September 2018. Association for Computing Machinery. doi:10.1145/3242744.3242756.

51 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2628136.2628161.

52 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158141.

53 Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. Halo: Haskell to logic through denotational semantics. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 431–442, 2013.

54 Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990. doi:10.1016/0304-3975(90)90147-A.

55 Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

56 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Nagoya termination tool. In *Rewriting and Typed Lambda Calculi*, pages 466–475. Springer, 2014.