REST: Rewriting for SMT Verification with User-Defined Functions

ZACHARY GRANNAN, IMDEA Software Institute, Spain
MUSTAFA HAFIDI, IMDEA Software Institute, Spain
NIKI VAZOU, IMDEA Software Institute, Spain
EVA DARULOVA, MPI SWS, Germany

We introduce REST, a rewrite technique for SMT-based verifiers that supports user-defined terminating functions. Our technique builds upon proof-by-logical evaluation (PLE) that decidably encodes user-defined functions as SMT equalities. REST extends PLE to automatically instantiate provably correct rewrite rules. It uses size-change termination to ensure that rewriting does not diverge, preserving predictable verification. We implement REST in Liquid Haskell and use it to automate proof generation. The automation provided by REST sets the ground for further proof tactic development: we used metaprogramming to implement a structural induction tactic. Our evaluation of REST and the induction tactic combined concludes that 1) they are not subsumed by existing rewriting techniques, 2) they can automate existing proof benchmarks, and 3) having the expected verification slowdown cost, they greatly ease Liquid Haskell proof development.

Additional Key Words and Phrases: SMT-verification, theorem proving, refinement types, Haskell, term rewriting

1 INTRODUCTION

Program verifiers that rely on satisfiability modulo-theory (SMT) solvers, *e.g.*, refinement type systems such as Liquid Haskell [Vazou et al. 2014] or automated verifiers such as Dafny [Leino 2010] and Frama-C [Signoles et al. 2012], enjoy high automation over decidable theories. However, when they derive SMT-queries outside of decidable fragments (*e.g.*, queries that depend on quantifiers), verification becomes unpredictable [Leino and Pit-Claudel 2016]. Worse, most of these verifiers use the SMT solvers as a black box, that is, the user cannot access "proof terms", making it impossible to generate any notion of a proof tactic. On the other hand, type-theory-based theorem provers (*e.g.*, Coq, Isabelle/HOL, Lean) provide explicit proof terms to the user facilitating proof metaprogramming and tactic development. For example, most type-theory provers support a rewriting tactic.

We present REST: a rewriting tactic developed for SMT-based and predictable verification. Our design builds upon Proof By Logical Evaluation (PLE) [Vazou et al. 2017] that encodes user-defined, terminating functions as logical equations that strengthen the SMT verification environment. PLE is defined as a fixpoint algorithm: while new redexes exist in the SMT queries, PLE unfolds them by generating equations that in turn might contain new redexes. Since user-defined functions are terminating, it eventually reaches a fixpoint. REST extends the PLE algorithm with rewriting. First, a rewrite rule $r \equiv e_l \longrightarrow e_r$ is generated when the open terms e_l and e_r are decided equal. Next, at each PLE iteration, if a subterm of the SMT query e can get unified with e_l, for some substitution σ, then the equality $\sigma \cdot e = \sigma \cdot e_r$ strengthens the SMT verification environment.

Rewrites can themselves lead to divergence. For instance, a rule capturing right list identity of append $r \equiv x \longrightarrow x$ ++ [] can get instantiated forever. To enforce termination, REST keeps track of

Authors' addresses: Zachary Grannan, zachary.grannan@imdea.org, IMDEA Software Institute, Madrid, Spain; Mustafa Hafidi, mustafa.hafidi@imdea.org, IMDEA Software Institute, Madrid, Spain; Niki Vazou, niki.vazou@imdea.org, IMDEA Software Institute, Madrid, Spain; Eva Darulova, eva@mpi-sws.org, MPI SWS, Kaiserslautern, Germany.

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

the rewrite derivations as a graph and stops the generation when the rewrite graph size-change diverges: intuitively, when there exists a term that is getting bigger—for some ordering of the function symbols—and no term is getting smaller.

Our first contribution is to use size-change termination and formalize the REST algorithm that combines PLE with term rewriting. In § 5 we prove correctness, completeness, and termination of the formalization of our algorithm.

Our second contribution is to implement REST as an extension to PLE, in the SMT-interaction back-end of Liquid Haskell [Vazou et al. 2014]. REST greatly simplifies proof development: a Liquid Haskell user can now simply hint the use of universally quantified equalities (expressed as a refinement type specification) without the need to explicitly instantiate them. As illustrated in § 2, this automation greatly simplifies the user-provided proofs and sets the ground for the development of further, standard tactics used by type-theory based theorem provers. Concretely, using REST many proofs get simplified to plain structural induction. In § 6 we further develop an induction tactic: we use metaprogramming to automatically derive the boilerplate code of structural induction.

Our final contribution is to evaluate REST along three axes. In § 7 we claim that 1) REST is not subsumed by existing rewrite tactics of commonly used theorem provers: these tactics diverge or return unknown on a representative set of benchmarks automatically verified by REST. 2) REST, combined with the induction tactic, can be applied to simplify a subset of existing benchmarks taken from the Tons of Inductive Problems [Claessen et al. 2015] benchmark suite. 3) REST leads to a 24% decrease in lines of proof code on three case studies on data types that encode sets, lists, and trees. Overall verification time increased. On the 5.7s originally required by Liquid Haskell to run our three case studies, a 96% increase in verification time occurs when using rewriting, which is expected because rewriting is not goal-directed, and therefore makes many extraneous instantiations, and an overall 204% increase occurs when also adding induction, which is attributed to initialization cost of metaprogramming and, as such, is expected to drop on larger benchmarks.

2 OVERVIEW

In this section we provide an overview of REST and our induction tactic. We start with the required background: § 2.1 introduces theorem proving using refinement types by example, § 2.2 describes how proof validation reduces to implication checking, and § 2.3 introduces PLE and how it can be used for decidable SMT-implication checking under terminating user-defined functions. § 2.4 introduces how REST interacts with PLE to apply rewriting, setting the ground for further proof tactics (*e.g.*, induction of § 2.5).

2.1 Refinement Types for Theorem Proving

Consider the definition of natural numbers as either Z or the Successor of a natural number:

```
data Nat = Z | S Nat
```

Following the inductive definition, below we define addition (+) and comparison (<) of natural numbers in the textbook way.

```
(+) :: Nat → Nat → Nat                (<) :: Nat → Nat → Bool
Z + j   = j          -- +.Z           Z < Z         = False  -- <.ZZ
S i + j = S (i + j)  -- +.S           Z < (S _)     = True   -- <.ZS
                                      (S i) < Z     = False  -- <.SZ
                                      (S i) < (S j) = i < j  -- <.SS
```

We can use refinement types to prove properties about these user provided definitions. For example, the following refinement type expresses that plus is commutative:

```
{-@ com :: i:Nat → j:Nat → { i + j = j + i } @-}
```

`com` is a function that takes two arguments `i` and `j` and returns a unit result, refined with the commutativity property. We use `{ i + j = j + i }` to simplify the unit type `{ v:() | i + j = j + i }`. The proof of `com` is an inductively defined inhabitant:

```
com :: Nat → Nat → ()
com Z     _ = ()
com (S i) j = com i j
```

That is, the base case it trivial, while the inductive case requires the inductive hypothesis, expressed as a recursive call. Refinement types can be used to ensure that such a proof is a well founded mathematical proof (*i.e.,* the definition of `com` is total). The details of such proving style can be found in [Vazou et al. 2017]; here we will use `com` to analyse the proof of a more interesting theorem.

The theorem `thm` below expresses the property of natural numbers that for each `i` and `j`, `i` is always smaller than the successor of `j+i`.

```
{-@ thm :: i:Nat → j:Nat → { i < S (j + i) } @-}
thm :: Nat → Nat → ()
thm Z     j = ()
thm (S i) j = thm i j ? com i j ? com (S i) j
```

Similar to before, the proof goes by induction, but now the inductive case, apart from the inductive call `thm i j`, also requires commutativity, expressed as the two calls to `com`. To combine these calls we use the operator `x ? p = x` that intuitively strengthens its argument `x` will all the information captured by the refinement of its argument `p`. To understand the details of this combination one needs to take a deeper look at how the `thm` definition is verified.

2.2 Validity of Inductive Proofs Reduces to Implication Checking

To verify the definition of `thm`, a refinement type system checks that each case of the definition provides sufficient information to prove the property of the result. Let Γ be the environment with global information shared among both case definitions.

```
Γ = thm :: i:Nat → j:Nat → { i < S (j + i) }
  , com :: i:Nat → j:Nat → { i + j = j + i }
  , (+) :: Nat → Nat → Nat
  , (<) :: Nat → Nat → Bool
  , (?) :: a → a → a
```

The environment Γ contains the type specifications of all the global functions (`com`, `(+)`, `(<)`, and `(?)`), as well as the specification of `thm`, since it is recursively defined. To ensure well-formedness of the definition of `thm`, the assumed type should be refined to ensure that `thm` is only used on smaller arguments. This can be achieved using a notion of sized types [Abel 2010], but here is omitted for simplicity.

Refinement type checking will extend the global environment Γ with the per-case binders to ensure that the two cases of the `thm` definition are sufficient to prove the result, leading in the following two type checking rules:

```
Γ, j:Nat ⊢ ()                         Γ, i,j:Nat ⊢ thm i j ? com i j ? com (S i) j
       :: { Z < S (j + Z) }                     :: { S i < S (j + S i) }
```

Finally, refinement type checking reduces these rules to the below verification conditions (VCs).

```
∀ j. true                         ∀ i j. i < S (j + i), i + j = j + i, S i + j = j + S i
        ⇒  Z < S (j + Z)                   ⇒ S i < S (j + S i)
```

For the base case (left), the requirement `Z < S (j + Z)` should hold under no assumptions. For the inductive case (right), the requirement `S i < S (j + S i)` should hold under the inductive hypothesis `i < S (j + i)` which was introduced in the VC by the call `thm i j` and the two commutativity hypothesis.

The original `thm` definition type checks if the above two VCs are valid, which can be checked by SMTs [Barrett et al. 2016]. Unsurprisingly, they are not valid: if the functions `(<)` and `(+)` are left uninterpreted, there exist models that can refute the two implications. To ensure the validity of the VCs, the models of the two user defined functions should comply with the user provided definitions of § 2.1. There exist various ways to impose this, including axiomatization of the user-specified definitions (which then leads to unpredictable verification [Leino and Pit-Claudel 2016]) and proof by logical evaluation [Vazou et al. 2017]. Here, we choose the latter.

2.3 Encoding of User Specified Definitions via PLE

Proof by Logical Evaluation (PLE) combines SMT implication checking with user-provided, well-founded function definitions while preserving decidability and completeness of validity checking.

PLE searches the VC for redexes of user-specified functions whose evaluation can be statically determined. For all such redexes, it strengthens the assumption of the VC with the evaluation step, encoded as an equality. For example, the VC of the `thm` base case contains the redex `Z < S (j + Z)`. By the definition of `(<)`, this redex can only evaluate to `True` (using the `<.ZS` case). Thus, the equality `Z < S (j + Z) = True` is used to strengthen the VC:

```
∀ j. Z < S (j + Z) = True          -- PLE <.ZS on Z < S (j + Z)
     ⇒  Z < S (j + Z)
```

The strengthened VC is now valid. Similarly, the inductive VC will be strengthened as below.

```
∀ i j. i < S (j + i), i + j = j + i, S i + j = j + S i
     , S i < S (j + S i) = i < j + (S i) -- PLE <.SS on S i < S (j + S i)
     , S i + j = S (i + j)               -- PLE +.S  on S i + j
     ⇒ S i < S (j + S i)
```

which is also decided as valid by an SMT solver that encodes the user specified operators `(<)`, `(+)` as well as `Z` and `S` as uninterpreted functions, but is using the theory of equality.

In general, PLE (summarised in § 5) uses an iterative algorithm to strengthen the VC while new redexes are generated. Since all the user-specified functions are well-formed (*i.e.,* terminating) the algorithm reaches a fixpoint. Further, it is using the SMT solver itself to decide whether redexes can be unfolded to precisely one expression under the current assumptions of the VC, thus ensuring completeness of validity checking.

2.4 Rewriting Under PLE

The main contribution of this work is to combine PLE with rewriting while preserving termination and completeness of verification. In our running example, we wish the proof of `thm` to be as below:

```
{-@ rewriteWith thm [com] @-}
{-@ thm :: i:Nat → j:Nat → { i < S (j + i) } @-}
thm Z     _ = ()
thm (S i) j = thm i j
```

That is, the pragma `rewriteWith thm [com]` directs type checking to use `com` to complete the proof, yet the user does not have to explicitly specify the instantiations of `com`.

With the above program, the inductive VC is now simply the following:

```
0. ∀ i j. i < S (j + i)  ⇒ S i < S (j + S i)
```

PLE generates the equality:

```
1. S i < S (j + S i) = i < j + S i  -- PLE <.SS on S i < S (j + S i)
```

No further PLE steps apply. Now the `com` rewriting takes action to introduce two further equations:

```
2.  j + i   = i + j                  -- rewrite com j i,     from j + i   of 0
3.  j + S i = S i + j                -- rewrite com j (S i), from j + S i of 1
```

Finally, the right-hand-side of `3.` generates the `S i + j` redex that can be expanded by PLE:

```
4. S i + j = S (i + j)              -- PLE +.S on S i + j
```

Strengthening the assumption of the VC `0` with the equalities `1-4` is sufficient to prove validity.

This example illustrates the need for PLE and rewrite steps to interact with each other. In § 4 we formalize an algorithm in which the PLE steps are also encoded as rewrite rules (that satisfy strong normalization) and prove that our rewrite algorithm is terminating and complete. The core idea of our algorithm is that we use "runtime" size-change termination checking [Nguyen et al. 2019] to determine if applying a rewrite may introduce divergence. Our notion of size-change termination is somewhat more liberal than is typically presented: the lack of an infinitely descending sequence is not sufficient to show divergence. In addition, we require that there is a possibly infinite ascending path.

In § 5 we present how we extended the PLE implementation to allow for rewrites. In § 7 we provide bigger case studies where more than one rewrite is used and we compare our technique to alternative approaches. In short, the obvious approach, *i.e.*, to encode rewrite rules as SMT axioms was not an option as this would quickly render verification unpredictable. Rewriting tactics have been extensively used by theorem provers, by either directly using rewrite theory or normalization. To the best of our knowledge, none of these is directly compatible with reasoning within SMT.

2.5 Automating Inductive Proofs

A great benefit of automating rewriting is that it leaves the proofs unlittered by explicit lemma invocations (*i.e.*, calls to `com` in our running example), setting the ground for further tactic development. For example, the definition of `thm` in § 2.4 is a boilerplate inductive call, easily derived using metaprogramming.

We used Template Haskell [Sheard and Peyton Jones 2002] to implement a `lhp` metafunction that expands boilerplate, structural induction. Thus, the definition of `thm` is totally automated using rewriting and `lhp`.

```
{-@ rewriteWith thm [com] @-}
{-@ thm :: i:Nat → j:Nat → { i < S (j + i) } @-}
[lhp|noSpec|ple|induction|caseExpand
thm :: Nat → Nat → ()
thm _ _ = () |]
```

In § 6 we discuss the details and challenges of our implementation, while in § 7 we evaluate the benefits of the inductive tactic combined with rewriting.

3 BACKGROUND

Before explaining our rewriting algorithm in detail, we review necessary background. Our formalism of rewriting is standard; we base our terminology on [Klop 2000]. Our language consists of the following:

(1) An infinite set of variables $\mathcal{V}$ with elements $x, y, \ldots$
(2) A finite set of operators $\mathcal{F}$ with elements $f, g, \ldots$
Each operator is associated with a fixed numeric arity.
(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 variables in t.

3.1 Rewrite Rules

Definition 3.1. A *substitution* $\sigma \subseteq \mathcal{V} \times \mathcal{T}$ is a mapping from variables to terms. We use the notation $\sigma \cdot t$ to denote the simultaneous application of the substitution: namely, $\sigma \cdot t$ replaces each occurrence of x in t with $\sigma(x)$ if $x \in \sigma$.

Definition 3.2. A substitution σ *unifies* two terms t and u if $\sigma \cdot t = \sigma \cdot u$.

Definition 3.3. A *context* C is a term-like object that contains exactly one placeholder $\bullet$. If t is a term, then $C[t]$ is the term generated by replacing the $\bullet$ in C with t.

Definition 3.4. 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 $\mathcal{R}^r$, such that for all contexts C and substitutions σ, $(C[\sigma \cdot t], C[\sigma \cdot u]) \in \mathcal{R}^r$.

We write $v \rightarrow_r w$ if $(v, w) \in \mathcal{R}^r$ and $\cdot \rightarrow_r^* \cdot$ to encode the reflexive, transitive closure of $\cdot \rightarrow_r \cdot$.

We use $\mathcal{R}$ to denote a set of rewrite rules. We write $v \rightarrow_{\mathcal{R}} w$ if there exists an $r \in \mathcal{R}$ so that $v \rightarrow_r w$ and $\cdot \rightarrow_{\mathcal{R}}^* \cdot$ to encode the reflexive, transitive closure of $\cdot \rightarrow_{\mathcal{R}} \cdot$.

Definition 3.5. A *path* on a set of rewrite rules $\mathcal{R}$, *i.e.*, $\mathcal{P}_{\mathcal{R}} \doteq t_1, \ldots, t_n$, is a list of terms for which, for all $i, 0 < i < n$, $t_i \rightarrow_{\mathcal{R}} t_{i+1}$.

Definition 3.6. A a set of rewrite rules $\mathcal{R}$ is *strongly normalizing* if it does not admit any infinite paths.

3.2 Ordering

Definition 3.7. A relation $>$ is *well-founded*, if it does not admit infinitely descending chains $x_1 > x_2 > \ldots$.

Definition 3.8. A relation $\geqslant$ is *well-quasi-ordered*, if for all infinite chains $x_1 \geqslant x_2 \geqslant \ldots$ there exists an $i, j, i < j$ such that $x_j \geqslant x_i$.

4 FORMALIZATION OF REST

We present the algorithm REST that, given an initial term t_0 and a list of rewrite rules, generates a set of terms that can be obtained from (repeatedly) applying the rewrite rules to t_0. The key idea is that REST will terminate, even if the rewrite rules themselves allow for infinite derivations. REST achieves this by keeping track of each term path generated by the rewrite rules, only applying rewrites if the size-change graphs for the resulting path do not diverge. Because any infinite derivation will have a finite prefix of terms with a diverging size-change graph, REST's termination is guaranteed.

Algorithm 1: REST :: $\mathcal{R} \times \mathcal{R} \times \mathcal{T} \rightarrow \mathcal{P}(\mathcal{T})$

```
Input: $\mathcal{R}_U, \mathcal{R}_E, t_0$
Result: $o$
$o \leftarrow \emptyset$ ;
$P \leftarrow [([], t_0)]$;
while $P \neq \emptyset$ do
    pop $(ts, t)$ from $P$;
    $o \leftarrow o \cup \{t\}$;
    $ts' \leftarrow ts +\!\!+ [t]$;
    forall $r, t'$ s.t. $r \in \mathcal{R}_U \cup \mathcal{R}_E, t \rightarrow_r t'$ and $t' \notin ts'$ do
        if $r \in \mathcal{R}_E$ or there exists an ordering $>$ over $\mathcal{F}$ such that $ts' +\!\!+ [t']$ does not size-change
          diverge with respect to $\geqslant^*$
         then push $(ts', t')$ to $P$ ;
    end
end
return $o$;
```

Algorithm 1 defines REST$(\mathcal{R}_U, \mathcal{R}_E, t_0)$ where $\mathcal{R}_E$ is a strongly normalizing (Definition 3.6) set of rewrite rules, $\mathcal{R}_U$ is a set of rewrite rules (not required to be strongly normalizing), and t_0 is an initial term. REST$(\mathcal{R}_U, \mathcal{R}_E, t_0)$ returns a set of terms such that $\forall t \in \text{REST}(\mathcal{R}_U, \mathcal{R}_E, t_0). t_0 \rightarrow^*_{\mathcal{R}_U \cup \mathcal{R}_E} t$.

The algorithm is using P that contains a list of pairs (ts, t) capturing the developed paths ts starting for the term t_0 and leading to t. It initializes the result o to empty and P to the trivial pair $([], t_0)$. While the developed paths can be expanded, *i.e.*, P is not empty, the pair (ts, t) is popped from P. Since the path ts is about to get further explored, t is appended to the path ts and t is added to the output. For all proper rewrites of t—-the ones that come from $\mathcal{R}_E$ or the ones that come from $\mathcal{R}_U$ and might not diverge—the rewritten expression is pushed to P together with the path $ts +\!\!+ [t]$ that leads to this expression from t_0.

The crucial part of the algorithm is the decision of whether or not to push a rewritten expression to the path. Because $\mathcal{R}_U$ may not be strongly normalizing, it is possible that there could be an infinite derivation. We use the concept of size-change termination [Lee et al. 2001] at runtime to determine whether or not to continue exploring a path. The core idea is that an infinite derivation must have a finite prefix of terms whose path generates an ascending size-change graph (§ 4.2). The size-change analysis requires an underlying ordering over terms. For this, we choose a well-quasi-ordering over terms (§ 4.1).

Because REST does analysis at runtime, *i.e.*, during algorithm execution, the decision to rewrite a term from t_i to t_{i+1} depends on the size-change graph of the path of terms $t_0, \ldots, t_i$. That is, the termination of the algorithm does not consider the rewrite rules themselves, instead it performs analysis only on the paths generated via rewriting. We discuss the REST's correctness, completeness and termination properties in detail in § 4.3 and § 4.4.

Example. Consider the example from § 2.1, from which we can map the PLE rules to $\mathcal{R}_E = \{Z + j \longrightarrow j, S(i) + j \longrightarrow S(i + j)\}$ and the rewrite rules deriving from com as $\mathcal{R}_U = \{i + j \longrightarrow j + i\}$. REST$(\mathcal{R}_U, \mathcal{R}_E, j + S(i))$ generates the following set of terms from $t_0 = j + S(i)$:

$$\text{REST}(\mathcal{R}_U, \mathcal{R}_E, j + S(i)) = \{j + S(i), S(i) + j, S(i + j), S(j + i)\}$$

That is, these terms are equivalent to using PLE *and* rewriting. Using only PLE, we cannot generate any rewrites from this expression:

$$\mathsf{REST}(\emptyset, \mathcal{R}_E, j + S(i)) = \{j + S(i)\},$$

and using only rewriting, we only get:

$$\mathsf{REST}(\mathcal{R}_U, \emptyset, j + S(i)) = \{j + S(i), S(i) + j\}$$

4.1 Term Ordering

A term ordering is necessary to encode the size-change of terms in size-change graphs. We base our ordering on recursive path orderings [Dershowitz 1982], which are used to prove the termination of rewrite systems. Termination can be proven for a rewrite system $\mathcal{R}$ if, for a well-founded ordering $>$ over terms, $t \rightarrow_{\mathcal{R}} u$ implies $t > u$.

However, because we are not interested in obtaining normal forms, we do not require a well-founded ordering. Instead, we choose a well-quasi-ordering $\geq$ over terms. By making this choice, two terms can be equivalent with respect to the ordering if $t \geq u$ and $u \geq t$ (in a well-founded ordering, two terms could only be equivalent if they are syntactically equal). Furthermore, if every term has a finite set of equivalent terms w.r.t. a well-quasi-ordering, all finite paths of terms that respect the ordering (i.e $t_i \rightarrow t_{i+1}$ implies $t_i \geq t_{i+1}$) can be generated simply by excluding those with duplicates.

Definition 4.1. Given an well-founded partial ordering $>$ over $\mathcal{F}$, we define a quasi-ordering $\geqslant^*$ over $\mathcal{T}$, based off the recursive path ordering, as follows:

(1) $\forall x, y \in \mathcal{V} \Rightarrow x \geqslant^* y$
(2) $\forall x \in \mathcal{V}, f \in \mathcal{F}, t_1, \ldots, t_n \in \mathcal{T} \Rightarrow f(t_1, \ldots, t_n) \geqslant^* x$
(3) $\forall u = f(u_1, \ldots, u_m), t = g(t_1, \ldots, t_n)$
   (a) If $f > g$ and $\{u\} \underline{>}^* \{t_1, \ldots, t_n\}$, then $u \geqslant^* t$.
   (b) If $g > f$ and $\{u_1, \ldots, u_m\} \underline{\geqslant}^* \{t\}$, then $u \geqslant^* t$.
   (c) Otherwise, if $\{u_1, \ldots, u_m\} \underline{\geqslant}^* \{t_1, \ldots, t_n\}$, then $u \geqslant^* t$.

Where $t >^* u$ *iff* $t \geqslant^* u$ and $u \not\geqslant^* t$. $\underline{>}^*$ is the multiset ordering [Dershowitz and Manna 1979] of $>^*$ such that $\bar{t} \underline{>}^* \bar{u}$ *iff* $\bar{u}$ can be obtained from $\bar{t}$ by removing one or more terms from $\bar{t}$ and replacing them with any number of smaller (with respect to $>^*$) terms. $\bar{t} \underline{\geqslant}^* \bar{u}$ means $\bar{t} \underline{>}^* \bar{u}$ or $\bar{t} = \bar{u}$. Two terms are considered equal if they are the same modulo permutations amongst their subterms.

As an example, we can show that using the ordering $+ > s$, the expression $t = s(x) + y \geqslant^* s(x + y) = u$. Since $+$ is the outermost symbol of t and s is the outermost symbol of u, and $+ > s$, $t \geqslant^* u$ if $\{s(x) + y\} \underline{>}^* \{x + y\}$. By the property of the multiset ordering, this holds if $s(x) + y >^* x + y$. The outermost symbol for both expressions is $+$, so we must check the multiset ordering: $\{s(x), y\} \underline{>}^* \{x, y\}$, which holds because $\{x, y\}$ can be obtained from $\{s(x), y\}$ by replacing $s(x)$ with the smaller element x.

Unlike the recursive path ordering of [Dershowitz and Manna 1979], our ordering is not stable under substitution (that is $t \geqslant^* u$ does not imply $\forall \sigma. \sigma \cdot t \geqslant^* \sigma \cdot u$). Although this could potentially be problematic in the context of static analysis, such a restriction is not necessary for our approach, which performs analysis at runtime.

Lemma 4.2. *Given a finite set of variables $\mathcal{V}$, for any term t, the set of terms $\{u \mid t \geqslant^* u, u \geqslant^* t\}$ is finite.*

Proof. The proof goes by structural induction on the term t.

When $t \in \mathcal{V}$, the set of terms $\{u \mid t \geqslant^* u, u \geqslant^* t\}$ is finite because $t \geqslant^* u$ implies $u \in \mathcal{V}$, and $\mathcal{V}$ is finite.

When $t = f(t_1, \ldots, t_n)$, the set cannot contain any terms $u \in \mathcal{V}$. Furthermore, for every $u = g(u_1, \ldots, u_m)$ in the set, it cannot be the case that $f > g$ or $g > f$, as the relationship between terms t and u would be either $t >^* u$ or $u >^* t$. Therefore, for each $u = g(u_1, \ldots, u_m)$ in the set, we must have $\{t_1, \ldots, t_n\} \underline{\geq}^* \{u_1, \ldots, u_m\}$ and $\{u_1, \ldots, u_m\} \underline{\geq}^* \{t_1, \ldots, t_n\}$. Therefore, the two multisets must only differ with respect to permutations amongst the subterms. Since there are only a finite amount of permutations amongst the subterms, the resulting set must also be finite. □

Theorem 4.3. *$>^*$ is transitive and well-founded. $\geqslant^*$ is transitive and well-quasi-ordered.*

Proof. We begin with the proof of transitivity of $\geqslant^*$, *i.e.*, for each three terms t_1, t_2, t_3, if $t_1 \geqslant^* t_2$ and $t_2 \geqslant^* t_3$, then $t_1 \geqslant^* t_3$.

The proof goes by structural induction on t_3.

If $t_3 \in \mathcal{V}$, we have $t_1 \geqslant^* t_3$ directly.

If $t_3 = h(t_{31}, \ldots, t_{3o})$, then we have $t_2 = g(t_{21}, \ldots, t_{2m})$ and $t_1 = f(t_{11}, \ldots, t_{1n})$. We can show transitivity based on the relationship between the outermost function symbols, defined inductively over the size of terms. This proof closely follows the structure of [Dershowitz 1982].

(1) If we have $f > h$, we must show that $t_1 >^* t_{3i}$ for all $i \in \{1, \ldots, o\}$. By the inductive hypothesis, we have $t_1 \geqslant^* t_2 \geqslant^* t_{3i}$. In fact, we have $t_1 \geqslant^* t_2 >^* t_{3i}$. Otherwise, we would have $t_{3i} \geqslant^* t_2$, which would imply $t_2 \not\geqslant^* t_3$, since t_{3i} is a subterm of t_3. Thus, since we have $t_1 >^* t_{3i}$ for all i, and t_{3i} is smaller than t_3, we have $t_1 \geqslant^* t_3$.
(2) If $f > g$ and $g \not> h$ and $h \not> g$. We have that $t_1 >^* t_{2i}$ for all t_{2i}. Since $\{t_{21}, \ldots, t_{2m}\} \geqslant^* \{t_{31}, \ldots, t_{3o}\}$, there cannot be a t_{3j} greater than any t_{2i}. Thus, $t_1 \geqslant^* t_{3j}$ holds for all t_{3j} inductively, and $t_1 \geqslant^* t_3$.
(3) If $f < g$ then there is a subterm t_{1i} of t_1 such that $t_{1i} \geqslant^* t_2$. Therefore, we have $t_{1i} \geqslant^* t_2 \geqslant^* t_3$ and $t_1 \geqslant^* t_3$ by the inductive hypothesis.
(4) If $f \not> g$ and $g \not> f$ and $h > g$ we have $t_1 \geqslant^* t_{2i} \geqslant^* t_3$, $t_1 \geqslant^* t_3$ by the inductive hypothesis.
(5) If $f \not> g$ and $g \not> f$ and $g > h$, we have that t_2 is greater than all t_{3i}. Therefore, we have $t_1 \geqslant^* t_2 >^* t_{3i}$ by the inductive hypothesis and therefore and $t_1 \geqslant^* t_3$.
(6) If $f \not> g$ and $g \not> f$ and $h \not> g$ and $g \not> h$, the relationships $t_1 \geqslant^* t_2$ and $t_2 \geqslant^* t_3$ is determined by the multiset extension of $\geqslant^*$. Since the multiset extension of a transitive relationship is also transitive, we have $t_1 \geqslant^* t_3$.

That $>^*$ is transitive follows from the transitivity of $\geqslant^*$. Assume $t_1 >^* t_2$ and $t_2 >^* t_3$, we need to show $t_1 >^* t_3$. By assumption we have $t_1 \geqslant^* t_2$ and $t_2 \geqslant^* t_3$, that by transitivity gives us $t_1 \geqslant^* t_3$. By assumption we also have $t_2 \not\geqslant^* t_1$ and $t_3 \not\geqslant^* t_2$. If $t_3 \geqslant^* t_1$, by transitivity we would get $t_3 \geqslant^* t_2$ which is a contradiction. Thus $t_3 \not\geqslant^* t_1$.

A proof of that $>^*$ is well-founded can be found in [Dershowitz 1982].

That $\geqslant^*$ is well-quasi-ordered comes from the well-foundedness of $>^*$, and that it does not have any infinite antichains.

□

4.2 Size Change

We now describe our formalism of size change, based on the size-change principle of [Lee et al. 2001]. We use size change graphs to determine when to employ a rewrite rule: if the size change graph indicates that employing a rewrite will cause divergence, then the rule is not applied. Our formalism for size-change requires a well-quasi ordering $\geqslant$ over $\mathcal{T}$. In our formulation of REST we use the ordering $\geqslant^*$ defined in the previous section, in principle it is applicable to any wqo over terms.

Size Change Direction. The function $sc_{\geqslant}(t,u)$ computes the size change, *i.e.,* $\mathsf{SC} \doteq \{\mathsf{EQ}, \mathsf{DESC}, \mathsf{ASC}\}$, between two terms as follows:

$$sc_{\geqslant}(t,u) \doteq \begin{cases} \mathsf{EQ}, & \text{if } t \geqslant u \text{ and } u \geqslant t \\ \mathsf{DESC}, & \text{if } t \geqslant u \text{ and } u \not\geqslant t \\ \mathsf{ASC}, & \text{otherwise} \end{cases}$$

For example, the expressions $sc_{\geqslant}(i+j, j+i)$ would be EQ, because $i+j \geqslant j+i$ and $j+i \geqslant i+j$. $sc_{\geqslant}(i+0, i)$ would be DESC because we have $i+0 \geqslant i$ and $i \not\geqslant i+0$. Correspondingly, we have $sc_{\geqslant}(i, i+0) = \mathsf{ASC}$.

Our definition of SC differs from the typical presentation by keep track of ascending edges. Later, we will show how these edges are used to identify divergence.

Size Change Entries. A size change entry is a triple $(\alpha, \beta, d) \in (\mathcal{F} \times \mathbb{N}) \times (\mathcal{F} \times \mathbb{N}) \times \mathsf{SC}$. An entry $((f,m),(g,n),d)$ represents the relationship between the n'th argument of g and the m'th argument of f.

We overload the function $sc_{\geqslant}(t,u)$ to generate size change entries between two terms $t = f(t_1, \dots, t_m), u = g(u_1, \dots, u_n)$ as:

$$sc_{\geqslant}(t,u) \doteq \{((f,i),(g,j), sc_{\geqslant}(t_i, u_j)) \mid i \in \{1, \dots, m\}, j \in \{1, \dots, n\}\}$$

Each element $(\alpha, \beta, d) \in sc_{\geqslant}(t,u)$ represents a directed edge from α to β, with size-change d.

The Size-Change Graph. The size-change graph shows the relationships between immediate subterms of each pair of adjacent terms in a path. REST uses this graph to determine whether or not to apply a rewrite. The underlying idea is that any path of infinite terms will have a finite prefix for which the size-change path diverges. Thus, REST will not employ a user-defined rewrite rule if doing so would cause the size-change graph to diverge.

Definition 4.4. The size-change graph $\mathcal{G} \subseteq ((\mathcal{F} \times \mathbb{N}) \times \mathbb{N}) \times ((\mathcal{F} \times \mathbb{N}) \times \mathbb{N}) \times \mathsf{SC}$ of a path ts consisting of n terms is defined as:

$$scg_{\geqslant}(ts) = \{((\alpha, i), (\beta, i+1), d) \mid (\alpha, \beta, d) \in sc_{\geqslant}(t_i, t_i + 1), i \in [1, \dots, n-1]\}$$

Each element $((\alpha, i), (\beta, i+1), d) \in scg_{\geqslant}(ts)$ represents a directed edge from (α, i) to $(\beta$, i+1), with size-change d. The construction of size-change graphs ensures that they are directed acyclic graphs.

Definition 4.5. A size-change graph for a path of n terms is *descending* if for some $\alpha, \beta \in (\mathcal{F} \times \mathbb{N})$, there exists a directed path from $(\alpha, 1)$ to (β, n) that contains one edge labeled DESC and does not contain any edge labeled ASC, and $\alpha = \beta$.

A descending size-change graph corresponds to a decreasing parameter in a path. If a size-change graph is descending for a path from $t_1 = f(t_{11}, \dots, t_{1m})$ to $t_n = f(t_{n1}, \dots, t_{nm})$, then there must exist some argument index i such that $t_{ni} > t_{1i}$.

The intuition behind this definition is that if a sequence of rewrite rule applications generates a path with a descending size-change graph, that same sequence cannot be applied indefinitely, because it would cause an infinite descent for some parameter.

Definition 4.6. A size-change graph for a path of n terms is *ascending* if for some $\alpha, \beta \in (\mathcal{F} \times \mathbb{N})$, there exists a directed path from $(\alpha, 1)$ to (β, n) that contains at least one edge labeled ASC and $\alpha = \beta$.

An ascending size-change graph corresponds to an increasing parameter in a path. If a path from $t_1 = f(t_{11}, \dots, t_{1m})$ to $t_n = f(t_{n1}, \dots, t_{nm})$, has an argument index i such that $t_{ni} > t_{1i}$, then the size-change graph must be ascending.

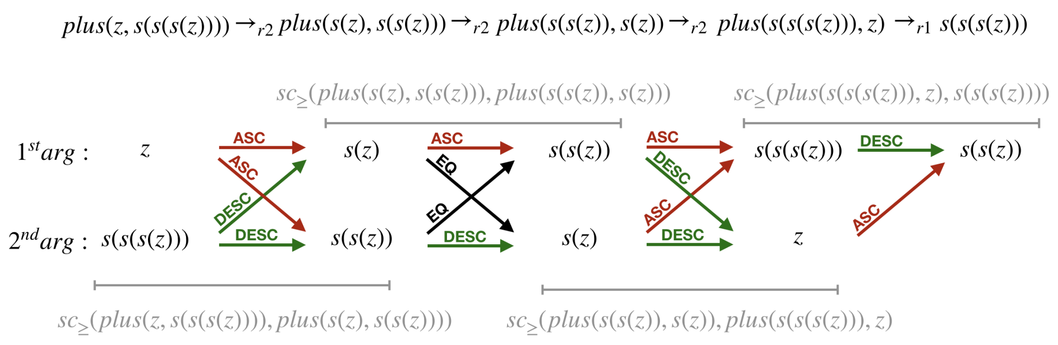
$plus(z, s(s(s(z)))) \rightarrow_{r2} plus(s(z), s(s(z))) \rightarrow_{r2} plus(s(s(z)), s(z)) \rightarrow_{r2} plus(s(s(s(z))), z) \rightarrow_{r1} s(s(s(z)))$
$sc_{\geq}(plus(s(z), s(s(z))), plus(s(s(z)), s(z)))$
$sc_{\geq}(plus(s(s(s(z))), z), s(s(s(z))))$
$1^{st} arg :$
$2^{nd} arg :$
ASC
DESC
EQ
$sc_{\geq}(plus(z, s(s(s(z)))), plus(s(z), s(s(z))))$
$sc_{\geq}(plus(s(s(z)), s(z)), plus(s(s(s(z))), z)$

Fig. 1. Size Change Graph for $plus(z, s(s(s(z))))$.

The intuition for ascending size-change graphs is that all infinite paths must contain a subsequence ts' such that $scg_{\geqslant}(ts')$ is ascending.

Definition 4.7. A path of terms ts *size-change diverges* with respect to $\geqslant$ if it contains a subsequence ts' such that $scg_{\geqslant}(ts')$ is ascending and is not descending.

A set of rewrite rules $\mathcal{R}$ is size-change diverging with respect to $\geqslant$ if it admits a path of terms that size-change diverges with respect to $\geqslant$.

The notion of size-change diverge corresponds to a sequence of rewrite rules that can be applied indefinitely to create an infinite path of distinct terms. Therefore, REST relies on this property to detect divergence. However, even a normalizing set of rewrite rules could yield a path that is size-change diverging.

Theorem 4.8. *If ts diverges with respect to $\geqslant$ and ts is a subsequence of ts', then ts' also diverges with respect to $\geqslant$.*

Proof. Follows by transitivity of subsequence. □

Example. As an example of size change analysis, consider an operator $plus$ of arity 2, s of arity 1 and operator z of arity 0. Suppose we have the following rewrite rules:

(1) $r_1 = plus(x, z) \longrightarrow x$
(2) $r_2 = plus(x, s(y)) \longrightarrow plus(s(x), y)$

Starting from the term $plus(z, s(s(s(z))))$ we get the following rewrite sequence:

$$plus(z, s(s(s(z)))) \longrightarrow_{r_2} plus(s(z), s(s(z))) \longrightarrow_{r_2} plus(s(s(z)), s(z)) \longrightarrow_{r_2} plus(s(s(s(z))), z) \longrightarrow_{r_1} s(s(s(z)))$$

Figure 1 summarizes the size change graph for this sequence, that can be visualized as the corresponding vertices between each adjacent terms. Edges labeled by ASC are drawn in red, EQ are drawn in black, and DESC are drawn in green.

The size change edges in the from $plus(z, s(s(s(z))))$ to $plus(s(z), s(s(z)))$ is as follows:

(1) $(plus, 1), (plus, 1)$, ASC), because $z \ngeqslant^* s(z)$.
(2) $(plus, 1), (plus, 2)$, ASC), because $s(s(z)) \ngeqslant^* z$.
(3) $(plus, 2), (plus, 1)$, DESC), because $s(s(s(z))) >^* s(z)$.
(4) $(plus, 2), (plus, 2)$, DESC), because $s(s(s(z))) >^* s(s(z))$.

Note that $(plus, 1)$ and $(plus, 2)$ corresponds to the first and second arguments of $plus$ respectively. Since recursive calls to plus are always increasing in the first argument and decreasing in the second, top-level applications of the rule r_2 will always result in an ASC edge from $(plus, 1)$ to $(plus, 1)$ and an DESC edge from $(plus, 2)$ to $(plus2)$.

The size change edges from $plus(s(z), s(s(z)))$ to $plus(s(s(z)), s(z))$ are as follows:

(1) $(plus, 1), (plus, 1)$, ASC), because $s(z) \not\geq^* s(s(z))$.
(2) $(plus, 1), (plus, 2)$, EQ), because $s(z) \not\geq^* s(z)$ and $s(z) \not\geq^* s(z)$.
(3) $(plus, 2), (plus, 1)$, EQ), because $s(s(z)) \not\geq^* s(s(z))$ and $s(s(z)) \not\geq^* s(s(z))$.
(4) $(plus, 2), (plus, 2)$, DESC), because $s(s(z)) >^* s(z)$.

The EQ edges arise from the fact that at this point in the path, the arguments to $(plus, 1)$ and $(plus, 2)$ have transposed.

The size change graph from $plus(s(s(z)), s(z))$ to $plus(s(s(s(z))), z)$ is as follows:

(1) $(plus, 1), (plus, 1)$, ASC), because $s(s(z)) \not\geq^* s(s(s(z)))$.
(2) $(plus, 1), (plus, 2)$, DESC), because $s(s(z)) >^* s(z)$.
(3) $(plus, 2), (plus, 1)$, ASC), because $s(z) \not\geq^* s(s(s(z)))$.
(4) $(plus, 2), (plus, 2)$, DESC), because $s(z) >^* z$.

The final call from $plus(s(s(s(z))), z)$ to $s(s(s(z)))$

(1) $(plus, 1), (s, 1)$, DESC), because $s(s(s(z))) >^* s(s(z))$.
(2) $(plus, 2), (s, 1)$, ASC), because $z \not\geq^* s(z)$.

This sequence $plus(z, s(s(s(z)))), plus(s(z), s(s(z))), plus(s(s(z)), s(z)), plus(s(s(s(z))), z), s(s(s(z)))$ does not diverge with respect to $\geqslant$, because for each subsequence ts' of ts either:

(1) $scg_{\geqslant}(ts')$ does not contain an ascending size-change path
(2) $scg_{\geqslant}(ts')$ contains a descending size-change-path

Subsequences containing zero or one term trivially meet the first condition. The remaining subsequences of ts can be divided into two distinct sets: those that contain the final term $s(s(s(z)))$, and those that do not.

For every ts' that includes the final term, there is only one vertex that corresponds to the final term (namely, $((s, 1), n)$. The vertices that correspond to the initial term are $((plus, 1), 1)$ and $((plus, 2), 1)$. Since $(s, 1) \neq (plus, 1)$ and $(s, 1) \neq (plus, 2)$, the graph is not ascending.

For every ts' that does not include the final term, there exists a descending size-change graph, because there is a path from the vertices $((plus_2), 1), ((plus_2, 2)), \ldots$ that takes only DESC edges.

Because the graph is either descending or not ascending for any subsequence of ts, the path ts does not size-change diverge.

Diverging Example. As an example of a diverging sequence, let's examine another set of rules:

(1) $r_1 = f(s(x)) \longrightarrow f(x)$
(2) $r_2 = g(s(x)) \longrightarrow g(x)$
(3) $r_3 = f(x) \longrightarrow g(x)$
(4) $r_4 = g(x) \longrightarrow f(s(x))$

Figure 2 presents the size change path for the sequence $f(s(s(z))) \longrightarrow_{r_1} f(s(z)) \longrightarrow_{r_3} g(s(z)) \longrightarrow_{r_2} g(z) \longrightarrow_{r_4} f(s(z))$. This sequence of terms diverges because it contains a size-change circuit, which starts and ends at $f_1 = s(z)$, and contains an ASC edge from $g_1 = z$ to f_1.

4.3 Correctness and Completeness

Lemma 4.9 (Invariant of REST). *In* REST$(\mathcal{R}_U, \mathcal{R}_E, t_0)$*, if* $(ts, t) \in P$ *then* $ts +\!\!+ [t]$ *is a path of* $\mathcal{R}_U \cup \mathcal{R}_E$ *starting from* t_0.

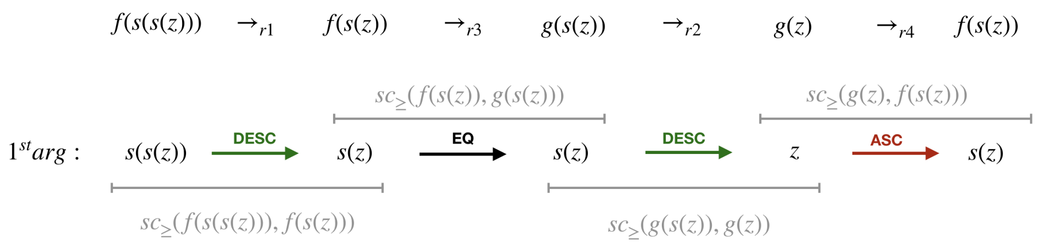

Fig. 2. Size Change Graph for $f(s(s(z))) \rightarrow_{r_1} f(s(z)) \rightarrow_{r_3} g(s(z)) \rightarrow_{r_2} g(z) \rightarrow_{r_4} f(s(z))$.

Proof. By induction on the loop iterations of the algorithm. P is initialized with the single element $([], t_0)$. $[] +\!\!+ [t_0] = [t_0]$ and $[t_0]$ is a valid path of $\mathcal{R}_U \cup \mathcal{R}_E$, 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 P is equal to (ts, t) at the beginning of the loop. The element to be pushed is a pair $(ts +\!\!+ [t], t')$ where $t \rightarrow_{\mathcal{R}_U \cup \mathcal{R}_E} t'$. This exactly satisfies the inductive hypothesis: if $ts +\!\!+ [t]$ is a path of $\mathcal{R}_U \cup \mathcal{R}_E$, then $(ts +\!\!+ [t]) +\!\!+ [t']$ is also a path of $\mathcal{R}_U \cup \mathcal{R}_E$. Furthermore, this operation preserves the head of the list: t_0 is still the first element. □

Theorem 4.10 (Correctness of REST). *If $u \in \text{REST}(\mathcal{R}_U, \mathcal{R}_E, t_0)$, then $t_0 \rightarrow^*_{\mathcal{R}_U \cup \mathcal{R}_E} u$.*

Proof. In each iteration of REST, the term t added to the output o is the second element of the pair $(ts, t) \in P$. By Lemma 4.9, t must be on the path of $\mathcal{R}_U \cup \mathcal{R}_E$ starting from t_0. □

Next we prove completeness of REST with respect to the two sets of rewrite rules.

Theorem 4.11 (Completeness of REST wrt $\mathcal{R}_E$). *For all sets of rewrite rules $\mathcal{R}_U, \mathcal{R}_E$ and terms t, u, if $\mathcal{R}_E$ is strongly normalizing and $t \rightarrow^*_{\mathcal{R}_E} u$, then $u \in \text{REST}(\mathcal{R}_E, \mathcal{R}_U, t)$.*

Proof. By Lemma 4.9, every (ts, t') in P represents a path of terms such that $ts +\!\!+ [t']$ is in the path of $\mathcal{R}_U \cup \mathcal{R}_E$. Furthermore, observe that the output o will contain the term t' for every such pair that ever appears in P.

Therefore, it is sufficient to show that at some point in the execution of REST , there is a (ts, t') in P such that $t' = u$. Let $t_0 \rightarrow_{\mathcal{R}_E} t_1 \rightarrow_{\mathcal{R}_E} \cdots \rightarrow_{\mathcal{R}_E} t_n \equiv u$ be the path of the assumed rewrite $t_0 \rightarrow^*_{\mathcal{R}_U \cup \mathcal{R}_E} u$. By induction on the length of the path, we show that we have $([t_0, \ldots, t_{n-1}], t_n)$ in P at some point of the algorithm.

For the base case, where $t = u$, P is initialized with $[([], u)]$.

For the inductive case, we assume that $t_0, \ldots, t_{n-1}$ is a path of $\mathcal{R}_E$ and that $([t_0, \ldots, t_{n-2}], t_{n-1})$ is in P. We want to show that if $t_{n-1} \rightarrow_{\mathcal{R}_E} t_n$, then $([t_0, \ldots, t_{n-1}], t_n)$ is also in P. Because the rule to generate t_n from t_{n-1} comes from $\mathcal{R}_E$, we know that $([t_0, \ldots, t_{n-1}], t_n)$ will be added to P, if $t_n \notin t_0, \ldots, t_{n-1}$. However, $t_n \notin t_0, \ldots, t_{n-1}$ will always hold. Otherwise, $\mathcal{R}_E$ would admit an infinite sequence, which contradicts strongly-normalizing hypotheses. So, the inductive case holds. □

Theorem 4.12 (Completeness of REST wrt $\mathcal{R}_U$). *For all sets of rewrite rules $\mathcal{R}_U, \mathcal{R}_E$ and terms t, u, if $t \rightarrow^*_{\mathcal{R}_E \cup \mathcal{R}_U} u$ and there exists an ordering over terms $>$ such that $\mathcal{R}_E \cup \mathcal{R}_U$ does not size-change diverge with respect to $\geqslant^*$, then $u \in \text{REST}(\mathcal{R}_E, \mathcal{R}_U, t)$.*

Proof. This proof follows essentially the same structure as the proof of Theorem 4.11. In Theorem 4.11, the `if` clause would always be entered because the rewrite rule came from $\mathcal{R}_E$, in this case the clause is always entered because there is an ordering $>$ such that any path in $\mathcal{R}_E \cup \mathcal{R}_U$ does not size-change diverge with respect to $\geqslant^*$.

However, in this case there is a possibility of encountering duplicates in the list. We can show that if a term appears in a path from t_0 that contains duplicates, it also appears in a duplicate-free path from t_0. Assume that there is a path of terms $ts = t_0, \ldots, t_i, \ldots, t_j, \ldots, t_k$ such that $i < j, t_i = t_j$ and $t_k \notin ts$. If we remove the cycle (i.e the terms from t_{i+1} to t_j), the resulting path is still a valid path.

Because the algorithm collects terms from all duplicate-free paths, and allowing duplicated terms in paths does not enable the generation of new terms, REST $(\mathcal{R}_E, \mathcal{R}_U, t)$ must contain all the terms u such that $t \rightarrow^*_{\mathcal{R}_E \cup \mathcal{R}_U} u$. We note that a similar argument cannot apply to Theorem 4.11 that requires strong normalization of $\mathcal{R}_E$: $\mathcal{R}_U$ could size-change diverge even if it was strongly normalizing. In that case, there would be no ordering over functions to prevent size-change divergence, but the algorithm would still accept terms generated via $\mathcal{R}_U$. □

4.4 Termination

Lemma 4.13. *Given a finite set of rewrite rules $\mathcal{R}$ and an initial term t_0 there exists an n such that for all well-founded orderings $>$ over $\mathcal{F}$, all duplicate-free paths ts with length greater than n diverge with respect to $\geqslant^*$.*

Proof. We provide an algorithm that generates the longest duplicate-free list ts that does not diverge with respect to $\geqslant^*$ and prove its termination. The termination of this algorithm depends on the fact that the set of well-founded orderings over $\mathcal{F}$ is finite, and that any infinite duplicate-free path of $\mathcal{R}$ will eventually size-change diverge with respect to a well-quasi-founded ordering over terms.

Suppose we have a partial function *oracle* that for any path $ts = t_0, \ldots, t_i$ of $\mathcal{R}$, $oracle(ts)$ generates the next element t_{i+1}, such that $t_0, \ldots, t_{i+1}$ is a prefix of the longest duplicate-free path ts' of $\mathcal{R}$ where ts' does not diverge with respect to $\geqslant^*$ for any ordering $>$ over $\mathcal{F}$. If there is no such term, $oracle(ts)$ is not defined.

The algorithm is defined as follows:

```
$ts \leftarrow [t_0]$ ;
while $oracle(ts)$ is defined do
    $ts \leftarrow ts \mathbin{+\!\!+} [oracle(ts)]$ ;
return $ts$;
```

At the initial iteration of our loop, there must be a candidate ordering $>$, such that ts does not size-change diverge with respect to $\geqslant^*$. If the oracle generates a new term, then either:

(1) ts diverges with respect to $\geqslant^*$, or
(2) ts does not diverge with respect to $\geqslant^*$.

In the first case, we are now certain that the ordering $>$ is an invalid candidate ordering at any future iteration of the loop (by Theorem 4.8). There must exist some other candidate ordering $>'$ where $ts \mathbin{+\!\!+} [oracle(ts)]$ does not diverge with respect to $\geqslant^{'*}$.

By Theorems 4.3 and 4.2, we know that the second case is not applicable indefinitely. Eventually we must generate the maximum finite list ts' that satisfies the call graph and the next additional element will cause it to diverge.

We can now show that $oracle(ts)$ must eventually be undefined (ensuring that the algorithm terminates). If $oracle(ts)$ was always defined, then the loop would never terminate, and the first case would be encountered an infinite amount of times. This is impossible, since there is only a finite amount of well-founded orderings over $\mathcal{F}$. Therefore, by contradiction, there must be a point when $oracle(ts)$ is undefined.

By the definition of the oracle function, the generated list has length n. □

Theorem 4.14 (Termination of REST). *For each term t_0 and finite sets of rewriting rules $\mathcal{R}_U$ and $\mathcal{R}_E$, if $\mathcal{R}_E$ are strongly normalizing, then* REST$(\mathcal{R}_U, \mathcal{R}_E, t_0)$ *terminates.*

Proof. By Lemma 4.13, there exists some natural number $maxlen$ such that for all duplicate-free paths ts of $\mathcal{R}_E \cup \mathcal{R}_U$ with more than $maxlen$ elements, ts size-change diverges with respect to $\geqslant^*$ for all well-founded orderings $>$ over $\mathcal{F}$.

First, note that at each iteration of the loop, an element (ts, t) is popped from P and a finite number of additional elements (ts', t') is pushed to P. The length of ts' is strictly greater than the length of ts.

Our goal is to show that the stack P will decrease in size. Specifically, we seek to prove that if an element (ts, t) is popped from the stack with elements P at the top of the loop, leaving remaining elements P' in the stack, after a finite amount of iterations of the loop, eventually the stack will consist of only P' at the top of the loop. Since the stack is initialized with a single element $([], t_0)$ and the algorithm terminates when the stack is empty, this is sufficient to ensure termination.

We prove this with induction on the length of the list ts. To prove the base case, we show that if ts has length $maxlen - 1$, the stack will eventually become P'. Since the length of $ts \mathbin{+\!\!+} [t]$ is greater than or equal to $maxlen$, the inner if statement will only accept rules from $\mathcal{R}_E$, since other lists $ts' \mathbin{+\!\!+} [t]$ would diverge due to their length being larger than $maxlen$.

At that point, only rules from $\mathcal{R}_E$ are accepted, and since $\mathcal{R}_E$ is a strongly normalizing rewrite sequence, the set of terms t' such that $t \rightarrow^*_{\mathcal{R}_E} t'$ must be finite and therefore the loop will eventually enter a state where the stack is P'.

For the inductive case, assume that if ts has length n, then the stack will become P'. We must show that this same property holds when ts has length $n - 1$. During an iteration of the loop, (ts, t) is removed from the stack and a finite amount m of pairs (ts', t') are added to the stack, where ts' has length n. By applying the inductive hypothesis m times, we prove the inductive case. □

5 IMPLEMENTATION OF REST

We implemented REST as an extension to the back-end of Liquid Haskell that interacts with the SMT solver, in approximately 250 lines of code. The implementation of REST is an extension to the existing PLE implementation (§ 5.1) based on a user specified set of rewrite rules (§ 5.2).

5.1 Proof By Logical Evaluation, Extended with Rewriting

Liquid Haskell, via refinement typing rules, reduces program verification to SMT-validity checking of implications of the form $\Phi \Rightarrow p$, where p is a logical predicate and Φ is a set of logical predicates. To ensure decidability of SMT queries, and thus predictable program verification, Liquid Haskell is not directly encoding Haskell functions to SMT axioms (as opposed to other SMT-based verifiers *e.g.,* F* [Swamy et al. 2016] or Dafny [Leino 2010]). Instead, it is using Proof By Logical Evaluation (PLE) [Vazou et al. 2017] to achieve terminating and complete verification over user defined functions, by strengthening the implication environment Φ.

Figure 3 presents the PLE algorithm and the extension (in blue) to allow for rewriting. PLE$(\mathcal{R}, \Psi, \Phi, p)$ decides the validity of the implication $\Phi \Rightarrow p$ given two parameters Ψ and $\mathcal{R}$:

- The function environment Ψ contains one entry of the form $f \mapsto \lambda\overline{x}.\overline{p \Rightarrow b}$ for each reflected function. The entry notation means that the function $f\ \overline{t}$ reduces to $b_i[\overline{x}/\overline{t}]$ when the guard

```
PLE                           :  $(\mathcal{R}, \Psi, \Phi, p) \longrightarrow$ Bool

PLE($\mathcal{R}, \Psi, \Phi, p$)             =  loop($0, \Phi \cup \bigcup_{f\ \overline{t} \preccurlyeq p}$ Instantiate($\mathcal{R}, \Psi, \Phi, f, \overline{t}$))
  where
    loop($i, \Phi_i$)
    | SMTValid($\Phi_i, p$)       =  true
    | $\Phi_{i+1} \subseteq \Phi_i$            =  false
    | otherwise               =  loop($i + i, \Phi_{i+1}$)
      where $\Phi_{i+1}$           =  $\Phi \cup$ Unfold($\mathcal{R}, \Psi, \Phi_i$)

Unfold                        :  $(\mathcal{R}, \Psi, \Phi) \longrightarrow \Phi$

Unfold($\mathcal{R}, \Psi, \Phi$)             =  $\Phi \cup \bigcup_{f\ \overline{t} \preccurlyeq \Phi}$ Instantiate($\mathcal{R}, \Psi, \Phi, f, \overline{t}$)

Instantiate                   :  $(\mathcal{R}, \Psi, \Phi, f, \overline{t}) \longrightarrow \Phi$

Instantiate($\mathcal{R}, \Psi, \Phi, f, \overline{t}$)   =  $\{(f(\overline{x}) = b_i)[\overline{t}/\overline{x}] \mid (p_i \Rightarrow b_i) \in d,$ SMTValid$(\Phi \Rightarrow p_i[\overline{t}/\overline{x}])\}$
{- rewrite extension -}       $\cup$  $\{f\ \overline{t} = t' \mid f\ \overline{t} \longrightarrow_{\mathcal{R}} t' \wedge$ SCTerm $t'\}$
  where $\lambda\overline{x}.d$              =  $\Psi(f)$
```

Fig. 3. PLE algorithm from [Vazou et al. 2017] extended with rewriting.

$p_i[\overline{x}/\overline{t}]$ is SMT-valid. For example, the function `f x = if x < 0 then 0 else f (x - 1)` gets encoded as $f \mapsto \lambda x.\{x < 0 \Rightarrow 0, \neg(x < 0) \Rightarrow f\ (x - 1)\}$.

- The set $\mathcal{R}$ of user provided rewrite rules as defined by the user derived `rewriteWith` pragmas.

PLE first strengthens the environment Φ with the instantiations of all function redexes within the predicate p (denoted as $f\ \overline{t} \preccurlyeq p$) and then enters a loop with the strengthened environment, *i.e.,* loop(i, Φ_i). If $\Phi_i \Rightarrow p$ is SMT valid, it returns true, otherwise it calls Unfold$(\mathcal{R}, \Psi, \Phi_i)$ to generate the environment Φ_{i+1}. If the environment has not increased (*i.e.,* $\Phi_{i+1} \subseteq \Phi_i$) then it terminates with false (*i.e.,* $\Phi \not\Rightarrow p$). Otherwise, it loops with the unfolded environment.

The function Unfold$(\mathcal{R}, \Psi, \Phi)$ extends the logical assumption Φ with the unfoldings of all the redexes appearing in the assumption (noted as $f\ \overline{t} \preccurlyeq \Phi$).

Finally, the function Instantiate$(\mathcal{R}, \Psi, \Phi, f, \overline{t})$, the only one that was modified, "instantiates" the redex $f\ \overline{t}$, that is, it derives the equalities of $f\ \overline{t}$ that will strengthen the logical assumption. If Ψ maps f to the entry $\lambda\overline{x}.\overline{p \Rightarrow b}$, then the equality $(f(\overline{x}) = b_i)[\overline{t}/\overline{x}]$ is introduced for each $p_i[\overline{t}/\overline{x}]$ that is SMT valid.

We extended the instantiation function to perform rewriting. That is, the instantiation of $f\ \overline{t}$ now also includes the equalities $f\ \overline{t} = t'$, if $f\ \overline{t}$ rewrites to t' for some rewrite rule in $\mathcal{R}$, written as $f\ \overline{t} \longrightarrow_{\mathcal{R}} t'$ (§ 3). The check SCTerm t' very compactly (for space) encodes the size-change termination check elaborated in § 4. Concretely, for our implementation to terminate, we need to ensure that the size-change graph of the rewrites does not diverge (for some ordering; see the `if` condition of Algorithm 1). Our implementation keeps track of all the equality paths generated by Instantiation, both from function unfoldings and rewrites, and checks that the graph does not size-change diverge before it adds a new rewrite. For clarity of exposition, we do not explicitly encode the bookkeeping of the paths in Figure 3; instead we encode the divergence check using

SCTerm. Given this interpretation of SCTerm, it is straightforward to argue that the critical properties of completeness, correctness, and termination proved in § 4 also hold for our implementation.

5.2 Derivation of Rewrite Rules

Next, we explain how $\mathcal{R}$, the extra argument to the PLE algorithm is generated.

Potential Rewrite Rules. The potential elements of the set $\mathcal{R}$ of rewriting rules in the implementation are user-defined, top-level functions with refinements of the form:

```
{-@ rrule :: x1:t1 → x2:t2 → ... → xn:tn → {v:() | el = er} @-}
```

The specification `rrule` can be proven correct after the user provides a definition that type checks with Liquid Haskell. The correctness of `rrule` implies that the expressions `el` and `er` are equal and thus can be used as a rewrite rule. Concretely, using $FV(e)$ to capture the free variables of the expression e, if $FV(e_r) \subseteq FV(e_l)$ and $e_l \notin \{x_1, \ldots, x_n\}$ then $e_l \to e_r$ is a potential rewrite rule. Symmetrically, if $FV(e_l) \subseteq FV(e_r)$ and $e_r \notin \{x_1, \ldots, x_n\}$ then $e_r \to e_l$ is also a potential rewrite rule.

Activation of Rewrite Rules. The user can activate a rewrite rule globally, *i.e.,* to be used to strengthen the environment of each generated verification condition, by using the pragma `rewrite`:

```
{-@ rewrite [rrule] @-}
```

Alternatively, the user can activate a rewrite rule locally, per function, by using the pragma `rewriteWith`, *e.g.,*

```
{-@ rewriteWith theorem [rrule] @-}
```

will only activate the rule `rrule` when verifying the `theorem` function.

Preventing Circular Reasoning. Our implementation takes care to ensure that rewrites cannot be used to justify circular reasoning. For example, the below, unsound, circular dependency will be rejected with a rewrite error by our system.

```
{-@ rewriteWith p1 [p2] @-}
{-@ p1 :: x:Int → { x = x + 1 } @-}
p1 _ = ()
```

```
{-@ rewriteWith p2 [p1] @-}
{-@ p2 :: x:Int → { x = x + 1 } @-}
p2 = p1
```

In general, to prevent circular dependencies, we analyze the dependency graph of the rewrite rules and ensure that functions used as rewrites do not depend on proofs that refer to them.

5.3 Current and Future Optimizations

In the case of a diverging rewrite system, REST iterates over every partial ordering of $\mathcal{F}$ in order to ensure that the corresponding size-change graph diverges. While this is manageable when the number of function symbols is small, the number of partial orderings grows quickly with respect to the number of function symbols. For example, there are 4,231 partial orderings for a set of five operators, but 431,723,379 partial orderings for a set of eight [Sloane 2020]. Therefore, our implementation provides an option to check for size-change divergence using a subset of all possible orderings, in which case it will only check orderings that are defined over subsets of $\mathcal{F}$ having a maximum size dictated by the user. The end result is that REST can be made to terminate faster, but may fail to generate the necessary equalities to discharge the proof.

One performance improvement (not present in our implementation) is to add a "target" term u such that REST can terminate immediately if $u \in o$. Alternatively, because our rewriting is used in the context of solving individual proofs, we could check for satisfiability with SMT during the execution of the algorithm, and abort when enough information is available to prove the theorem or its negation. Either of the above modifications could potentially enable a further efficiency improvement, by using a priority queue in place of a stack in REST. Paths that are more likely to lead to the target according to some heuristic could be prioritized.

Another potential performance improvement is to prioritize particular orderings over $\mathcal{F}$ when checking for size-change divergence. In order to decide to include a rewrite rule, REST must show that there is an ordering over $\mathcal{F}$ that does not cause the path to size-change diverge. First considering orderings that are less likely to cause divergence (for example, by choosing one that was statically proven to ensure termination) would improve performance.

6 IMPLEMENTATION OF THE INDUCTION TACTIC

In this section we describe the implementation of an inductive tactic that is using metaprogramming of Template Haskell [Sheard and Peyton Jones 2002] to automate the boilerplate code of structurally inductive proofs. The implementation and design of the inductive tactic is orthogonal to the rewrite tactic. That said, the metafunction we designed allows invocations to the rewrite tactic, thus the combination of the two tactics leads to concise and highly automated proofs (§ 7).

Induction On Single Argument Functions. Consider the simple inductive property that list appending is right identity:

```
{-@ rightIdP :: xs:[a] → { xs ++ [] == xs } @-}
rightIdP :: [a] → ()
rightIdP []     = ()
rightIdP (_:xs) = rightIdP xs
```

The proof goes trivially by structural induction: it exhaustively case splits over the inductive list argument. For the base case the proof concludes by function unfolding (automated by PLE), while for the inductive case it also requires the inductive hypothesis.

We used Template Haskell to define the Liquid Haskell Prover metafunction `lhp` that automates these simple steps. Using `lhp` the `rightIdP` is automated as follows:

```
[lhp|caseExpand|induction
rightIdP   :: [a] → ()
rightIdP _ = ()          |]
```

The `caseExpand` option is doing case expansion on the argument of the function. Concretely, it is using the type of the argument (here `[a]`) to query the Haskell compiler GHC for the constructors of the type and generates one case per constructor. The `induction` option applies the inductive hypothesis on each inductive case. That is, for the `rightIdP` proof, it will generate the `rightIdP xs` call for the `(x:xs)` case. Hence, the code generated by the usage of `lhp` as above, will be exactly the same as the inductive, user-provided proof for `rightIdP`.

Induction On Multi Argument Functions. Functions with multiple recursive parameters require more sophisticated treatment. For example, consider the list associativity proof below that is defined over three inductive, list arguments.

```
{-@ assocP :: x:[a] → y:[a] → z:[a] → { x ++ (y ++ z) == (x ++ y) ++ z } @-}
assocP :: [a] → [a] → [a] → ()
assocP [] _ _     = ()
assocP (_:x) y z = assocP x y z
```

A naïve application of the inductive tactic leads to an excessively verbose proof. Concretely, the below naïve call leads to $8 = 2^3$ cases, *i.e.*, the cross product of the expansion of each argument.

```
[lhp|caseExpand|induction
assocP :: [a] → [a] → [a] → ()
assocP _ _ _ = ()                  |]
```

The body of each case is implemented with calls to inductive hypotheses whose number range from 0 to 7, depending on the case: In the base case `assocP [] [] []` no inductive calls are generated. In inductive cases with one cons, *e.g.*,`assocP (x₁:xs₁) [] []`, only one inductive call is generated: `assocP xs₁ [] []`. In inductive cases with two cons, *e.g.*,`assocP (x₁:xs₁) (x₂:xs₂) []`, three inductive calls are generated: `assocP xs₁ (x₂:xs₂) []`, `assocP (x₁:xs₁) xs₂ []`, `assocP xs₁ xs₂ []`. Finally, in the inductive case with three cons, *i.e.*,`assocP (x₁:xs₁) (x₂:xs₂) (x₃:xs₃)` seven calls are generated: `assocP xs₁' xs₂' xs₃'`, where $\texttt{xs}_i\texttt{'} = \texttt{x}_i\texttt{:xs}_i$ or $\texttt{xs}_i$, excluding the non well-founded call where all three arguments are cons.

In general, the generated inductive calls for a function `f` with arguments $\texttt{x}_1, \ldots \texttt{x}_n$ are all the calls $\texttt{f sub(x}_1\texttt{) ... sub(x}_n\texttt{)}$, where $\texttt{sub(x}_i\texttt{)}$ contains $\texttt{x}_i$ and all its structural subterms, while in each call there should exist at least one i for which $\texttt{sub(x}_i\texttt{)}$ is not equal to $\texttt{x}_i$.

This excessive information is sufficient but often not required. For instance, for the `assocP` proof, our tactic will generate 8 cases with $0-7$ inductive calls, while only two cases with 0 and 1 inductive calls, respectively, are required. In practice, the extra information highly increases the verification time. To reduce the excessive information, the `lhp` tactic admits a `caseExpandP:1` option forcing induction to occur only on the first argument. Thus, the `assocP` that only requires induction on the first argument, will get metaprogrammed as follows:

```
[lhp|induction|caseExpandP:1
assocP :: [a] → [a] → [a] → ()
assocP _ _ _ = ()                  |]
```

The `caseExpandP:i` option is used to force induction on the first `i` arguments. For example, in `assoc2` below, the proof goes by induction on the first 2 arguments.

```
{-@ assoc2 :: x:[a] → y:[a] → z:[a] → w:[a]
           → { x ++ (y ++ (z ++ w)) == ((x ++ y) ++ z) ++ w } @-}
[lhp|induction|caseExpandP:2
assoc2 :: [a] → [a] → [a] → [a] → Bool
assoc2 _ _ _ _ = () |]
```

Intuitively, induction on the second argument proves associativity on `y z w` and induction on the first argument completes the proof.

7 EVALUATION

This section evaluates LH+, *i.e.*, Liquid Haskell extended with REST and the inductive tactic of § 5 and § 6, respectively. Our evaluation addresses three main questions:

Q1: How does REST *compare to existing rewriting tactics?* § 7.1 compares our rewriting tactic, implemented in the context of an SMT solver, with rewriting tactics of standard theorem provers.

Q2: Does REST *simplify common general problems?* § 7.2 combines our rewriting tactic with the induction tactic to prove benchmarks from the Tons of Inductive Problems (TIP) [Claessen et al. 2015] benchmark suite and to showcase that our two tactics 1. work well with each other and 2. greatly automate common inductive problems.

Q3: Does REST *simplify Liquid Haskell proofs?* § 7.3 presents Liquid Haskell proofs over three different datatypes and compares the proofs with and without the rewrite and induction tactics.

Property	LH+	Coq	Agda	Lean	Isabelle/HOL	Zeno	CVC4
Diverge	OK	loops	loops	fails	loops	OK	loops
List Associativity	OK	OK	OK	OK	OK	OK	OK
List Identity	OK	loops	OK	OK	OK	OK	OK
Plus AC	OK	loops	loops	fails	fails	OK	loops
Optimization	OK	OK	OK	OK	OK	fails	OK

Fig. 4. Comparison of REST with existing rewriting techniques. LH+ is Liquid Haskell with the novel rewrite and induction tactics the rest are commonly used theorem provers with existing rewriting techniques. The potential outcomes are **OK** when the property is proved; **loops** when the prover does not produce any result after 300 seconds; and **fails** when the property cannot be proven.

7.1 Comparison with Existing Rewriting Techniques

Figure 4 is using five toy examples to compare REST against six proving systems that support rewriting. Coq [Coq Development Team 2020], Agda [Norell 2008], Lean [Avigad et al. 2018], and Isabelle/HOL [Nipkow et al. 2020] allow user-defined rewrites. In Lean and Isabelle/HOL , the tactic for applying rewrite rules multiple times is called `simp`; for simplification. Coq, Agda, 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. Lean's approach is somewhat similar to ours: it defines a well-founded ordering over terms and only applies a rewrite if the rewritten term is less than the original one in the ordering [Avigad et al. 2018].

Zeno [Sonnex et al. 2012] and CVC4 [Barrett et al. 2011] do not allow for user-defined rewrite rules. Our theorems in these cases are expressed as implications, where the rewrite rules are encoded as universally-quantified antecedents. Our implementation of rewriting is similar to that of CVC4 in that in both systems equalities are added to an SMT environment. But, while our system generates equalities based on unification, CVC4 generates equalities based on quantifier instantiation using heuristics such as E-matching [de Moura and Bjørner 2007].

To compare these six rewrite techniques against ours we used five toy examples. Three common list theorems, one (Diverge) to test divergence of the systems' rewrite tactic, and one (Optimization) to test compatibility with generalized theorem proving. In all the cases, we used user-defined datatypes and functions to prevent the theorem prover from using built-in theories to complete the proofs. The code of the examples can be found in [Supplementary-Material 2020].

The test Diverge is designed to test if a diverging rewrite system will cause the theorem prover to loop. As expected, Coq, Agda, and Isabelle/HOL diverge on this example. Lean does not diverge, but it also fails to prove the theorem, which requires reasoning about induction. In CVC4, the divergence is caused by matching loops during quantifier instantiation.

The List Associativity test determines whether or not the law of associativity for list concatenation can be used as a rewrite to prove "associativity" over four lists. All systems are able to prove this example.

The List Identity is used to test if the equality `xs = xs ++ []` can be used as a rewrite to prove `xs = xs ++ [] ++ []` . This single equality alone leads to an infinite derivation of the form `xs` → `xs ++ []` → `xs ++ [] ++ []` → . . .. However, with the exception of Coq, all theorem provers were able to use this rewrite rule.

The test Plus AC shows that for `p`, `q`, and `r`, user-defined natural numbers, `(p + q) + r` can rewritten to `(r + q) + p`, which requires application of both associativity and commutativity. Lean is unable to use rewriting in this example, because it is unable to find a rewrite ordering that would permit applications of these rewrites. Using commutativity as a rewrite leads to a nonterminating rewrite system, therefore causing the other theorem provers to loop.

The final example `Optimization` shows that the expressions $f(g(x))$ and $f(g'(x))$ can be proven equal if there exists a rewrite rule of the form $g(x) \longrightarrow g'(x)$. This is useful for provably correct optimizations, *i.e.*, when $g'(x)$ is an optimized version of $g(x)$. In our benchmark, we defined $g(n)$ to be the integer series $1 + 2 + \ldots + n$, and $g'(n)$ as the closed-form solution $(n * (n + 1))/2$. Zeno was not able to prove the equality of $g(n)$ and $g'(n)$ automatically, and even asserting $g(n) = g'(n)$ as an axiom did not enable it to prove the trivial consequence $f(g(n)) = f(g'(n))$.

From this evaluation, we conclude that, REST, the rewriting tactic of LH+ does not trivially overlap with rewriting tactics of commonly used theorem provers.

7.2 Some Inductive Proofs

Prove Technique	Total Number	LH+		CVC4 + Ind	
		Solved	Max Time	Solved	Max Time
(1) Structural Induction	50	50	5s	46	84s
(2) General Induction	15	0	N/A	9	54s
(3) Induction & Lemmata	21	21	11s	0	N/A
Summary	86	71	11s	55	84s

Fig. 5. Evaluation of the tactics on inductive proofs. LH+ is Liquid Haskell with the two tactics and CVC4 + Ind is CVC4 with inductive quantification. Proofs in category (3) in LH+ required explicit lemmata: 11 out of 21 lemmata were equalities and 7 of them were automated using the rewrite tactic.

Next, we evaluated LH+ in a subset of the *TIP* [Claessen et al. 2015] benchmarks. We chose a subset of 86 benchmarks presented by Johansson et al. [2010] who also combine rewriting with induction to automate proofs. All the benchmarks are mathematical properties for sorting and manipulating natural numbers (as defined in § 2.1), and can be proved by case analysis and induction.

Figure 5 evaluates the performance of LH+ in this set of benchmarks and compares it with CVC4 with induction quantification enabled (we name it `CVC4 + Ind`), using the setting of Reynolds and Kuncak [2015].

The benchmarks are classified in three categories based on the required proof technique:

(1) Structural Induction: the inductive hypothesis is simply called on subterms of the arguments.
(2) General Induction: the inductive hypothesis needs non trivial calls, *e.g.*, the inductive case `f (S i) j` requires an inductive call of `f i (g j)`, for some function `g`.
(3) Induction & Lemmata: structural induction that also requires invocation of external lemmata.

LH+ provides only a tactic for structural induction (as discussed in § 6) thus can quickly and automatically prove all the problems of category (1) but none of category (2), since our induction tactic does not support general induction heuristics. On the contrary, `CVC4 + Ind` can prove 46/50 problems of category (1) and 9/15 of category (2). This is due to how the inductive automation works in CVC4 [Reynolds and Kuncak 2015]. While our tactic works on the proof structure (by automating a specific proof pattern), CVC4's works on the SMT goal where it applies skolemization with inductive strengthening and subgoal generation. Specifically, it attempts to prove the unsatisfiability of the *negated* goal by strengthening it with the smallest counter-example.

In category (3) the proofs require structural induction and invocation of external lemmata. CVC4 + Ind was unable to automatically prove any of these problems. In LH+ we were able to prove all of them, but by explicitly providing the lemmata. Out of the 21 properties that required lemmata, 11 required equality theorems and the rewrite tactic could automate 7 of them.

```
unionMono' :: m1:Set → m2:Set → m2':{Set | m2' ⊆ m2 }
          → { (m1 ∪ m2) ⊆ (m1 ∪ m2')}
unionMono' m1 m2 m2' =
(m1 ∪ m2') ∪ (m1 ∪ m2)  ? unionAssoc m1 m2' (m1 ∪ m2)
=== m1 ∪ (m2' ∪ (m1 ∪ m2))  ? unionAssoc m2' m1 m2
=== m1 ∪ ((m2' ∪ m1) ∪ m2)  ? unionComm mp m1 m2'
=== m1 ∪ ((m1' ∪ m2') ∪ m2) ? unionAssoc m1 m2' m2
=== m1 ∪ (m1 ∪ (m2' ∪ m2))
=== m1 ∪ (m1 ∪ m2) ? unionAssoc m1 m1 m2
=== (m1 ∪ m1) ∪ m2 ? unionIdemp m1
=== m1 ∪ m2
*** QED
```

Fig. 6. A proof of the monotonicity of ∪ using equational reasoning.

From this experiment we conclude that our tactics can be used to automate common inductive proofs. Most of the proofs (71/86) required structural induction, automated by our inductive tactic, while from the few cases of external, equality lemmata required, some (7/11) were automatically instantiated by our rewrite tactic.

7.3 Case Studies

Finally, we evaluate our two tactics by comparing Liquid Haskell proofs with and without the tactics over three datatypes, *i.e.,* sets (§ 7.3.1), lists (§ 7.3.2), and trees (§ 7.3.3).

7.3.1 Set Properties. In this case study, we prove that set union is monotonic with respect to the subset relation. We use the (non directly inductive) definition of a set as a list of integers.

```
type Set = [Int]

unionMono :: m1:Set → m2:Set → m2':{Set | m2' ⊆ m2 }
          → { (m1 ∪ m2) ⊆ (m1 ∪ m2') }
```

where $x \subseteq y$ is defined as $x \cup y = y$.

The proof of this theorem relies on associativity (`unionAssoc`), commutativity (`unionComm`), and idempotency (`unionIdemp`).

```
unionAssoc :: x:Set → y:Set → z:Set → { (x ∪ y) ∪ z = x ∪ (y ∪ z) }
unionComm  :: x:Set → y:Set → { x ∪ y = y ∪ x }
unionIdemp :: x:Set → { x ∪ x = x }
```

Using our rewrite technique, the proof of `unionMono` is trivial:

```
{-@ rewriteWith unionMono [unionAssoc, unionComm, unionIdemp] @-}
unionMono _ _ _ = ()
```

For comparison, Figure 6 presents the explicit (*i.e.,* without the rewrite tactic) Liquid Haskell proof. In our benchmark we compared the proof using rewriting to the one using explicit lemma application; the total lines of code were reduced from 14 to 9, while verification time changed 0.5 seconds to 4.4 seconds. The increase in time can be explained by the fact that REST generates many unused SMT equalities for lemmas such as commutativity and associativity. Since our proof is not by induction, the induction tactic as not used.

7.3.2 List Properties. Next, we present a proof that list `reverse` is an involution, *i.e.,* `reverse (reverse xs) = xs`. The Liquid Haskell proof of the property can be found in [Vazou et al. 2018] and consists of 30 LoC (with PLE). Here, we present the same proof, highly automated by our two tactics and PLE.

To begin with, we prove that list `reverse` distributes over append.

```
{-@ rewriteWith distributivityP [rightIdP , assocP] @-}
{-@ distributivityP :: xs:[a] → ys:[a]
                    → { reverse (xs ++ ys) == reverse ys ++ reverse xs } @-}
[lhp|noSpec|ple|induction|caseExpandP:1
distributivityP :: [a] → [a] → ()
distributivityP _ _ = ()        |]
```

The proof goes by induction on the first argument (as noted by `caseExpandP:1`) while it is using right identity and associativity of append (as proved in § 6).

Using `distributivityP` as a rewrite rule we prove involution of `reverse`, simply by induction.

```
{-@ rewriteWith involutionP [distributivityP] @-}
{-@ involutionP :: xs:[a] → { reverse (reverse xs) == xs } @-}
[lhp|noSpec|ple|induction|caseExpand
involutionP :: [a] → ()
involutionP _ = ()       |]
```

7.3.3 Tree Properties. In this final case study we optimize flattening of an inductive tree into a list. In particular, we prove the equivalence between a naïve flattening implementation that uses ++ and one that avoids it.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

-- Slow implementation
flatten :: Tree a → [a]
flatten (Leaf n)   = [n]
flatten (Node l r) = flatten l ++ flatten r

-- Fast implementation
flatten' :: Tree a → [a]
flatten' l = flattenApp l []

flattenApp :: Tree a → [a] → [a]
flattenApp (Leaf n)   ns = n:ns
flattenApp (Node l r) ns = flattenApp l (flattenApp r ns)
```

First we need to prove that `flattenApp t ls` behaves like `flatten t ++ ls` for any tree `t` and list `ls`. The proof goes by induction and requires the previously seen (`assocP`) associativity theorem of ++. The inductive hypothesis required is general (it has to be called on a flattened right subtree) and cannot be automated by our induction tactic, however we can use rewriting to automate the use of `assocP`, slimming down the proof to:

```
{-@ rewriteWith flattenAppP [assocP] @-}
{-@ flattenAppP :: t:Tree a → ls:[a] → { flattenApp t ls == flatten t ++ ls } @-}
flattenAppP :: Tree a → [a] → ()
flattenAppP (Leaf _)   _  =  ()
```

Properties	LH		LHRW		LH+	
	LoC	Time	LoC	Time	LoC	Time
§ 7.3.1 Set Properties	14	0.5s	9	4.4s	*9	*4.4s
§ 7.3.2 Lists Properties	30	3.2s	27	3.8s	22	7.9s
§ 7.3.3 Tree Properties	28	2.0s	27	3.0s	24	5.0s
Summary	72	5.7s	63	11.2s	55	17.3s

Fig. 7. LH, LHRW and LH+ are respectively Liquid Haskell without any tactic, Liquid Haskell with rewriting, and Liquid Haskell with both rewriting and induction tactics. **LoC** is total lines of code and comments required to express and prove the properties and helper lemmata. **Time** is verification time in seconds.
*The Set Properties could not use the induction tactic.

```
flattenAppP (Node l r) ls = flattenAppP r ls ? flattenAppP l (flattenApp r ls)
```

Finally, we prove that the two implementations `flatten` and `flatten'` are equivalent. The proof requires the use of the right identity and `flattenAppP`; hence, can be fully automated by rewriting:

```
{-@ rewriteWith flattenP [rightIdP, flattenAppP] @-}
{-@ flattenP :: t:Tree a → { flatten t == flatten' t } @-}
flattenP :: Tree → Bool
flattenP t = ()
```

Although the tactics only saved four lines of code, the important aspect was that they removed the need for manual instantiation of rewrite rules.

7.3.4 Summary. Table 7 summarizes the lines of code and verification times of our three case studies in Liquid Haskell with and without the use of the two tactics. All original LoC numbers are those of the minimal possible proof bodies with PLE enabled (intermediate steps are omitted for a fair comparison with the tactics). Our tactics have a clear advantage since they lead to compact, yet informative proofs: the required rules are mentioned, but there is no need to get explicitly instantiated by the user. This comes at the cost of a verification time slowdown. There 96% increase from LHRW was expected because rewriting is not goal-directed, and therefore makes many extraneous instantiations; the overall 204% increase when the induction tactic is added, is attributed to metaprogramming that needs to interpret the original code. We expect this to be an initialization cost, *i.e.,* it should drop for larger benchmarks with bigger original verification time.

8 RELATED WORK

8.1 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.

The Coq plugin Equations [Sozeau and Mangin 2019] enables dependent pattern matching by converting user-provided equalities into definitions and propositional equalities. PLE generated equalities for terminating user-defined functions of non dependently typed Haskell. We could, in the future, combine both techniques to target dependently typed Haskell functions (that use GADTs and type families) and use REST for rewriting over such equations.

8.2 SMT Verification & Rewriting

Our rewrite rules could be encoded in the 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.

CVC4 [Barrett et al. 2011] is a rapidly advancing SMT solver that supports both induction [Reynolds and Kuncak 2015] and rewrite [Nötzli et al. 2019] tactics. As detailed in § 7, the approach CVC4 is using for both tactics are different than ours. CVC4 uses E-Matching for rewriting and thus can diverge, and uses Skolemization with inductive strengthening as an inductive heuristic, thus addressing a different set of inductive problems than our structural induction heuristic.

8.3 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.

8.4 Size-Change Termination & Orderings

The size-change termination of REST is similar to Arts and Giesl [2000]; Thiemann and Giesl [2007]. However, unlike Thiemann and Giesl [2007] that builds a size-change graph based on static analysis of function definitions, our algorithm builds and checks the size-change graph at "runtime", *i.e.,* the graph is re-build each time the size-change-divergence condition is checked, based on rewrite applications. In fact, our implementation of dynamic size-change termination checking is similar to Nguyen et al. [2019], where termination is checked at runtime as a contract. Unlike our system which is based on rewriting, theirs is based on lambda calculus. They are able to ensure termination of a broader range of functions (like the ackermann function), because they apply size-change termination recursively on the applications in the function body. In contrast, our system only applies size-change termination on strict subterms of the outermost term. On the other hand, the semantics of rewriting are less constrained compared to those of function application in the lambda calculus. For example, our system does not demand that a rewriting system be confluent nor imposes that innermost redexes be rewritten first.

Different orderings can be used to decide side-change termination, including recursive path orderings and lexicographic path orderings [Dershowitz 1982], which were recently used, for example, in dependency pairs [Arts and Giesl 2000]. We choose to use a well-quasi-ordering [Kruskal 1972] as the basis of our size-change termination principle, because it leads to a natural notion of terms that have the same size and because we do not need to reduce terms to a normal form.

9 CONCLUSION

SMT-based verification highly and efficiently automates reasoning on decidable theories, but can become unpredictable when quantifiers are used, *e.g.,* to encode user defined functions. We extend the design space provided by PLE to allow for decidable SMT-verification in the presence of user-defined, terminating functions with rewriting automation. We used size-change termination to design REST, a terminating rewrite algorithm that we implemented and evaluated on top of Liquid Haskell. The evaluation of REST shows that 1/ it is not subsumed by existing rewriting techniques and 2/ at the expected cost of an increase in verification time, it can be used to greatly shrink proofs and ease proof development.

REFERENCES

Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. *Electronic Proceedings in Theoretical Computer Science* 43, 14–28. https://doi.org/10.4204/EPTCS.43.2

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.

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, 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)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). *t* www.SMT-LIB.org.

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

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2015. TIP: Tons of Inductive Problems, Vol. 9150. https://doi.org/10.1007/978-3-319-20615-8_23

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.

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

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

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 (Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/2804302.2804303

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

Moa Johansson, Lucas Dixon, and Alan Bundy. 2010. Case-Analysis for Rippling and Inductive Proof. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–306.

jan willem Klop. 2000. Term Rewriting Systems. (08 2000).

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 (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 (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

Phúc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, Phoenix, AZ, USA, 845–859. https://doi.org/10.1145/3314221.3314643

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 (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.

Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–98.

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh* (proceedings of the 2002 haskell workshop, pittsburgh ed.). 1–16. https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/

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

N. J. A. Sloane. 2020. The Encyclopedia of Integer Sequences.

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.

Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.

Supplementary-Material. 2020. Non Anonymous Supplementary Material.

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/

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 (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