# REST: Integrating Term Rewriting with Program Verification

ANONYMOUS AUTHOR(S)

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. (3) integrating external oracles that allow steps that cannot be expressed as rewrite rules. We implemented REST as a Haskell library and incorporated it into Liquid Haskell's evaluation strategy, thus extending Liquid Haskell with rewriting rules. We evaluated our REST implementation by comparing it against both existing rewriting techniques and E-matching and by showing that it can be used to supplant manual lemma application in many existing Liquid Haskell proofs.

Additional Key Words and Phrases: term rewriting, program verification, theorem proving

## 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 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: 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 technique for instantiating universally quantified equalities.

For the wide range of practical program verifiers that are built upon SMT solvers (e.g., [Filliâtre and Paskevich 2013; Leino 2010; Müller et al. 2016; Signoles et al. 2012; Swamy et al. 2016; Vazou et al. 2014]), quantified equalities can naturally be expressed in the SMT solver's logic. However, relying solely on such solvers' E-matching techniques [Detlefs et al. 2005a] for quantifier instantiation (as the majority of these verifiers do) can lead to both non-termination and incompletenesses that may be unpredictable [Leino and Pit-Claudel 2016] and challenging to diagnose [Becker et al. 2019].

A classical alternative approach to automating equality reasoning is *term rewriting systems* [Huet 1977], 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 [Dershowitz 1987]. 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, all mainstream proof assistants (e.g., `Coq` [Coq Development Team 2020], `Isabelle/HOL` [Nipkow et al. 2020], `Lean` [Avigad et al. 2018]) provide automated, customizable term rewriting tactics.

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. For verifiers built around SMT, this provides equational reasoning with complementary strengths to E-matching-based techniques. While term rewriting in general does not guarantee termination, REST 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) We define a generalization of the well-established *recursive path ordering* (hereafter,

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

| *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* | $(x \cap y) \cup z$ | $=$ | $(x \cup z) \cap (y \cup z)$ |

Fig. 1. Set identities used for examples in this section. By convention, variables $x, y, z$ are implicitly quantified. We write the binary functions $\cup, \cap$ infix; along with (nullary) $\emptyset$ these are fixed function symbols.

RPO) technique [Dershowitz 1982] for termination of term rewriting systems, which we call *recursive path quasi-orderings* (hereafter, RPQOs), designed to accommodate common and important rules such as commutativity and associativity properties. (2) We dynamically and iteratively select custom RPQOs based on the terms encountered *during term rewriting itself*. (3) We allow 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 termination orderings to permit as many rewriting steps as possible while guaranteeing termination (Sec. 3).
(2) We formalize and prove key results for our technique: soundness, relative completeness, and termination (Sec. 4).
(3) We introduce and formalize well-quasi orderings (WQOs), that are more permissive than classical RPOs, and so let us prove more properties (Sec. 5).
(4) We provide an implementation of REST as an extension of Liquid Haskell, including efficient means of exploring candidate orderings (Sec. 6).
(5) We evaluate REST in three ways: comparing to other term rewriting tactics, to E-matching-based axiomatization, and substantially simplifying equational reasoning proofs (Sec. 7).

We discuss related work in Sec. 8; we begin (Sec. 2) by identifying four key problems that all need solving for a reliable and automatic integration of term rewriting into a program verification tool.

## 2 FOUR CHALLENGES FOR AUTOMATING EQUATIONAL REASONING

In this section, we consider the application of term rewriting to program verification and illustrate *four key challenges* that naturally arise. We illustrate each with simple verification goals involving mathematical set operators ($\emptyset$,$\cup$, $\cap$) as well as uninterpreted functions. The standard properties we will assume for the set operators in these examples are listed in Figure 1. The variables $x, y, z$ are implicitly quantified[1] in these rules. In formalizations of set theory, such properties may be assumed as (quantified) axioms, or may be proven as lemmas and then used in future proofs.

Term rewriting systems (defined formally in Sec. 4.1) are a standard approach for formally expressing and applying equational reasoning (rewriting terms via known identities). A term

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

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 pair $(x \cup \emptyset, x)$ denotes a rewrite rule $x \cup \emptyset \longrightarrow x$ that 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 corresponding 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.

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 $\emptyset \longrightarrow x \cap \emptyset$, and those that trivially lead to infinite derivations. With these exceptions, the first four identities induce rewrite rules from left-to-right (which we denote by e.g., *idem-inter*$\longrightarrow$, v.s. *idem-inter*$\longleftarrow$), while the remaining induce rewrite rules in both directions.

Next, we present a simple proof obligation taken from Leino and Polikarpova [2013] in the style of equational reasoning (*calculational proofs*) supported in the Dafny program verifier [Leino 2010].

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 $f((s_0 \cup s_1) \cap s_0) = f(s_0)$.*

*Equational Proof:*

$$\begin{array}{rcll} f((s_0 \cup s_1) \cap s_0) & = & f((s_0 \cap s_0) \cup (s_1 \cap s_0)) & (\textit{distrib-union}\longrightarrow) \\ & = & f(s_0 \cup (s_1 \cap s_0)) & (\textit{idem-inter}\longrightarrow) \\ & = & f(s_0 \cup \emptyset) & (\textit{disjointness ass.}\longrightarrow) \\ & = & f(s_0) & (\textit{empty-union}\longrightarrow) \end{array}$$

This manual proof closely follows the user annotations employed in the corresponding Dafny proof [Leino and Polikarpova 2013]; 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 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 $(s_0 \cup s_1) \cap s_2 \longrightarrow (s_0 \cap s_2) \cup (s_1 \cap s_2) \longrightarrow (s_0 \cup (s_1 \cap s_2)) \cap (s_2 \cup (s_1 \cap s_2)) \longrightarrow \ldots$

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

A classical approach to ensure the termination of a term rewriting system is to require that rewrite applications decrease the size of the term with respect to a well-founded order. A rather-flexible approach is that of *recursive path ordering* [Dershowitz 1982], which induces such a well-founded order $>_{\mathcal{T}}$ on terms $\mathcal{T}$ based on an underlying well-founded strict partial order $>$ on *function symbols*. Intuitively, this ordering uses $>$ to order terms with different top-level function symbols, combined with the properties of a *simplification order* [Dershowitz 1979] (e.g., compatibility with the subterm relation). Notably, an RPO does not necessarily restrict terms from sometimes rewriting into larger ones (more function symbols). For example, if one fixes $\cap > \cup$ in the underlying ordering, the left-to-right application of *distrib-union* would be permitted by the corresponding RPO. In fact, this yields a terminating rewrite system which allows all four steps of the proof in Example 1. However, while this particular RPO restriction of our term rewriting rules works well for Example 1, it is easy to find very similar examples for which it does not suffice.

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 $f((s_0 \cap s_1) \cup s_0) = f(s_0)$.*

*Equational Proof:*

$$
\begin{array}{rcll}
f((s_0 \cap s_1) \cup s_0) & = & f((s_0 \cup s_0) \cap (s_1 \cup s_0)) & \textit{(distrib-inter}\longrightarrow\textit{)} \\
 & = & f(s_0 \cap (s_1 \cup s_0)) & \textit{(idem-union}\longrightarrow\textit{)} \\
 & = & f(s_0 \cap (s_0 \cup s_1)) & \textit{(commut-union}\longrightarrow\textit{)} \\
 & = & f(s_0 \cap s_0) & \textit{(subset ass.}\longrightarrow\textit{)} \\
 & = & f(s_0) & \textit{(idem-inter}\longrightarrow\textit{)}
\end{array}
$$

Unfortunately, given the prior choice to order the function $\cap$ before $\cup$ in the underlying $>$, the corresponding RPO relation does not allow the first step of this proof (essentially, since $\cap$ is considered the larger function symbol, the increased complexity of the arguments to $\cap$ outweighs, in the RPO ordering, the decrease in complexity of the arguments to $\cup$). In fact, this particular RPO allows identity *distrib-union* to be applied only left-to-right, and *distrib-inter* only *right-to-left*, which is useless for Example 2; picking the alternative RPO relation generated by the opposite choice of $\cup > \cap$ instead allows this proof step but correspondingly fails to handle Example 1.

**Challenge 2:** Different term orderings are needed to solve different proof goals.

Furthermore, since RPOs are well-founded relations on terms, there are in fact *no* RPOs which permit the application of general commutativity/associativity properties such as *commut-union* in the proof above. Such reasoning steps are, however, ubiquitous when reasoning about either datatypes built into a programming language or mathematical types used to abstract them.

**Challenge 3:** Well-founded orderings rule out commutativity and associativity steps.

Finally, although equational reasoning is powerful enough for these examples, general verification problems necessarily involve logical entailments and theory 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.

**Challenge 4:** Program verification needs proof steps not expressible with term rewriting.

## 3 THE REST APPROACH

We develop REST to tackle the above four challenges, providing a flexible means of integrating expressive and guaranteed-terminating term rewriting into a verification tool. Based on Challenge 1, we borrow the core idea of the classical RPO technique for ensuring termination; we search for sequences of rewrite steps (hereafter, *rewrite paths*) such that some term ordering (e.g., an RPO) orients each consecutive pair of terms (hereafter, *orients the path*). By employing term orderings which preclude infinite paths, we can guarantee termination. However, as observed in Challenge 2, fixing such a term ordering up front can prevent necessary proof steps. Instead, our algorithm tracks an existential condition: it requires that an ordering that orients the path *exists*; the set of orderings that witness this existential may *shrink dynamically* as terms are added to a path.

Checking exhaustively for the existence of an ordering that orients a path can be an expensive or intractable problem. REST allows this to be avoided via an indirection. We define an abstraction that we call the Ordering Constraint Algebra (OCA) (formally defined in Sec. 4.2) which allows a

$$\mathsf{REST} : (\mathcal{R} \times \mathcal{T} \; \times \; (\mathcal{T} \to \mathcal{P}(\mathcal{T}))) \to \mathcal{P}(\mathcal{T})$$

```
REST(R, t₀, ℰ) =
   o := ∅;
   p := [([t₀], ⊤)];
   while{p is not empty}{
      pop(ts, c) from p;
      t := last ts;
      o := o ∪ {t};
      foreach {t′ such that t′ ∉ ts ∧ (t →_R t′ ∨ t′ ∈ ℰ(t))}{
         if {t′ ∈ ℰ(t) ∨ (t →_R t′ ∧ SAT(refine(c, t, t′)))}{
            push (ts ++ [t′], refine(c, t, t′)) to p
         }
      }
   }
   return o;
```

Fig. 2. The REST algorithm.

custom language of *constraints* to be used to symbolically represent conditions on the orderings that orient a path. For instance, the RPO orderings directing the path in Example 1 are those whose function ordering satisfies $\cap > \cup$. We call $c_1 = \cap > \cup$ the *ordering constraint* of the path in Example 1 (and in general use the name $c$ to range over ordering constraints). Similarly, the ordering required for the first step of Example 2 is $c_2 = \cup > \cap$, which is also satisfiable. But, if the path of Example 1 were extended with a term requiring the ordering constraint $c_2$, then the derived ordering constraints would be the conjunction: $c_{12} = c_1 \wedge c_2$. Since $c_{12}$ cannot be satisfied there exists no ordering that can orient this path. REST uses three functions on constraints that an OCA must define: (1) $\mathsf{SAT}(c)$ checks satisfiability of the constraint $c$, (2) $\mathit{refine}(c, t_l, t_r)$ extends the constraint $c$ to further capture the ordering requirements for $t_l$ to be greater than $t_r$, and finally, (3) $\top$ is the empty ordering constraint. In this way, our algorithm remains completely generic over both the initial set of candidate orderings and the choice of OCA employed.

Figure 2 presents our core REST algorithm. The algorithm takes three explicit parameters; it is also implicitly parameterized by the set of candidate term orderings and an OCA over them, as discussed above. The algorithm's first 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 second parameter $t_0$ is the term from which term rewrites are sought. $\mathcal{E}$ 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 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* candidate ordering allows the rewrite path; this condition, along with the flexibility to dynamically change term orderings on the fly, addresses Challenge 1 and Challenge 2 above (each candidate ordering is required not to admit infinite paths).

Our algorithm 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 (down any rewrite path) equal to $t_0$ via the rewriting paths explored.

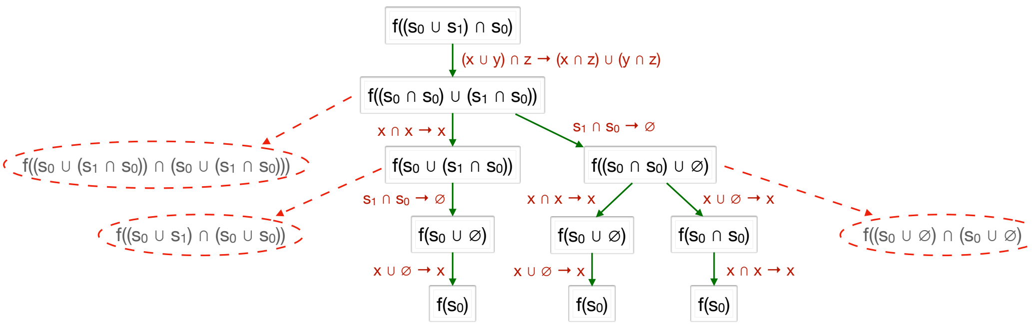


Fig. 3. A visualization of REST running on the term from Example 1. Each path through the tree shown represents a rewrite path uncovered by our algorithm; the edge labels show the rewrite rule applied. The red dotted lines indicate rewrite steps rejected by REST.

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 $t \longrightarrow t'$ is handled in the **if** check of REST. This check is to guarantee termination, by enforcing that we only add rewrite steps which would leave the extended path still justifiable by *some* term ordering, as enforced by the SAT check.

Figure 3 visualizes the rewrite paths explored by our algorithm for a run corresponding to the problem from Example 1[2]. 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).

Challenge 3 motivates that even the full set of candidate RPOs (which are each well-founded term orderings) may not always be a flexible enough choice for examples that require commutativity or associativity properties (such as Example 2 above). To this end, in Sec. 5 we develop a generalization of the RPO concept, lifting both the (well-founded) input ordering on function symbols, and the generated (well-founded) ordering on terms to the more permissive notion of *WQOs*, which admit commutativity and associativity steps. In practice, this turns out to be a very powerful tool since our generalized RPOs are rather permissive; as we show in Sec. 7, this means that we easily solve example problems in practice. In Sec. 5 we prove that all key classical properties of RPOs indeed lift to our generalized version, which we also employ in our implementation.

Finally, to tackle Challenge 4, 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 $s_0 \cap s_1 \longrightarrow \emptyset$ 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 [Vazou et al. 2017], 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 the binary relation on terms generated by calls to it is bounded (finitely-branching) and strongly normalizing (cf. Sec. 4). Our

[2]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.

algorithm therefore flexibly allows the interleaving of term rewriting steps and those justified by the external oracle; 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 explained by at least one of our candidate orderings.

The combination of our search that selects candidate orderings on the fly, our generalized RPO notion as a powerful default set of orderings to use in practice, and the flexible possibility of combination with external solvers via the oracle parameter makes REST very adaptable and powerful in practice. We turn next to the formal results underpinning these practical claims.

## 4 REST METAPROPERTIES: SOUNDNESS, COMPLETENESS, AND TERMINATION

We now present the metaproperties of the REST algorithm defined in Figure 2. We show correctness (Theorem 4.2), completeness (Theorem 4.4) relative to the ordering relations checked, and termination (Theorem 4.7) which requires that the checked ordering relations be decidable and well-founded on duplicate-free paths (no term occurs more than once, e.g., those that REST generates).

### 4.1 Formal Definitions

Our formalism of rewriting is standard; based on the terminology of Klop [1993]. Our language consists of the following:

(1) An infinite set of meta-variables (the variables for rewrite rules) $\mathcal{V}$ with elements $X$, $Y$, ....
(2) A finite set of operators $\mathcal{F}$ with elements $f, g, \ldots, x, y, \ldots$
Each operator is associated with a fixed numeric arity and types for its arguments and result (elided here, for simplicity). As common we use the variables $x$, $y$ to for zero-arity operators.
(3) A set of terms $\mathcal{T}$ with elements $t, u, \ldots$ inductively defined as follows: (a) $X \in \mathcal{V} \Rightarrow X \in \mathcal{T}$ and (b) $f \in \mathcal{F}$, $f$ has arity $n$, $t_1, \ldots, t_n \in \mathcal{T} \Rightarrow f(t_1, \ldots, t_n) \in \mathcal{T}$.

We use $FV(t)$ to refer to the set of meta-variables in $t$. A term $t$ is *ground* if $FV(t) = \emptyset$.

A *substitution* $\sigma \subseteq \mathcal{V} \times \mathcal{T}$ is a mapping from meta-variables to terms. We write $\sigma \cdot t$ to denote the simultaneous application of the substitution: namely, $\sigma \cdot t$ replaces each occurrence of each meta-variable $X$ in $t$ with $\sigma(X)$. A substitution $\sigma$ *grounds* $t$ if, for all $X \in FV(t)$, $\sigma(X)$ is a ground term. A substitution $\sigma$ *unifies* two terms $t$ and $u$ if $\sigma \cdot t = \sigma \cdot u$.

A *context* $E$ is a term-like object that contains exactly one term placeholder $\bullet$. If $t$ is a term, then $E[t]$ is the term generated by replacing the $\bullet$ in $E$ with $t$.

A *rewrite rule* $r$ is a pair of terms $r \doteq (t, u)$ such that $FV(u) \subseteq FV(t)$ and $t \notin \mathcal{V}$. Each rewrite rule $r \doteq (t, u)$ defines a binary relation $\rightarrow_r$ which is the smallest relation such that, for all contexts $E$ and substitutions $\sigma$ grounding $t$ (and therefore $u$), $E[\sigma \cdot t] \rightarrow_r E[\sigma \cdot u]$.

We use $R$ to range over sets of rewrite rules. We write $v \rightarrow_R w$ iff $v \rightarrow_r w$ for some $r \in R$.

For oracle functions (terms to sets of terms) $\mathcal{E}$ , we write $t \rightarrow_{\mathcal{E}} t'$ *iff* $t' \in \mathcal{E}(t)$. We write $t \rightarrow_{R+\mathcal{E}} t'$ if $t \rightarrow_R t'$ or $t \rightarrow_{\mathcal{E}} t'$. For a relation $\rightarrow$ we write $\rightarrow^*$ for its reflexive, transitive closure.

A *path* is a list of terms. A binary relation $\geqslant$ *orients* a path $t_1, \ldots, t_n$ if $\forall 1 \leq i < n, t_i \ \geqslant \ t_{i+1}$.

### 4.2 Ordering Constraint Algebras

The REST algorithm explores only finite rewrite paths; this is achieved via the input candidate term orderings, each of which is required to admit only finite paths. Our algorithm explores only paths which are admitted by *at least one* of these input orderings, conceptually by tracking the *set* of orderings that still accept a path as it is being constructed, and checking for non-emptiness.

To avoid insisting on this set being computed and iterated through at each and every rewriting step, our algorithm is defined to be parametric with any chosen representation of these sets of orderings along with some basic operations on this representation that REST requires; together, these form what we call an *ordering constraint algebra*.

*Definition 4.1 (Ordering Constraint Algebra).* An *Ordering Constraint Algebra* $\mathcal{A}_{(T,\Gamma)}$ *over a set of terms* $T$ *and set of candidate term orderings* $\Gamma$, is a five-tuple $\mathcal{A}_{(T,\Gamma)} \doteq \langle C, \gamma, \top, \mathit{refine}, \mathsf{SAT}\rangle$, 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. $\gamma$, the *concretization function* of $\mathcal{A}_{(T,\Gamma)}$, is a function from elements of $C$ to subsets of $\Gamma$.
3. $\top$, the *top constraint*, is a distinguished constant from $C$, satisfying $\gamma(\top) = \Gamma$.
4. *refine*, the *refinement function*, is a function $C \longrightarrow T \longrightarrow T \longrightarrow C$, satisfying (for all $c, t_l, t_r$) $\gamma(\mathit{refine}(c, t_l, t_r)) = \{\geqslant \;\mid\; \geqslant \in \gamma(c) \;\wedge\; t_l \geqslant t_r\}$.
5. SAT, the *satisfiability function*, is a function $C \longrightarrow \mathit{Bool}$, satisfying (for all $c$) $\mathsf{SAT}(c) = \mathit{true} \Leftrightarrow \gamma(c) \neq \emptyset$.

The functions $\top$, *refine* and SAT are all called from our REST algorithm (Figure 2), and must be implemented as (terminating) functions when implementing REST. Specifically, REST instantiates the initial path with constraints $c = \top$. When a path can be extended via a rewrite application $t_l \rightarrow_R t_r$, 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 ($\mathsf{SAT}(c')$ holds); that is, if $c'$ admits an ordering that orients the generated path. The function $\gamma$ need *not* be implemented in practice; it is purely a mathematical concept used to give semantics to the algebra.

Given terms $T$ and a finite set of candidate orderings $\Gamma$, a trivial Ordering Constraint Algebra is obtained by letting $C = \mathcal{P}(\Gamma)$, and making $\gamma$ 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 candidate 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 candidate orderings $\Gamma = \{\geqslant_{lex(\geqslant)} \mid \geqslant \textit{ is a total order on } Q\}$ writing $\geqslant_{lex(\geqslant)}$ to mean the corresponding lexicographic ordering on $p(q_1, q_2)$ terms generated from an ordering $\geqslant$ on $Q$.

A possible Ordering Constraint Algebra over these $T$ and $\Gamma$ 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 $\gamma$ is given by $\gamma(c) = \{\geqslant_{lex(\geqslant)} \mid \geqslant \;\textit{satisfies}\; c\}$, 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 $p(a_1, a_2) \longrightarrow p(b_1, a_2) \longrightarrow p(a_1, a_1)$. 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, without explicitly constructing any orderings.

Next, we prove the metaproperties of REST independently of the specific choice of Ordering Constraint Algebra, while in Sec. 5.3 we introduce a particularly flexible example of such an algebra, designed to be efficient to implement.

## 4.3 Soundness

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

Our proof relies on the following simple invariant of REST: any path stored in the stack during the execution of the algorithm can be derived by the rewrite rules in $R$ or the external oracle $\mathcal{E}$.

REST Invariant 1 (Path Invariant). *For any execution of* $\mathsf{REST}(R, t_0, \mathcal{E})$*, at the start of any iteration of the main loop, for each* $(ts, c) \in p$*, the list* $ts$ *is a path of* $R + \mathcal{E}$ *starting from* $t_0$*.*

Proof. By induction on the loop iterations of the algorithm. $p$ is initialized with the single element $([t_0], c)$. $[t_0]$ is a valid path of $R + \mathcal{E}$, because it only contains a single term; clearly this path also starts with $t_0$.

At each loop iteration, new elements are potentially pushed to $p$. Suppose the path $ts$ is popped from $p$ at the beginning of the loop. The element to be pushed is a pair $(ts +\!\!+ [t'], c)$ where $last(ts) \rightarrow_{R+\mathcal{E}} t'$. This exactly satisfies the inductive hypothesis: if $ts$ is a path of $R + \mathcal{E}$, then $ts +\!\!+ [t']$ is also a path of $R + \mathcal{E}$. Furthermore, this operation preserves the head of the list: $t_0$ is still the first element. □

Theorem 4.2 (Soundness of REST). *For all* $R$*,* $u$*, and* $t_0$*, if* $u \in \mathsf{REST}(R, t_0, \mathcal{E})$*, then* $t_0 \rightarrow^*_{R+\mathcal{E}} u$*.*

Proof. In each iteration of REST, the term $t$ added to the output $o$ is the last element of the list $ts$ for the tuple $(ts, c) \in p$. By Invariant 1, $t$ must be on the path of $R + \mathcal{E}$ starting from $t_0$. □

## 4.4 Completeness

A naïve completeness statement for REST might be that, for any terms $t_0$ and $u$, if $t_0 \rightarrow^*_{R+\mathcal{E}} u$ then $u$ is in our output ($u \in \mathsf{REST}(R, t_0, \mathcal{E})$). This result doesn't hold in general by design, since REST explores only paths permitted by at least one of its input candidate orderings. We prove this *relative* completeness result in two stages. First (Theorem 4.3), we show that completeness always holds if all steps only involve the external oracle. Then (Theorem 4.4), we prove relative completeness of REST with respect to the ordering relation. We begin by stating another simple invariant of our algorithm: that any term appearing in a path in the stack $p$, will belong to the final output:

REST Invariant 2. *For any execution of* $\mathsf{REST}(R, t_0, \mathcal{E})$*, at the start of any iteration of the main loop, if* $t \in ts$ *and* $(ts, c) \in p$*, then, when the algorithm terminates, we will have* $t \in \mathsf{REST}(R, t_0, \mathcal{E})$*.*

Proof. (Sketch:) We can prove inductively that terms contained in any list in $p$ either remain in $p$ or end up in $o$; since $p$ is empty on termination, the result follows. □

Theorem 4.3 (Completeness w.r.t. $\mathcal{E}$). *For all* $R$*,* $u$*, and* $t_0$*, if* $t_0 \rightarrow^*_{\mathcal{E}} u$*, then* $u \in \mathsf{REST}(R, t_0, \mathcal{E})$*.*

Proof. The proof goes by induction on the number of steps of the path.

Assume the path has $n$ steps: $t_0 \rightarrow_{\mathcal{E}} t_1 \rightarrow_{\mathcal{E}} \ldots \rightarrow_{\mathcal{E}} t_{n-1} \rightarrow_{\mathcal{E}} t_n \equiv u$.

For the base case, $n = 0$ and $u \equiv t_0$. Since $p$ is initialized with $([t_0], \top)$, by the Invariant 2, $t \in \mathsf{REST}(R, t_0, \mathcal{E})$.

For the inductive case, assume that $t_0 \rightarrow^*_{\mathcal{E}} t_{n-1} \rightarrow_{\mathcal{E}} t_n$. By inductive hypothesis, $t_{n-1} \in \mathsf{REST}(R, t_0, \mathcal{E})$. When $t_{n-1}$ was added in the result, it was the last element of a path $ts$ that was popped from the stack $p$. Since $t_{n-1} \rightarrow_{\mathcal{E}} t_n$, we split cases on whether or not $t_n \in ts$. If $t_n \in ts$, then by Invariant 2 $t_n \in \mathsf{REST}(R, t_0, \mathcal{E})$. Otherwise, $(ts +\!\!+ [t_n], c)$ will be pushed into $p$ and, again, by Invariant 2 it will appear in the output. □

Before stating our main completeness result, we observe the (somewhat standard) property that if any path justifies $t_0 \rightarrow^*_{R+\mathcal{E}} u$, there is a *duplicate-free variant* of such path (intuitively, obtained by cutting out all subpaths leading from a term to itself).

Below, we prove that if $t_0 \rightarrow^*_{R+\mathcal{E}} u$ and the ordering $\geqslant$ orients the path, then a duplicate-free variant path $ts$ belongs in the stack $p$ with some constraints $c$ and $\geqslant \in \gamma(c)$.

REST Invariant 3. *For any execution of* $\mathsf{REST}(R, t_0, \mathcal{E})$*, if* $t_0 \rightarrow^*_{R+\mathcal{E}} t_n$ *and* $\geqslant \in \gamma(\top)$ *is an ordering that orients* $t_0 \rightarrow^*_{R+\mathcal{E}} u$*, then at some iteration of the main loop, a duplicate-free variant path* $ts$ *of this path is stored in* $p$*, with some ordering constraints* $c$ *and* $\geqslant \in \gamma(c)$*.*

Proof. The proof goes by strong induction on the length $n+1$ of the path justifying $t_0 \rightarrow^*_{R+\mathcal{E}} t_n$.

First, consider the case $n = 0$, where the path is $[t_0]$ and the constraints $\top$. $([t_0], \top) \in p$ by initialization and trivially $\geqslant \in \gamma(\top)$.

Otherwise, when $n > 0$, assume that $t_0 \rightarrow^*_{R+\mathcal{E}} t_{n-1} \rightarrow_{R+\mathcal{E}} t_n$. If there are any duplicate terms in this path, a duplicate-free variant exists of shorter length, and we can conclude by our induction hypothesis. Otherwise, consider this path with the last element $t_n$ removed. Being already duplicate-free, by our induction hypothesis we must have that, at some iteration of our main loop, this path is contained in $p$ along with a constraint $c_{n-1}$ such that $\geqslant \in \gamma(c_{n-1})$. By the assumption that $\geqslant$ orients the original path, in particular we must have $t_{n-1} \geqslant t_n$, and so, by Def. 4.1, $\geqslant \in \gamma(\mathit{refine}(c_{n-1}, t, t'))$ and therefore $\mathit{refine}(c_{n-1}, t, t')$ is satisfiable. Therefore, the original path will be pushed to $p$ with this constraint in this loop iteration. □

Theorem 4.4 (Relative Completeness). *For all* $R$*,* $u$*, and* $t_0$*, if* $t_0 \rightarrow^*_{R+\mathcal{E}} u$ *and there exists an ordering* $\geqslant \in \gamma(\top)$ *that orients the path justifying* $t_0 \rightarrow^*_{R+\mathcal{E}} u$*, then* $u \in \mathsf{REST}(R, t_0, \mathcal{E})$*.*

Proof. The proof is similar to Theorem 4.3, but now we need to also show that the relation that orients the path satisfies all the ordering constraints generated by the respective REST path. By Invariant 3, at some iteration of the main loop, there must be some path ending in $u$ contained in $p$. Then, by Invariant 2 it follows that all the elements of the path, thus also $u$, belong in the result. □

## 4.5 Termination

Termination of REST requires appropriate conditions on the candidate orderings employed, the external oracle $\mathcal{E}$ and the ordering constraints algebra $\mathcal{A}$ employed. We formally define these requirements and then prove termination of REST.

*Definition 4.5 (Well-Founded ordering constraint algebras).* For ordering constraint algebras $\mathcal{A} = \langle C, \top, \mathit{refine}, \mathsf{SAT}, \gamma\rangle$, 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.

*Definition 4.6.* A relation $t_l \rightarrow t_r$ is *normalizing* if it does not admit an infinite path and *bounded* if for each $t_l$ it only admits finite $t_r$. A relation $\geqslant$ is *thin well-founded* if it cannot orient a duplicate-free infinite path.

Theorem 4.7 (Termination of REST). *For any finite set of rewriting rules* $R$*, if:*

(1) $\rightarrow_{\mathcal{E}}$ *is normalizing and bounded,*

*(2) every candidate ordering (element of* $\Gamma$*) is a thin well-founded relation,*
*(3) The refine and* SAT *functions from* $\mathcal{A}$ *are decidable (always-terminating, in an implementation),*
*(4)* $\mathcal{A}$ *is well-founded,*

*then, for all terms* $t_0$*,* REST$(R, t_0, \mathcal{E})$ *terminates.*

Proof. At every iteration of REST, a path with length $n$ is popped off the stack and due to Requirement 1, and the fact that only a finite number of new terms can be generated by single applications of the rules $R$ to an arbitrary term, a finite number of paths with length $n+1$ is pushed on. Therefore, REST implicitly builds (via its set of paths $p$) a *finitely-branching* tree starting from $t_0$. For REST to not terminate, there must be an infinite path down the tree (note that Requirement 3 eliminates the possibility that the operations called from the ordering constraint algebra cause non-termination).

Consider an arbitrary path down the tree explored by REST, represented by the $(ts, c)$ pairs iteratively generated. Firstly, due to the first condition in the `foreach` of REST (cf. Figure 2), this path will remain duplicate-free. By Requirement 4, at only finitely many steps is the constraint tracked *strictly* refined. Consider then, the postfix of the path after the last time that this happens; at every step, the constraint $c$ remains identical. The normalization assumption (Requirement 1) of $\mathcal{E}$ entails that this path contains no infinite sequence of steps all justified by $R$. However, for each step justified instead by a rewriting step from $R$, the additional condition SAT$(c)$ must hold; by Def. 4.1 this means that there is some $\geqslant\, \in \gamma(c)$ which orients all of these steps. Then the number of steps must be finite, otherwise we would obtain an infinite number of distinct terms which are all oriented by $\geqslant$, contradicting Requirement 2.

Since every path in the finitely-branching tree explored is finite, the algorithm (always) terminates. □

Note that any deterministic, terminating external oracle function satisfies the first requirement. Next, we define a family of ordering functions along with an accompanying ordering constraint algebra that satisfy the second, third and fourth requirements, while being flexible enough to accept most of the interesting paths (as required for completeness).

## 5 TERM ORDERING

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 compatible with symmetrical rules such as commutativity and associativity (cf. Challenge 3, Sec. 2). In (Sec. 5.1) we formally define the term orderings and illustrate how they are used both to generate terms and derive the ordering constraints; next (Sec. 5.2) we prove that the orderings satisfy the termination requirements making them compatible with REST. Finally (Sec. 5.3) we define an efficient ordering constraints algebra based on a compact representation of sets of these orderings, and show that it is well-founded.

### 5.1 Recursive Path Quasi-Orderings

We introduce a term ordering closely following the classic strict ordering definitions for term-rewriting systems [Dershowitz 1982], but with the additional flexibility of enabling rewriting to terms in the same equivalence class with respect to some quasi-ordering (cf. Sec. 5.2). For example, the classic termination criteria of [Dershowitz 1982] would reject the rewrite rule $x + y \longrightarrow y + x$ which is of high importance when reasoning about commutative operators (cf. Challenge 3 in Sec. 3). Since REST already ensures that the generated paths are duplicate free, it gives us the flexibility to allow rewrites on equivalent terms without sacrificing termination of the overall system.

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 $\geqslant_{\mathcal{F}}$ can be any quasi-ordering over $\mathcal{F}$
- The corresponding *multiset quasi-ordering* $\geqslant_{M(X)}$ lifts an ordering $\geqslant_X$ over $X$ to an ordering $\geqslant_{M(X)}$ over multisets of $X$. Intuitively $T \geqslant_{M(X)} U$ when $U$ can be obtained from $T$ by replacing zero or more elements in $T$ with the same number of equal (with respect to $\geqslant_X$) elements, and replacing zero or more elements in $T$ with a finite number of smaller ones. (Def. 5.1).
- Finally, the corresponding *recursive path quasi-ordering* $\geqslant_{\mathcal{T}}$ is an ordering over terms. Intuitively $f(ts) \geqslant_{\mathcal{T}} g(us)$ uses $\geqslant_{\mathcal{F}}$ to compare the function symbols $f$ and $g$ and the corresponding $\geqslant_{M(\mathcal{T})}$ to compare the argument sets $ts$ and $us$. (Definition 5.2).

Below we provide the formal definitions of the multiset quasi-ordering and recursive path quasi-ordering respectively generalized from the multiset ordering of [Dershowitz and Manna 1979] and the recursive path ordering [Dershowitz 1982] to operate on quasi-orderings. For all the three orderings, we write $x_l < x_r \doteq x_l \not\geqslant x_r$ and $x_l > x_r \doteq x_l \geqslant x_r \wedge x_r \not\geqslant x_l$.

*Definition 5.1 (Multiset Ordering).* Given a ordering $\geqslant_X$ over a set $X$, the *derived multiset ordering* $\geqslant_{M(X)}$ over finite multisets of $X$ is defined as $T \geqslant_{M(X)} U$ iff:

(1) $U = \emptyset$, or
(2) $t \in T \wedge u \in U \wedge t \approx u \wedge (T - t) \geqslant_{M(X)} (U - u)$, or
(3) $t \in T \wedge (T - t) \geqslant_{M(X)} (U \setminus \{u \in U \mid u <_X t\})$.

*Definition 5.2 (Recursive Path Quasi-Ordering).* Given a basic ordering $\geqslant_{\mathcal{F}}$, the *recursive path quasi-ordering (RPQO)* is the ordering $\geqslant_{\mathcal{T}}$ over $\mathcal{T}$ defined as follows: $f(t_1, \ldots, t_m) \geqslant_{\mathcal{T}} g(u_1, \ldots, u_n)$ iff

(1) $f >_{\mathcal{F}} g$ and $\{f(t_1, \ldots, t_m)\} >_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$, or
(2) $g >_{\mathcal{F}} f$ and $\{t_1, \ldots, t_m\} \geqslant_{M(\mathcal{T})} \{g(u_1, \ldots, u_n)\}$, or
(3) $f \approx g$ and $\{t_1, \ldots, t_m\} \geqslant_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$.

Example 3. *As a first example, any RPQO $\geqslant_{\mathcal{T}}$ used to restrict term rewriting will accept the rule $x + y \longrightarrow y + x$, since $x + y \geqslant_{\mathcal{T}} y + x$ always holds. Since the top level function symbol is the same $+ \approx +$, by Def. 5.2(1) we need to show $\{x, y\} \geqslant_{M(\mathcal{T})} \{y, x\}$. By Def. 5.1(2) (choosing both $t$ and $u$ to be $x$), we can reduce this to $\{y\} \geqslant_{M(\mathcal{T})} \{y\}$; the same step applied to $y$ reduces this to showing $\emptyset \geqslant_{M(\mathcal{T})} \emptyset$ which follows directly from Def. 5.1(3).*

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

Example 4. *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 $s(x) + y \geqslant_{\mathcal{T}} s\ (x + y)$, the 1st case of Definition 5.2 further requires $\{s(x) + y\} >_{M(\mathcal{T})} \{x + y\}$, which holds if $s(x) + y >_{\mathcal{T}} x + y$. The outermost symbol for both expressions is $+$, so we must check the multiset ordering: $\{s(x), y\} >_{M(\mathcal{T})} \{x, y\}$, which holds because by case splitting on the relation between $s$ and $x$, we can show that $s(x)$ is always smaller than $x$. In short, if $+ >_{\mathcal{F}} s$, then $s(x) + y \geqslant_{\mathcal{T}} s\ (x + y)$.*

## 5.2 Properties of the 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.
(3) *total*, when for all $t, u \in S$ either $t \geqslant s$ or $s \geqslant t$.

Developing on our *RPQO* notion (Def. 5.2), 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 4.7. Concretely:

Theorem 5.3. *If $\geqslant_{\mathcal{F}}$ is a total, well-quasi-ordering, then*

*(1) $\geqslant_{\mathcal{T}}$ is a well-quasi-ordering,*
*(2) $\geqslant_{\mathcal{T}}$ is thin, and*
*(3) $\geqslant_{\mathcal{T}}$ is thin well-founded.*

Proof. The detailed proofs can be found on the appendix (§ A). (1) uses the well-foundedness theorem of Dershowitz [Dershowitz 1982] and the fact that $\geqslant_{\mathcal{T}}$ is a quasi-simplification ordering. (2) relies on the fact that a finite number of function symbols can only generate a finite number of equal terms. (3) is a corollary of (1) and (2) combined. □

### 5.3 An Ordering Constraints Algebra for $\geqslant_{\mathcal{T}}$

Having defined the necessary metaproperties for the recursive path quasi-orderings, we now show an effective way to integrate them into REST. Namely, we provide an ordering constraints algebra enable REST to accept a rewrite path so long as it can be oriented by *some* RPQO. However, REST does not depend on a specific term ordering; for example, it could use a lexicographic ordering instead. We present the implementation for RPQOs here to highlight the approach we use in our implementation and prove its correctness.

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. 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, as described in Sec. 6.4, 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: satisfiability of ordering constraints.* Consider the following rewrite path given by the rules $r_1 \doteq f(g(x), y) \to g(f(y, y))$ 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 $\geqslant_{\mathcal{T}}$ such that $f(g(h), k) \geqslant_{\mathcal{T}} g(f(h, h))$. Following the Definition 5.2, we obtain three possibilities:

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

We can further simplify these using the definition of the multiset quasi-ordering (Def. 5.1). 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 $\geqslant_{\mathcal{T}}$ to satisfy $f(g(h), k) \geqslant_{\mathcal{T}} g(f(h, h))$ 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 $g(f(z,z)) \geqslant_{\mathcal{T}} g(f(k,z))$ entails the constraints $c_1 \doteq z \geqslant_{\mathcal{F}} k$. 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 $f >_{\mathcal{F}} g \wedge z \geqslant_{\mathcal{F}} k$ is satisfiable by an RPQO, thus the path is satisfiable.

*Example: unsatisfiable ordering constraints.* As a second example, consider the rewrite rules $r_1 \doteq f(x) \longrightarrow g(s(x))$ and $r_2 \doteq g(s(x)) \longrightarrow f(h(x))$. 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 constraints algebra can show the unsatisfiability of the diverging path:

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

$f(z) \geqslant_{\mathcal{T}} g(s(z))$ requires $c_0 \doteq f > g \wedge f > s$ which is satisfiable, but $g(s(z)) \geqslant_{\mathcal{T}} f(h(z))$ 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 on it's own, conflicts with $c_0$. Since no *RPQO* can satisfy both $c_0$ and $c_1$, the rewrite path is not satisfiable.

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 derive functions: *rpqo* : $\mathcal{T} \longrightarrow \mathcal{T} \longrightarrow C_{\mathcal{F}}$, and *mul* : $(\mathcal{T} \longrightarrow \mathcal{T} \longrightarrow C_{\mathcal{F}}) \longrightarrow M(\mathcal{T}) \longrightarrow M(\mathcal{T}) \longrightarrow C_{\mathcal{F}}$. *rpqo* is derived by a straightforward translation of Def. 5.2:

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

where *mul'* is the strict multiset comparison given by $mul'(f,T,U) = mul(f,T,U) \wedge \neg mul(f,U,T)$. $\neg : C_{\mathcal{F}} \longrightarrow C_{\mathcal{F}}$ 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 somewhat more complex. Recall that $T \geqslant_{M(X)} U$ when $U$ can be obtained from $T$ by replacing zero or more elements in $T$ with the same number of equal (with respect to $\geqslant_X$) elements, and by replacing zero or more elements in $T$ with a finite number of smaller ones. Therefore, each justification for $\{t_1,\dots,t_m\} \geqslant_{M(X)} \{u_1,\dots,u_n\}$ can be represented by a bipartite graph with nodes labeled $t_1,\dots,t_m$ and $u_1,\dots,u_n$, 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`

*mul*$(f, \{t_1,\dots,t_m\}, \{u_1,\dots,u_n\})$ generates all such graphs, and for each graph converts each labeled edge $(t, u, \texttt{EQ})$ to the formula $f(t,u) \wedge f(u,t)$ and each edge $(t, u, \texttt{GT})$ to the formula $f(t,u) \wedge$

```
{-@ 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
```

Fig. 4. Liquid Haskell version of the proof from Example 1.

$\neg f(u,t)$, 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 the recursive path quasi-ordering to the language of constraints, we can now define our ordering constraints algebra $\mathcal{A}_{(\mathcal{T},\Gamma)}$ by the tuple $\langle C_{\mathcal{F}}, \top, \mathit{refine}, \gamma, \mathsf{SAT}\rangle$ where:

- $\mathit{refine}(c,t,u) = c \wedge \mathit{rpqo}(t,u)$
- $\Gamma$ is the set of all RPQOs.
- $\gamma(c)$ is the set of RPQOs derived from the underlying quasi-orders $\succcurlyeq_{\mathcal{F}}$ that satisfy $c$.
- $\mathsf{SAT}(c) = \mathit{true}$ if there exists a quasi-order $\succcurlyeq_{\mathcal{F}}$ satisfying $c$, $\mathit{false}$ otherwise.

In Sec. 6.4 we discuss how the satisfiability check is mechanized and implemented using an SMT solver. Note that the following properties or ordering constraint algebras:

- $\mathsf{SAT}(c)$ iff $\gamma(c) \neq \emptyset$
- $\succcurlyeq \in \gamma(\mathit{refine}(c,t,u))$ iff $\succcurlyeq \in \gamma(c)$ and $t \succcurlyeq u$

hold by definition.

Finally, we must also show that $\mathcal{A}_{(\mathcal{T},\Gamma)}$ is well-founded (definition 4.5).

Theorem 5.4. *If $\mathcal{F}$ is finite, then $\mathcal{A}_{(\mathcal{T},\Gamma)}$ is well-founded.*

Proof. Recall that $\mathcal{A}_{(\mathcal{T},\Gamma)}$ represents a set of RPQOs derived from quasi-orders over $\mathcal{F}$. If $\mathcal{F}$ is finite, then the range of $\gamma$ is also finite. If $\mathcal{A}_{(\mathcal{T},\Gamma)}$ were not well-founded, then there must exist an infinite sequence $c_1 \sqsupset c_2 \sqsupset \ldots$. Then, there is also some corresponding infinite sequence $\gamma(c_1) \supset \gamma(c_2) \supset \ldots$. But since the range of $\gamma$ is finite, this would yield a contradiction, as $\subset$ is well-founded on finite sets. Therefore, $\mathcal{A}_{(\mathcal{T},\Gamma)}$ is well-founded. □

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 constraints algebra, we now show how to implement REST and the ordering constraints algebra for *RPQOs* into a real theorem prover.

## 6 IMPLEMENTATION OF REST

We implemented REST along with the ordering constraint algebra of § 5) as a standalone library, comprising 2006 lines of Haskell code. We integrated this library into a version of the Liquid Haskell program verifier [Vazou et al. 2014], where we chose the task of applying *lemmas* in Liquid Haskell proofs as a suitable target problem for automation via REST.

## 6.1 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, Figure 4 shows a Liquid Haskell adaptation of the set example of Example 1 (without any integration of `REST`). The function `example1` is to prove the proof obligation from the example; 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 [Vazou et al. 2018]. The penultimate step requires no lemma; the verifier can discharge it based on the refinement on the `s1` parameter.

Lemmas already proven can be used in the proof of further lemmas; as is standard for program verification, care needs to be taken to avoid circular reasoning. Liquid Haskell ensures this via well-founded recursion: lemmas can only be instantiated recursively with smaller arguments.

## 6.2 `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 :: $x_1$:$t_1$ → ... → $x_n$:$t_n$ → {v:() | $e_l = e_r$ } @-}

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 $e_l \longrightarrow e_r$ 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 $e_r \longrightarrow e_l$ 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.

Since the extracted rewrite rules are defined as refinement-typed expressions, our implementation technically goes beyond simple term rewriting, since instantiations of these rules in our implementation are also refinement-type-checked; i.e., it instantiates only the rules with expressions of the proper refined type, achieving a simple form of conditional rewriting [Kaplan 1984].

*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`.

*Preventing Circular Reasoning.* Our implementation finally ensures that rewrites cannot be used to justify circular reasoning, by checking that there are no cycles induced by our `rewrite` and `rewriteWith` pragmas. For example, the below, unsound, circular dependency will be rejected with a rewrite error by our implementation.

```
{-@ rewriteWith p1 [p2] @-}
{-@ rewriteWith p2 [p1] @-}
```

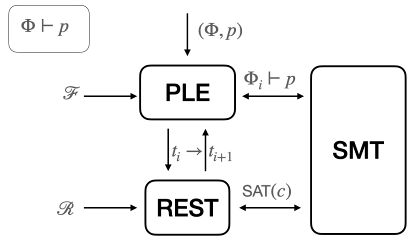


Fig. 5. Interaction between PLE and REST.

```
{-@ p1, p2 :: x:Int → { x = x + 1 } @-}
p1 _ = () ; p2 = p1
```

To prevent circular dependencies, we check that the dependency graph of the rewrite rules (which are made available for proving which) has no cycles. This simple restriction is stronger than strictly necessary; a more-complex termination check could allow rewrites to be mutually justified by ensuring that recursive rewrites are applied with smaller arguments. In practice, our coarse check isn't too restrictive: because Haskell's module system enforces acyclicity of imports, rewrite rules placed in their own module can befreely referenced by importing the library.

*Lemma Automation.* Using our implementation, the same Example 1 proven manually in Figure 4 can be alternatively proven (with all relevant extracted 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: no manual lemma calls are needed as these are all handled by REST. Integrating REST into Liquid Haskell required around 500 lines of code, mainly for surface syntax.

### 6.3 Mutual PLE and REST interaction

Liquid Haskell includes a general technique called *Proof by Logical Evaluation* (PLE) [Vazou et al. 2017] for automating the expansion of terminating program function definitions. PLE expands function calls into single cases of their (possibly conditional) bodies exactly when the verifier can prove that a unique case definitely applies. This check is performed via SMT and so can condition on arbitrary logical information; in our implementation, this forms a natural complement to the term rewriting of REST, and plays the role of its external oracle (cf. Sec. 3). Since PLE is proven terminating [Vazou et al. 2017], the termination of this collaboration is also guaranteed (cf. Sec. 4).

Figure 5 summarizes the mutual interaction between PLE and REST on a verification condition $\Phi \vdash p$, where $\Phi$ is an environment of assumptions. PLE also 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); 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 } →
```

```
-- Interface of OC Algebra
data OC C T = OC
  { top    :: C
  , refine :: C → T → T → C
  , sat    :: C → IO Bool
  }

-- Language of Logical Formulas
data LF = LTrue     | LFalse
        | 𝓕 :>: 𝓕   | 𝓕 :=: 𝓕
        | LF :∧: LF | LF :∨: LF
```

```
-- Implementation of OC Algebra
rpoOC :: OC LF 𝓣
rpoOC = OC LTrue refine sat where

  refine :: LF → 𝓣 → 𝓣 → LF
  refine c t u =
    c :∧: rpo t u -- As in Def 5.2

  sat :: LF → IO Bool
  sat = smtSat . toSMT -- SMT Interface
```

Fig. 6. The implementation of our RPQO Ordering Constraint Algebra

```
                    { 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 in particular 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.

### 6.4 An Efficient Implementation of the RPQO Ordering Constraint Algebra

Figure 6 presents REST's library interface for ordering constraint algebras, and the implementation Liquid Haskell uses. The interface `OC` is parametric in the language of constraints `C`, and the type of terms `t`. Liquid Haskell's implementation uses logical formulas `LF` for the language of constraints $C$ (cf. Def. 4.1) to represent the constraints. The logical formulas `LF` are tailored to our RPQO orderings, tracking properties of the underlying *function* ordering $\mathcal{F}$. Concretely, they contain true, false, comparisons (`:>:`) and equality (`:=:`) between functions in $\mathcal{F}$, and logical conjunction (`:∧:`) and disjunction (`:∨:`).

Our implementation `rpoOC` defines the initial constraints **top** to be `LTrue`, (intuitively, permitting any RPQO). The function `refine c t u`, conjoins the current constraints `c` with the constraints `rpo t u`, ensuring $t \succcurlyeq u$. Finally the `sat` function converts the constraints into an equisatisfiable SMT formula, by encoding each distinct function symbol as an SMT integer variable, encoding the logical operators as their SMT equivalent, and checking for satisfiability of the resulting formula.

REST's interface supports arbitrary implementations for ordering constraints and is not dependent on any particular ordering, constraint language, or solver. For example, one could implement a trivial ordering constraint algebra enforcing maximum rewrite paths of length `n`, by defining `top = n`, `refine c _ _ = c -1`, and `sat c = return (c > 0)`. The ordering constraint algebra interface is straightforward to implement, yet powerful enough to support arbitrary complex functionality.

### 6.5 Further Optimizing the REST algorithm

When a rewrite system is branching, REST may encounter different rewrite paths from an initial term $t$ to an arbitrary term $u$. For example, in Figure 7 (a), the term $(b + a) + a$ is explored in 5 different paths. In general, REST cannot always ignore the repeat encounters of $u$, as a new path from $t$ to $u$ may impose ordering constraints enabling more rewrites in the future. Nonetheless,

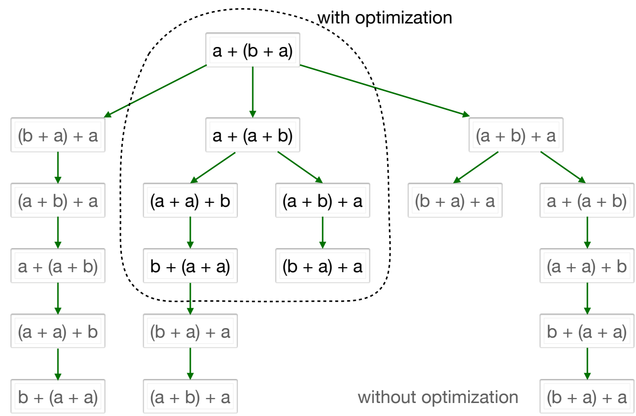


Fig. 7. Associative-commutative rewrites of $a + (b + a)$ generated by REST. Paths explored by REST with the explored terms optimization are within the dashed line. Using the explored terms optimization, REST only considers each term once.

reducing the number of explored paths naturally improves performance. Therefore, we optimize REST based on the following observations:

(1) A term $t$ does not need to be revisited if all of it's rewrites have already been visited.
(2) If a term $t$ was previously visited at constraints $c$, revisiting $t$ at constraints $c'$ is not necessary if $c$ permits all orderings permitted by $c'$, i.e., $\gamma(c') \subseteq \gamma(c)$.

To implement this optimization, REST maintains a mapping $M$ from terms to the logical constraints $c$ each term was explored with (initially mapping all terms to **top**). To explore a term $t$ under logical constraints $c$, the algorithm checks that this term is *explorable*, formally defined by:

$$\text{explorable}(t, c) \doteq t \notin M \vee (\neg(c \Rightarrow M[t]) \wedge \exists u.(t \rightarrow_R u \wedge \text{explorable}(u, c)))$$

This predicate ensures that either this term was not explored before or it comes with weaker constraints that can derive at least one new term in the path.

After exploring a new term, REST weakens the mapping $M$ for this term to the disjunction of the constraints under which it was newly explored and those previously mapped to in $M$. With this optimization, a term will appear in more than one paths in the REST graph only when it can lead to different terms in the path. This optimization critically reduces the number of explored terms, as shown in Figure 7 where 19 vertices of the REST graph on the left reduced to only 6 on the right.

## 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 [Leino 2010] by porting Dafny's calculational proofs to Liquid Haskell, using rewriting to handle axiom instantiation. 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.

| Property | LH+ | Coq | Agda | Lean | Isabelle | Zeno | Isa+ |
|---|---|---|---|---|---|---|---|
| Diverge | OK | loop | loop | fail | loop | OK | OK |
| Plus AC | OK | loop | loop | fail | fail | OK | OK |
| Congruence | OK | OK | OK | OK | OK | fail | OK |

Table 1. Comparison of REST with existing theorem provers. LH+ is Liquid Haskell with rewriting. The potential outcomes are **OK** 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 the Sledgehammer tactic.

### 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 four challenges described in Sec. 2, and compare our implementation to that of other solvers. We chose to evaluate against Agda [Norell 2008], Coq [Coq Development Team 2020], Lean [Avigad et al. 2018], Isabelle/HOL [Nipkow et al. 2020], and Zeno [Sonnex et al. 2012], 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 [Agda Developers 2020; Coq Development Team 2020; Nipkow et al. 2020]. 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 [Avigad et al. 2020]. Zeno [Sonnex et al. 2012] does not allow for user-defined rewrite rules, rather it generates rewrites internally based on user-provided axioms. Sledgehammer [Meng and Paulson 2008; Paulson and Susanto 2007; Paulsson and Blanchette 2012] 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 first challenge and fourth challenges: restricting the rewrite system to ensure termination, and integrating external oracle steps. This example encodes a single (terminating) rewrite rule $f(x) \longrightarrow g(s(s(x)))$ and terminating, mutually recursive function definitions for $f$ and $g$. However, the combination of the rules and function expansions can cause divergence. This test also requires a simple proof that follows directly from the function definitions.

2. Plus AC tests the second and third challenges, 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 $f(g(x))$ and $f(g'(x))$ can be proved equal if there exists a rewrite rule of the form $g(x) \longrightarrow g'(x)$.

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. For this selection of simple but illustrative examples, the only tools to succeed on all cases are our implementation, and Isabelle's Sledgehammer. The latter combines a great many techniques which go beyond term rewriting. Nonetheless, we note that 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 provides strong formal guarantees for such combinations. In particular, `REST` provides general termination and relative completeness guarantees, which Sledgehammer (via its timeout mechanism) does not.

### 7.2 Comparison with E-matching

To evaluate `REST` against the E-matching based approach to axiom instantiation, we compared with Dafny [Leino 2010], a state-of-the-art program verifier. Dafny supports equational reasoning via calculational proofs [Leino and Polikarpova 2013] and calculation with user-defined functions [Amin et al. 2014]. We ported the calculational proofs of Leino and Polikarpova [2013] to Liquid Haskell, using rewriting to automatically instantiate the necessary axioms.

*7.2.1 List Involution.* Figure 8 shows that the reverse operation on lists is an involution, i.e., ∀xs.*reverse(reverse(xs)) = xs*. In this example, both Liquid Haskell and Dafny operate on inductively defined lists with user-defined functions ++ and `reverse`. The proof goes through via the lemma $reverse(xs +\!\!+ ys) = reverse(ys) +\!\!+ reverse(xs)$ and induction on the size of the list.

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

In Dafny, a similar simplification of the calculational proof is not possible. We experimented and found that the lemma `ReverseAppendDistrib` can be alternatively encoded as an 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; analysis with the Axiom Profiler [Becker et al. 2019] 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.

*7.2.2 Set Properties.* Figure 9 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.

Using `REST`, it is safe to encode arbitrary lemmas as rewrite rules, as the termination is guaranteed; in this case the distributivity lemma can be used to complete the proof (and is permitted as a rewrite rule with the precedence $\cap > \cup$).

In conclusion, we have shown that using `REST` to apply rewrites could 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.

```
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 [Leino and Polikarpova 2013].

```
{-@ 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

Fig. 8. List Involution proofs in Liquid Haskell and Dafny

### 7.3 Simplification of Equational Proofs

Finally, we evaluate how REST can simplify equational proofs. We chose to include the set example from [Leino and Polikarpova 2013] (described in Sec. 7.2.2), data structure proofs from [Vazou et al. 2018], examples from the Liquid Haskell test suite, as well as our own case studies. We developed each example in Liquid Haskell both with and without rewriting, and compared the timing and proof complexity. The proofs in [Vazou et al. 2018] were selected because the proofs 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. For our case studies, we included an additional proofs on set properties, arithmetic properties, and program equivalences.

Our case study evaluates the performance of our implementation using a large 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 to prove such kind of equivalences used in techniques such as supercompilation [Bolingbroke and Peyton Jones 2010; Tate et al. 2009; Wadler 1990], by encoded 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 the appendix (§ B). 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.

```
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.

Fig. 9. Set Proofs in Liquid Haskell and Dafny

| No. | Name | Orig. | Removed | Rules | Ind. | Other | Time (Orig.) | Time (RW) |
|---|---|---|---|---|---|---|---|---|
| 1 | Set-Dafny | 4 | 4 | 5 | 0 | 0 | 4.0s | 4.2s |
| 2 | Set-Mono | 7 | 7 | 4 | 0 | 1 | 4.3s | 14.8s |
| 3 | List | 7 | 3 | 3 | 4 | 0 | 7.0s | 6.0s |
| 4 | Tree | 7 | 3 | 3 | 4 | 0 | 5.0s | 5.8s |
| 5 | DSL | 43 | 43 | 23 | 0 | 0 | 9.6s | 16.7s |
| 6 | LH-FingerTree | 3 | 1 | 1 | 1 | 1 | 15.6s | 16.9s |
| 7 | LH-T1013 | 2 | 1 | 1 | 0 | 0 | 4.0s | 3.5s |
| 8 | LH-T1025 | 2 | 2 | 2 | 0 | 0 | 3.8s | 3.9s |
| 9 | LH-T1548 | 1 | 1 | 2 | 0 | 0 | 5.0s | 4.3s |
| 10 | LH-T1660 | 1 | 1 | 1 | 0 | 0 | 3.9s | 4.3s |
| 11 | LH-MapReduce | 5 | 3 | 2 | 1 | 1 | 19.6s | 40.1s |
| Total | | 82 | 69 | 47 | 10 | 3 | 76.8s | 120.5s |

Table 2. Results from simplification of proofs with rewriting. **Set-Dafny** is the set example from[Leino and Polikarpova 2013], **Set-Mono** describes a similar property. **List** and **Tree** are equational proofs from [Vazou et al. 2018]. **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 lemma applications in the original proof. **Removed** is the number of lemma applications that were removed by rewriting. **Rules** is the number of axioms encoded as rewrite rules. **Ind.** is the number of inductive lemmas (not handled by our technique). **Other** are lemma applications or equalities that could not be handled via rewriting. **Time (Orig.)** is verification time in seconds for the original proof. **Time (RW)** is verification time in seconds for the ported proof using rewriting.

We present our results in table 2. By using rewriting, we were able to eliminate all but three of the non-inductive axiom instantiations, while maintaining a reasonable verification time.

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 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 either manually instantiate the lemma. As future work, a limited form of E-matching [de Moura and Bjørner 2007] could be used to address this issue in the general case.

The remaining test case `Set-Mono` cannot be entirely automated via rewriting for a more fundamental reason: the necessary rewrite steps cannot be oriented. This example proves that set union is monotonic for the subset operation, i.e $(s_1 \subset s_2 \implies (s \cup s_1) \subset (s \cup s_2))$. Perhaps surprisingly, assuming the standard set axioms as rewrite rules, no RPQO can orient the necessary step: $(s \cup s_1) \cup (s \cup s_2) \to s \cup (s_1 \cup (s \cup s_2))$. Therefore, after Liquid Haskell generates all permitted rewrites (in this case terminating in less than five seconds), it indicates to the user that the termination check prevented some rewrite applications. The proof was completed by mentioning the equality to this intermediate term; as initializing `REST` from the term $s \cup (s_1 \cup (s \cup s_2))$ enables the appropriate rewrites to successfully complete the proof.

We note that other term orderings could support this proof without the need for intermediate steps. For example, a naïve quasi-ordering based on the size of the term would suffice, as the proof does not require expansion into larger terms.

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 [Hsiang et al. 1992] 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 generated during program verification.

*SMT Verification & Rewriting.* Our rewrite rules could be encoded in SMT solvers as universally quantified equations and instantiated using *E-matching* [de Moura and Bjørner 2007], i.e., a common algorithm for quantifier instantiation. E-matching might generate matching loops leading to unpredictable divergence. Leino and Pit-Claudel [2016] refer to this unpredictable behavior of E-matching as the "the butterfly effect" and partially address it by detecting formulas that could give rise to matching loops. Our approach circumvents unpredictability by using the terminating `REST` algorithm to instantiate the rewrite rules outside of the SMT solver.

Z3 [De Moura and Bjørner 2008] and `CVC4` [Barrett et al. 2011] are state-of-the-art SMT solvers; both support theory-specific rewrite rules internally. Recent work [Nötzli et al. 2019] 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. Breitner [2018] proposes Inspection Testing as a way to check such rewrite rules using runtime execution and metaprogramming, while Farmer et al. [2015] prove rewrite rules via metaprogramming and user-provided hints. In a work closely related to ours, Zeno [Sonnex et al. 2012] is using rewriting, induction, and further heuristics to provide lemma discovery and fully automatic proof generation of inductive properties.

Unlike our approach, the 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 [Vytiniotis et al. 2013] 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 [Dershowitz 1982]. More recent work involves dependency pairs [Arts and Giesl 2000] and applying the size-change termination principle [Lee et al. 2001] in the context of rewriting [Thiemann and Giesl 2007]. Tools like AProVE [Giesl et al. 2017] 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 choose to use a well-quasi-ordering [Kruskal 1972] because it enables rewriting to terms that are not strictly decreasing in a simplification ordering. WQOs are commonly used in online termination checking [Leuschel 2002], especially for program optimization techniques such as supercompilation [Bolingbroke et al. 2011].

*Equality Saturation.* In our implementation, REST passes equalities to the SMT environment, ultimately used for *equality saturation* via an E-graph data structure [Detlefs et al. 2005b]. Equality saturation has also been used for supercompilation[Tate et al. 2009]. 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., [Willsey et al. 2021]) for caching the equivalence classes generated via rewrite applications.

*Associative-Commutative Rewriting.* Associative-Commutative (AC) rewriting [Dershowitz et al. 1983] considers rewrite systems containing associative-commutative operators. It is well known that the inclusion of AC axioms can lead to an explosion in the search space. One solution is to convert terms with AC operators into canonical representations [Conchon et al. 2012]. Another is to handle some AC operations via theory-specific solvers, for example as in SMT solvers.

REST currently does not make any attempt directly address the search state explosion due to the introduction of AC axioms. However, this issue is not significant in practice; as it can be used alongside other solvers supporting theory-specific AC reasoning, or by using an external oracle to generate canonical forms for AC expressions.

## 9 CONCLUSION

We've presented REST, a novel approach to rewriting that can be integrated into program verifiers. We proved correctness, relative completeness, and (online) termination of REST in a very general way, using the abstraction of an ordering constraints algebra. Next, we defined RPQO, an ordering that both satisfies the (abstract) termination requirements of REST and allows for an efficient, algorithmic implementation of ordering constraints algebra. We implemented REST with RPQOs in Liquid Haskell and showed that the resulting system compares well with existing rewriting techniques, it can be used as an alternative to E-matching based axiomatizations approaches, and can substantially simplify equational proofs. In the future, we plan to integrate REST with E-matching to make rewriting facilities available to other program verifiers or SMT solvers.

## REFERENCES


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

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

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

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

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

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

N. Becker, P. Müller, and A. J. Summers. 2019. The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2019 (LNCS)*. Springer-Verlag, 99–116.

Maximilian Bolingbroke and Simon Peyton Jones. 2010. Supercompilation by Evaluation. *SIGPLAN Not.* 45, 11 (Sept. 2010), 135–146. https://doi.org/10.1145/2088456.1863540

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

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

Sylvain Conchon, Evelyne Contejean, and Mohamed Iguernelala. 2012. Canonized rewriting and ground AC completion modulo Shostak theories: design and implementation. *arXiv preprint arXiv:1207.3262* (2012).

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

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

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

Nachum Dershowitz. 1979. A note on simplification orderings. *Inform. Process. Lett.* 9, 5 (1979), 212–215. https://doi.org/10.1016/0020-0190(79)90071-1

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

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

Nachum Dershowitz, Jieh Hsiang, N Alan Josephson, and David A Plaisted. 1983. Associative-Commutative Rewriting.. In *IJCAI*. 940–944.

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

David Detlefs, Greg Nelson, and James B. Saxe. 2005a. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

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

Andrew Farmer, Neil Sculthorpe, and Andy Gill. 2015. Reasoning with the HERMIT: Tool Support for Equational Reasoning on GHC Core Programs. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/2804302.2804303

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—Where Programs Meet Provers. In *Programming Languages and Systems (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128.

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

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

Gerard Huet. 1977. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, USA, 30–45. https://doi.org/10.1109/SFCS.1977.9

Stéphane Kaplan. 1984. Conditional rewrite rules. *Theoretical Computer Science* 33, 2 (1984), 175–193. https://doi.org/10.1016/0304-3975(84)90087-2

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

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

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The Size-Change Principle for Program Termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/360204.360210

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

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

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

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

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

P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*.

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

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Heijen, The Netherlands) *(AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266.

Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2019*, Mikoláš Janota and Inês Lynce (Eds.). Springer International Publishing, Cham, 279–297.

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

Lawrence C Paulsson and Jasmin C Blanchette. 2012. 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*, Vol. 2.

Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. 2012. Frama-c: a Software Analysis Perspective. *Formal Aspects of Computing* 27. https://doi.org/10.1007/s00165-014-0326-7

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems*, Cormac Flanagan and Barbara König (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–421.

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. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: a New Approach to Optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (Savannah, GA, USA). ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

René Thiemann and Jürgen Giesl. 2007. Size-Change Termination for Term Rewriting, Vol. 2706. 264–278. https://doi.org/10.1007/3-540-44881-0_19

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

3242744.3242756

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

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

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

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

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

# A PROOFS ON ORDERINGS

Lemma A.1. *If $T \geqslant_{M(X)} U$, then $T \geqslant_{M(X)} U'$ for all $U' \subset U$.*

Proof. It is sufficient to show that $T \geqslant_{M(X)} U$ implies $T \geqslant_{M(X)} (U - u')$, for any $u' \in U$, since the subset can be obtained by removing a finite number of elements. That is, if $U'$ was obtained by removing elements $u_1, \ldots, u_n$ from $U$, we can show that $T \geqslant_{M(X)} (U \setminus \{u_1\})$ implies $T \geqslant_{M(X)} (U \setminus \{u_1, u_2\})$ and so on.

The proof goes by induction on the size of $T$ and case analysis on $T \geqslant_{M(X)} U$.

For case one there are no $u'$ in $U$, so the proof holds vacuously.

For case two, we have either $u = u'$ or $u \neq u'$. If $u = u'$, a proof of $T \geqslant_{M(X)} (U - u)$ can be made by modifying the proof of $(T - t) \geqslant_{M(X)} (U - u)$. The base case of that proof must be of the form $T' \geqslant_{M(X)} \emptyset$. We modify the base case to be $(T' + t) \geqslant_{M(X)} \emptyset$. Each recursive case is also modified to replace $T'$ with $(T' + t)$, yielding $(T' + t) \geqslant_{M(X)} (U - u) = T \geqslant_{M(X)} (U - u)$, as required. The proof that $T \geqslant_{M(X)} (U - u')$ for all other $u' \in U$ is obtained by induction. By the inductive hypothesis, we have $(T - t) \geqslant_{M(X)} (U - u - u')$, since $u \neq u'$, we also have $u \in (U - u')$. Therefore applying case two we get $T \geqslant_{M(X)} (U - u')$.

For case three, we have either $u' < t$ or $u' \nless t$. If $u' < t$, then the proof $(T - t) \geqslant_{M(X)} (U \setminus \{u \in U \mid u < t\})$ is also a proof of $(T - t) \geqslant_{M(X)} ((U - u') \setminus \{u \in U \mid u < t\})$, thus we obtain obtain the proof directly. The proof for all other $u' \in U$ is obtained by induction. By the inductive hypothesis we have $(T - t) \geqslant_{M(X)} ((U \setminus \{u \in U \mid u < t\}) - u')$. Then, applying the same top-level proof yields $T \geqslant_{M(X)} (U - u')$, since $u'$ is not in the set $\{u \in U \mid u < t\}$.

□

Lemma A.2. *If $\geqslant_X$ is a quasi-order, then the multiset extension $\geqslant_{M(X)}$ is also a quasi-order.*

Proof. To show that $\geqslant_{M(X)}$ is a quasi-order, we define a single-step version $\geqslant_{mul}$, and show that $T \geqslant_{M(X)} U$ if and only if $T \geqslant_{mul*} U$, where $\geqslant_{mul*}$ is the reflexive transitive closure of $U$.

We define $\geqslant_{mul}$ as:

(1) For all elements $t, u$ if $t \in T$ and $u \approx t$, then $T \geqslant_{mul} (T - t + u)$
(2) For all elements $t \in T$ and finite multisets $U$, if $t > u$ for all $u \in U$, then $T \geqslant_{mul} ((T - t) \cup U)$

First, observe that $\geqslant_{mul*}$ is monotonic with respect to multiset union: for all multisets $T$, $U$, and $V$, $T \geqslant_{mul*} U$ implies $(T \cup V) \geqslant_{mul*} (U \cup V)$.

The reflexive case is given by $T \cup V = T \cup V$; we show the transitive case by showing there is a correspondence for each single-step. The proof for each case assumes an arbitrary multiset $V$.

In case one we must show for all $t, u \in T$, $T \geqslant_{mul*} (T - t + u)$ implies $(T \cup V) \geqslant_{mul*} ((T - t + u) \cup V)$. $t$ and $u$ are also in $T \cup V$, therefore we have $(T \cup V) \geqslant_{mul*} ((T \cup V) - t + u)$. We have $(T \cup V) - t + u = (T - t + u) \cup V$, giving us the desired result. Case two is similar: $t \in T$ implies $t \in (T \cup V)$, and $((T \cup V) - t) \cup U = ((T - t) \cup U) \cup V$ for all $U, V$.

Now we show the if direction by case analysis.

Case 1: $U = \emptyset$.
If $T = \emptyset$, then we have $T \geqslant_{mul*} U$ via reflexivity. Otherwise we can select an arbitrary $t$ to remove from $T$, and by definition of $\geqslant_{mul}$ we have $T \geqslant_{mul} ((T - t) \cup \emptyset)$. Then by induction on the size of $T$ we have $((T - t) \cup \emptyset) \geqslant_{mul*} \emptyset$. Then $T \geqslant_{mul} ((T - t) \cup \emptyset) \geqslant_{mul*} \emptyset$, as required.

Case 2: $t \in T \wedge u \in U \wedge t \approx u \wedge (T - t) \geqslant_{mul} (U - u)$.
Let $T' = T - t$ and $U' = U - u$. Then we have $(T' + t) \geqslant_{mul} (T' + u)$ by definition and $T' \geqslant_{M(X)} U'$ implies $T' + u \geqslant_{mul*} U' + u$ via the inductive hypothesis and monotonicity.

Thus $T = (T' + t) \geqslant_{mul} (T' + u) \geqslant_{mul*} (U' + u) = U$ as required.

Case 3: $t \in T \wedge (T - t) \geqslant_{mul} (U \setminus \{u \in U \mid u < t\})$
Partition $U$ into two sets $U_1$ and $U_2$ where $U_1 = \{u \in U \mid u \nless t\}$ and $U_2 = \{u \in U \mid u < t\}$. By definition we have $U = U_1 \cup U_2$. As before $T' = T - t$. Then we have $(T' + t) \geqslant_{mul} (T' \cup U_2)$. $T' \geqslant_{M(X)} U_1$ implies $(T' \cup U_2) \geqslant_{mul*} (U_1 \cup U_2)$ via monotonicity and induction. Thus $T = (T' + t) \geqslant_{mul} (T' \cup U_2) \geqslant_{mul*} (U_1 \cup U_2) = U$ as required.

Now the only-if direction. First we have that $\geqslant_{M(X)}$ is reflexive via induction on size with base case $T = U = \emptyset$ handled by case 1, and recursive case by case 2, similar to above, we remove an arbitrary $t$ from $T$. Now we show how to handle one or more steps from $\geqslant_{mul}$ in a single step of $\geqslant_{M(X)}$.

The key observation is that all elements $u$ of $U$, have exactly one "responsible" element $t$ in $T$ that justifies $T \geqslant_{mul*} U$: we must have either $t > u$ or $t \approx u$ (in which case $t$ is uniquely responsible for $u$ and no other elements of $u$). To prove $T \geqslant_{M(X)} U$, for each $t$ in $T$, we recursively build a tuple $(T', U', p)$ where $T'$, and $U'$ are multisets and $p$ is the proof that $T' \geqslant_{M(X)} U'$. The tuple is initialized to $(\emptyset, \emptyset, U = \emptyset)$.

For each $t$ uniquely responsible for one $u$, we update the tuple to $(T' + t, U' + u, t \in T \wedge u \in U \wedge t \approx u \wedge p)$. The new proof state is valid because by induction we have $p$ being a proof of $T' \geqslant_{M(X)} U'$, as required.

Now consider each $t \in T$ where $t$ justified some multiset $U''$. By induction, we have a proof of $T' \geqslant_{M(X)} U'$; we need a proof that $T' \geqslant_{M(X)} ((U' \cup U'') \setminus \{u \in (U' \cup U'') \mid u < t\})$. Since we have $t > u$ for all $u \in U''$, this simplifies to: $T' \geqslant_{M(X)} (U' \setminus \{u \in U' \mid u < t\})$, which we can obtain via the hypothesis $T' \geqslant_{M(X)} U'$ and lemma A.1.

□

Lemma A.3. *If $\geqslant_X$ is a well-quasi-order, the strict part of it's multiset extension defined as $t >_{M(X)} u$ if $t \geqslant_{M(X)} u$ and $u \ngeqslant_{M(X)} t$ is a well-founded order.*

Proof. This proof operates on the single-step relation defined in A.2. Proving the well-founded property is done by showing that an infinite descent in $>_{M(X)}$ would correspond to an infinite descent in the underlying ordering.

Now, consider a tree built from an infinite path $T_1, T_2, \ldots$ of multisets related by $\geqslant_{M(X)}$. With the exception of special nodes $\top$ and $\bot$, each node in the tree represents an element in a multiset, and the vertices connect the elements to the smaller ones they were replaced with via an application of $\geqslant_{mul}$. Crucially, every edge represents an descent in a well-founded order.

The tree is constructed as follows: let $\top$ be the root of the tree, and let the elements of $T_1$ be the children of $\top$. Then, for each $T_i$ in the infinite list, it was either obtained by replacing some element in $T_{i-1}$ with a same-sized element, or by removing some element $t$ and replacing it with a finite number of smaller elements $ts$.

In the former case, the tree is not modified.

In the latter case, if $ts = \emptyset$, add a single child $\bot$ to the $t$ in the tree. Otherwise, let $ts$ be the children of $t$.

Now, we note that the case one of $\geqslant_{mul}$ is symmetric. Therefore, each pair of terms related by $>_{M(X)}$ must correspond to at least one step in case two of $\geqslant_{mul}$, Therefore in an infinite path of terms related by $>_{M(X)}$ contains an infinite number of applications of case two in $\geqslant_{mul}$.

Therefore, an infinite number of vertices will be added to the tree. Since the tree is finitely branching, it must have an infinitely descending path. However, this infinitely descending path would correspond to an infinite descent in the underlying ordering, contradicting that hypothesis that $\geqslant_X$ is a WQO. □

Lemma A.4. *If $\geqslant_{\mathcal{F}}$ is a total quasi-ordering, then $\geqslant_{\mathcal{T}}$ is a quasi-simplification ordering.*

Proof. We must show that $\geqslant_{\mathcal{T}}$ is a quasi-ordering, i.e it is reflexive and transitive; and also that it satisfies the replacement, subterm, and deletion properties.

Reflexivity occurs via case 3 and A.2.Replacement and deletion follow from case 3 of RPO and the definition of the multiset ordering.

To prove the subterm property, we show a slightly stronger property: for all terms $t = f(t_1, \ldots, t_m)$ and (not necessarily immediate) subterms $u = g(u_1, \ldots, u_n)$, $t >_{\mathcal{T}} u$. The proof goes by induction on the term size, where terms are bigger than their subterms, and by case analysis on the relationship between $f$ and $g$. Because $\geqslant_{\mathcal{F}}$ is total, we have either $f >_{\mathcal{F}} g$, $f \approx g$, or $g >_{\mathcal{F}} f$.

If $f >_{\mathcal{F}} g$, then to get $t \geqslant_{\mathcal{T}} u$ we must show $\{t\} >_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$. Via induction, we have $t >_{\mathcal{T}} u_i$ for all $1 \leqslant i \leqslant n$, as each $u_i$ is a subterm of $u$. To show $u \not\geqslant_{\mathcal{T}} t$, observe that we need $\{u_1, \ldots, u_n\} \geqslant_{M(\mathcal{T})} \{t\}$. This is impossible via the inductive hypothesis and the definition of $\geqslant_{M(\mathcal{T})}$: we already have $t >_{\mathcal{T}} u_i$ for all $u_i$.

If $f \approx g$, then we must show $\{t_1, \ldots, t_m\} >_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$. If $u$ is a direct subterm of $t$, then $u = t_i$ for some $i$. By the inductive hypothesis we have $t_i \approx u >_{\mathcal{T}} u_j$ for all $u_j$, which implies $\{t_1, \ldots, t_m\} >_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$. If $u$ is a nested subterm, then we have some $t_i >_{\mathcal{T}} u_j$ for all $u_j$ via the induction hypothesis: all $u_j$ are subterms of $t_i$.

If $g >_{\mathcal{F}} f$, to get $t \geqslant_{\mathcal{T}} u$ then we must show $\{t_1, \ldots, t_m\} \geqslant_{M(\mathcal{T})} \{u\}$. If $u$ was a direct subterm, then $t_i = u$ gives us the desired result; otherwise we have $t_i >_{\mathcal{T}} u$ via the inductive hypothesis. To show $u \not\geqslant_{\mathcal{T}} t$, observe that showing $u \geqslant_{\mathcal{T}} t$ would require $\{u\} >_{M(\mathcal{T})} \{t_1, \ldots, t_m\}$. However we already have some $t_i \approx u$, which prevents this possibility.

Transitivity is also proven via induction on size. Assume we have $s = f(s_1, \ldots, s_m) \geqslant_{\mathcal{T}} t = g(t_1, \ldots, t_n)$ and $t \geqslant_{\mathcal{T}} u = h(u_1, \ldots, u_p)$. We proceed to show $s \geqslant_{\mathcal{T}} u$ by for each relationship between $f$, $g$, and $h$.

(1) $f >_{\mathcal{F}} g >_{\mathcal{F}} h$, or $f >_{\mathcal{F}} g > h$: Via transitivity of $>_{\mathcal{F}}$ we have $f >_{\mathcal{F}} h$, therefore we must show $\{s\} >_{M(\mathcal{T})} \{u_1, \ldots, u_p\}$. $\{s\} \geqslant_{M(\mathcal{T})} \{t\}$ follows from our assumption $s \geqslant_{\mathcal{T}} t$, and $\{t\} >_{M(\mathcal{T})} \{u_1, \ldots, u_p\}$ follows from $t \geqslant_{\mathcal{T}} u$. By the inductive hypothesis, we have $s \geqslant_{\mathcal{T}} t \geqslant_{\mathcal{T}} u_i$ for all $u_i$, and therefore $\{s\} \geqslant_{M(\mathcal{T})} \{t\} >_{M(\mathcal{T})} \{u_1, \ldots, u_p\}$.

(2) $h >_{\mathcal{F}} g$: There must exist some subterm $t_i$ such that $t_i \geqslant_{\mathcal{T}} u$. Therefore we have $s \geqslant_{\mathcal{T}} t_i$ and $t_i \geqslant_{\mathcal{T}} u$, the inductive hypothesis gives us $s \geqslant_{\mathcal{T}} t_i \geqslant_{\mathcal{T}} u$.

(3) $g >_{\mathcal{F}} f$: There must exist some subterm $s_i$ such that $s_i \geqslant_{\mathcal{T}} t$. As above, using the induction hypothesis allows us to show $s_i \geqslant_{\mathcal{T}} u$, by the subterm property we have $s \geqslant_{\mathcal{T}} s_i$. We show $s \geqslant_{\mathcal{T}} u$ by the definition of $\geqslant_{\mathcal{T}}$.

(4) $f \approx g \approx h$. We clearly have $f \approx h$, we need to show $\{s_1, \ldots, s_m\} \geqslant_{M(\mathcal{T})} \{u_1, \ldots, u_p\}$, which we have via A.2.

□

Theorem A.5. *If $\geqslant_{\mathcal{F}}$ is a total WQO, then $\geqslant_{\mathcal{T}}$ is a WQO.*

Proof. To show that $\geqslant_{\mathcal{T}}$ is WQO, via the well-foundedness theorem of Dershowitz [Dershowitz 1982], which states that a quasi-simplification ordering $\geqslant'$ is WQO if there exists a well-quasi ordering $\geqslant$ such that $f \geqslant g$ implies $f(t_1, \ldots, t_n) \geqslant' g(t_1, \ldots, t_n)$.

By A.4 we have that $\geqslant_{\mathcal{T}}$ is a quasi-simplification ordering, and there exists an ordering over function symbols to satisfy the condition of the well-foundedness theorem: namely the underlying order $\geqslant_{\mathcal{F}}$ from which $\geqslant_{\mathcal{T}}$ is constructed.

□

Theorem A.6. *If $\geqslant_{\mathcal{F}}$ is a total WQO, then $\geqslant_{\mathcal{T}}$ is thin*

Proof. We show that for any term $t = f(t_1, \ldots, t_m)$, the set of terms $\{u \mid t \approx u = g(u_1, \ldots, u_m)\}$ is finite.

If $t \approx u$, then we must have $t \succcurlyeq_{\mathcal{T}} u$ and $u \succcurlyeq_{\mathcal{T}} t$. Assume we have $t \succcurlyeq_{\mathcal{T}} u$.

First, we show that if $f > g$ then $u \not\succcurlyeq_{\mathcal{T}} t$. Assume $u \succcurlyeq_{\mathcal{T}} t$, then there must have some $u_i$ such that $u_i \succcurlyeq_{\mathcal{T}} t$. But via the subterm property, we have $u >_{\mathcal{T}} u_i \succcurlyeq_{\mathcal{T}} t$, contradicting $t \succcurlyeq_{\mathcal{T}} u$.

Likewise, if $g > f$, then there is some $t_i \succcurlyeq_{\mathcal{T}} u$. Then $t >_{\mathcal{T}} t_i \succcurlyeq_{\mathcal{T}} u$. Therefore we also have $u \not\succcurlyeq_{\mathcal{T}} t$.

Therefore, $t \approx u$ only if $f \approx g$. Since there are only a finite number of function symbols, then to show thinness we must show that only a finite number of multisets $\{u_1, \ldots, u_n\}$ such that $\{t_1, \ldots, t_m\} \succcurlyeq_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$ and $\{u_1, \ldots, u_n\} \succcurlyeq_{M(\mathcal{T})} \{t_1, \ldots, t_m\}$. If $\{t_1, \ldots, t_m\} = \emptyset$, then the only such set is $\emptyset$. Otherwise, only such multisets are those where $\{u_1, \ldots, u_n\}$ is obtained from $\{t_1, \ldots, t_m\}$ by removing zero or more terms $t_i$ and replacing them the same number of terms $u_j$ where $t_i \approx u_j$. If $\{t_1, \ldots, t_m\} \succcurlyeq_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$ was justified by removing $t_i$ from $\{t_1, \ldots, t_m\}$ and removing smaller terms $\{u' \mid u' < t_i\}$ from $\{u_1, \ldots, u_n\}$, then we would have $\{t_1, \ldots, t_m\} >_{M(\mathcal{T})} \{u_1, \ldots, u_n\}$: this corresponds to the irreflexive single-step operation shown to form a well-founded order in lemma A.3.

Since the multisets contain a finite number of elements, and each term only has a finite number of equivalent terms (by induction on term size), there are only a finite number of such multisets. □

# B BASIC EQUALITIES AND PROVED THEOREMS IN THE PROGRAM EQUIVALENCE CASE STUDY

| | Name | Formula |
|---|---|---|
| 1. | addDist | $(x * y) + (z * y) = (x + z) * y$ |
| 2. | subDist | $(x * y) - (z * y) = (x - z) * y$ |
| 3. | times2Plus | $x * 2 = x + x$ |
| 4. | plus0 | $x + 0 = x$ |
| 5. | mul0 | $x * 0 = 0$ |
| 6. | mul1 | $x * 1 = x$ |
| 7. | subSelf | $x - x = 0$ |
| 8. | divSelf | $x/x = 1$ |
| 9. | subAdd | $x - y = x + (-y)$ |
| 10. | mulSym | $e * e' = e' * e$ |
| 11. | addSym | $e + e' = e' + e$ |
| 12. | mulAssoc | $(x * y) * z = x * (y * z)$ |
| 13. | addAssoc | $(x + y) + z = x + (y + z)$ |
| 14. | ifT | `if True then lhs else rhs = lhs` |
| 15. | ifF | `if False then lhs else rhs = rhs` |
| 16. | seqNop | `seq lhs nop = lhs` |
| 17. | seqNop' | `seq nop rhs = rhs` |
| 18. | repeatNop | `repeat 0 body = nop` |
| 19. | repeatN1 | `repeat (S n) body = seq body (repeat n body)` |
| 20. | ifJoin | `if c1 then (if c2 then op else nop) else nop` <br> `= if (c1 and c2) then op else nop` |
| 21. | mapFusion | `map g (map f xs) = map (g . f) xs` |
| 22. | foldMap | `(foldr f e) . (map g) = foldr (f . g) e` |
| 23. | foldFusion | `∀ x y . h (f x y) = f' x (h y)` <br> `⟹ h . (foldr f e) xs = foldr f' (h e) xs` |

Table 3. Basic Equality Axioms used in our Program Equivalence Case Study

| Formula | Rewrites |
|---|---|
| $(-(x+x)) + (x+x) = 0$ | 7, 11, 9 |
| $(x * 2) * 2 = (x + x + x + x)$ | 3, 13 |
| $(x * y) + (y * x) = (x * 2 * y)$ | 3, 10, 12 |
| $(x * y) + (y * z) - ((x + z) * y) = 0$ | 1, 7, 10 |
| $(x * y) - (0 * y) = x * y$ | 2, 9, 7, 4 |
| $x * (1 - (x/x)) = 0$ | 5, 7, 8 |
| $x * 1 = x + 0$ | 4, 6 |
| `if true then (seq nop hw) else nop = hw` | 17, 14 |
| `repeat (S (S Z)) hw = seq hw hw` | 16, 18, 19 |
| `if True then (if False then hw else nop) else nop` | |
| `= if (True and False) then hw else nop` | 20 |
| `map p1 (map p2 list) = map p3 list` | 21 |
| `( (foldr add 0) . (map p1)) list = foldr addP1 0 list` | 22 |
| `double . (foldr add 0) list = foldr twicePlus 0 list` | 23 |

Table 4. Theorems Proved via Rewriting using the Basic Equality axioms in 3