IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL *

Hanna Lachnitt[1], Mathias Fleury[4], Leni Aniva[1], Andrew Reynolds[2], Haniel Barbosa[3], Andres Nötzli[5], Clark Barrett[1], and Cesare Tinelli[2]

[1] Stanford University, Stanford, USA
[2] The University of Iowa, Iowa City, USA
[3] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[4] University of Freiburg, Freiburg, Germany
[5] Cubist, Inc.

Abstract. Satisfiability modulo theories (SMT) solvers are widely used to ensure the correctness of safety- and security-critical applications. Therefore, being able to trust a solver's results is crucial. One way to increase trust is to generate independently checkable proof certificates, which record the reasoning steps done by the solver. A key challenge with this approach is that it is difficult to efficiently and accurately produce proofs for reasoning steps involving term rewriting rules. Previous work showed how a domain-specific language, Rare, can be used to capture rewriting rules for the purposes of proof production. However, in that work, the Rare rules had to be trusted, as the correctness of the rules themselves was not checked by the proof checker. In this paper, we present IsaRare, a tool that can automatically translate Rare rules into Isabelle/HOL lemmas. The soundness of the rules can then be verified by proving the lemmas. Because an incorrect rule can put the entire soundness of a proof system in jeopardy, our solution closes an important gap in the trustworthiness of SMT proof certificates. The same tool also provides a necessary component for enabling full proof reconstruction of SMT proof certificates in Isabelle/HOL. We evaluate our approach by verifying an extensive set of rewrite rules used by the cvc5 SMT solver.

1 Introduction

Satisfiability modulo theories (SMT) [8] solvers provide the back-end reasoning power for many formal methods applications. These applications are often used to provide safety or security guarantees for critical systems [1, 15, 21, 23]. For such applications, an incorrect result from a solver could have catastrophic consequences. Thus, ensuring the correctness of a solver's results is crucial. However, industrial-strength SMT solvers are large and complex software systems which are under constant active development. As with any other large software project,

* This work was supported in part by the Stanford Center for Automated Reasoning and by a gift from Amazon Web Services.

even when employing software engineering best practices, it is unrealistic to expect that solvers do not contain implementation bugs that could, in the worst case, compromise the correctness of their answers.

One solution is to formally verify the SMT solver itself. Unfortunately, that would be a massive effort. It would likely require performance compromises [17] and impose a tremendous maintenance burden, as changes to solvers are frequent, and each change would require revisiting the verification.

Fortunately, there is a less expensive alternative: we can independently check each result produced by a solver. This is generally easy when the result is "satisfiable," at least for quantifier-free inputs. The solver can produce a model and we can check via evaluation that the input formula indeed holds in it. To have a similar ability to check a result of "unsatisfiable," solvers must be instrumented to produce *proof certificates* that can be independently verified by a separate proof checker. To maximize trustworthiness, the proof checker should be small, simple, and, ideally, formally verified. Alternatively, the checker can be embedded in a highly trusted system such as a skeptical interactive theorem prover. The SMT community is increasingly embracing proof production, with it becoming a major focus in recent years [3, 4, 19, 29].

One of the main challenges faced by SMT proof production efforts is the extensive use of theory-specific *term rewriting rules*. There are hundreds of such rules in modern solvers, each of which must be justifiable using some proof rule. Nötzli et al. [28] introduced a methodology for producing proofs for term rewriting rules by using the RARE domain-specific language. In that work, rules are defined in RARE, imported by a solver, and then used to elaborate the solver's term rewriting proof steps into finer-grained proofs using the RARE rules. This approach has proved to be viable in the cvc5 SMT solver [2]. However, previous work did not address the *correctness* of the rules, i.e., it does not ensure that an incorrect RARE rule does not compromise the correctness of proof certificates.

An incorrect rule can have severe consequences. First of all, it may affect the ability of the solver to produce a proof certificate at all: if the incorrect rule does not match what the solver code does, then the elaboration of the term rewriting proof steps with RARE may fail. More concerningly, if both the code and the proof rule are incorrect in the same way (perhaps because one was modeled after the other), then proof elaboration may succeed, but the proof certificate will be incorrect because it uses an invalid rule. This is especially problematic when using proof checkers that consider proof rules as trusted—that is, they only check whether rules are applied correctly and do not check the rules themselves.

There are two ways to fill this gap. One is to separately verify the proof rules; another is to use a more sophisticated proof checker, for example, one embedded in a skeptical interactive theorem prover, that will fail if an invalid rule is used. In this paper, we introduce IsaRARE, a new plugin for the Isabelle/HOL proof assistant [27] (abbreviated to just Isabelle going forward), which can do the former and is a necessary step towards the latter. The plugin translates RARE rules into the language of Isabelle where they can then be formally proved as lemmas. Note that when using IsaRARE simply as a rewrite rule verifier, the translation from Rare to Isabelle becomes another trusted component. We mitigate this by reusing extensively-tested infrastructure in Isabelle for the translation.

To show the effectiveness of IsaRare, we implemented a large number of new rules in Rare (beyond those in [28]) needed to elaborate term rewriting steps in proofs generated by the cvc5 SMT solver [2]. We show that IsaRare can translate all of these rules into corresponding lemmas in Isabelle and can prove the majority of them automatically. In ongoing work, we are manually providing proofs for the rest, and have already proven most of them.

Our long-term vision is to enable the full integration of cvc5 and Isabelle via proof certificate reconstruction. Currently, Isabelle can send proof obligations to cvc5, but it is unable to automatically reconstruct Isabelle proofs from cvc5's proof certificates. Our goal is to enable Isabelle to reconstruct *every* step in these proof certificates. In order to reach this goal, it is essential to have rewrite lemmas available for reconstructing rewrite steps, as they appear in almost all proofs, and without dedicated support for discharging rewrite proof steps, reconstruction in Isabelle can fail [11, 31].

In summary, we make the following contributions:

– we introduce IsaRare, an Isabelle plugin for generating correctness lemmas for Rare rules;
– we add several new features to Rare itself and implement 163 new rewrite rules in Rare, almost tripling the size of the rule database from [28];
– we evaluate IsaRare, showing that it can translate all of the Rare rules into Isabelle lemmas and can prove the majority of them automatically.

In the rest of the paper, after surveying related work, we give an overview of proof production and the interface to Isabelle (Section 2). Then, we present the Rare language and our extensions (Section 3). We next introduce IsaRare and explain the challenges in transforming a Rare rule to an Isabelle lemma (Section 4). Finally, we present an evaluation of our approach (Section 5).

1.1 Related Work

Various attempts at proof production in SMT solvers have been implemented in the past [7, 13, 14, 22, 25], though these implementations typically either produce proofs certificates that are too coarse-grained (that is, they do not provide enough information for efficient proof checking) or produce them only if critical components are disabled, making solving while producing proofs slow or incomplete. Producing complete, independently-checkable proofs remains challenging.

One major challenge is solved by the modular framework by Barbosa et al. [3]. It enables proof production during term rewriting and formula processing and has been implemented in the SMT solver veriT [13] using the Alethe proof format [32]. Hoenicke and Schindler [19] introduce an alternative approach, implemented in the solver SMTInterpol [14], which also allows proof production for term rewriting and formula processing. Both of these approaches assume that the set of rewrite rules that can be used in proofs is fixed. Their sets include rules for rewriting over equality, rules for rewriting Boolean formulas, and rules for reasoning about arithmetic. Notably absent, however, are rules for string and bit-vector rewrites. In other work, Barbosa et al. [4] describe a general architecture where the only holes in the generated proof certificates are those from rewrite steps. One of their key ideas is to support lazy proof production via a post-processing proof reconstruction step. This capability is leveraged in the work by Nötzli et al. [28] to produce proofs for rewrite steps based on rules written in RARE, which is the starting point for this work.

The interactive theorem prover Isabelle [30] includes a popular tool called Sledgehammer [9], which encodes proof obligations as SMT problems and uses SMT solvers to solve them. Sledgehammer currently supports proof reconstruction [12, 18] for two SMT solvers: z3 [26] and veriT [13]. However, z3 provides only coarse-grained proofs, which can cause reconstruction to fail. This issue was addressed for veriT by manually translating and proving correct in Isabelle the predefined set of rewrite rules in Alethe [18, 31]. Our work improves on this effort by providing an *automatic* mechanism for translating an *extendable* set of rewrite rules into Isabelle and includes support for bit-vector and string rewrites unsupported by veriT.

2 Preliminaries

2.1 Satisfiability Modulo Theories (SMT)

The underlying logic of SMT is many-sorted first-order logic with equality (see e.g., [16]). A *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq S$ of sort symbols and a set Σ^{f} of sorted function symbols with sorts from Σ^{s}. We assume the usual definitions of well-sorted terms, literals, and formulas. We also use the usual definition of interpretations and of a satisfiability relation $\models$ between Σ-interpretations and Σ-formulas. A *Σ-theory T* is a non-empty class of Σ-interpretations closed under variable reassignment. A Σ-formula φ is *T-satisfiable* (resp., *T-unsatisfiable*, *T-valid*) if it is satisfied by some (resp., no, all) interpretation(s) in T. For the rest of the paper, we assume (un)satisfiability is always with respect to some given background theory T.

2.2 SMT Proofs and Rewriting

A *proof (of unsatisfiability)* is a series of inference steps starting from an input formula and terminating with $\bot$, showing that the input formula is unsatisfiable. The *granularity* of a proof step refers to how much reasoning it requires and roughly corresponds to the complexity of checking that the step is correct. In particular, steps (and thus the proofs containing them) are *fine-grained* if they can be efficiently checked, and *coarse-grained* otherwise. We will often refer to coarse-grained steps as *holes*.

One approach for the efficient production of proofs is to introduce coarse-grained proof steps for certain performance-critical deductions made while solving and then go back and fill in these holes with fine-grained steps as a post-processing step. We refer to this as *proof elaboration*, and it is particularly appealing for rewriting steps, since SMT solvers have hundreds of different rewrites to simplify and normalize terms, and instrumenting the rewriting code to produce fine-grained proofs is difficult and may introduce an unacceptable degradation in performance.

The approach taken by Nötzli et al. [28], and the one we also follow in this paper, is to assume that the SMT solver uses *generic* proof steps for all rewrites during solving and then elaborates these steps during post-processing by consulting a database of specific rewrite rules. The database is constructed by defining a set of rewrite rules in the domain-specific language Rare, which we discuss in Section 3. The elaboration tries to find one or more rules from the database to justify each generic, coarse-grained rewrite step. Additionally, it uses a built-in *evaluate* rule to justify steps that hold purely via constant folding. If elaboration is successful, the generic step is replaced by the fine-grained steps from the database.

2.3 SMT in Isabelle

As mentioned above, Sledgehammer [9] is an Isabelle tactic that applies automated reasoning tools, including SMT solvers, to prove goals in Isabelle. When targeting an SMT solver, the goal is encoded as an SMT-LIB [5] problem which is unsatisfiable iff the goal is valid. Sledgehammer also selects facts that it thinks will be relevant for solving the goal and includes encodings of them as well. The problem is given to the solver which reports back to Sledgehammer whether it was able to prove the goal [9]. Proving the goal externally, however, is not enough since Isabelle is a *skeptical* proof assistant, in the sense that it does not trust external solvers. Thus, a proof of the goal must somehow be constructed and checked inside Isabelle.

Finding such a proof internally can be challenging. One useful technique is to query the external solver for an *unsat core*, i.e., a subset of the facts it was given that are sufficient to prove the goal valid. Sometimes, this information is enough for Isabelle to search for an internal proof on its own. However, this process can be greatly improved, if, instead of just communicating the result and the core back to Sledgehammer, the solver also communicates a fine-grained proof. Then, with the appropriate proof reconstruction machinery, each step in the proof can be reconstructed as one or more steps using Isabelle's internal inference engine. As mentioned in Section 1.1, Sledgehammer can do this for proofs from the veriT and z3 solvers, though the former supports only a limited set of theories, and the latter produces only coarse-grained proofs.

Still, this means that Isabelle already has an integration with solvers supporting the SMT-LIB standard and is able to translate to and from SMT-LIB and internal terms. We build on this integration and extend it. Notice that such an integration requires each SMT-LIB operator to be matched with a term in Isabelle with the same semantics. Isabelle has built-in operators that match well with those in the uninterpreted function and arithmetic SMT theories, and both formalisms support quantifiers [18]. However, Isabelle only has partial support for bit-vector operators. A more complete development of bit-vectors in Isabelle is described by Böhme et al. [11], but unfortunately, parts of their work (including parsing bit-vector proofs) never made it into Isabelle and now appear to be lost. As we describe below, part of our effort includes improving support for SMT theories in Isabelle, including bit-vectors and strings.

2.4 Approximate Sorts

RARE rules are meant to be easy and effortless to write. This is not the case when users have to specify sort information that is either inferable from the context or too restrictive. As an example of the latter, consider any rewrite rules involving bit-vector sorts. The SMT-LIB standard provides bit-vectors sorts that are parameterized by their size, or *bit-width*. However, to keep sort checking simple, it requires all bit-widths in SMT-LIB scripts to be concrete as, for instance, in `(_ BitVec 8)`. A similar argument applies to polymorphic sorts because, although SMT-LIB allows the definition of theories with such sorts (such as, for instance, array, set, and sequence sorts), it restricts scripts to monomorphic instantiations of polymorphic sorts — e.g., `(Set Int)`.

Unfortunately, these restrictions are too strong for RARE. They make it impossible, for example, to write any rewrite rule involving bit-vector terms that is naturally parametric in the bit-width of those terms, or any rule involving terms with a polymorphic sort. The ideal solution would then be to introduce dependent types (or sorts, to maintain the SMT-LIB terminology) in RARE, allowing both value and type parameters in sorts — e.g., `(_ BitVec n)` with `n` an integer variable, and `(Array A B)` with `A` and `B` type variables. However, this would make it difficult for SMT solvers, cvc5 included, to process RARE rules since, effectively, they only support non-dependent, monomorphic sorts.

RARE's compromise solution is to add instead *approximate sorts* to the sort system, following an approach analogous to gradual typing in programming languages [33], a hybrid type-checking discipline where some program types are checked statically and others are checked dynamically. In our case, where there is no notion of dynamic checking, we have instead two sort-checking phases in the SMT solver for RARE rules: (*i*) as the rules are read by the solver, when sort checking is done with respect to the declared approximate sorts, and (*ii*) during proof elaboration, when the approximate sorts in the RARE rules are matched against the exact sorts in the proof steps that correspond to those rules.

Approximate sorts are obtained by extending the sort system of SMT-LIB with a distinguished unknown value and a distinguished unknown sort, both denoted by `?`, that can be used in place of a value or parameter in a sort. This allows the construction of approximate sorts such as `(_ BitVec ?)`, `(Set ?)`, and `(Array ? ?)` (abbreviated as `?BitVec`, `?Set`, and `?Array`), while still allowing precise sorts such as `(_ BitVec 1)`, `(Set Real)`, and `(Array Int Real)`. Approximate sorts can be used to approximate dependently-sorted/polymorphic rewrite rules, as we see in the next section.

```
⟨rule⟩    ::= ( define-rule ⟨symbol⟩ ( ⟨par⟩* ) [⟨defs⟩] ⟨expr⟩ ⟨expr⟩ )
            | ( define-rule* ⟨symbol⟩ ( ⟨par⟩* ) [⟨defs⟩] ⟨expr⟩ ⟨expr⟩ [⟨expr⟩] )
            | ( define-cond-rule ⟨symbol⟩ ( ⟨par⟩* ) [⟨defs⟩] ⟨expr⟩ ⟨expr⟩ ⟨expr⟩ )

⟨par⟩     ::= ⟨symbol⟩ ⟨sort⟩ [:list]

⟨sort⟩    ::= ⟨symbol⟩ | ? | ?⟨symbol⟩ | ( ⟨symbol⟩ ⟨numeral⟩+ )

⟨expr⟩    ::= ⟨const⟩ | ⟨id⟩ | ( ⟨id⟩ ⟨expr⟩+)

⟨id⟩      ::= ⟨symbol⟩ | ( ⟨symbol⟩ ⟨numeral⟩+ )

⟨binding⟩ ::= ( ⟨symbol⟩ ⟨expr⟩ )

⟨defs⟩    ::= ( def ⟨binding⟩+ )
```

Fig. 1: Overview of the grammar of Rare.

An additional advantage of this approach is that, by relieving the Rare user from the burden of specifying the precise sort of variables in rewrite rules, it makes them both easier to write and less error-prone. At the same time, the loss of precision introduced by approximate sorts is not a serious hindrance in practice: both the SMT solver, which relies on Rare rules for proof elaboration, and IsaRare, which uses them during proof reconstruction, are able to infer the exact sort represented by an approximate one thanks to their knowledge of the (exact) sort of the constant and function symbols in the supported SMT theories. Subsection 4.3 explains how IsaRare recovers exact sorts by type inference fully automatically during the translation to Isabelle.

3 The RARE Language

The Rare language[6] was introduced by Nötzli et al. [28]. As part of this work, we have extended the language to be able to represent more rewrite rules. We present the full updated language here and summarize the differences with [28] at the end of the section.

A Rare file contains a list of rules whose syntax is defined by the grammar in Figure 1. Expressions use SMT-LIB syntax with a few exceptions. These include the use of approximate sorts for parameterized sorts (e.g., arrays and bit-vectors) and the addition of a few extra operators (e.g., `bvsize`, described below). Rare uses SMT-LIB 3 syntax [6], which is very close to SMT-LIB 2 and mostly differs from its predecessor in that it uses higher-order functions for indexed operators.

We say that an expression e *matches* a match expression m if there is some *matching substitution* σ that replaces each variable in m by a term of the same sort to obtain e (i.e., $m\sigma$ is syntactically identical to e). For example, the expression `(or (bvugt x1 x2) (= x2 x3))`, with variables `x1`, `x2`, `x3`, all of sort `?BitVec`,

[6] Rare comes from Rewrites, Automatically REconstructed.

matches `(or (bvugt a b) (= b a))` but not `(or (bvugt a b) (= c a))`, with `a`, `b`, and `c` bit-vector constant symbols of the same bit-width.

Rare Rules A Rare rewrite rule is defined with the `define-rule` command which starts with a parameter list containing variables with their sorts. These variables are used for matching as explained below. After an optional *definition list* (see below), there follow two expressions that form the main body of the rule: the *match* expression and the *target* expression. The semantics of a rule with match expression m and target expression t is that any expression e matching m under some sort-preserving matching substitution σ can be replaced by $t\sigma$. With approximate sorts, the sort preservation requirement is relaxed as follows. In Rare, for any sort constructor S of arity $n > 0$, there is a corresponding approximate sort `(`S `?` $\cdots$ `?`) with n occurrences of `?` which is always abbreviated as `?`S. A variable x with sort `?`S (e.g., `?BitVec`) in a match expression matches all expressions whose sort is constructed with S (e.g., `(BitVec 1)`, `(BitVec 2)`, and so on). Variables with sort `?` match expressions of any sort.

An optional *definition list* may appear in a Rare rule immediately after the parameter list. It starts with the keyword `def` and provides a list of local variables and their definitions, allowing the rewrite rule to be expressed more succinctly. A rule with a definition list is equivalent to the same rule without it, where each variable in the definition list has been replaced by its corresponding expression in the body of the rule. For a rule to be well-formed, all variables in the match and target expressions must appear either in the parameter list or the definition list. Similarly, each variable in the parameter list must appear in the match expression (while this second requirement could be relaxed, it is useful for catching mistakes). Consider the following example.

```
(define-rule bv-sign-extend-eliminate ((x ?BitVec) (n Int))
  (def (s (bvsize x)))
  (sign_extend n x)   (concat (repeat n (extract (- s 1) (- s 1) x)) x))
```

In this rule, there are two parameters, `x` and `n`. The sort annotation `?BitVec` indicates that `x` is a bit-vector without specifying its bit-width. The latter is stored in the local variable `s` using the `bvsize` operator. The rule says that a `(sign_extend n x)` expression can be replaced by repeating `n` times the most significant bit of `x` and then prepending this to `x`.

The `define-cond-rule` command is similar to `define-rule` except that it has an additional expression, the *condition*, immediately after the parameter and definition lists. This restricts the rule's applicability to cases where the condition can be proven equivalent to true under the matching substitution. In the example below, the condition `(> n 1)` can be verified by evaluation since in SMT-LIB, the first argument of `repeat` must be a numeral.

```
(define-cond-rule bv-repeat-eliminate-1 ((x ?BitVec) (n Int))
  (> n 1)   (repeat n x)   (concat x (repeat (- n 1) x)))
```

Note that the rule does not apply to terms like `(repeat 1` t`)` or `(repeat 0` t`)`.

Fixed-point Rules The `define-rule*` command defines rules that should be applied repeatedly, to completion. This is useful, for instance, in writing rules that iterate over the arguments of n-ary operators. Its basic form, with a body containing just a match and target expression, defines a rule that, whenever is applied, must be applied again on the resulting term until it no longer applies.

The user can optionally supply a *context* to control the iteration. This is a third expression that must contain an underscore. The semantics is that the match expression rewrites to the context expression, with the underscore replaced by the target expression. Then the rule is applied again to the target expression only. In the example below, the `:list` modifier is used to represent an arbitrary number of arguments, including zero, of the same type.

```
(define-rule* bv-neg-add ((x ?BitVec) (y ?BitVec) (zs ?BitVec :list))
  (bvneg (bvadd x y zs))   (bvneg (bvadd y zs))   (bvadd (bvneg x) _))
```

This rule rewrites a term `(bvneg (bvadd` s t $\cdots$`))` to the term `(bvadd (bvneg` s`)` r`)` where r is the result of recursively applying the rule to `(bvneg (bvadd` t $\cdots$`))`.

Changes to Rare Here, we briefly mention the changes to Rare with respect to [28]. First, we have support for a richer class of approximate sorts, including approximate bit-vector and array sorts. Also, we replaced the `let` construct by the new `def` construct. The definition list is more powerful as it applies to the entire rest of the body (whereas `let` was local to a single expression).

Additionally, to aid with bit-vector rewrite rules, we added several operators: `bvsize`, which returns the width of an expression of sort `?BitVec`; `bv`, which takes a integer n and natural w, and returns a bit-vector of width w and value n mod 2^w; `int.log2` which returns the integer (base 2) logarithm of an integer, and `int.islog2`, which returns true iff its integer argument is a power of 2.

We also removed the `:const` modifier, which was used previously to indicate that a particular expression had to be a constant value. We found that this adds complexity and is usually unnecessary. For rules that actually manipulate specific constant values, we can specify those values explicitly, e.g., by using the `bv` operator above.

4 IsaRare: from Rare Rewrites to Isabelle Lemmas

In this section, we introduce IsaRare, a plugin for Isabelle that automatically translates a Rare rule into an Isabelle lemma stating the correctness of the rule. Being able to generate such lemmas automatically is highly desirable, as Rare rules may be added and/or changed frequently for a given solver, or differ significantly between solvers, and manually translating Rare rules into lemmas is time-consuming and error-prone. IsaRare can also suggest a proof sketch which is sometimes sufficient to prove the lemma. If this automatic proof fails, the user must provide the proof or determine that the lemma does not hold. In the latter case, Isabelle's counterexample-finder Nitpick [10] can be helpful.

```
(define-cond-rule str-len-replace-inv ((t String) (s String) (r String))
  (= (str.len s) (str.len r))
  (str.len (str.replace t s r))   (str.len t))
```

```
lemma str_len_replace_inv:
  fixes t::string and s::string and r::string
  shows "smtlib_str_len s = smtlib_str_len r ⟶
   smtlib_str_len (smtlib_str_replace t s r) = smtlib_str_len t"
```

Fig. 2: Rare rule and corresponding lemma.

Figure 2 shows an example of a Rare rule (which simplifies the length of the result of a string replacement) and the Isabelle lemma generated from it by IsaRare. Roughly speaking, a rule with parameters $x_1, \ldots, x_m$, definition list $((y_1\ d_1)\ \cdots\ (y_n\ d_n))$, condition c, match expression s, and target expression t is converted by IsaRare into a lemma of the form $\forall\, x_1, \ldots, x_m.\, (c \Rightarrow s = t)\sigma$ where σ is the substitution $\{y_1 \mapsto d_1, \ldots, y_n \mapsto d_n\}$. Type inference in Isabelle is used to suitably instantiate the `?` wildcards in any approximate sorts in the rules.

Next we discuss the main challenges we encountered while implementing the translation from Rare to Isabelle.

4.1 Adding New Theories

Since IsaRare uses Isabelle's SMT-LIB parser, it was necessary to extend it to handle SMT theories not previously supported and, in case there was no corresponding Isabelle theory, to define new types, definitions and theorems corresponding to the SMT-LIB theory. For sets and arrays, Isabelle already provides the required data structures (`Set.set` and `Map.map` respectively) and definitions (e.g., `union`, and `store`). Translation from the SMT operators and types is thus straightforward, requiring only simple extensions to the parser.

The SMT-LIB parser also had to be extended for the operators and sorts of the SMT-LIB theory of strings. String terms are represented with Isabelle's `HOL.string`, and regular expressions are represented as sets of strings. We developed a new theory with auxiliary definitions and theorems meant to facilitate the proving of lemmas generated by IsaRare. Since strings are defined as lists of characters, we were able to reuse many list operators for our definitions. For example, string concatenation is defined as concatenation of lists.

As mentioned, bit-vectors are encoded in Isabelle using the `word` type, which represents integers modulo 2^n, where n is a type parameter (see Subsection 4.3). Isabelle has support for reasoning about this type, but we still had to provide a number of extensions. For example, to translate bit-vector rewrite rules, we had to extend Isabelle's SMT-LIB parser significantly. We added support for all of the standard SMT-LIB operators, as well as some additional operators that cvc5

```
(define-rule bv-extract-extract
 ((x ?BitVec) (i Int) (j Int) (k Int) (l Int)))
 (extract l k (extract j i x)))
 (extract (+ i l) (+ i k) x))
```

(a) A Rare rule

```
t0 = (extract j i x) ∧
size t0 = j + 1 - i ∧
t1 = (extract l k t0) ∧
size t1 = l + 1 - k ∧
t2 = (extract (i+l) (i+k) x) ∧
size t2 = (i+l) + 1 - (i+k) ∧
j < size x ∧ 0 ≤ i ∧ i ≤ j ∧
l < size t0 ∧ 0 ≤ k ∧ k ≤ l ∧
(i+l) < size x ∧ 0 ≤ (i+k) ∧
(i+k) ≤ (i+l)
```

(b) Additional Assumptions

Fig. 3: Implicit Assumption Generation

supports, such as `bvuaddo` (which checks for overflow from unsigned addition). It was also necessary to add several new definitions and basic theorems to Isabelle, for example for reasoning about the `extract` operator.

4.2 Mismatch between Isabelle and SMT-LIB operators

An important challenge for the translation concerns the mismatch between SMT-LIB operators and Isabelle functions. One of the main difficulties concerns *implicit* assumptions. As an example, consider the bit-vector extract operator. The term `(extract` i j t`)` denotes the sub-vector of bit-vector t from index i through index j, where i is the more significant index. SMT-LIB specifies that the second index j must be at most i, and both indices must be in the range $[0, n)$, where n is the bit-width of t — making the result a bit-vector of width $i + 1 - j$. These assumptions are necessary to correctly capture the semantics of SMT-LIB's `extract` since the extract operator in Isabelle is more permissive.

There are several ways to address this issue. First, we could make the implicit assumptions explicit in the Rare rules. However, this would be tedious and error-prone and would greatly clutter the Rare rules. It is also superfluous to always manually add them since the constraints are inherent in the SMT-LIB semantics. A second option is to write custom definitions for SMT-LIB operators in Isabelle that exactly match the SMT-LIB semantics (i.e., are undefined if the implicit assumptions do not hold). The main disadvantage of this approach is that it complicates proving the translated Rare rules, as those proofs cannot directly use any existing Isabelle lemmas that use the standard definitions. It also works against one of our long-term goals, which is to be able to use proof reconstruction to provide proofs for Isabelle conjectures, conjectures which will naturally use the existing Isabelle operators.

The last option, which we adopted, is to automatically add the implicit assumptions during the translation of Rare rules to Isabelle lemmas. This does make the lemmas a bit more complicated, but it is the minimal complexity needed to bridge the semantic gap between the two extract operators. And, we can be confident that these implicit assumptions will easily be discharged when using the lemmas for proof reconstruction, since SMT proofs only use operators in ways that are consistent with SMT-LIB semantics (unless there is a bug, in which case proof reconstruction *should* fail). Figure 3 shows an example of a Rare rule with three applications of the extract operator, together with the assumptions added by IsaRare.

In a few cases, we had to fall back on the custom definition approach. For example, we had to do this for the bit-vector `concat` operator for bit-vector concatenation. To see why, note that the SMT-LIB operator can take two or more arguments (abbreviating nested binary applications), each with arbitrary bit-width. Recall that the `:list` annotation in Rare can be used to specify a variable number of arguments. There is no way to even state lemmas corresponding to rewrite rules involving concatenations of a variable number of arguments in Isabelle using its built-in binary concatenation operator. For this case, we thus define a custom concatenation operator that matches the SMT-LIB semantics. The implicit assumption that the bit-width of the result is the sum of the bit-widths of the arguments is embedded in the custom definition. Using the new definition, we can translate the problematic rules into Isabelle lemmas. As expected, proving these lemmas requires extra work. Specifically, it requires formulating and proving bridging theorems between Isabelle's built-in concatenation operator and the new one we defined.

4.3 Supporting Approximate Sorts

With the addition of approximate sorts to Rare, we had to extend Isabelle's SMT-LIB translator to support them. We observe that Isabelle/HOL is not based on a dependently-typed logic. However, it supports an encoding of sorts depending on integer values into polymorphic types with parameters that range over types expressing ordinals. In particular, bit-vectors of width w are represented by the type `(n word)` of integers modulo 2^w; for instance, `3::(8 word)` represents an integer with value 3 modulo 2^8. In fact, thanks to polymorphism, it is possible for the bit-width to be a type variable (e.g., `3::('a::len word)`). Note that this is more precise than allowing the bit-width in the type to be completely unknown, as in approximate sorts: with type parameters one can state, for instance, that two terms of unknown bit-width have the same width, whereas two terms both of sort `?BitVec` may have different bit-widths.

Conveniently then, all the approximate sorts in Rare correspond to polymorphic types in Isabelle. For instance, `?BitVec` corresponds to `'a word` and `?Array` corresponds to `('a,'b) map` where `'a` and `'b` are type variables. During parsing, each occurrence of a approximate sort is converted into an instance of the corresponding polymorphic type obtained by instantiating each sort variable with a fresh dummy type. For some bit-vector operators, the output sort is dependent on the input sorts (e.g., `extract` and `concat` as mentioned above). For applications of such operators, we also use a dummy type for the bit-width of each argument for which the width is not known. Once translation is done, we use Isabelle's type inference algorithm to concretize each dummy type to a monomorphic one. For example, during translation of the rule `bv-ugt-eliminate` below, the variables `x` and `y` would both be assigned dummy types.

```
(define-rule bv-ugt-eliminate ((x ?BitVec) (y ?BitVec))
  (bvugt x y)   (bvult y x)
)
```

However, `bvugt` requires that both of its arguments be bit-vectors of the same width in SMT-LIB. This restriction is either already present in the definition in Isabelle that we map an operator to, or added during parsing as an implicit assumption, as we describe in Section 4.2. The type inference algorithm then computes the most general type for `x` and `y` that satisfies all assumptions. In this case, it correctly infers that they are bit-vectors of arbitrary but equal bit-width.

4.4 List Parameters

As mentioned earlier, SMT-LIB supports multi-arity syntax for certain binary operators, and Rare supports a variable number of arguments via the `:list` annotation. In contrast, in Isabelle all operators are fixed-arity. To facilitate the translation in these cases we added a new datasort, `'a rare_ListVar`, with a single constructor `ListVar :: 'a list` $\rightarrow$ `'a rare_ListVar` to encapsulate multiple arguments in a list. We also introduced two second-order operators, called `rare_list_left` and `rare_list_right`, to encode Rare left-associative and right-associative operators, respectively. As an example, a Boolean term of the form `(and` $x_1 \ \cdots \ x_n \ y \ z$`)` is translated to the Isabelle term `(rare_list_right` $(\wedge)$ `(ListVar` $[x_1, \ldots, x_n]$`)` $(y \wedge z)$`)`. The `rare_list_left` and `rare_list_right` functionals fold the operator passed as first argument over the list stored in their second argument to obtain properly nested binary applications. For example, if $n = 2$, the Isabelle term above is translated to $(x_1 \wedge (x_2 \wedge (y \wedge z)))$.

For every multi-arity SMT-LIB operator, we prove that it can be built up from Isabelle's built-in binary version using `fold(r)` functions. For Rare rules with list parameters, these *transfer lemmas* become part of the correctness proof automatically generated by IsaRare. When proving the corresponding lemma, we can take advantage of the many lemmas in Isabelle's libraries about fold functions without having to know the internals of the translation process.

If we have a Rare rule in which all arguments to an operator are lists, we must handle the special case when the lists are all empty. When the operator has an identity element, we return that. For example, applications of `and` to just empty lists are translated as standing for `true`. So far, we have only encountered one operator without an identity: bit-vector concatenation. Since neither SMT-LIB nor Isabelle support bit-vectors of bit-width 0, for that operator, we explicitly add an assumption ruling out the case where all lists are empty.

4.5 Writing Lemmas and their Proofs

To generate a lemma from a Rare rewrite rule, IsaRare first introduces the parameters with their types using Isabelle's **`fixes`** construct. Next, it generates the statement of the lemma, the *goal*, which states that the implicit assumptions and conditions imply the equality of the match and target terms. The types of any bit-vector constants are fully specified (via type ascription), because otherwise the lemma may be too general and not hold.

Lastly, IsaRare adds an Isabelle proof of the lemma. For lemmas that do not contain lists, this is simply a call to the main automatic tactic `auto`. Otherwise, the list constructs are eliminated as explained above, and any transfer lemmas are applied to the resulting terms. This ensures that goals will not contain any IsaRare list definitions. We then invoke induction for every list and use the `simp_all` tactic to attempt to solve and simplify the goals.

The proof is printed in *apply* style so that it can be easily modified and completed manually if Isabelle is unable to discharge all its sub-goals automatically.

4.6 Availability

IsaRare currently supports the theories of uninterpreted functions, linear arithmetic, bit-vectors, arrays, strings, and sets. It is publicly available[7] under the BSD 3-Clause license. We plan to submit IsaRare to the Archive of Formal Proofs [20]. We have also been working with the Isabelle maintainers to have our extensions to Isabelle itself (e.g., to the SMT-LIB parser) included in the official Isabelle distribution. Many features were already included in the latest release. IsaRare requires the Word_Lib library (which is also included in the Archive of Formal Proofs) if it is used on Rare rules containing bit-vector operators not present in Isabelle itself.

5 Evaluation and Experience

We used IsaRare to help develop, translate, and verify new Rare rewrite rules. These rules were designed to address coarse-grained rewrite steps appearing in cvc5 proofs, i.e., steps that could not be elaborated into fine-grained steps using the existing Rare rules and the approach mentioned in Section 2.2. In this section, we report on this experience and also discuss challenges arising from particular rewrites and theories.

5.1 Impact of New Rewrites on cvc5 Proof Holes

Previous work developed 85 Rare rules for cvc5 [28]. For our evaluation, we ran cvc5 with these plus our 163 new rules, bringing the total number of Rare rules in the cvc5 database to 248. We evaluated the impact of the new rules on cvc5's ability to produce fine-grained proof steps by comparing the *success rate* of the elaboration (i.e., percentage of rewriting proof steps that are successfully elaborated into fine-grained steps) before and after the addition of the new rules. We ran cvc5 on 70,709 unsatisfiable benchmarks, as determined by cvc5 [2, Sec. 4], in the SMT-LIB logics containing quantifier-free problems with equality and uninterpreted functions, arrays, linear arithmetic, strings, and bit-vectors.

[7] https://github.com/cvc5/IsaRARE

	rewrites			
theory	**old**	**new**	**proven**	**autoproven**
EUF	22	43	43	37
Arithmetic	23	22	22	14
Sets	0	7	7	7
Arrays	0	4	4	4
Strings	40	57	57	37
Bit-vectors	0	115	84	62

Table 1: Rule and rule verification counts per theory

The results were generated with a cluster equipped with 16 x Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz, 62.79 GiB RAM machines, with one core per solver/benchmark pair, 1200s time limit, and 8gb memory limit.

For string benchmarks (the only set evaluated in [28]), the success rate went from 92% to 98%. Results on the logics with equality and uninterpreted functions, arrays, and linear arithmetic were similar. By far the most challenging theory, in terms of rewrite rules, is the bit-vector theory. Prior to our work, there were no Rare rules for this theory, so no bit-vector rewrite steps could be turned into fine-grained steps. With our 115 new Rare rules for bit-vectors, 92% of coarse-grained bit-vector rewrite steps are successfully elaborated into fine-grained steps. We see this as tremendous progress towards full fine-grained proofs for bit-vector problems.

5.2 Translating and Verifying Rewrites

In Table 1, we list the number of new rules in each theory, distinguishing between how many were there before (old) and the total including both the old rules and our new rules (new).[8] We also show how many of the lemmas we have successfully proven and how many of these were done automatically, i.e., either by the proof suggested by IsaRare or by a single call to Sledgehammer. The **proven** column shows that all non-bit-vector rules as well as most of the bit-vector rules have now been proven. The numbers in the last column show that most of the proofs were provided automatically by IsaRare.

For the theory of strings, the number of lemmas automatically proven is not clear-cut. For other theories, libraries with useful background lemmas already existed, but for strings we had to add many new general-purpose lemmas ourselves and then decide whether these should count as background lemmas or as part of the proof effort for a rewrite rule. We were rather conservative in that decision, i.e., we did not count a lemma as automatically proved if it used a lemma whose classification as a background lemma was in doubt. Many of the

[8] Consolidation in the set of arithmetic rules actually resulted in one fewer rule than existed previously.

translated string rewrites had to be proved manually because they required induction on string length, especially since many operators are defined inductively. However, we found that most of these manual proofs were fairly easy once an appropriate induction variable was selected.

There are no performance issues—IsaRARE translates most files in milliseconds. Even for our biggest RARE database, the one containing bit-vector rules, IsaRARE took only around 1-2 seconds on our machine.

5.3 Bugs Found in String Rules

We found several bugs in the existing Rare rules for strings by using Isabelle's counterexample finder Nitpick [10] on the translated Isabelle lemmas. We diagnosed and fixed each of them, so that now they can all be verified.[9] The bugs fall into three main categories.

Misinterpreted Semantics: The `str.substr` operator takes three arguments and returns the substring of the first argument, starting at the position given by the second argument, and continuing for the number of characters specified by the third argument. The following (corrected) rule simplifies a substring expression to the empty string whenever the third argument is 0 or negative.

```
(define-cond-rule str-substr-empty-range ((x String) (n Int) (m Int))
  (>= 0 m)   (str.substr x n m)   "")
```

However, the first version of the rule had the wrong condition: `(>= n m)` rather than `(>= 0 m)`. This is likely due to the rule's author mistaking the third argument of `str.substr` for an absolute index instead of a relative offset.

Forgotten Condition: The corrected rule below says that, under some assumptions, the length of a substring term is equal to the offset (third) argument.

```
(define-cond-rule str-len-substr-in-range ((s String) (n Int) (m Int))
  (and (>= n 0) (>= m 0) (>= (str.len s) (+ n m)))
  (str.len (str.substr s n m))   m)
```

The earlier version of the rule did not include the condition `(>= m 0)`. This however, makes it unsound, because according to the semantics of str.substr, if the offset is negative, the result is just the empty string. This led to a counterexample with a negative value for `m`. Note that this condition is not automatically added by IsaRare since `str.substr` *is* defined for negative offsets.

Misunderstanding the Rewrite: One rule was designed to closely mirror a piece of cvc5 code implementing a rewrite, but it failed to properly capture all cases. The code involved included several conditionals resulting in two different ways a term could be rewritten. The original rule only captured one of the two cases and even missed one of the conditions for the case it included. Since this rule was quite complex and was only incorrect for some corner cases, it would have been challenging to find this bug without our verification effort.

[9] Fortunately, none of the bugs in rules corresponded to buggy code in cvc5 itself. However, cvc5 could have used those rules to construct incorrect proofs.

5.4 Bit-vector Rewrite Rules

Bit-vector theory solvers make extensive use of rewriting, employing large numbers of rewrite rules. In order to define Rare rules for cvc5's bit-vector theory, we began by analyzing the cvc5 rewriting code, which implements a total of 99 rewrite methods. We then wrote Rare rules to try to capture the behavior of these methods. There are 5 methods that are too complex to be captured by Rare (or by any straightforward extension of it). For each of these, we instead added new hard-coded proof rules to the cvc5 proof rule database.[10] These hard-coded proof rules are not included in Table 1, but they *are* used to help demonstrate the overall progress on SMT-LIB proofs (Section 5.1). The long-term plan for reconstruction of proofs using these rules is to write custom Isabelle tactics for reconstructing those proof steps.

Unlike with the string rules, where we applied IsaRare to already-written rules, we used IsaRare extensively to help debug the bit-vector rules as they were being written. We were able to quickly and easily find many kinds of mistakes this way. For example, rule authors mixed up `bvneg` (unary 2's complement negation) and `bvnot` (bit-wise Boolean negation). In other cases, rules used inconsistent bit-widths. The type inference that IsaRare performs is particularly helpful in such cases, as it is stricter than the cvc5 Rare parser.

Many of the bit-vector rules can be proved automatically, but others must be proved manually and are quite challenging, especially those involving signed arithmetic or division. Despite this, as shown in Table 1, the process of manually proving the full set of bit-vector lemmas is largely complete. This is important for our long-term goal of reconstructing SMT proofs in Isabelle.

6 Conclusion

We presented IsaRare, a tool providing an automatic pipeline for verifying rewrite rules. We showed the effectiveness of our approach by proving the correctness of a large number of rewrite rules used in cvc5 proofs. Our experiments show that many lemmas can be proved with minimal user interaction.

This work is also part of a long-term project that aims to further automate proof search in Isabelle. The goal is to be able to reconstruct any cvc5 proof in Isabelle's internal inference engine. This, of course, also includes reconstructing rewrite steps. The lemmas IsaRare generates are directly applicable to this effort. We plan to provide a detailed description and evaluation of this larger effort in future work.

Data Availability Statement The datasets generated and analyzed during the current study are available in the Zenodo repository: `https://zenodo.org/records/10048664` [24].

[10] This is analogous to the handling of polynomial normalization in [28].

References

1. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9. IEEE (2018)
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., et al.: cvc5: a versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
3. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. Journal of Automated Reasoning **64**(3), 485–510 (2020)
4. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning. pp. 15–35. Springer International Publishing, Cham (2022)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
6. Barrett, C., Fontaine, P., Tinelli, C.: SMT-LIB Version 3.0 - Preliminary Proposal (2021), `https://smtlib.cs.uiowa.edu/version3.shtml`
7. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: Delahaye, D., Woltzenlogel Paleo, B. (eds.) All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 23–44. College Publications, London, UK (Jan 2015)
8. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021)
9. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction – CADE-23. pp. 116–130. Springer Berlin Heidelberg (2011)
10. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 131–146. Springer Berlin Heidelberg (2010)
11. Böhme, S., Fox, A.C., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: International Conference on Certified Programs and Proofs. pp. 183–198. Springer (2011)
12. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 179–194. Springer, Berlin, Heidelberg (2010)
13. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. pp. 151–156. Springer (2009)
14. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) Model Checking Software. pp. 248–254. Springer (2012)
15. Cook, B.: Formal reasoning about the security of Amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 38–47. Springer (2018)

16. Enderton, H.B.: A mathematical introduction to logic. Elsevier (2001)
17. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods. Lecture Notes in Computer Science, vol. 11460, pp. 148–165. Springer (2019)
18. Fleury, M., Schurr, H.J.: Reconstructing veriT proofs in isabelle/HOL. Electronic Proceedings in Theoretical Computer Science **301**, 36–50 (2019)
19. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) International Workshop on Satisfiability Modulo Theories (SMT). CEUR Workshop Proceedings, vol. 3185, pp. 54–70. CEUR-WS.org (2022)
20. Jaskelioff, M., Merz, S.: Proving the correctness of disk paxos. Archive of Formal Proofs (June 2005), `https://isa-afp.org/entries/DiskPaxos.html`, Formal proof development
21. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: a certified string solver. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Produms and Proofs. pp. 210–224. Association for Computing Machinery (2022)
22. Katz, G., Barrett, C., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy proofs for DPLL (T)-based SMT solvers. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 93–100. IEEE (2016)
23. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal verification of an OSa kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220. Association for Computing Machinery (2009)
24. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Noetzli, A., Barrett, C., Tinelli, C.: IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL (Oct 2023), `https://doi.org/10.5281/zenodo.10048664`
25. de Moura, L., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)
26. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
27. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
28. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) 2022 Formal Methods in Computer-Aided Design (FMCAD). p. 65 (2022)
29. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Theory-specific proof steps witnessing correctness of SMT executions. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). pp. 541–546. IEEE (2021)
30. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to Isabelle/HOL. Formal Aspects of Computing **31**(6), 675–698 (2019)
31. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 12699, pp. 450–467. Springer (2021)
32. Schurr, H.J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). Electronic Proceedings in Theoretical Computer Science **336**, 49–54 (2021)

33. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: ACM (ed.) Proceedings of Scheme and Functional Programming Workshop (2006)