



Improving the SMT Proof Reconstruction Pipeline in Isabelle/HOL

Hanna Lachnitt  
Stanford University, Stanford, USA

Mathias Fleury  
University of Freiburg, Freiburg, Germany

Haniel Barbosa  
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Jibiana Jakpor  
Stanford University, Stanford, USA

Bruno Andreotti  
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Andrew Reynolds  
University of Iowa, Iowa, USA

Hans-Jörg Schurr  
University of Iowa, Iowa, USA

Clark Barrett  
Stanford University, Stanford, USA

Cesare Tinelli  
University of Iowa, Iowa, USA

Abstract

Sledgehammer is a tool that increases the level of automation in the Isabelle/HOL proof assistant by asking external automatic theorem provers (ATPs), including SMT solvers, to prove the current goal. When the external ATP succeeds it must provide enough evidence that the goal holds for Isabelle to be able to reprove it internally based on that evidence. In particular, Isabelle can do this by replaying fine-grained proof certificates from proof-producing SMT solvers as long as they are expressed in the Alethe format, which until now was supported only by the veriT SMT solver. We report on our experience adding proof reconstruction support for the cvc5 SMT solver in Isabelle by extending cvc5 to produce proofs in the Alethe format and then adapting Isabelle to reconstruct those proofs. We discuss several difficulties and pitfalls we encountered and describe a set of tools and techniques we developed to improve the process. A notable outcome of this effort is that Isabelle can now be used as an independent proof checker for SMT problems written in the SMT-LIB standard. We evaluate cvc5's integration on a set of SMT-LIB benchmarks originating from Isabelle as well as on a set of Isabelle proofs. Our results confirm that this integration complements and improves Sledgehammer's capabilities.

2012 ACM Subject Classification Theory of computation → Automated reasoning

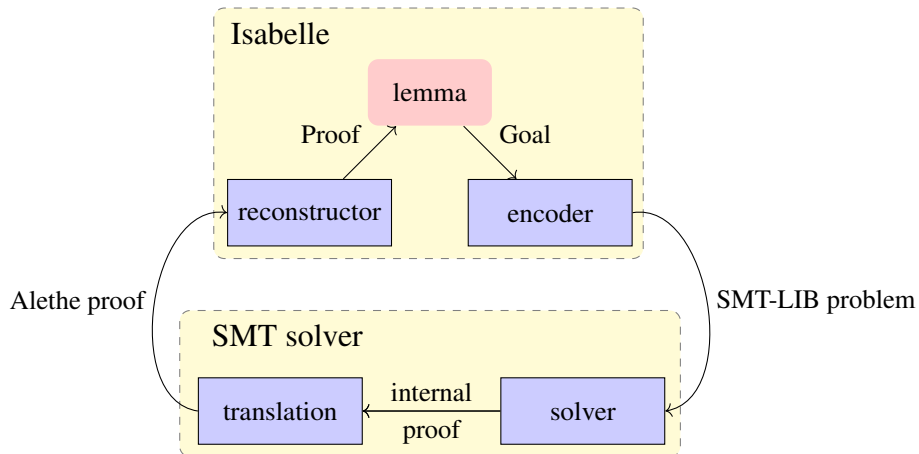
Keywords and phrases interactive theorem proving, proof assistants, Isabelle/HOL, SMT, certification, proof certificates, proof reconstruction, proof automation

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.26

Funding This work was supported in part by the Stanford Center for Automated Reasoning, a gift from Amazon Web Services, a gift from Intel Corporation, and the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-24-2-1001. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA.

1 Introduction

Interactive theorem provers (ITPs) such as Isabelle/HOL [34] are used in many applications ranging from formal mathematics to software and hardware verification [28]. One reason for their popularity is the high assurance they offer, which, however, comes at the cost of requiring users to develop fine-grained machine-checkable proofs. To increase their usability, ITPs implement various automation tactics that allow users to concentrate on high-level arguments without being impeded by simpler but tedious proof obligations. A particularly successful approach is to provide automation via integration with fast external automatic theorem provers (ATPs), such as satisfiability modulo theories (SMT) solvers [23].



■ **Figure 1** Supporting a new solver in the Isabelle smt tactic by translation to Alethe.

Isabelle/HOL provides access to SMT solvers in the form of the `smt` tactic [10]. Through that tactic, a proof goal is translated into a problem in the SMT-LIB format [9] and given to either the `z3` [18] or the `veriT` [13] solver. If successful, these solvers can produce *proof certificates* that can be *reconstructed* in Isabelle, that is, used to construct a complete proof of the original goal fully within Isabelle/HOL. The `smt` tactic is particularly useful in combination with Isabelle’s Sledgehammer component [10], which automatically selects additional facts from the current proof context that it expects to be useful in proving the goal dispatched to the external ATP. Experiments have shown that the `smt` tactic is regularly recommended by Sledgehammer as the fastest method to prove Isabelle/HOL goals automatically [23].

Because different solvers have different strengths, the best results are obtained with Sledgehammer by using a portfolio of SMT solvers. Adding a new SMT solver to Sledgehammer’s portfolio thus has the potential to directly and positively impact the user experience, and so it should ideally be as straightforward as possible. Since proof reconstruction is specific to a particular proof certificate format, integrating a new proof-producing SMT solver requires a choice between two possible alternatives: (i) integrate the solver’s own proof format into Isabelle’s reconstruction pipeline, or (ii) extend the solver to use a format already supported by Isabelle. Unfortunately, the two options are not equally feasible. Supporting a new proof certificate format in Isabelle is rather challenging [37] as it requires writing a custom parser and adding a reconstruction function for each proof rule. This is an immense manual effort that requires considerable expertise in both the proof format and the Isabelle code base. In this work, we thus focus on the second option.

Before this work, Isabelle supported two proof formats for SMT: a coarse-grained one produced by the version of the `z3` solver used by the `smt` tactic [17], and the fine-grained Alethe proof format supported by the `veriT` solver [7]. The `z3` proof format is not the best vehicle for a new SMT solver integration for a number of reasons. Even the authors of the `z3` integration in Isabelle point out that it is difficult to reconstruct `z3` proof certificates due to their coarseness [12]. Furthermore, there are some known bugs that the developers do not plan to address soon [22], and the format has limited documentation. The Alethe format seems to be a better choice, both in contrast with the `z3` format option and in its own right. Although originally developed specifically for `veriT`, after its integration into Isabelle, it has evolved into a stand-alone proof format [36], with its own independent proof checker Carcara [1] and a detailed specification [7], both of which are publicly available. Alethe supports reasonably fine-grained natural-deduction-style proofs, is based on the SMT-LIB language, and is designed to be easy to parse and extend.

Figure 1 shows an overview of our chosen approach. For efficiency, we produce proof certificates in Alethe via an internal translation in the CVC5 solver itself, though in principle, this could also be done by an external tool. In theory, the main advantage of our approach is that even though building a translation to Alethe requires a significant amount of work, it can be done independently from the proof assistant, and so does not need significant expertise in Isabelle development. Ideally, the existing Isabelle infrastructure supporting Alethe should be reusable as is.

Unfortunately, our experience in practice revealed something quite different. We discovered several issues, including: (i) Isabelle’s Alethe reconstruction was missing several proof rules available in the full Alethe standard; (ii) ambiguities in the Alethe documentation made it difficult to ensure that solver outputs matched the input expected by Isabelle; and (iii) Isabelle’s reconstruction was in some cases over-specialized towards the output generated by veriT rather than output that is just compliant with the Alethe standard. As we investigated these issues, we further discovered that: (iv) the different parts of Isabelle’s reconstruction code were tightly coupled—it was difficult to fix a bug in one part without understanding all components; (v) proof generation and proof reconstruction could not be tested separately, making for long and difficult debug cycles; and (vi) there were not enough or sufficiently diverse test cases to ensure that each proof rule in the full Alethe standard was appropriately supported.

As a result, we refocused a large portion of our work on addressing the issues above. We refactored the Isabelle reconstruction code to improve its modularity and provide clean interfaces. We contributed extensions and clarifications to the Alethe standard. Finally, we developed an analysis and debugging toolkit aimed at improving the workflows both for members of the Isabelle community as well as SMT solver developers looking to integrate their solver into Isabelle. To our knowledge, our work is the first project that targets reusing an existing format for a new solver. This also allows us to compare the effectiveness of different solvers for proof automation in a more granular and direct way. Our contributions can be summarized as follows:

1. We show that translation to Alethe from a modern full-featured proof-producing SMT solver like CVC5 is possible with minimal overhead.
2. We present an extensively refactored smt tactic implementation in Isabelle/HOL, with a focus on modularity and support for the full Alethe standard, including some new rules and language features.
3. We extend the smt tactic with a plug-in capability so that it can easily integrate Isabelle lemmas for simple rules not easily translated to Alethe. We demonstrate the utility of this feature by using it to reconstruct CVC5 rewrite rules.
4. We present a toolkit for Isabelle users and developers for gaining insight into the smt back end.
5. We conduct a thorough experimental evaluation showing the effectiveness of our approach and providing a baseline for future solvers to compare against.

We start in Section 2 with some background on different tools and concepts relevant for this work. Section 3 describes how we instrumented CVC5 to produce proofs in Alethe, Section 4 describes how we refactored the Isabelle smt tactic, and Section 5 describes our toolkit for analyzing and debugging. Finally, we present an experimental evaluation of our work in Section 6 and conclude in Section 7.

2 Preliminaries and Related Work

2.1 Satisfiability modulo theories and proofs

The underlying logic of SMT is many-sorted first-order logic with equality (see e.g., [21]). We assume the usual definitions of well-sorted terms, literals, and formulas in this logic. We also assume the usual definitions of satisfiability and unsatisfiability with respect to one or more theories. SMT solvers

determine the satisfiability of input formulas with respect to specific sets of supported background theories. Examples of SMT solvers include CVC5 [5], veriT [13], and z3 [18].

A *refutation proof* in a theory T is a series of inference steps starting from an input formula φ and terminating with the unsatisfiable formula \perp . Such a proof certifies the unsatisfiability of φ in T . A *proof format* encompasses a list of *proof rules* specifying how specific conclusions can be drawn from specific assumptions and under what conditions. A (*refutation*) *proof certificate* is a realization of a refutation proof in a particular proof format consisting of a number of *proof steps*. We require each step in a proof certificate to be an application of a proof rule from the corresponding proof format. As an example, the Cooperating Proof Calculus (CPC) defines a set of proof rules used internally by the CVC5 SMT solver [2], and a CPC proof certificate is a sequence of steps based on those rules. Despite the fact that several SMT solvers can produce proof certificates [8, 14, 17, 25, 36], no standard proof format has emerged in the SMT community yet. A program that checks the well-formedness of a proof certificate is called a *proof checker*.

A useful informal notion for capturing the difficulty of checking a proof is its *granularity*, which roughly corresponds to the computational complexity of checking that a step is correct. In particular, steps (and thus the proofs containing them) are *fine-grained* if they can be checked efficiently, and *coarse-grained* otherwise. We are deliberately vague about what *efficiently* means in this context, but one suitable general definition would be requiring polynomial time in the worst case. We will often refer to very coarse-grained steps as *holes*. An *elaborator* is a tool that transforms coarse-grained steps into fine-grained steps. A *translator* transforms proof certificates from one proof format to another. Elaboration and translation are often performed simultaneously.

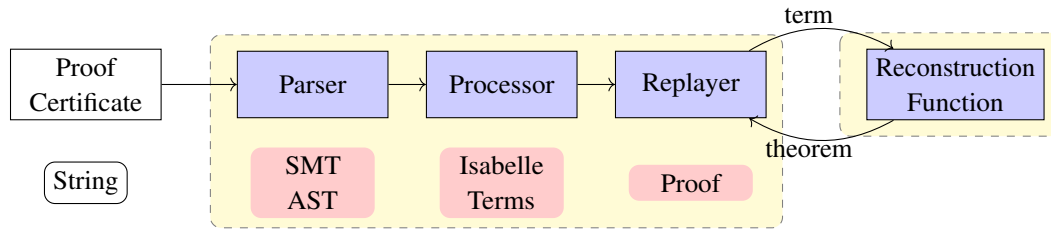
One way to define simple fine-grained rules in a user-friendly way is the domain-specific language RARE [33]. It is based on the SMT-LIB language and is designed for non-experts. Most rules that capture basic simplifications of SMT-LIB terms can be expressed in RARE. On top of this, RARE's abstract types allow type-generic rules. For example, reflexivity can be expressed as a single rule that applies to all SMT-LIB terms. Lachnitt et al. [29] show that Isabelle lemmas can be generated from RARE rules automatically.

2.2 Alethe

The Alethe proof format evolved from what was initially a custom proof certificate format for the veriT SMT solver. Its syntax is based on SMT-LIB [9]. Each step in an Alethe proof is either an assumption, an anchor opening a new subproof, or a regular step. Steps have a *rule* label, indicating the rule applied in that step, and a conclusion clause of the form $(\text{cl } F_1 \cdots F_n)$, where $n \geq 0$ and each F_i is a formula. Additionally they might have *premises* and *arguments*. Anchors introduce an additional context and can have a *context argument* to express transformations under binders like quantifiers. Assumptions either repeat input assertions or add new assumptions locally in a subproof.

Alethe has a natural-language specification that defines a total of 103 proof rules [7]. The SMT theory currently supported is the (quantified) theory of linear (real and integer) arithmetic with uninterpreted functions. Only two kinds of proof steps in Alethe proofs are considered holes. The first one is applications of a rule called `hole`, which is used to represent steps that do not correspond to any other Alethe rule. The second one is application of a rule called `lia_generic`, which is used to introduce any valid disjunction of integer linear inequalities, without any hints for proof checking. These steps are considered holes because they are extremely coarse-grained — filling them in can require super-polynomial work in principle.

Carcara [1] is an independent proof checker and elaborator for Alethe written in Rust. It can elaborate most `lia_generic` steps from veriT proofs by calling CVC5, which can output a fine-grained proof for unsatisfiable linear integer arithmetic queries. Carcara itself is not verified, meaning that it must be considered part of the trusted computing base when it is used to check proof certificates.



■ **Figure 2** The architecture of the smt tactic in Isabelle/HOL

2.3 Isabelle/HOL

Isabelle/HOL is a proof assistant based on higher-order logic with top-level polymorphism [32]. It follows the LCF [34] dogma; that is, its kernel only allows a small set of inference rules. From these, *tactics* can be built, which are programs that discharge proof goals. Techniques that can find proofs automatically help the user concentrate on the creative argument of their proof, without dealing with more tedious parts. Sledgehammer is a tool inside of Isabelle that calls external solvers to improve automation. To maintain high trustworthiness and avoid extending the trusted computing base beyond the Isabelle kernel, any result found by an outside tool has to be proven again inside Isabelle, based on some sort of directions from the external prover. This process is called *replay* and can easily fail unless the ATP provides detailed directions. These are typically in the form of a proof certificate, allowing Isabelle to implement a specialized tactic for each proof rule.

The smt tactic has been developed in Isabelle to interact with SMT solvers and replay their proof certificates. Figure 1 shows how the tactic works at a high level. The current goal (which is to be proven valid in the current context) is negated and encoded into an SMT-LIB problem together with a selection of premises from the context. The premises can be definitions or previously-stated lemmas and are selected by Sledgehammer based on relevance heuristics. A big part of the encoding consists in bridging the expressiveness gap between higher-order logic and the SMT solver's many-sorted first-order logic. If the external solver finds a proof, the smt tactic checks that the assumptions in the returned proof certificate match the original premises, and if every proof step is correct. For the latter check, a set of specialized tactics, which we call *reconstruction functions*, are called. Each specific tactic called depends on the proof rule used in the step being checked. For the most part, there are separate tactics for each of the proof rules in the Alethe standard.

Figure 2 provides more details on the proof reconstruction process for Alethe proof certificates. First, a parser translates the certificate from a string to an AST capturing Alethe terms. Then, the proof processor transforms it to a different AST now using Isabelle terms. Finally, the replay function builds an Isabelle context which contains variable and type definitions. It checks if the assumptions in the Alethe proof correspond to the original assertions selected by Sledgehammer. Then, for each proof step, it calls the appropriate reconstruction function for the corresponding Alethe proof rule, producing an Isabelle theorem stating the soundness of the proof step with respect to a formalization in Isabelle/HOL of Alethe's theory (i.e., linear arithmetic with uninterpreted functions).

2.4 Related work

Solvers for the Boolean satisfiability (SAT) problem support proof certificates of various granularities. External and internal elaborators and translators between coarse-grained formats, such as DRAT, intermediate formats like FRAT, and more fine-grained formats like LRAT or GRAT have been implemented [4, 16], although producing detailed proofs directly has recently become faster than producing coarse-grained proofs and then elaborating them [35]. Certified checkers for these formats

199 exist in several ITPs [16, 30, 31].

200 While several proof formats exist for SMT solvers, translations between them have not been the
 201 focus of community efforts, probably because no one format has prevailed yet. Instead, most proof
 202 formats are specific to individual solvers [13, 17, 26]. CVC5’s internal elaboration capabilities allow
 203 the user to either output coarse-grained proofs quickly, or fine-grained proofs at the cost of some
 204 performance overhead [8, 33]. CVC5 also features internal translations to different formats such as the
 205 LFSC [38] and CPC formats [3]. SMTCoq translates LFSC proofs into the certificate format used by
 206 Rocq [20]. There are also ongoing efforts to translate proof certificates from various theorem provers
 207 to Dedukti [15, 24].

208 While many checkers for SMT proof certificates exist, certified checkers or checkers written
 209 in a trusted environment are more rare. They include SMTCoq (verification in Rocq) and the
 210 reconstruction in Isabelle by Schurr et al. [37] that we extend in this work. Schurr et al. rely on veriT’s
 211 fine-grained proofs, but show that proof granularity is a trade-off and acknowledge that some steps
 212 are not fine-grained enough. For example, Alethe allows proof steps whose conclusion can contain
 213 equalities that have been implicitly reordered. In contrast, CVC5 proofs are more fine-grained and
 214 provide such extra operations explicitly as proof steps. However, some other rules establish facts not
 215 needed by Isabelle. For example, the bind rule, which renames bound variables, is superfluous in
 216 Isabelle which uses De Bruijn indices.

217 Blanchette et al. [10] extended Sledgehammer so that it could call SMT solvers as oracles, i.e.
 218 using one of Isabelle’s internal tactics and without replaying a proof certificate. The authors point
 219 out that only after working together with SMT solver developers were they able to obtain reasonable
 220 results. This work represents the groundwork for our project.

221 The proof reconstruction approach for the incorporation of external provers into ITPs was
 222 pioneered by Böhme and Webers’ reconstruction of z3 proofs [12]. The closest work to ours is
 223 the aforementioned paper by Schurr et al. [37], which extends the original reconstruction of veriT
 224 proofs [23] by fine-tuning veriT and Isabelle, compressing proofs, and supporting somewhat fine-
 225 grained simplification rules. Along the way, they recognized that documentation on the proof format
 226 was needed, which resulted in the Alethe specification.

227 There have been several large-scale evaluations of proof automation in Isabelle. Böhme et al. [11]
 228 compared three first-order theorem provers on goals from seven Isabelle theory files, creating the
 229 original *Judgement Day* benchmark set. The aforementioned work by Blanchette et al. [10] extended
 230 the set and for the first time also included SMT solvers in an evaluation. Desharnais et al. [19] set a
 231 new standard by generating a set of 5000 benchmarks from 50 randomly selected Isabelle libraries.

232 To our knowledge, our work is the first to contain a comparative evaluation, over the same
 233 problems, of two solvers that use the same reconstruction functions, enabling more interesting and
 234 detailed comparisons, as we discuss in Section 6.

235 **3 Adding Alethe Support to an SMT Solver**

236 While Alethe is intended as a generic SMT proof format, we encountered a number of limitations
 237 when instrumenting CVC5 to produce Alethe proofs. The main issues came from the specificity of
 238 the Alethe proof calculus to the particular way in which veriT operates, which is a consequence of
 239 Alethe having evolved from the original veriT proof format. As we discuss below, we addressed these
 240 issues either with particular translation schemas or by adding new proof rules when a translation
 241 was not suitable. Other relevant setbacks were due to errors, ambiguities, or missing assumptions in
 242 the format’s evolving specification. We made numerous improvements to the Alethe specification to

<p>CPC proof rule:</p> $\frac{F_1 \quad F_1 \rightarrow F_2}{F_2} \text{modus_ponens}$	<p>Alethe translation:</p> $\frac{(cl \ F_1) \quad \frac{(cl \ (F_1 \rightarrow F_2))}{(cl \ \neg F_1 \ F_2)} \text{implies}}{(cl \ F_2)} \text{resolution}$
--	---

■ **Figure 3** An Example of a CPC rule without a straightforward Alethe translation.

mitigate these issues for future endeavors in producing Alethe proofs from SMT solvers.¹

3.1 Rules without a direct counterpart

The most common issue when attempting to produce Alethe proofs in CVC5 was the lack of direct counterparts for some rules in the CPC format, which closely follows CVC5's internal calculus. This was addressed in most cases with a dedicated translation routine. For example, Figure 3 contains the CPC `modus_ponens` rule which concludes the formula F_2 from the premises F_1 and $F_1 \rightarrow F_2$. Since there is no corresponding rule in Alethe, the same conclusion must be derived via a combination of other rule applications, as shown in the figure. This example also illustrates how in Alethe there is an emphasis on clausal forms and inference by resolution, with auxiliary rules that derive clauses from more general formulas, whereas in CPC there is more emphasis on Natural Deduction-style rules which apply directly to general formulas. As a result, numerous CPC rules have to be converted to Alethe by first applying Alethe rules that convert CPC rule premises to valid clauses and then applying the Alethe `resolution` rule to those clauses. The expansion of single CPC inferences to multiple Alethe inferences can lead to noticeably larger sizes for the resulting Alethe proof.

The above case is actually relatively benign. Some cases showed more serious incompatibilities between the Alethe and CPC proof calculi. A specific example is the way they handle the normalization of associative commutative (AC) operators, which involves flattening and removal of duplicates. In Alethe, the `ac_simp` rule establishes an equality between a term and its normalized form, where *all* nested occurrences of AC operators are flattened to completion.² In CPC, on the other hand, a similar normalization is performed by the `ACI_NORM` rule, but only on the top-level applications of the AC operator in the original term. To translate an `ACI_NORM` step concluding $t_0 = t_1$, we make use of the fact that `ac_simp` is strictly more powerful, so it can be used to derive two intermediate conclusions of the form $t_0 = t'$ and $t_1 = t'$, where t' is the result of the more aggressive normalization achieved by `ac_simp`. The CPC conclusion $t_0 = t_1$ can then be derived via Alethe rules for symmetry and transitivity of equality. Alternatively, a rule could be added to Alethe to represent a more local version of AC simplification. This could be a possible avenue for improving the performance of CVC5 proof reconstruction in Isabelle.

3.2 Costly translation

In some cases, there is no reasonable way to translate a CPC step into Alethe — because it would require a huge number of steps, for example. In those cases, we extended the Alethe calculus itself. We only considered this as a last resort, however, since adding a rule to the standard impacts all of its users.

¹ We also fixed bugs we found in the veriT solver and added support for the updates to the Alethe standard, which were reflected in the release of veriT 2024.12.

² The motivation for this rule is to directly represent a performance-critical simplification [6, Sec. 4.6] at the cost of a more coarse-grained rule.

275 An example of a set of CPC rules that could not be directly represented or otherwise conveniently
 276 translated to Alethe is the set containing variants of `ARITH_MULT_POS`, which allows the two sides
 277 of an (in)equality to be multiplied by a positive factor. For equalities, the rule is as follows:

$$\frac{}{(m > 0 \wedge l = r) \rightarrow m * l = m * r} \text{ ARITH_MULT_POS}$$

279 While Alethe does include arithmetic rules, they usually only work on normalized (linear)
 280 inequalities, so for the rule above it would be necessary to break up the equality into two inequalities,
 281 normalize both $m * l$ and $m * r$, and then perform the arithmetic step via Alethe's `la_generic`
 282 rule, which is unfortunately expensive to check [1]. Moreover, using this translation would add a
 283 number of steps linear in the degree of nesting present in the input formula, which can be large. We
 284 deemed this unacceptable. We instead introduced new rules in Alethe which capture the rule above
 285 and each of its variants.

286 Another example of a CPC rule that is difficult to implement in Alethe is `ARITH_POLY_NORM`,
 287 which infers the equality $s = t$ of any two terms s and t that can be normalized to the same sum of
 288 monomials. Since this rule performs a full normalization on the given terms s and t , a translation
 289 into Alethe would require a full proof of how the two terms are normalized according to particular
 290 arithmetic simplifications. Thus, in this case too, we added a corresponding Alethe rule with the same
 291 semantics as the CPC rule.

292 3.3 Consistent decimal representation

293 A subtle aspect of Alethe, inherited from the SMT-LIB language, is that the sort of a number literal
 294 depends on the SMT-LIB logic associated with the problem. For example, the literal `123` is an
 295 integer in most logics, including those that permit real numbers. However, in logics with reals but
 296 not integers, the same literal becomes a real number. To complicate matters further, decimal literals
 297 (e.g., `2.45`) are allowed in SMT-LIB only in logics with real numbers whereas they are allowed in
 298 Alethe also in proofs that do not involve reals. For example, the `la_generic` rule uses decimal
 299 coefficients to certify the validity of a set of linear inequalities in both real and integer arithmetic.

300 To simplify this situation, we changed the Alethe standard to be not so liberal with the type
 301 assigned to numeric literal. Specifically, now a numeral (e.g., `123`) is always an integer, while a
 302 decimal (e.g., `12.0`) is always a real. Furthermore, both numerals and decimals are allowed in rules,
 303 independently of the logic set in the input problem. Finally, Alethe now has a dedicated syntax for
 304 writing rational constants in fractional form. In SMT-LIB, they must be written using the division
 305 operator, e.g., `(/ 1 3)`. This complicates parsing, since the entire term must be parsed to detect
 306 that it is to be treated as a constant. In Alethe, this rational number can now be written as the single
 307 literal `1/3` and parsed directly as a (real) constant.

308 3.4 Incompatibility

309 The only case we encountered where a translation from CPC to Alethe is not possible is when the
 310 CPC proof contains steps introducing fresh terms *without* a proper justification, which is not allowed
 311 in Alethe. Each such step results in a hole in the Alethe proof. This case is exceedingly rare, however,
 312 and thus does not significantly impact our results. Nevertheless, finding a good long-term solution for
 313 such incompatibilities is an important ongoing challenge.

314 4 Refactoring the Isabelle SMT Code Base

315 As mentioned earlier, our work revealed several problems with the code and architecture in Isabelle's
 316 Standard ML implementation of the `smt` tactic. We discuss our efforts to address these in this section.

4.1 Decoupling parts of the code

A common occurrence we encountered was needing to examine many different parts of the code base in order to fix a single bug or issue. This is a heavy burden for maintainers and potential developers, as they first must become experts on the whole system before being able to make a change. It would be much better if the code were more modular so that local changes could be made without having to understand everything else.

One might hope, for instance, that the code implementing reconstruction could be modified by a non-expert who just needs to add support for a single new feature or fix a bug with one operator. Unfortunately, this is not the case. For one thing, some transformations are done during parsing that affect what is seen in reconstruction in non-intuitive ways. For example, the SMT-LIB term $(\text{xor } x \ y)$ is converted during parsing to the disequality $\neg(y = x)$. This is to avoid introducing an xor operator in Isabelle. Therefore, a new developer looking to make a change to how xor is handled in the reconstruction function would be confused to discover that xor no longer appears after parsing.

To address such issues, we typically took one of two courses of action. In the cases where such design decisions seemed unjustified, we changed them. When changing was deemed too risky or difficult, we instead produced better documentation to explain the non-intuitive design decisions.

Fortunately, in many cases, we were able to successfully decouple different parts of the code. For instance, previously, the Alethe node data structure inside of Isabelle did not fully mirror the Alethe grammar. Specifically, in the grammar, both anchor steps and normal steps can have arguments, called *step arguments* and *context arguments*, respectively. The Alethe node data structure inside of Isabelle only contained a single field for arguments, which was used for both step and context arguments. Furthermore, for some steps, the proof postprocessor needs to communicate additional information to the reconstruction function. Any such information was also transmitted using the same argument field. Thus, the field was used for three different purposes. A developer working on the reconstruction function would have had to understand all the possible ways the argument field could be populated and take the appropriate context-dependent action. In this case, we changed the implementation so that the Alethe node data structure uses different fields for the different kinds of information.

We also tried to use strong modular design principles when adding new features. For example, in some cases, different strategies work better for the same Alethe rule, depending on which solver it comes from. This can happen, for example, when a rule is quite general and two solvers tend to use it in different ways. To address this, we made it easy to have the reconstruction code dispatch to different routines based on which solver generated the proof. As another example, we refactored a monolithic debug tracing capability to instead support debug traces for different components that can be turned on or off independently.

In general, a major goal of our work was to make it possible to work on or test many parts of the reconstruction pipeline separately and independently. We describe several tools we developed to aid with this goal in Section 5.

4.2 Supporting (new) Alethe features

As we dug into the reconstruction code in Isabelle, we discovered that only a fragment of the full Alethe standard was supported. Worse still, there was no explicit indication of which fragment was supported exactly, and even active developers did not know for sure. Finally, it was unclear whether some parts of the code were there for backwards compatibility with older versions of veriT or whether the code was dead and should be removed.

We took several steps to address these issues. First of all, we added support for parts of the standard that were clearly missing. An example of this is the *tautology* rule, which was always pruned from veriT proof certificates due to a veriT option that was always enabled when called by Isabelle.

As another example, reasoning about `xor` had not been implemented because none of the input problems from Isabelle contained it and `veriT` did not generate it. This, however, might not be the case for other solvers. We also discovered that some rules, such as `div_simplify`, were missing even though they do show up in `veriT` proofs. We attribute this to the fact that an insufficiently diverse set of test cases were used for the `veriT` reconstruction. We now support all of these rules as well as several others we discovered to be unsupported.

It was trickier to find and fix the rules that were only partially supported. For this, we created a library of regression tests. We discovered that many errors came from rules involving variadic operators. Terms such as `(and a b c)` are transformed into nested binary applications during parsing. This makes it harder to reconstruct rules that need to distinguish between applications of operators to multiple arguments and their equivalent representation as nested binary applications.

We also discovered cases where the Isabelle reconstruction supported proofs inconsistent with the Alethe standard. It was unclear whether this was the result of bugs or inconsistencies in `veriT` or of old code supporting previous versions of `veriT` or Alethe. Such code made it difficult to upgrade to a newer version of `veriT`, where these inconsistencies had been fixed. It also led to difficulties in the CVC5 integration. We made the decision to remove and update any code incompatible with the current Alethe standard and encouraged the `veriT` developers to fix any resulting discrepancies.

4.3 Supporting fine-grained rules automatically

For some proof steps, especially those that are very fine-grained, providing a full translation into Alethe is overkill. In this section, we discuss how the proof rules generating these steps can be supported much more easily. This functionality is also useful when updating a solver to a new version containing additional proof rules or when supporting a completely new solver that outputs Alethe proof certificates but extends it with some custom rules.

We assume in Section 3 that a solver has a fixed proof calculus with rules that are rarely changed. However, this often only holds for coarse-grained proof rules. More fine-grained rules are more likely to change frequently for solvers under active development. A good example of this is the set of term rewriting rules. Modern SMT solvers have hundreds of different rewrite rules for simplifying and normalizing terms. Developers frequently add new rules or modify previous ones as part of their ongoing performance tuning efforts. The approach implemented in CVC5 is to use a DSL called RARE, initially proposed by Nötzli et al. [33], which makes it easy to add or modify the rewrite rules used by the proof calculus.

For a large subset of rewrite rules expressible in RARE, we offer an automatic solution for supporting them that does not require knowledge of the SMT solver or of the Isabelle ML code at all. Lachnitt et al. [29] already showed how the Isabelle plug-in IsaRARE can automatically translate a rewrite rule in RARE to a lemma in Isabelle. We build on this work by adding infrastructure that can automatically use IsaRARE-generated lemmas (or even hand-written lemmas) for proof reconstruction. All that is required is to register such lemmas with the reconstruction algorithm using a simple interface we provide.

Note that even if a solver does not use RARE, this feature could still be useful for supporting a solver's fine-grained proof rules, as RARE is easy to learn and the reconstruction interface we provide for it is easy to use.

5 Toolbox

In the course of our efforts, we developed a suite of tools to aid with debugging and analysis. In this section, we describe these tools, which we hope will be useful for future developers and maintainers of the proof reconstruction code in Isabelle.

5.1 Isabelle internal testing tools

We call any testing code implemented in Isabelle that doesn't interact with an external ATP *internal*. Before our work, internal testing tools were very limited, and debugging was a tedious process. Developers were able to see the generated SMT-LIB problem and the proof certificate coming back from an external SMT solver. And, they could see when a reconstruction function failed and could also see the Isabelle term corresponding to the conclusion of a proof step. While these utilities were useful, there were two significant problems with this workflow.

First, if the conclusion had not been translated correctly, it was not easy to determine which part of the integration was responsible for the mistake. For example, the context arguments of bind steps can rename bound variables. These arguments were, however, not fully parsed in Isabelle according to the Alethe standard — a problem that had not been detected earlier since veriT would only output certain combinations. The available internal debugging information would point to a failure in a reconstruction function, but the problem was actually during replay, where the code for adding new variables to the context was implemented.

Second, since the contracts between different modules were not clearly defined, it was not always clear what to expect when looking at a reconstruction function. For example, the parser sometimes removes or changes proof steps to increase performance. This is done, e.g., for subproofs that merely rename variables. For an inexperienced developer, this could come as a surprise if they did not know about this functionality.

The standard approach for debugging such issues was to add print statements to figure out which part of the code had failed. Thus, we added a general trace-based debugging capability. Tracing can be turned off or on for each part of the pipeline independently and with different verbosity levels. We then added output traces that show not only relevant variables with their values but also explain what the code is doing.

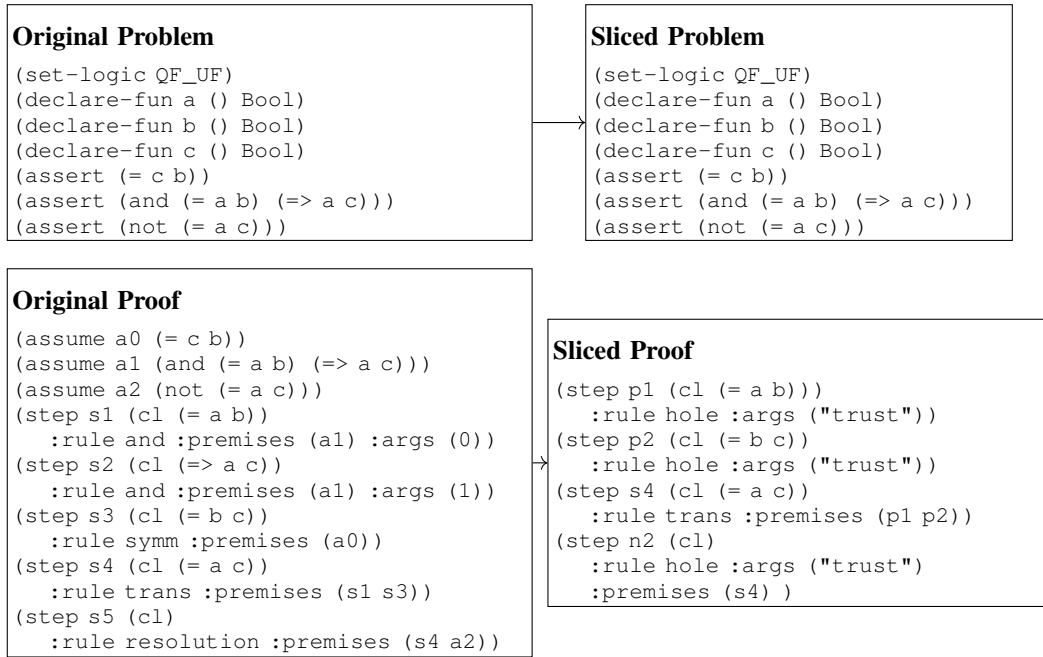
5.2 Checking SMT proof certificates

We have implemented a new function in Isabelle called `check_smt` that can check proof certificates generated by SMT solvers. `check_smt` takes two arguments: a file containing an SMT problem in SMT-LIB format and a file containing a proof certificate for that problem in Alethe format. The function parses the SMT-LIB input file, declares the constants and types in a new Isabelle context, imports the assumptions, and then checks the proof certificate via reconstruction.

While in principle the function is easy to implement (ignoring the inconveniences created by SMT-LIB's very liberal rules on what can be an identifier), its development is complicated by the need to check that the assumptions in the proof certificate match the assertions in the SMT-LIB problem. When the `smt` tactic produces an SMT-LIB file, it always gives names to each formula being asserted, and solvers use these names in their proof certificates. This makes it easy to recognize the assumptions again during proof reconstruction. However, there is no guarantee that the file and certificate read by `check_smt` will follow the same policy. To address this, we added the capability to match assertions with assumptions syntactically. If that fails, we allow Isabelle to do some lightweight reasoning (using the `auto` tactic) to try to match assumptions with assertions. In effect, `check_smt` makes it possible to use Isabelle as a trusted proof checker for SMT (see Section 6.1). It also makes it easy to create diverse test cases for reconstruction simply by generating SMT-LIB problems.

5.3 Testing reconstruction functions

While adding the capability to check external proofs helps to decouple the translation of proofs from their reconstruction, when the process fails, it might still not be clear if the error originates in the



■ **Figure 4** Basic configuration of slice. The slicing is done on step `s4` with depth limit 1.

reconstruction function, the certificate parser, or an incorrectly built context during replay.

For instance, we had trouble replaying a certain rule, and only after a long and tedious debugging session did it become clear that the reconstruction function was not the issue. Instead, the replay had not properly deleted an underscore in an internal representation of a term.

To aid with debugging in such situations, we introduce a new Isabelle method that can be used with most rules and that directly applies a reconstruction function to a lemma written in Isabelle. Users can then write an Isabelle lemma corresponding to the expected result of parsing an Alethe proof step and check that the reconstruction function works correctly. The main challenge was to emulate the context built up during the normal reconstruction process. To keep this simple the method does not support for now rules that introduce subproofs since those rules create nontrivial dependencies on contexts.

The new functionality allowed us to discover several bugs in the reconstruction functions and significantly helped in the development of new rules. In general, it also allows examples to be kept as regression tests, since complete proofs are more fragile and may change during a solver update.

5.4 Slicing proof steps

While it is helpful to check the reconstruction functions directly on Isabelle terms, this is mostly useful with hand-written tests. Checking the reconstruction of a step in an actual proof certificate would require translating that step to an Isabelle term. Even though this is possible, it is quite tedious and error-prone, and it becomes infeasible once variables have to be added to the context, e.g., as a consequence of reconstructing a subproof. This approach is thus not well suited for extensive testing of individual steps coming from actual proofs. Note also that, although our `check_smt` tool can be used to check the parser, processor, and replayer all at once, it is not suitable for testing the reconstruction function on a *specific* proof step: one would have to rely on that step being present in an end-to-end proof of an SMT-LIB input problem and, even then, the reconstruction could fail for reasons unrelated to the targeted proof step, or time out before reaching it.

477 To fill this gap, we developed a new subtool in the Carcara checker called *slice* that slices Alethe
 478 proofs with respect to any proof step in them. Specifically, *slice* takes a problem Q , a proof P , the
 479 identifier s of a *target* proof step in P , and a *depth limit* d . It then produces a new problem Q' and
 480 new proof P' . In the most basic configuration of *slice*, Q' is the same as Q while proof P' contains
 481 only the steps needed to generate step s , as well as the anchors of any subproofs of P that rely on
 482 s . The input value d indicates the maximum depth that P' can have as a proof tree with root s . If s
 483 has a deeper proof tree in P , the proof is cut at depth s by justifying any steps at that depth in P' as
 484 the application of the *hole* rule with argument "trust". Such steps are skipped during testing in
 485 Isabelle. The *hole* rule is similarly applied to step s to conclude the empty clause (cl) from it.³
 486 Checking the new proof P' can be used to reveal any problems with step s directly and results in a
 487 unit test for the proof rule used in that step. An example of a sliced proof with depth limit 1 is shown
 488 in Figure 4, where we can see that neither the original assumptions nor step $s2$ appear in the sliced
 489 proof, as they are not relevant for justifying target step $s4$.

Maximally Sliced Problem	Maximally Sliced Proof
<pre>(set-logic QF_UF) (declare-fun a () Bool) (declare-fun b () Bool) (declare-fun c () Bool) (assert (= a b)) (assert (= b c)) (assert (not (= a c)))</pre>	<pre>(assume a0 (= a b)) (assume a1 (= b c)) → (assume a2 (not (= a c))) (step s4 (cl (= a c)) :rule trans :premises (a0 a1)) (step s5 (cl) :rule resolution :premises (a2 s4))</pre>

■ **Figure 5** Optimized sliced problem and proof. The slicing is done on step $s4$.

490 An alternative configuration of *slice* allows the generation of even smaller problems and sliced
 491 proofs by focusing entirely on the targeted step. It can be used for a large class of steps not embedded
 492 in subproofs. In these cases, P' can be minimized by simply adding the premises and negated
 493 conclusion of the target step s as assumptions. An example is shown in Figure 5, where all the
 494 assertions in the original problem from Figure 4 are discarded and a new SMT problem Q' is built
 495 out of the premises of step $s4$. These premises then become assumptions in the sliced proof.

496 We considered extending this configuration to steps within subproofs by always adding as an
 497 assumption either the negated conclusion of the target step s or that of the last step t in the subproof
 498 in which s appears. Unfortunately, a combination of the limitations of the Isabelle reconstruction and
 499 the Alethe standard currently prevents us from taking advantage of this improvement. In future work,
 500 we plan to address this issue, which would extend the scope of this configuration.

501 6 Evaluation

502 In this section, we report on three evaluations of our work on the proof reconstruction pipeline in
 503 Isabelle. We first show how our *check_smt* tool can be used to turn Isabelle into a proof checker for
 504 SMT solvers. Second, we demonstrate the functionality of the Carcara *slice* command, which allows
 505 us to essentially do unit testing for specific proof rules. Finally, we report end-to-end results of using
 506 Sledgehammer with the new addition of CVC5, showing that it allows us to solve more goals.

³ Alethe proofs can only be refutations and so are required to end with the empty clause.

■ **Table 1** Solved benchmarks with a 5min timeout. Time and size are measured only on benchmarks solved by all solvers (common). Time is total, cumulative time (in seconds). The size is the average proof size.

Benchmark Set	common	CVC5 CPC			CVC5 Alethe			veriT Alethe		
		solved	time	size	solved	time	size	solved	time	size
max facts 16	1291	1333	80	53	1327	82	115	1305	505	101
max facts 32	1644	1741	123	62	1735	125	136	1657	1809	119
max facts 64	1981	2140	541	78	2130	549	177	2003	3430	147
max facts 128	2306	2526	991	87	2524	998	202	2331	2526	175
max facts 256	2535	2789	1928	101	2788	1942	242	2564	3590	175
max facts 512	2604	2895	2735	109	2892	2734	264	2646	5615	197
max facts 1024	2551	2926	4451	110	2924	4484	270	2607	7647	195

6.1 Using Isabelle as a proof checker

In their “Seventeen Provers Under the Hammer” evaluation, Desharnais et al. [19] generate several benchmark sets from 50 randomly selected files from Isabelle’s Archive of Formal Proofs, selecting 100 goals per file (they avoid selecting consecutive goals because those are often similar to each other). Using these 5000 goals, several families of SMT-LIB benchmarks are generated (they also generate non-SMT-LIB benchmarks, but we ignore those here). The families vary based on how many facts are selected and added from Isabelle’s definitions and lemmas. In general, adding more facts increases the chance that the solver will prove the goal, although adding too many of them increases the risk of slowing down the search. The families are called “max facts n ” for different values of n . Each family is based on the same set of 5000 goals, and benchmarks are generated by using the MePo relevance filter to add n facts to each goal. The value of n varies from 16 up to 1024.

For this experiment, we use our `check_smt` tool to run veriT and CVC5 on these benchmarks, have both of them generate Alethe proof certificates, and then check these proof certificates via reconstruction in Isabelle. Since we are only interested in comparing solvers using the Alethe pipeline, we did not run z3 as part of this experiment. All experiments are run on a cluster equipped with Intel(R) Xeon E5-2637 v4 CPUs and 8GB of memory per job.

We test the success of our translation approach with problems originating from Isabelle rather than on benchmarks from the SMT-LIB library, since Isabelle-generated benchmarks have a different structural profile than those generally found in SMT-LIB. For example, SMT-LIB benchmarks often contain huge (nested) conjunctions, while Isabelle-generated benchmarks do not. As another example, all Isabelle benchmarks that mention natural numbers use a particular encoding into integers, while SMT-LIB benchmarks use more heterogeneous encodings.

6.1.1 Solving and translating

To test the efficiency of the Alethe translation in CVC5, we run CVC5 twice, once producing proof certificates in its native format, CPC, and once producing Alethe proof certificates. We also run veriT. Table 1 reports the results. For each family of benchmarks, we report the number solved by each solver within a 5 minute time limit, where solved means that the solver successfully proves the problem unsatisfiable and prints a proof certificate. We also report the number of benchmarks solved by all three solvers (common) as well as, for each solver, the total time taken on the common set of solved benchmarks and the average proof size on that subset.

The first observation is that there is essentially no overhead in the time taken to generate Alethe certificates compared to that required to generate CPC certificates, suggesting that the translation is not particularly expensive time-wise. A small number of cases cannot be translated into Alethe

■ **Table 2** Reconstruction Success. The average (reconstruction) time in ms only takes benchmarks into account that were solved both by veriT and CVC5 and does not take the solving time into account.

Benchmark Set	CVC5 Alethe				veriT Alethe			
	solved	recon	unique	avg time	solved	recon	unique	avg time
max facts 16	1327	1326	45	386	1305	1295	14	131
max facts 32	1735	1727	101	344	1657	1643	17	127
max facts 64	2130	2121	168	417	2003	1980	27	156
max facts 128	2524	2508	236	522	2331	2307	35	209
max facts 256	2788	2767	271	693	2564	2536	40	209
max facts 512	2892	2869	311	874	2646	2608	50	246
max facts 1024	2924	2906	407	1303	2607	2560	61	334

due to the issue mentioned in Section 3.4. Note that the cases where the time for Alethe proofs is slightly lower can be explained by the fact that CPC proofs are a bit more verbose in their symbol declarations, so printing can take a bit longer.

Our second observation is that CVC5 is up to an order of magnitude more efficient at solving and constructing proofs than veriT. It also solves more problems, suggesting that the addition of CVC5 to Isabelle’s proof reconstruction pipeline will benefit Isabelle users, at least in some cases.

Our third observation is that the proofs produced by CVC5 in Alethe format are larger than both those produced by CVC5 in CPC format and those produced by veriT, possibly due to the translation from native CPC to Alethe in CVC5. It should be added, however, that our size measure is pretty crude as it simply counts the number of lines in a proof and not, for instance, the size of individual steps or their number. Note that more steps can be good, if that means that the proof is more fine-grained and so reconstruction is easier. However, the proof parser on the Isabelle side is rather inefficient, so producing proofs that are too detailed can result in performance losses due to parsing.

Compared with Desharnais et al.’s results, where both CVC5 and veriT peaked at 512 facts, CVC5 now does better with 1024 facts than with 512 (veriT still peaks at 512). This suggests that CVC5 got better at filtering out unnecessary assumptions. To rule out differences resulting from a larger timeout, we did a similar analysis to that in Table 1, but with a 30 second timeout, and saw the same trends.

6.1.2 Reconstruction

For each solved benchmark, we attempt to reconstruct the proof in Isabelle. Table 2 reports the results. For each family, we repeat the number solved from Table 1 and then report, of those, the number of proof certificates that are successfully reconstructed (recon) in Isabelle, as well as the number uniquely solved (unique) and the average time (avg time) taken on the commonly solved benchmarks. Each benchmark is given a timeout of 20 min.

Our first observation on these experiments is that most proofs are successfully reconstructed, suggesting that the Alethe pipeline is functioning well for these benchmarks. Proofs that fail to reconstruct often time out during reconstruction of holes or harder to check rules (such as `la_generic`). Further work could elaborate some of these rules into smaller steps.

A second observation is that Isabelle takes significantly longer to reconstruct CVC5 Alethe proofs than it does to reconstruct veriT proofs. This is due to several factors. First of all, CVC5 proofs are a bit larger on average, partly due to the translation from CVC5’s native format, as mentioned above. Second, the Isabelle translation was co-developed with the veriT proof generation, and the two systems were tuned and optimized together. We do expect that additional work and optimization will improve the CVC5 workflow as well.

■ **Table 3** Reconstruction success when using automatically generated lemmas and when using a general simplification tactic

Benchmark Set	total	reconstructed gen	reconstructed lemma	time gen	time lemma
max facts 16	1650	1638	1650	76	66
max facts 32	1085	1085	1066	64	80
max facts 64	1259	1256	1246	214	227
max facts 128	1329	1327	1317	74	87
max facts 256	1381	1381	1365	78	89
max facts 512	1386	1371	1354	99	104
max facts 1024	1378	1331	1311	87	99

Finally, we observe that, on average, a higher percentage of the CVC5 proofs are successfully reconstructed. This suggests that a potential benefit of larger proof sizes is some additional detail that helps the reconstruction succeed.

6.2 Slicing

Our second experiment demonstrates that proof steps can be supported without writing reconstruction functions and provides a case study for the functionality of our slicing tool.

We focus on CVC5's term rewrite rules, which are supported via automatically generated lemmas as explained in Section 4.3. For our experiment, we use every rewrite rule that appears at least once in the CVC5 Alethe proofs of the Seventeen Prover benchmark sets. Then, we randomly select up to 50 instances of each rewrite rule.

We use our slice tool to generate benchmarks for each selected rewrite step and then check the resulting problems and proofs with `check_smt`. We compare two ways of reconstructing the proofs. The first, which we call `gen`, uses a general tactic that first tries the simplifier with additional facts that we manually select for the goal. If that fails, it calls `auto_tac`. In general, we tried to be generous with the power this tactic has. In the second approach, we add lemmas proven by Lachnitt et al. as part of their effort to verify CVC5's term rewriting rules and use their tool IsaRARE to generate lemmas for any newly added rules. We also instruct CVC5 to add the instantiations for the variables in the lemma and reconstruct the step by matching against the lemma. This can be slow since the right lemma needs to be fetched and the instantiation can be more costly than the matching the simplifier does.

The results are summarized in Table 3. They show that using lemma during reconstruction is faster and more robust, which is a confirmation of the value of developing custom reconstruction lemmas rather than just relying on generic Isabelle techniques.

6.3 cvc5 as a Sledgehammer back end

Our final experiment looks at how the integration of CVC5 affects overall Sledgehammer performance. Sledgehammer is somewhat complex; however, it roughly works as follows. Given a goal to prove in Isabelle, it first calls a number of ATPs in parallel to attempt to solve the goal. Note that in this first phase, no proofs are requested and no reconstruction is attempted. Instead, whenever an ATP succeeds, it is asked for an *unsatisfiable core*, i.e., a (preferably small) subset of the problem that is unsatisfiable on its own. Next, all of the collected unsatisfiable cores are passed to a set of built-in simplification tactics in Isabelle, which often can then produce an Isabelle proof. When this fails, these problems are sent to the `smt` tactic which calls all of the available proof-producing SMT solvers and attempts to reconstruct any proof certificate obtained.

For our experiment, we use Mirabelle [11], a tool that automates calls to Sledgehammer. We run the experiment on Isabelle’s HOL Library.⁴ Following Schurr et al. [37], we set the timeout to 10 s for the entire call to the `smt` tactic. In general, we followed their setup as closely as possible. Unfortunately, some Mirabelle options are no longer available in the latest version of Isabelle.

Our results⁵ show that for the HOL Library, the `smt` tactic is called 267 times. Of these 267 queries, 257 can be solved by reconstructing `veriT` proof certificates, and 119 can be solved by reconstructing `z3` certificates. Together `veriT` and `z3` succeed on 259. On the other hand, proof reconstruction using `CVC5` succeeds on 243, including half (4/8) of the remaining problems unsolved by `veriT` and `z3`. Overall, `CVC5` is significantly more successful than `z3` but not as successful as `veriT`. We expect this is largely due to the observation already made that `CVC5` proofs are slightly larger and take a bit longer to reconstruct. With only a 10 second timeout, `veriT` has an advantage. We expect that with a longer timeout or with more optimization effort, `CVC5` would catch up and probably surpass `veriT`, as is seen in the first experiment which has a longer timeout.

7 Conclusion and Future Work

We reported on our experience adding `CVC5` as a new external ATP in Isabelle’s proof reconstruction pipeline. We found and fixed various problems in the reconstruction and in the Alethe standard. We also developed a toolkit that could benefit future users and developers of proof reconstruction in Isabelle. Our evaluation shows that `CVC5` improves the capabilities of Isabelle’s `smt` tactic.

Isabelle currently uses strategies for `CVC5` based on the configuration used in the annual SMT-COMP competition. However, as mentioned in Section 6.1, Isabelle-generated problems are often quite different from those in SMT-COMP, which are selected from the SMT-LIB library. Thus, one promising direction for future work is to explore what strategies yield the best results on Isabelle benchmarks specifically. Furthermore, for some reconstruction tasks, `CVC5` and `veriT` might have different needs. Currently, the reconstruction code in Isabelle is optimized for `veriT`. Additionally, Isabelle compresses certain proof elements such as subproofs that rename variables. That benefits `veriT` more since the solver renames all variables in a proof as a first step. This was implemented to speed up reconstruction after first results with `veriT` were not performant enough. It would be interesting to see what optimizations could improve `CVC5`’s performance in a similar fashion.

This paper focuses on comparing Alethe proof certificates from `veriT` and `CVC5` to provide a baseline for additional solvers that may decide to adopt the Alethe format. However, `CVC5` supports more theories than Alethe currently does, including the theory of bit-vectors which is essential in many verification applications. The machine-word library in Isabelle and its extension in AFP have been used for important and safety-critical projects such as the verification of the seL4 microkernel [27]. Thus, improved automation for machine words is highly desirable. `CVC5` has a strong bit-vector solver capable of producing proof certificates. Therefore, another promising direction for future work is to extend Alethe with bit-vector rules, translate `CVC5` bit-vector proofs, and add reconstruction functionality to Isabelle. Other theories that `CVC5` supports that might be interesting targets for reconstruction are the theory of strings and regular expressions and the theory of finite sets.

⁴ Schurr et al.’s original experiments with `veriT` also included three AFP entries, but there were not many calls to the `smt` tactic for those files, so we decided to use only the HOL Library.

⁵ Run on an Intel i9-12900 with 128 GB RAM, running 3 Mirabelle instances at the same time with a limit of 30 GB given to the underlying Standard ML implementation.

References

- 1 Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara: An efficient proof checker and elaborator for SMT proofs in the Alethe format. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*, volume 13993 of *Lecture Notes in Computer Science*, pages 367–386. Springer, 2023. doi:10.1007/978-3-031-30823-9_19.
- 2 The authors of cvc5. cvc5’s C++ API. <https://cvc5.github.io/docs-ci/docs-main/api/cpp/enums/prooofrule.html>, 2025. [Online; accessed 12-February-2025].
- 3 The authors of cvc5. Proof production. <https://cvc5.github.io/docs-ci/docs-main/proofs/proofs.html>, 2025. [Online; accessed 10-Mar-2025].
- 4 Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. *Log. Methods Comput. Sci.*, 18(2), 2022. URL: [https://doi.org/10.46298/lmcs-18\(2:3\)2022](https://doi.org/10.46298/lmcs-18(2:3)2022), doi:10.46298/LMCS-18(2:3)2022.
- 5 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.
- 6 Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning*, 64(3):485–510, 2020. doi:10.1007/s10817-018-09502-y.
- 7 Haniel Barbosa, Mathias Fleury, Pascal Fontaine, and Hans-Jörg Schurr. The Alethe Proof Format. An Evolving Specification and Reference. <https://verit.gitlabpages.uliege.be/alethe/specification.pdf>, 2024. [Online; accessed 22-November-2025].
- 8 Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark Barrett. Flexible proof production in an industrial-strength SMT solver. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022. doi:10.1007/978-3-031-10769-6_3.
- 9 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- 10 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013. URL: <https://doi.org/10.1007/s10817-013-9278-5>, doi:10.1007/s10817-013-9278-5.
- 11 Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010. doi:10.1007/978-3-642-14203-1_9.
- 12 Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 1*, pages 179–194. Springer, 2010.
- 13 Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009. doi:10.1007/978-3-642-02959-2_12.

- 694 **14** Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In
695 Alastair F. Donaldson and David Parker, editors, *Model Checking Software - 19th International Workshop,*
696 *SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer*
697 *Science*, pages 248–254. Springer, 2012. doi:10.1007/978-3-642-31759-0_19.
- 698 **15** Alessio Coltellacci, Stephan Merz, and Gilles Dowek. Reconstruction of SMT proofs with Lambdapi. In
699 Giles Reger and Yoni Zohar, editors, *Proceedings of the 22nd International Workshop on Satisfiability*
700 *Modulo Theories co-located with the 36th International Conference on Computer Aided Verification (CAV*
701 *2024), Montreal, Canada, July, 22-23, 2024*, volume 3725 of *CEUR Workshop Proceedings*, pages 13–23.
702 CEUR-WS.org, 2024. URL: <https://ceur-ws.org/Vol-3725/paper8.pdf>.
- 703 **16** Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp.
704 Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26*
705 *- 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017,*
706 *Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017.
707 doi:10.1007/978-3-319-63046-5_14.
- 708 **17** Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Proofs and refutations, and Z3. In Piotr Rudnicki,
709 Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Proceedings of the*
710 *LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th*
711 *International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, volume 418
712 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. URL: [https://ceur-ws.org/Vol-418/](https://ceur-ws.org/Vol-418/paper10.pdf)
713 [paper10.pdf](https://ceur-ws.org/Vol-418/paper10.pdf).
- 714 **18** Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan
715 and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th*
716 *International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory*
717 *and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*,
718 volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/
719 978-3-540-78800-3_24.
- 720 **19** Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under
721 the hammer. In June Andronick and Leonardo de Moura, editors, *13th International Conference on*
722 *Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPICs*, pages
723 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: [https://doi.org/10.](https://doi.org/10.4230/LIPICs.ITP.2022.8.4230/LIPICs.ITP.2022.8)
724 [4230/LIPICs.ITP.2022.8](https://doi.org/10.4230/LIPICs.ITP.2022.8.4230/LIPICs.ITP.2022.8), doi:10.4230/LIPICs.ITP.2022.8.
- 725 **20** Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark
726 Barrett. Smtcoq: A plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kuncak,
727 editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany,*
728 *July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages
729 126–133. Springer, 2017. doi:10.1007/978-3-319-63390-9_7.
- 730 **21** Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.
- 731 **22** Mathias Fleury and Nikolaj Bjørner. Internal variables in proofs. [https://github.com/](https://github.com/Z3Prover/z3/issues/5073)
732 [Z3Prover/z3/issues/5073](https://github.com/Z3Prover/z3/issues/5073), 2021. [Online; accessed 22-November-2025].
- 733 **23** Mathias Fleury and Hans-Jörg Schurr. Reconstructing veriT proofs in Isabelle/HOL. In Giselle Reis and
734 Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP*
735 *2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 36–50, 2019. doi:10.4204/
736 EPTCS.301.6.
- 737 **24** Mohamed Yacine El Haddad, Guillaume Burel, and Frédéric Blanqui. EKSTRAKTO A tool to reconstruct
738 Dedukti proofs from TSTP files (extended abstract). In Giselle Reis and Haniel Barbosa, editors,
739 *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August*
740 *26, 2019*, volume 301 of *EPTCS*, pages 27–35, 2019. URL: [https://doi.org/10.4204/EPTCS.](https://doi.org/10.4204/EPTCS.301.5)
741 [301.5](https://doi.org/10.4204/EPTCS.301.5).
- 742 **25** S Hitharth, Cayden Codel, Hanna Lachnitt, and Bruno Dutertre. Extending DRAT to SMT. pages 18–28.
743 TU Wien Academic Press, 2024.
- 744 **26** Jochen Hoenicke and Tanja Schindler. A simple proof format for SMT. In David Déharbe and Antti
745 E. J. Hyvärinen, editors, *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories*

- co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70. CEUR-WS.org, 2022. URL: <https://ceur-ws.org/Vol-3185/paper9527.pdf>.
- 27 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. *seL4: Formal verification of an os kernel*. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.
- 28 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. *seL4: formal verification of an os kernel*. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSOP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.
- 29 Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. *IsaRare: Automatic verification of SMT rewrites in Isabelle/HOL*. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2024. doi:10.1007/978-3-031-57246-3_17.
- 30 Peter Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 457–463. Springer, 2017. URL: https://doi.org/10.1007/978-3-319-66263-3_29.
- 31 Peter Lammich. Fast and verified UNSAT certificate checking. In Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, volume 14739 of *Lecture Notes in Computer Science*, pages 439–457. Springer, 2024. URL: https://doi.org/10.1007/978-3-031-63498-7_26.
- 32 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
- 33 Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Reconstructing fine-grained proofs of rewrites using a domain-specific language. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 65–74. IEEE, 2022. URL: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12, doi:10.34727/2022/ISBN.978-3-85448-053-2_12.
- 34 Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *CoRR*, abs/1907.02836, 2019. URL: <http://arxiv.org/abs/1907.02836>, arXiv:1907.02836.
- 35 Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPICs*, pages 21:1–21:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: <https://doi.org/10.4230/LIPICs.SAT.2023.21>, doi:10.4230/LIPICs.SAT.2023.21.
- 36 Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021. doi:10.4204/EPTCS.336.6.

- 797 **37** Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs
798 in a proof assistant. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 -*
799 *28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*,
800 volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021. doi:10.1007/
801 978-3-030-79876-5_26.
- 802 **38** Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking
803 using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013. URL: [http:](http://dx.doi.org/10.1007/s10703-012-0163-3)
804 [//dx.doi.org/10.1007/s10703-012-0163-3](http://dx.doi.org/10.1007/s10703-012-0163-3), doi:10.1007/s10703-012-0163-3.