Reliable Reconstruction of Fine-Grained Proofs in a Proof Assistant

Hans-Jörg Schurr
Verification of Distributed Systems (VeriDis)
University of Lorraine, CNRS, Inria, and LORIA
Villers-lès-Nancy, France
hans-jorg.schurr@inria.fr

Mathias Fleury
Institute for Formal Models and Verification (FMV)
Johannes Kepler University
Linz, Austria
Research Group 1: Automation of Logic
Max-Planck Institute für Informatik
Saarbrücken, Germany
mathias.fleury@jku.at

Martin Desharnais
Research Institute Cyber Defence (CODE)
Universität der Bundeswehr München
München, Germany
martin.desharnais@unibw.de

Abstract

We present a reliable and fast reconstruction of proofs generated by the SMT solver veriT in Isabelle, building upon the work by Barbosa et al. The fine-grained proof format of veriT makes the reconstruction simple and efficient. For typical proof steps, such as arithmetic reasoning and skolemization, our reconstruction is able to avoid expensive search. We compare our procedure with the existing Z3 integration and show similar levels of robustness while solving more problems. Skipping some steps of the very detailed proofs is possible and improves the performance of proof checking.

Keywords: automatic theorem provers, proof assistants, proof verification

1 Introduction

Proof assistants are used in verification, formal mathematics, and other areas to provide trustworthy, machine-checkable formal proofs of theorems. Proof *automation* reduces the burden of proof on users, thereby allowing them to focus on the core of their arguments instead of technical details. A successful approach to automation is to invoke an external automatic theorem prover, such as a Satisfiability Modulo Theories (SMT) solver [10], and to reconstruct the generated proof using the proof assistant's inference kernel. Typically such invocations will be generated by a "hammer", like Sledgehammer for Isabelle [13]. Invoked on a proof goal, this tool heuristically selects facts from the background theory and then uses an external solver to remove facts not needed to discharge the goal. Since this will not directly result in a trusted proof, Sledgehammer subsequently attempts to find the fastest trusted method that can recreate the proof. This is often a call to an automated theorem prover for whom a proof reconstruction method exists.

Isabelle's smt tactic translates the current goal to the SMT-LIB format [9], runs an SMT solver, parses the proof, and replays it through Isabelle's kernel. It was originally developed for the SMT solver Z3 [19] by Böhme and Weber's [16] and we extended [25] to support veriT under the name `veriT_smt`.

CVC4 [8] is one of the best SMT solver on Sledgehammer generated problems [12], but it currently does not produce proofs for problems with quantifiers. To reconstruct CVC4 proofs, Sledgehammer's primary approach is to use the smt proof method based on Z3 [19]. However, this often fails because CVC4 uses more elaborated quantifier instantiation techniques: A provable problem for CVC4 might be unprovable for Z3. Because the state-of-the-art SMT solver veriT [17] supports theses technique and generated detailed proofs, we reconstructs its proofs to increase the efficiency of Sledgehammer.

We build upon the prototype `veriT_smt` by Barbosa et al. [6, 25], which reconstructs proofs generated by veriT (Sect. 2) in the interactive theorem prover Isabelle. We make the tactic more efficient and more robust.

To improve the existing reconstruction (Sect. 3), we extended veriT to make proofs more detailed. Instead of a single rule simplifying the term with a combination of propositional, arithmetic, and quantifier reasoning, veriT now gives several detailed steps to simplify checking. Similarly, more information is provided to avoid search, like for linear arithmetic and for normalizations performed automatically (Sect. 4).

Since users can call the `veriT_smt` tactic many times in a theory file, users expect that the entire method, proof search

CPP '21, January 17–22, 2021, Copenhagen, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and reconstruction, is fast. veriT exposes options to fine-tune the proof search. We generated typical SMT problems and selected four different combinations of options with complementary performance on these problems. We then evaluate the reconstruction with Sledgehammer (Sect. 5). The resulting tactic is as efficient as the existing reconstruction for Z3 and solves proof goals not solved before.

Then, we study the time required to reconstruct each kind of rule, both per application and cumulative. This shows where and how to improve reconstruction. Many simple rules are fast to reconstruct, but also occur often. Therefore, we have added a *step skipping* mode which removes some of those simple steps not needed for the reconstruction ahead of time (Sect. 6).

Finally, we discuss related work (Sect. 7)

2 veriT and Proofs

veriT is an open source, CDCL($\mathcal{T}$) SMT solver. In proof production mode, it supports the theories of uninterpreted functions with equality and linear real and integer arithmetic and quantifiers. To support quantifiers veriT uses quantifier instantiation and extensive preprocessing. It uses SMT-LIB [9] as the input format.

A proof produced by veriT is an indexed list of steps. Each step has a conclusion term and is annotated with a rule, a list of premises, and optionally some rule-dependent arguments. veriT distinguishes 90 different rules [36], ten of which deal with connectors not produced by Isabelle.

Subproofs enable the use of local assumptions or of an additional *context* to reason about binders, e.g., to express preprocessing steps like transformation under quantifiers.

The conclusion of rules with contexts are always equalities. The context models a substitution into the free variable of the term on the left-hand side of the equality.

Consider the following proof extract:

```
(define-fun X () A (choice ((vr A)) (f vr)))
(assume a0 (exists (x A) (P x))
(anchor :step t1 :args (:= x vr))
(step t1.t1 (= x vr) :rule cong)
(step t1.t2 (= (f x) (f vr)) :rule cong)
(step t1 (= (exists (x A) (f vr))
            (exists (vr A) (f vr))
   :rule bind)
(step t2 (= (exists (vr A) (f vr)) (f X))
   :rule sko_ex)
```

The output syntax of the proofs is inspired by the SMT-LIB format. Accordingly, the output is a list of commands. To skolemize terms, we use *Hilbert's choice*: $\epsilon x.\ \varphi$ is a value v, such that $\varphi[v/x]$ holds if any exists. Skolem constants are optionally defined by standard **define-fun** commands. Here, `X` is defined as $\mathrm{f}(\epsilon x.\ \mathrm{f}(x))$. The **assume** command is used to assert assumptions (here from the input problem). Subproofs starts by the **anchor** command and can introduce a context. Semantically, the context is a shorthand notation for a lambda abstraction of the free variable and an application of the substituted term. Here the context is $x \mapsto \mathrm{vr}$ and the step `t1.t1` means $\triangleright\ (\lambda x.\ x)\ \mathrm{vr} = \mathrm{vr}$. The step is proven by applying congruence (rule cong). Then congruence is applied again to prove that $(\lambda x.\ f\ x)\ \mathrm{vr} = f\ \mathrm{vr}$ (step `t1.t2`) and the α-renaming is finished in step `t1`. The sko_ex step `t2` proofs the skolemization $(\exists \mathrm{vr}.\ f\ \mathrm{vr}) \iff f\ X$.

To reduce the size of the proof, perfect sharing is achieved following the SMT-LIB format. This achieves a large reduction of the proof size.

Internally veriT records proofs in an eager fashion. During proof search each module is appending steps into an array. Every written step is final: there is no elaboration phase after the proof has been found. Premises are represented by list indices of the corresponding steps. If necessary, deduced terms can be associated with their proof via a hash table.

While steps are not modified after creation, veriT performs some cleanup before printing the proof. First, a pruning phase removes branches of the proof not connected to the root $\bot$. Such branches could, for example, be the instantiations of quantified formulas that were not needed to show unsatisfiability. Second, a merge phase detects shared conclusions and rewrites the proof such that those duplicates are only proved once. A final pass prepares the data structures for the optional term sharing via name annotations and the explicit definition of skolem functions.

3 Overview of `veriT_smt`

Isabelle is a generic proof assistant based on an intuitionistic logic framework, *Pure*, and is nearly only used parameterized with a logic. In this work we use only Isabelle/HOL, the instantiation of Isabelle with higher-order logic with rank-1 (top level) polymorphism. Isabelle adheres to the LCF [26] tradition that started in the 1970s. Only inferences that go through the small kernel are allowed. Tactics are programs that prove a goal by using only the kernel. This tradition also means that external tools, like SMT solvers, are not trusted.

Nevertheless, external tools are successfully used. Either, they provide information, such as UNSAT cores, which are useful to find tactics which can proof a goal, or they provide a detailed proof which can goes through the trusted kernel. In Isabelle/HOL the Sledgehammer tool implements the former, while tactics like smt (using Z3) and `veriT_smt` (using veriT) implement the latter approach. The focus of our work is improving the `veriT_smt` prototype. Our reconstruction procedure replays proofs generated by veriT and it is possible for Sledgehammer to suggest it to the user.

Isabelle's proof interaction is built around the *context*. It encompasses all the symbols and definitions available globally and locally in the current proof. Assumptions (e.g., to

introduce local assumption) and variables can extend contexts. Once a theorem is proven in an extended context, it can be exported back to the original context: new assumptions become as assumptions of the theorem and variables universally quantified.

Isabelle's smt tactic by Böhme and Weber [16] translates the current goal to the SMT-LIB format [9], runs an SMT solver, parses the proof, and replays it through Isabelle's kernel. We will call the first z3_smt (corresponding to the smt tactic in Isabelle) to keep the distinction clear. The proof formats and steps of Z3 and veriT are so different that the reconstruction is not shared.

The first step [16] is the negation of the proof goal to allow proof by contradiction and to encode the goal into the first-order logic supported by veriT and Z3. Although a veriT version with some higher-order support exists [7], it does not generate proofs yet. After the encoding, veriT is called. If it finds no proof, the tactic returns an error.

The second step [25, Sect. 3] parses the proof. We represent it as a DAG with $\bot$ as the only conclusion. The proof contains subproofs with local assertions. The proof structure is slightly amended to simplify reconstruction.

The third step consists of converting the SMT-LIB terms to typed Isabelle terms. We also reverse most of the encoding used to convert higher-order into SMT-LIB terms.

Now the reconstruction itself can start. The fourth step consists of preparing the choice terms. The proof format introduces choice terms as defined constants. For technical reasons, the constants are not added as definitions on the Isabelle side, but instead as constants and their definition is assumed. The extra-assumptions are discharged at the end to get a proof of $\bot$.

In the fifth step, we finally check each remaining proof step. We start by checking that the input assertions, as repeated in the proof output, match their Isabelle counterpart and then reconstruct the proof step by step.

4 Tuning the Reconstruction

In this section, we describe the changes we made to the reconstruction in order to improve efficiency. Most changes were backed up by a change in veriT to produce proofs with more details. We split the rule `connect_equiv`, which collected various prepocessing simplifications into small and well-defined rules (Sect. 4.1).

Previously, veriT implicitly normalized every generated step. For example, repeated literals were immediately deleted. We now produce proofs for this normalization (Sect. 4.2).

Finally, the linear-arithmetic steps now contains coefficients which allow Isabelle to reconstruct the step without relying on its limited arithmetic automation (Sect. 4.3). On the Isabelle side, we have implemented an efficient reconstruction for skolemization steps (Sect. 4.4) and selectively decode the first-order encoding to preserve the structure of terms (Sect. 4.5)

4.1 Preprocessing Rules

During preprocessing, veriT performs simplifications on the operator level. Those simplifications are often akin to simple calculations. For example, they replace the term $a \times 0 \times f(x)$ by the constant 0. Previously, the rule `connect_equiv` collected all those simplifications in a single step. Furthermore, no list of the operations performed by this rule existed. This forced Isabelle to fall back to a hand-crafted list of automatic tactics.

By reading the source code of veriT, we created a list of all simplifications emitted this rule. After some deliberation, we decided to split the rule into 17 new rules: one rule per operator, one to replace boolean operators with their definition (such as xor), and one to replace n-ary operator applications with binary applications. The example above now produces a `prod_simplify` step with the conclusion $a \times 0 \times f(x) = 0$. This is a compromise with having one rule for every possible simplification. Our approach has he benefit that the fallback to automated tactics utilized by the reconstruction can catch simplifications added to veriT in the future until the developer adapts the reconstruction.

On the Isabelle side, the reconstruction becomes easier and faster, because we can direct the search instead of trying automated tactics that can also work on other parts of the formula. We use the simplifier to reconstruct some of those rules. For example, the simplifier handles the numeral manipulations of the `prod_simplify` rule and we restrict it to only use arithmetic lemmas.

Moreover, since we know the performed transformations, we can ignore some parts of the terms by *abstracting* or replacing them by constants. This results in a more robust reconstruction, because terms are smaller, the search is more directed, and we are less likely to hit some search-depth limitation of Isabelle's auto tactic. The reconstruction is also easier to debug.

4.2 Implicit Steps

The internal API of veriT's proof module performs an automatic normalization on each recorded proof step. Namely, it removes repeated literals and double negations from literals and simplifies clauses that contain complementary literals to $\top$. While documented, the normalization is not proof producing.

We extend the normalization code such that it generates a new proof step. For the removal of duplicated literals and the detection of clauses with complementary literals we add a step that has the original clause as premises and the simplified clause as conclusion. To remove the double negation $\neg\neg t$ we introduce the tautology $\neg\neg\neg t \vee t$ and resolve it with the original clause. Naturally, the code creating this rule has to bypass the normalization process. Since the API hides the

normalization of steps, our changes do not affect any other part of veriT.

On the Isabelle side, the reconstruction becomes more regular with fewer special cases. For example, the rule `ite1` corresponds to the theorem (if b then Q else P) $\Longrightarrow b \vee P$. However, it could also produce (if P then Q else P) $\Longrightarrow P$, where $P \vee P$ was silently normalized to P. The reconstruction originally dealt with this by first generating the conclusion of the theorem and then running the simplifier to remove the duplicates to match veriT's conclusion. Now this done in two steps. First we apply the rule, which is efficient. After that, a separate step removes duplicates literals if needed. Furthermore, we can abstract over the literals when reconstructing this step. This improves the reliability of the reconstruction, because the tactic is only applied to constants and is not distracted by the potentially complex structure of P.

Tautological clauses are sometimes generated during the proof search. For example, a step can prove $\neg P \vee P$ from $P \rightarrow P$. Since such tautologies are normalized to $\top$, the structure of the step is lost and the reconstruction could not cope with those cases. Handling them is possible by considering that almost every step can also generate $\top$, but a separate proof step is a simpler solution.

We introduced an explicit quantified-clausification rule which is issued before instantiating. While this rule is not detailed, having an explicit clausification step helps not at least for debugging. It also improves reconstruction, since we no longer clausify the formula before every time. A more fine-grained proof can be achieved by performing the clausification after creating the instance, but this will require an extensive refactoring of the instantiation module of veriT. Since this will not require additional rules, the reconstruction will not require any change.

One implicit transformation remains without proof: veriT applies the symmetry of equality implicitly. This *reordering* of equalities enforces the global invariant that for equality $t_1 = t_2$ the term index of t_1 is less or equal than the term index of t_2. The term index is the position of the term in the global term array. To enforce this invariant the API used to create equalities silently reorders arguments. Since this API is also used during parsing, such reordering is applied even before initialization of the proof module. The initial reordering is the motivation for repeating the input assertions in the proof: most further proof steps do not have to handle reordering anymore [25].

We also improve the proof and reconstruction of quantifier instantiation steps. Since the instantiation scheme *conflicting instances* only works on clauses, universally quantified terms are clausified before instantiation. This was performed implicitly with the `forall_inst` rule.

4.3 Arithmetic Reasoning

Proof Production. veriT supports and produces proofs for linear real and integer arithmetic. The real arithmetic solver relies on the simplex method and follows the implementation in yices as described by Dutertre and de Moura's [21]. The integer solver is a restricted implementation of a branch-and-bound method.

The core proof rule used by the arithmetic solver is the `la_generic` rule. This rule produces a tautological clause of linear inequalities and equations. A `la_generic` step is created whenever the arithmetic solver determines that the current propositional model is unsatisfiable in the theory of linear arithmetic. The justification of the step is a list of coefficients such that the linear combination is a trivially contradictory inequality after simplification (e.g., $0 \geq 1$). Farkas lemma guarantees the existence of such coefficients for the reals. The coefficients are usually derived by the simplex algorithm like in veriT. Checking the step amounts to bring the negated inequalities into normal form, multiplying them with the provided coefficients, and to then form their sum. The sum is the trivial contradiction.

While previously this clause was presented without additional information, we now provide the coefficients to ensure that the step is a mechanically checkable certificate, without the need to rediscover the coefficients. This is similar to the approach of Z3 [18].

Additionally, veriT strengthens inequalities on integer variables before adding them to the simplex tableau. For example, the inequality $x + y > 3$ becomes $x + y \geq 4$, if x and y are integers. Rational coefficients for integer variables express strengthening: $2x < 3$ multiplied by $\frac{1}{2}$ gives $x \leq 1$. The reconstruction must replay this strengthening.

Since veriT's simplex stores the coefficients without a sort, we print them as real numbers (e.g., `1.0`) whenever the input problem uses a theory with real numbers and otherwise resort to integers (e.g., `1`) and integer division. It is the task of the reconstruction to correctly apply the coefficients.

Overall, a linear arithmetic proof step can look like:

```
(step t11 (cl (not (<= f3 0))
              (<= (+ 1 (* 4 f3)) 1))
    :rule la_generic :args (1 (div 1 4)))
```

In proofs generated by Z3, coefficients are also produced for arithmetic steps called `farkas-lemma`. z3_smt ignores them during the reconstruction. However, although the coefficients are rationals, they don't contain any strengthening because a separate step logs that.

Isabelle Reconstruction. To replay linear arithmetic steps, Isabelle can use the tactic linarith. It searches the coefficients necessary to verify the lemma. As already noted in [25], Isabelle's linarith tactic is too weak to reconstruct all lemmas, because it can only find integer coefficients and fails if strengthening is required. It is however strong enough to replay `farkas-lemma` steps from Z3.

The reconstruction of a `la_generic` step starts with a lemma of the form $\bigvee_i \neg c_i$ where each c_i is either an equality or a disequality. Before starting the reconstruction, we

abstract over the non-arithmetic parts by replacing the each term in the equations by constants. The abstraction corresponds to veriT's arithmetic solver which treats such terms as simplex variables and therefore it does not limit the reconstruction. Abstracting also stops the reconstruction from looking inside terms. We encountered an example where a term of the form ((if $\top$ then 0 else 1) $<$ 0) appeared in a lemma and was prematurely simplified to $\bot$. Then we transform the lemma into the equivalent formulation $c_1 \implies \cdots \implies c_n \implies \bot$ and remove all negations (e.g., replacing $\neg a \leq b$ by $b > a$).

After that we combine the theorems. As the first step in the reconstruction, we multiply the equation by the corresponding coefficient. We use defined theorems to do this. For example, for integers, if we have $A < B$ and the rational coefficient $\frac{p}{q}$ (with $p > 0$ and $q > 0$), we strengthen the equation to get

$$(A \mathit{div}\, q) + (\text{if } B \mathit{mod}\, q = 0 \text{ then } 1 \text{ else } 0) \leq (B \mathit{div}\, q),$$

then we multiply it by the constant p:

$$p \times (A \mathit{div}\, q) + p \times (\text{if } B \mathit{mod}\, q = 0 \text{ then } 1 \text{ else } 0) \leq p \times (B \mathit{div}\, q).$$

This is the strictest possible equation. The if-then-else term (if $B \mathit{mod}\, q = 0$ then 1 else 0) corresponds to the strengthening (and therefore, there is no such term for the corresponding theorem over reals): If $B \mathit{mod}\, q = 0$ is true, the result is an equation of the form $A' + 1 \leq B'$, which is equivalent to $A' < B'$.

After that, we can combine all the equations by summing them while being careful with the equalities that can appear. Internally, to combine two equations (that either involve $\leq$, $<$, or $=$), we currently do not try to find which theorem is needed for that combination of (in)equalities, but simply try to unify all possible theorems and see which single theorem remains at the end. Only one can remain. We simplify this inequality or equation using Isabelle's simplifier. If everything was correct, this will derive $\bot$.

Coefficients are typed in Isabelle as either real or integer. The sort used by veriT is depending on the SMT-LIB theory and not on the current equation. To handle this, we use two versions of the equations to multiply by the coefficients, one with integers and one with rounding from reals to integers.

When handling the assumptions of the equations to multiply with the coefficients, we experimented with aggressive simplification and discharging all assumptions together. The former allows us to shorten terms (by, e.g., replacing the term (if $2 \mathit{mod}\, 1 = 0$ then 1 else 0) by 1), while the latter calls the simplification procedure only once. The latter turned out to be slightly faster. Therefore, we import the assumption in the local context, combine all the equations together, and finally, we discharge all assumptions and simplify the combined equation to $\bot$ at once.

We experimented on one arithmetic-heavy example stating the integer sequence of the form $x_{i+2} = |x_{i+1}| - x_i$ is periodic. The proof heavily relies on the arithmetic procedure:

$$\begin{aligned}&\big(x_3 = |x_2| - x_1 \wedge x_4 = |x_3| - x_2 \wedge x_5 = |x_4| - x_3 \wedge\\ &\ x_6 = |x_5| - x_4 \wedge x_7 = |x_6| - x_5 \wedge x_8 = |x_7| - x_6 \wedge\\ &\ x_9 = |x_8| - x_7 \wedge x_{10} = |x_9| - x_8 \wedge x_{11} = |x_{10}| - x_9\big)\\ &\implies x_1 = x_{10} \wedge x_2 = x_{11}\end{aligned}$$

The translation to the SMT-LIB format replaces the absolute values by if-then-else terms. The veriT proof involves 101 `la_generic` steps and `veriT_smt` takes 3.6 s. If instead we use the linarith tactic to reconstruct the 101 arithmetic steps (the proof involves no strengthening), the reconstruction takes 38.6 s, meaning that finding the coefficients again can be a lot slower. Z3 produces a different proof that takes 4.3 s to reconstruct. We believe this is because the number of equalities in each step is smaller and, hence, fewer variables have to be eliminated.

To improve the efficiency of other rules we added a dedicated tactics sequence instead of relying on a generic tactic whenever possible. For example the arithmetic rule `la_disequality` produces a theorem of the form $a = b \vee \neg a \leq b \vee \neg b \leq a$. The prototype from Fleury and Schurr's [25] reconstructs these steps using linarith, which is less efficient than applying the theorem.

We also connected our arithmetic reconstruction to Z3, since it also gives the coefficients. The reconstruction works in a similar way, except that the rational part of the coefficients does not indicate rounding but is simply a side effect of the generation of the coefficients. To avoid using arithmetic over rational numbers, we multiply the coefficient by the LCM of the denominator. We, however, did not observe any speed-up on the example above.

4.4 Skolemization

During our careful evaluation and attempts to speed up the reconstruction, we realized that defining choice terms can take a significant amount of time. The solution is to shorten the term size, by ensuring that veriT replaces quantified formulas by their associated choice definition, e.g., when $X_1 = (\epsilon x_1.\ x_1 = 0)$, writing $X_2 = (\epsilon x_2.\ x_2 = X_1)$ instead of $X_2 = (\epsilon x_2.\ x_2 = (\exists x_1.\ x_1 = 0))$. In the extreme example where we identified the issue, the number of characters in the definition of the skolem term without sharing went down from over 3 million to around 800. The result is a large speedup, not only when converting to Isabelle terms, but also when defining the skolem constants in veriT.

There still are some issues related to this. veriT replaces terms by constants only when they match syntactically. For example, instead of producing $x_1 = (\epsilon x_1.\ x_1 = 0)$, veriT

can produce the equivalent term $x_1 = (\epsilon x_1.\ 0 = x_1)$. This term can not be folded in x_2, because of equality reordering. In those cases, the terms are larger than necessary, but we cannot easily detect those cases in either veriT or Isabelle.

The handling of skolemization is one of the most critical parts of the code, not only because it often occurs, but also because the reconstruction is not easy. Skolemization steps have the form $(\exists x.\ P\ x) \vee P\ X$ where X is defined as $(\epsilon x.\ P\ x)$. Since veriT can implicitly apply the symmetry of equalities, we might actually get $(\epsilon x.\ P'\ x)$ where P and P' are equal up to reordering of equalities. We have experimented with various ways to reconstruct such steps.

We now rely on the fact that the *name* of the dummy variable in the quantifier is unique within an equation and identical to the one in the choice term. Assume that for example the goal is $(\exists x_1\ x_2.\ f\ x_1\ x_2 = 0) \iff f\ X_1\ X_2 = 0$. We extract the names from the equations (x_1, x_2) and find the corresponding choice terms ($X_1 = (\epsilon x_1.\ \exists x_2.0 = f\ x_1\ x_2)$ and $X_2 = (\epsilon x_2.\ f\ X_1\ x_2 = 0)$). This also gives the order in which we have to replace the terms (first X_1, then X_2). Then we can fold the choice term in the term and discharge the proof obligation (first $f\ x_1\ x_2 = 0 \iff 0 = f\ x_1\ x_2$, then $f\ X_1\ x_2 = 0 \iff 0 = f\ X_1\ x_2$), *without* exploring the terms under the $\exists$ symbol. The replacement can be done at several locations because not all choice terms have been folded into each others definition.

Initially we tried to add this process as a congruence rule or simplification rule to Isabelle, but the simplifier did not perform higher-order unification efficiently (unifying with the wrong equations and only exploring that branch).

In the case where everything is ordered correctly, this is slower than the previous solution, because inspecting terms and reordering equations is not necessary. Therefore, we first assume that no reordering occurred. If the goal is not solved, we extract the names.

The reconstruction the proof is not the way the proof was intended to be checked. veriT provides a more detailed proof. If we have $\exists x\ y.\ P\ x\ y$, we get a proof that $(\exists y.\ P\ x\ y) \iff (P\ x\ Y)$ for any x. However, this does not solve the problem of detecting reorderings. Hence, we currently only check that the proof is correct but discard the outcome and reconstruct directly the full equation as explained above.

4.5 Selective Decoding of the First-order Encoding

One example of a reconstruction issue on the Isabelle side that shows the interplay of the higher-order encoding is the rule `eq_congruent_pred`. This rule expresses congruence on a predicate: $\neg(t_1 = u_1) \vee \cdots \vee \neg(t_n = u_n) \vee P(t_1, \ldots, t_n) = P(u_1, \ldots, u_n)$. For example, veriT can produce the theorem $f \neq f' \vee x \neq x' \vee \text{app}(f, x) \leftrightarrow \text{app}(f', x')$, where app is a function introduced by the first-order encoding which encodes function application. The idea of the reconstruction is to assume that $f = f'$, then to identify the arguments, and finally to rewrite them in the left side to produce $\text{app}(f, x) \leftrightarrow \text{app}(f', x)$. However, f and f' can be a construct which represents a lambda function like $f := \lambda x.\ x = a$ and $f' := \lambda x.\ a = x$. Isabelle generates such constructs if the goal contains a higher-order function like map.

We realized that unfolding all constructs which encode higher-order terms makes the reconstruction harder, because we change the term structure. If we aggressively unfold the encoding, as terms are directly β-normalized by Isabelle, we have to prove $(x = a) \leftrightarrow (a = x')$, where we cannot identify the arguments f and f' anymore. The reconstruction from [25] has a special case to detect lambda function, but the code is very involved and not well tested. Therefore, we no longer unfold the encoding and instead verify that $\text{app}(f, x) \leftrightarrow \text{app}(f', x)$ holds and only then unfold everything to get $(x = a) \leftrightarrow (a = y') \vee (\lambda x.\ x = a) = (\lambda x.\ a = x') \vee x = x'$. This minor change simplifies the reconstruction of `eq_congruent_pred` steps by avoiding special cases, makes the code easier to test, and makes the reconstruction tactic easier to implement and maintain.

5 Evaluation

During development we routinely tested our proof reconstruction to find bugs. As a side effect, we produced SMT-LIB files corresponding to the calls. We measured the performance of veriT with various options on them and selected five different strategies (Sect. 5.1). After that we evaluated the performance of Sledgehammer (Sect. 5.2).

All experiments have been performed on a computer with two Intel Xeon Gold 6130 CPUs with a total of 32 cores and 192 GB of RAM.

5.1 Strategies

We tested the reconstruction by using Isabelle's *mirabelle* tool. It takes theories and automatically runs Sledgehammer [12] on all proof steps. Sledgehammer selects facts from the background theory and calls various automatic provers (we used the SMT solvers CVC4, veriT, and Z3 and the superposition prover E [34]) to *filter* facts and then attempts to find a tactic that can prove the goal from the filtered facts. It will pick the fastest trusted tactic. We then called mirabelle on some Isabelle files from HOL-Library (an extended standard library containing various developments), a formalization of Green's theorem [1, 2], of the Prime Number Theorem [23], and of the KBO ordering [11].

The outcome are both a series of SMT files typical for the kinds of problems Isabelle users want to solve and a series of calls to `veriT_smt`. For failing `veriT_smt` (smt) calls three cases are possible: veriT (Z3) does not find a proof, the reconstruction works but takes more time than allocated, or it fails. In the last case for veriT, we fixed the reconstruction.

Table 1. Options corresponding to the different veriT strategies

Name	Options
default	(no option)
del_insts	--index-sorts --index-fresh-sorts --ccfv-breadth --inst-deletion --index-SAT-triggers --inst-deletion-loops --inst-deletion-track-var
ccfv_SIG	--triggers-new --index-SIG --triggers-sel-rm-specific
ccfv_insts	--triggers-new --index-sorts --index-fresh-sorts --triggers-sel-rm-specific --triggers-restrict-combine --inst-deletion --inst-deletion-loops --inst-deletion-track-vars --index-SAT-triggers --inst-sorts-threshold=100000 --ematch-exp=10000000 --ccfv-index=100000 --ccfv-index-full=1000
best	--triggers-new --index-sorts --index-fresh-sorts --triggers-sel-rm-specific

veriT exposes a wide range of options to fine-tune the proof search. In the SMT competition veriT utilizes a scheduler which runs the solver with different, hand-crafted, combinations of options (*strategies*) on short timeouts. To find strategies which work well on typical Sledgehammer problems we determined which benchmarks were solved by each strategy within a two second timeout. We then chose the strategy which solved the most benchmarks and three strategies which together solved the most benchmarks. For comparison, we also kept the default strategy.

The strategies are shown in Tab. 1 and mostly differ in the instantiation schemes. The strategy del_insts uses instance deletion as described by Barbosa [5] and uses a breadth-first algorithm to find conflicting instances. All other strategies utilize an implementation of triggers which incorporates Leino and Pit-Claudel's [28] trigger inference techniques. The strategy ccfv_SIG uses a different indexing method for instantiation. It also restricts enumerative instantiation [32], because the options "--index-sorts --index-fresh-sorts" are not used. The strategy ccfv_insts increases some thresholds. Finally, the strategy best utilizes a subset of the options used by the other strategies.

We have also considered using a scheduler as used in the competition in Isabelle. The advantage of this approach is that we do not need to select the strategy on the Isabelle side. However, this approach would make the tactic unreliable. A problem solved by only one strategy just before the end of the allocated time slice could become unprovable on slower hardware. Issues with z3_smt timeouts have been reported on the Isabelle mailing list, for example due to an antivirus delaying the startup [27]. Such a scheduler would make debugging even more complicated.

5.2 Sledgehammer Experiments

To measure the performance of the `veriT_smt` tactic we again used mirabelle. We extend Sledgehammer (using the solvers E, CVC4, veriT, and Z3) to try all the strategies including the default one and instrumented it to print the time of all tried tactics. This is similar to the Judgment Day experiments [15]. We used the full HOL-Library and the Archive of Formal Proofs theories Prime Distribution Elementary (a formalization of various properties of the distribution of prime numbers, abbreviated PDE) [22], a formalization of an executable resolution prover (abbreviated RP) [33], and a formalization of the Simplex algorithm used within SMT solvers [29].

Tab. 2 gathers the results. The upper part shows which smt tactic Sledgehammer selected. veriT_smt is selected roughly 30% less on all theories, but Simplex. Looking at the raw data for Simplex, we noticed that veriT finds proofs, but Z3 is much faster. If we look at the specific strategies, the *best* strategy outperforms the other strategies, but none of other strategy outperforms the remaining. This is not surprising, since we selected the strategies to be complementary. Overall, we can see that Sledgehammer suggests veriT_smt very often in proofs generated, but not as often as z3_smt.

The bottom part of Tab. 2 shows the number of problems uniquely solved by each SMT tactic, even if solved by one of Isabelle built-in tactic. They are generally less powerful but faster than the SMT-solver based ones because of less overhead. veriT solves more problems than Z3 on all of the theories. This effect is the least pronounce on Simplex. When looking at the detailed results per solver, most uniquely solved problems by veriT from PDE, RP, and Simplex proved by veriT itself as a backend for Sledgehammer. Without the problems solved by veriT, the solver Z3 would solve slightly more unique problems. The opposite effect is less strong for Z3.

Even if veriT solves more problems, it seems that it solves the "wrong" problems, in the sense that they are also solved by a built-in tactic which does not have the overhead of translating the problem. We did not take that effect into account when generating the problems for the strategies. This can explain the difference of usefulness between veriT and Z3: solving more "simple" problems also solved by internal tactics does not translate into more `veriT_smt` calls.

When running Sledgehammer, an important information is the number of reconstructed goals and the number of failures (Tab. 3). There are three possible outcomes: Sledgehammer (i) finds a one-liner, (ii) detects that all one-liners timeout, but finds a proof with intermediate steps, and (iii) finds no working proof (either an error or a timeout: the

Table 2. Tactic used to reconstruct proofs produced by Mirabelle

Theory		HOL-Library	PDE	RP	Simplex
`z3_smt`		1756	77	59	81
`veriT_smt` (all)		1240	102	40	18
	strategy default	236	13	8	1
	strategy del_insts	226	28	9	5
	strategy ccfv_threshold	228	9	5	5
	strategy ccfv_SIG	219	26	11	4
	strategy best	236	26	7	3
veriT	unique	7575	1389	1894	3262
Z3	unique	6978	441	653	2168

Table 3. Outcome of Sledgehammer calls showing when the reconstruction finds a single tactic to replay the proof (one-line), needs intermediate steps (Isar proof), or fails (oracle)

Prover	HOL-Library			Simplex			RP			PDE		
	One-line	Isar	Oracle	One-line	Isar	Oracle	One-line	Isar	Oracle	One-line	Isar	Oracle
CVC4	6956	+ 0	+ 69	582	+ 0	+ 6	1100	+ 0	+ 7	460	+ 0	+ 3
E	6575	+ 46	+ 40	1203	+ 13	+ 7	2061	+ 9	+ 10	1047	+ 10	+ 5
veriT	6308	+ 8	+ 26	454	+ 2	+ 1	1000	+ 0	+ 5	509	+ 2	+ 3
Z3	6442	+ 24	+ 22	973	+ 4	+ 4	2058	+ 2	+ 10	928	+ 2	+ 4

solver is an oracle). In the cases of veriT and Z3, if the reconstruction is fast and perfect, `veriT_smt` or `z3_smt` can replay every proof as a one-liner and there would be no failure. As we can see, here, veriT and Z3 have a similar failure rate, showing that our reconstruction is reliable.

We can confirm the observation from Blanchette et al.'s [12] that producing Isar proofs improves the efficiency of the Sledgehammer. While CVC4 cannot produce Isar proofs (because it does not produce a detailed proof), generation of Isar proof works better for Z3 than for veriT. For veriT's older proof format, Isar reconstruction [12] was supported, but some specific handling of the skolemization is missing now. However, the transformation to Isar proofs still works sometimes, as seen in Tab. 3.

Overall, we can see that veriT is not only a useful backend for Sledgehammer, but also that the reconstruction works as well as the one for Z3.

6 Further Improving the Reconstruction

The experiments performed for the previous sections allow us to determine the time needed to replay each rule to show which rules are critical for the reconstruction (Sect. 6.1). We explain some of the problems that affect the reconstruction and can cause issues (Sect. 6.2). In order to speed up the reconstruction, one solution is to skip the checking of some steps at the possible expense of missing errors in the certificate. This improves the performance of the reconstruction (Sect. 6.3).

6.1 Performance

To better understand what the key rules of our reconstruction are, we produced statistics on the time used to reconstruct each rule. Fig. 1 shows the distribution of the time spent on some rules.[1] We removed the slowest and fastest 5% of the applications, because garbage collection can trigger at any moment and even a trivial rule can take a long time. Fig. 2 on the other hand gives the total amount of time used to reconstruct the rules over all proofs. We have included `parsing` that is the time required to parse and convert the veriT into Isabelle terms, before reconstruction.

Overall, there are three kinds of rules: direct application of a sequence of theorems (e.g., `equiv_pos2` corresponds to theorems of the form $\neg(a \leftrightarrow b) \vee \neg a \vee b$), application of a sequence of direct theorems and calls to the simplifier (e.g., `sko_forall` involves a call to the simplifier to reorder terms), and calls to full-blown tactics (like `qnt_simplify`).

If we look at direct application of theorems, the reconstruction is usually fast, but they occur so often that the cumulative time is significant. The most extreme example is the `or` rule. veriT distinguishes between clauses and disjunctions and uses `or` steps to deconstruct a disjunction into a clause. Isabelle uses only clauses and the rule translates to returning the singular element of the premise. However, in our tests it used 1.1% of the time, more than the complicated `la_generic` rule overall.

[1] Note to reviewers, Figs. 5 and 6 of the appendix show the graph with all rules.

Another example is cong. Replaying consists of unfolding some equations and applying reflexivity of equality. When investigating why the reconstruction is sometimes taking more than 10 ms, we noticed that the theorems to certify can be very large, in particular much larger than theorems written by a human user. This slows down reconstruction and it is not clear how to improve those aspects further.

When we have to call the simplifier (sko_forall), the reconstruction needs more time, but such rules are less frequent and are less critical for efficient reconstruction.

Finally, the slowest rules are the ones calling full-blown tactics like qnt_simplify, ite_intro, and tmp_bfun_elim. We have not yet tried to write specialize tactics and instead rely on Isabelle's blast tactic, based on the tableau method. The rule la_generic seems slow on average, but searching the coefficients to replay the proof takes even more time as shown in Sect. 4.3.

Overall, we can also see that the time required to check the simplification steps that were formerly combined as a single connect_equiv rule is not significant anymore.

It is also interesting to note that some rules did not appear at all in our tests on real data. The biggest surprise to us is lia_generic. It produces a lemma of equations over integers specifically where no witness is guaranteed to exist. Fruthermore, no tautological clauses with complementary literals are produced, although we could create artificial examples which produced such steps. The rules unary_minus_simplify, qnt_join, and la_tautology normalize terms by simplifying unary minuses, combining quantifiers, and handling inequalities where the variables can be eliminated. Their absence shows that terms written in Isabelle are already normalized.

We have performed the same experiments with the reconstruction of the SMT solver Z3. Unlike for veriT, we do not have the amount of time required for parsing. The results are shown in Figs. 3 and 4. The rule distribution is very different. The nnf-neg and nnf-pos rules are the slowest rules and take a huge amount of time in the worst case. However, the quantifier instantiation that requires fewer reconstruction steps is *faster* than the one produced by veriT. We suspect that reconstruction is faster because the steps are easier to check. The step is only an implication, and no choice term is involved.

Böhme [14, Sect. 3.4] performed a similar study for version 2.15 of the SMT solver Z3. Currently, version 4.4.0 ships with Isabelle and is used by us. Since version 2.15, the proof format changed slighty (e.g., th-lemma-arith was introduced), fulfilling some of the wishes expressed by Böhme and Weber [16] to simplify reconstruction. Surprisingly, the nnf rules do not appear among the five rules that used the most runtime. Instead, the th-lemma and rewrite were the slowest rule. Similarly to veriT, the cong rule was among the most used (without appearing in the rules accounting for the most time), but it does not appear in our tests.

6.2 Overall Issues

Equality Ordering. One recurring challenge is the implicit reordering of equalities. It has a major impact on the reconstruction of skolemization steps and instantiation. Unfortunately, solving this problem is difficult. We implemented a prototypical version of veriT which normalized terms as an explicit, proof producing preprocessing step. Unfortunately, this resulted in significant performance regressions. Since the normalization was a separate step, the invariant would not hold at all times. This affected especially the preprocessing steps in subtle ways. We believe a more sustainable solution would be a post-processing of the proof which adds the missing normalization steps.

Arithmetic. The implicit reordering of equalities is not a problem for la_generic steps. Whenever an equality $t_1 = t_2$ is added to the arithmetic solver, this equality is always translated into the constraint $t_1 - t_2 = 0$. This makes the reconstruction of arithmetic steps easier and avoid backtracking to guess the direction of equalities. Overall, if we compare the amount of time spent on the reconstruction of arithmetic theorems, the Z3 reconstruction is faster. We believe that this due to a difference in the scheduling the arithmetic theory within the SMT solver, leading to simpler lemmas.

Another issue is the lia_generic rule. This rule is emitted by the branch-and-bound procedure. Similar to the la_generic rule, it produces a conjunction of inequalities and equalities that is unsatisfiable in the theory of integers specifically. Unfortunately, reconstructing it is still an issue, because no detailed information is provided. We currently use the same reconstruction method as for Z3 which can fail even on simple equations like $3 \times p \neq 1$. Such simple equations, however, did not appear a single time in our tests on practical theories. Our tests also show that the lia_generic rule does not occur often in our proofs.

6.3 Skipping Steps

A major difference between the proofs generated by Z3 and veriT is the number of steps. The latter produces more detailed proofs, but this can be detrimental to performance. The or rule only return its premise, but accounts for more than 1% of the overall runtime. A more involved example is skolemization: It is *one* step in Z3 and three in veriT—first $\exists x.\ P\ x$, then $\neg(\exists x.\ P\ x) \vee P\ (\epsilon x.\ P\ x)$, and finally $P\ (\epsilon x.\ P\ x)$ (by resolution on the two previous theorems).

One solution to speed up the proof compression is to avoid replaying some rule. There are two kind of steps that the reconstruction can skip. The first is the or rule where we simply replace every usage by its premise. The other rules we can skip come from renaming. veriT considers $\forall x.\ P\ x$ and $\forall y.\ P\ y$ to be two different terms and produces a detailed proof of the conversion. Isabelle, on the other hand, uses de Brujn indices and variable names are not relevant. Hence, we replace bind rules of the form $(\forall x.\ P\ x) \iff (\forall y.\ P\ y)$ by

Table 4. Reconstruction of Sledgehammer backed by Z3 with and without step skipping for veriT

		No step skipping			Step skipping		
Theory		PDE	RP	Simplex	PDE	RP	Simplex
z3_smt		31	15	22	18	8	35
veriT_smt (all)		29	10	2	41	16	3
	strategy default	1	2	0	11	4	0
	strategy del_insts	7	2	0	15	4	1
	strategy ccfv_threshold	4	1	1	3	1	1
	strategy ccfv_SIG	9	4	1	3	1	1
	strategy best	8	1	0	9	6	0

a single application of reflexivity. As veriT initially renames all variables to canonical names starting `veriT_vr`, we skip many such proof steps.

We have done some of the experiments from Sect. 5.2 with step skipping (Tab. 4). We used Z3 as the only backend solver for Sledgehammer. As Sledgehammer is *not deterministic*, no direct comparison is possible. The overall number of veriT and Z3 calls is different. However, the ratio of calls is interesting: for Simplex, step skipping does not help much, but for PDE and RP, Sledgehammer generates more `veriT_smt` calls than z3_smt calls by a factor of two. Hence, except for Simplex, `veriT_smt` outperforms z3_smt on Z3-found problems: finding and replaying the proof is faster for `veriT_smt`, which shows how efficient our reconstruction stack is and how useful our strategies are.

Step skipping has the drawback (for veriT developers, not for Isabelle users) that the reconstruction does not check every step. So an inconsistency could remain undiscovered, but it is unlikely with the steps we are skipping.

7 Related Work

The state-of-the-art SMT solvers CVC4 [8], Z3 [19], and veriT [17] produce proofs. CVC4's output does not record quantifier reasoning, whereas Z3 does not always produce fine-grained steps. Proofs generated by SMT solvers have also been used to generate unsatisfiability cores [20], and interpolants [31]. They are also useful as a debugging instruments for the automatic prover itself, since unsound steps often point to the origin of bugs. Our work also relates to systems like Dedukti [4] that focuses on translating the steps of the proofs, not on replaying detailed proofs.

Reconstruction of proofs generated by external theorem provers has been implemented in various systems, including CVC4 proofs in HOL Light [30], Z3 in HOL4 and Isabelle/HOL [16], and veriT [3] and CVC4 [24] in Coq. Only veriT produces detailed proofs for both preprocessing and skolemization. For veriT proofs, SMTCoq currently supports an old version of the proof output (version 1) that has different rules and is implemented in an older version of veriT (from 2016), which does not record detailed information for skolemization and has worse performance. SMTCoq also supports bitvectors and arrays.

The reconstruction of Z3 proofs in the proof assistants HOL4 and Isabelle/HOL is one of the most advanced and well tested. It is regularly used by Isabelle users. Böhme and Weber [16]'s report performance numbers for the reconstruction of proofs of preexisting SMT-LIB files, such as benchmarks from the SMT-LIB library. The study was performed on problems generated by a proof assistant [14] and from the SMT competition. This included some Isabelle problems. Unfortunately, the code to read the SMT-LIB problems was never included in the standard Isabelle distribution and is now lost.[2] The Z3 proof reconstruction procedure has been heavily tested. It succeeds in more than 90% of Sledgehammer benchmarks [12, Section 9] and is efficient. An older version of Z3 was used.

The SMT solver CVC4 follows a different philosophy from veriT and Z3: it produces proofs in a logical framework with side conditions [35]. The output can contain programs to check certain rules. The proof format is flexible, but currently CVC4 does not generate proofs for quantifiers.

Our approach for the normalization was very different in our prototype. For duplicates, we relied on the simplifier to remove them. We had not identified the issue with simplifications to $\top$, because they do not happen very often. To efficiently reconstruction `connect_equiv` that combines several rules, our prototype from [25] first attempt to reconstruct the steps using propositional logic only and then used Isabelle's auto tactic, because it also supports arithmetic and first-order transformation to some degree. The new detailed rules avoid such strange constructs.

8 Conclusion

We presented a performant and reliable reconstruction of proofs generated by as state-of-the-art SMT solver in an interactive theorem prover. Our improvements address reconstruction challenges for proof steps of typical inferences performed by SMT solvers.

[2] via private communication with the authors

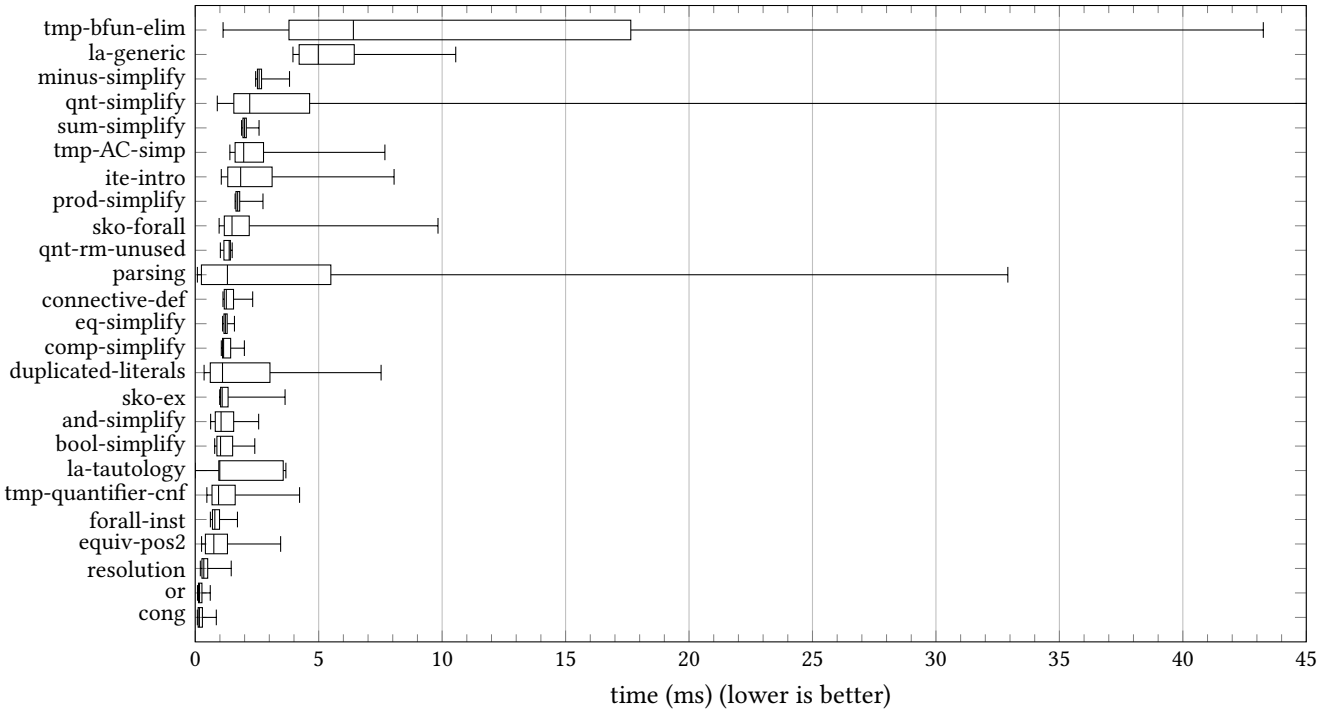

Figure 1. Timing of some of veriT's rules sorted by median. From left to right, the lower whisker marks the 5th percentile, the lower box line the first quartile, the middle of the box the median, the upper box line the third quartile, and the upper whisker the 95th percentile. qnt_simplify's 95th percentile is 240 ms.

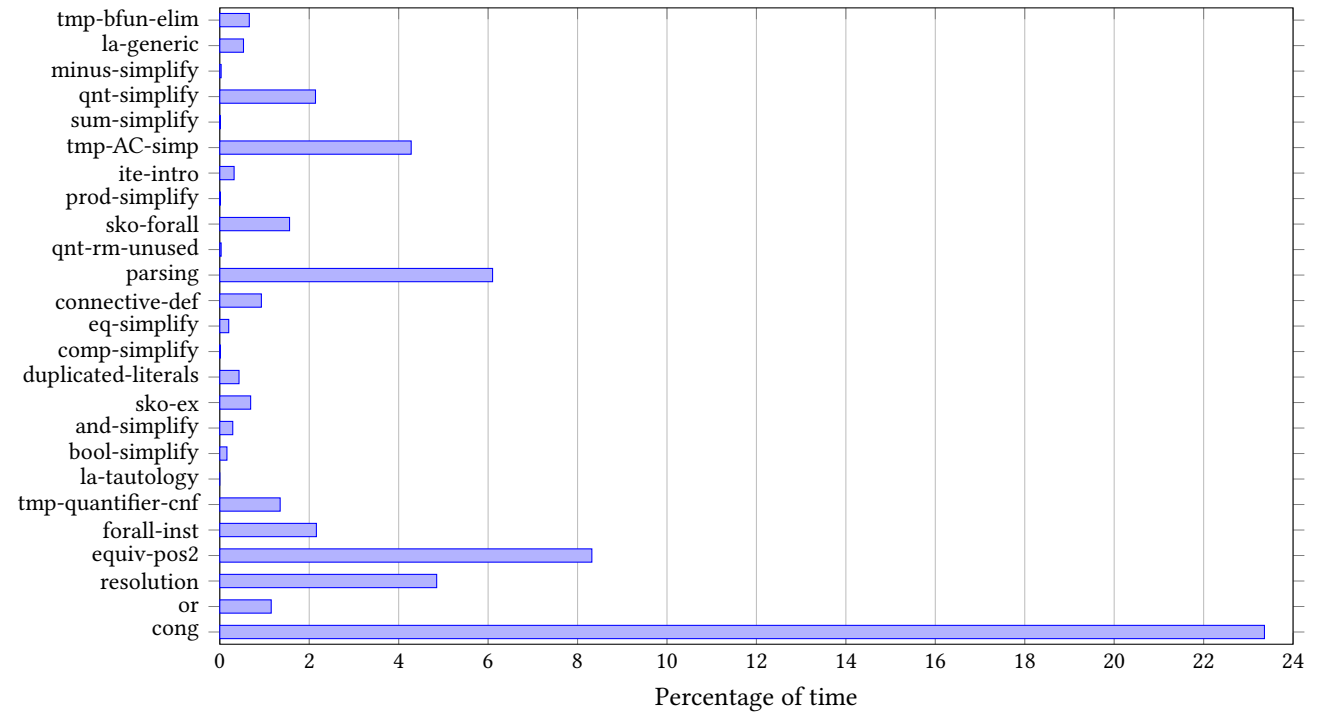

Figure 2. Total percentage spent on each rule for the SMT solver veriT in the same order as Fig. 1.

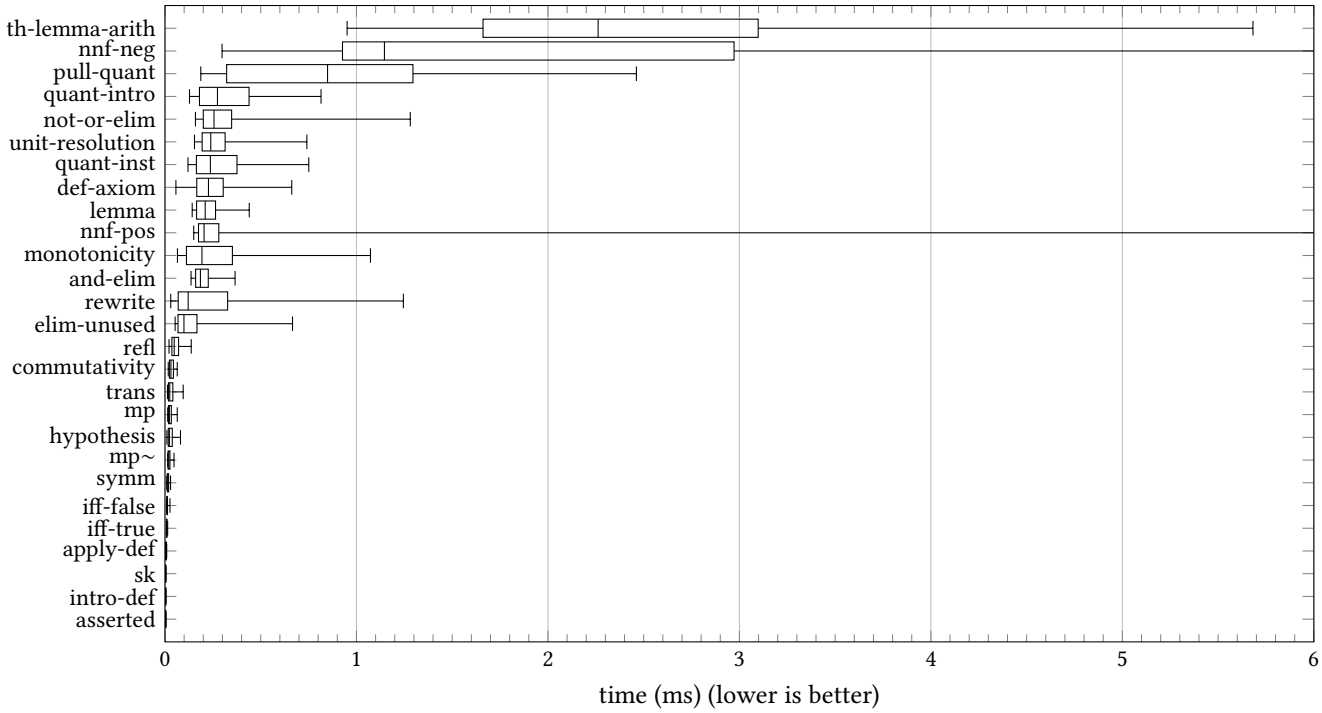

Figure 3. Timing of some of Z3's rules sorted by median. From left to right, the lower whisker marks the 5th percentile, the lower box line the first quartile, the middle of the box the median, the upper box line the third quartile, and the upper whisker the 95th percentile. nnf-neg's 95th percentile is 36 ms. nnf-pos's 95th percentile is 75 ms.

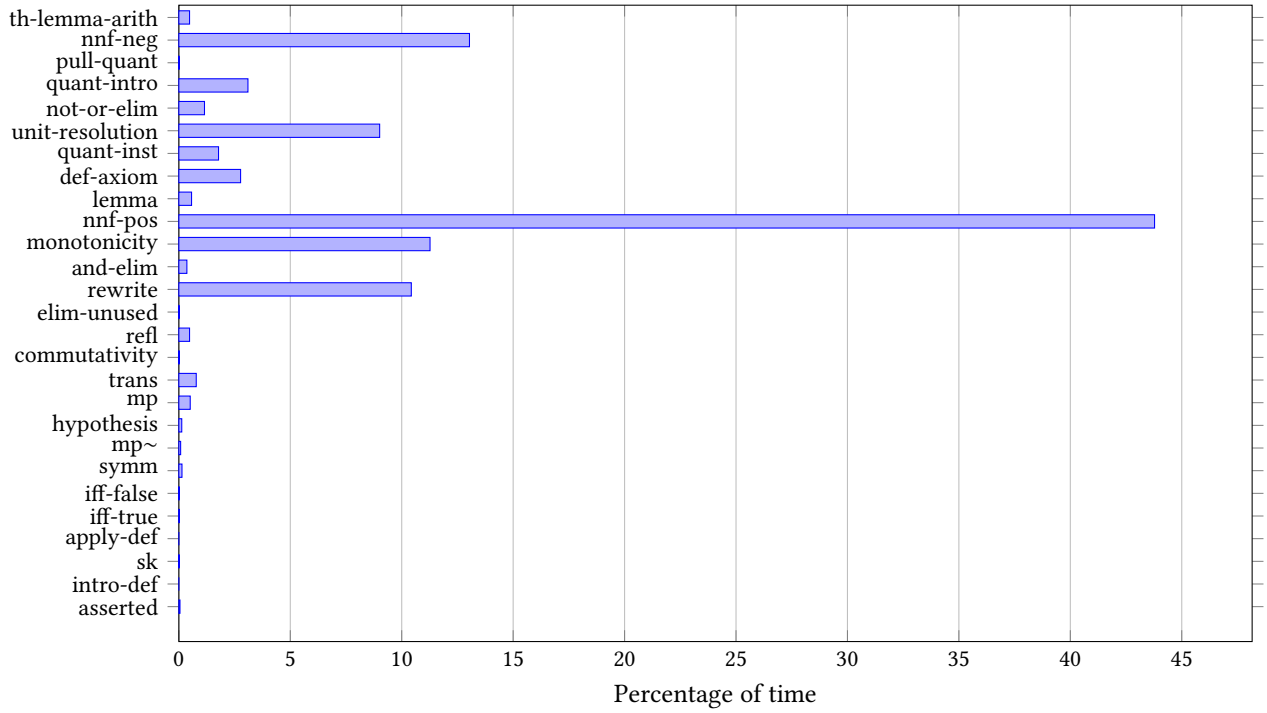

Figure 4. Total amount of time per rule for the SMT solver Z3.

By studying the time required to replay each rule, we were able to compare the reconstruction for two different proof formats with different design directions. The very detailed proof format of veriT makes the reconstruction easier and allows for more specialization of the tactics. Furthermore, it presents the opportunity to skip irrelevant steps.

We plan to improve the reconstruction of the slowest rules. We also want to investigate if replaying several proof step at once can improve performance (e.g., to skip some `equiv_pos2` steps). The developers of the SMT solver CVC4 are currently rewriting the proof generation (e.g., they now track the reordering of equalities) and plan to support a similar proof format. We hope to be able to reuse the current reconstruction (likely with additional rules). Generating and reconstructing proofs from Barbosa et al.'s [7] veriT version with higher order logic could also improve the usefulness of veriT on Isabelle problems. The current proof rule should be sufficient to express proofs in the more expressive logic.

Acknowledgments

We would like to thank Haniel Barbosa for his support with the implementation in veriT. We also thank Haniel Barbosa, Jasmin Blanchette, and Pascal Fontaine for many fruitful discussions and suggesting many textual improvements. The first author has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). The second author is supported by the LIT AI Lab funded by the State of Upper Austria. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Mohammad Abdulaziz and Lawrence C. Paulson. 2018. An Isabelle/HOL formalisation of Green's Theorem. *Archive of Formal Proofs* (Jan. 2018). http://isa-afp.org/entries/Green.html, Formal proof development.

[2] Mohammad Abdulaziz and Lawrence C. Paulson. 2019. An Isabelle/HOL Formalisation of Green's Theorem. *Journal of Automated Reasoning* 63, 3 (Nov. 2019), 763–786. https://doi.org/10.1007/s10817-018-9495-z

[3] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP 2011 (LNCS, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer Berlin Heidelberg, 135–150. https://doi.org/10.1007/978-3-642-25379-9_12

[4] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. 2016. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *TYPES: Types for Proofs and Programs*. Novi SAd, Serbia.

[5] Haniel Barbosa. 2016. Efficient Instantiation Techniques in SMT (Work In Progress) *(CEUR Workshop Proceedings, Vol. 1635)*, Pascal Fontaine, Stephan Schulz, and Josef Urban (Eds.). CEUR-WS.org, 1–10. http://ceur-ws.org/Vol-1635/#paper-01

[6] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury, and Pascal Fontaine. 2019. Scalable Fine-Grained Proofs for Formula Processing. *Journal of Automated Reasoning* (Jan. 2019). https://doi.org/10.1007/s10817-018-09502-y

[7] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark W. Barrett. 2019. Extending SMT Solvers to Higher-Order Logic. In *CADE 27 (LNCS, Vol. 11716)*, Pascal Fontaine (Ed.). Springer International Publishing, 35–54. https://doi.org/10.1007/978-3-030-29436-6_3

[8] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV 2011 (LNCS, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[10] Clark W. Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[11] Heiko Becker, Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. 2016. Formalization of Knuth–Bendix Orders for Lambda-Free Higher-Order Terms. *Archive of Formal Proofs* (Nov. 2016). http://isa-afp.org/entries/Lambda_Free_KBOs.html, Formal proof development.

[12] Jasmin C. Blanchette, Sascha Böhme, Mathias Fleury, Steffen J. Smolka, and Albert Steckermeier. 2016. Semi-intelligible Isar Proofs from Machine-Generated Proofs. *Journal of Automated Reasoning* 56, 2 (2016), 155–200. https://doi.org/10.1007/s10817-015-9335-3

[13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2011. Extending Sledgehammer with SMT Solvers. In *CADE 23 (LNCS, Vol. 6803)*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer Berlin Heidelberg, 116–130. https://doi.org/10.1007/978-3-642-22438-6_11

[14] Sascha Böhme. 2012. *Proving Theorems of Higher-Order Logic with SMT Solvers.* Ph.D. Dissertation. Technische Universität München. http://mediatum.ub.tum.de/node?id=1084525

[15] Sascha Böhme and Tobias Nipkow. 2010. Sledgehammer: Judgement Day. In *IJCAR 2010*, Jürgen Giesl and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, 107–121. https://doi.org/10.1007/978-3-642-14203-1_9

[16] Sascha Böhme and Tjark Weber. 2010. Fast LCF-Style Proof Reconstruction for Z3. In *ITP 2010 (LNCS, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, 179–194. https://doi.org/10.1007/978-3-642-14052-5_14

[17] Thomas Bouton, Diego C. B. de Oliveira, David Déharbe, and Pascal Fontaine. 2009. veriT: An Open, Trustable and Efficient SMT-solver. In *CADE 22 (LNCS, Vol. 5663)*, Renate A. Schmidt (Ed.). Springer Berlin Heidelberg, 151–156. https://doi.org/10.1007/978-3-642-02959-2_12

[18] Leonardo de Moura and Nikolaj Bjørner. 2008. Proofs and Refutations, and Z3. In *LPAR 2008 (CEUR Workshop Proceedings, Vol. 418)*, Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-418/#paper-10

[19] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008 (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[20] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. 2012. SMT solvers for Rodin. In *ABZ 2012 (LNCS, Vol. 7316)*, John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene (Eds.). Springer Berlin Heidelberg, 194–207. https://doi.org/10.1007/978-3-642-30885-7_14

[21] Bruno Dutertre and Leonardo de Moura. 2006. *Integrating Simplex with DPLL(T).* Technical Report. SRI International. http://www.csl.sri.com/users/bruno/publis/sri-csl-06-01.pdf

[22] Manuel Eberl. 2019. Elementary Facts About the Distribution of Primes. *Archive of Formal Proofs* (Feb. 2019). http://isa-afp.org/entries/Prime_Distribution_Elementary.html, Formal proof development.

[23] Manuel Eberl and Lawrence C. Paulson. 2018. The Prime Number Theorem. *Archive of Formal Proofs* (Sept. 2018). http://isa-afp.org/entries/Prime_Number_Theorem.html, Formal proof development.

[24] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *CAV 2017 (LNCS, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer International Publishing, 126–133. https://doi.org/10.1007/978-3-319-63390-9_7

[25] Mathias Fleury and Hans-Jörg Schurr. 2019. Reconstructing veriT Proofs in Isabelle/HOL. In *PxTP 2019 (EPTCS, Vol. 301)*, Giselle Reis and Haniel Barbosa (Eds.). 36–50. https://doi.org/10.4204/EPTCS.301.6

[26] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF: A Mechanised Logic of Computation.* LNCS, Vol. 78. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-09724-4

[27] Fabian Immler. 2019. Re: [isabelle] Isabelle2019-RC2 sporadic smt failures. Email. https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2019-May/msg00130.html

[28] K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *CAV 2016 (LNCS, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, 361–381. https://doi.org/10.1007/978-3-319-41528-4_20

[29] Filip Marić, Mirko Spasić, and René Thiemann. 2018. An Incremental Simplex Algorithm with Unsatisfiable Core Generation. *Archive of Formal Proofs* (Aug. 2018). http://isa-afp.org/entries/Simplex.html, Formal proof development.

[30] Sean McLaughlin, Clark Barrett, and Yeting Ge. 2006. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electronic Notes in Theoretical Computer Science* 144, 2 (2006), 43–51. https://doi.org/10.1016/j.entcs.2005.12.005

[31] Kenneth L. McMillan. 2011. Interpolants from Z3 proofs. In *FMCAD 2011*. FMCAD Inc, Austin, Texas, 19–27.

[32] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. 2018. Revisiting Enumerative Instantiation. In *TACAS 2018 (LNCS, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, 112–131. https://doi.org/10.1007/978-3-319-89963-3_7

[33] Anders Schlichtkrull, Jasmin C. Blanchette, Dmitriy Traytel, and Uwe Waldmann. 2018. Formalization of Bachmair and Ganzinger's Ordered Resolution Prover. *Archive of Formal Proofs* (Jan. 2018). http://isa-afp.org/entries/Ordered_Resolution_Prover.html, Formal proof development.

[34] Stephan Schulz. 2002. E - a brainiac theorem prover. *AI Communications* 15, 2-3 (2002), 111–126. http://content.iospress.com/articles/ai-communications/aic260

[35] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT Proof Checking Using a Logical Framework. *Formal Methods in System Design* 42, 1 (Feb. 2013), 91–118. https://doi.org/10.1007/s10703-012-0163-3

[36] The veriT Team and Contributors. 2020. Proofonomicon: A reference of the veriT proof format. Software Documentation. https://verit.loria.fr/documentation/proofonomicon.pdf Last Accessed: September 2020.

9 Appendix

Figure 5 ist the complete version of Figure 1. Figure 6 is the complete version of Figure 2.

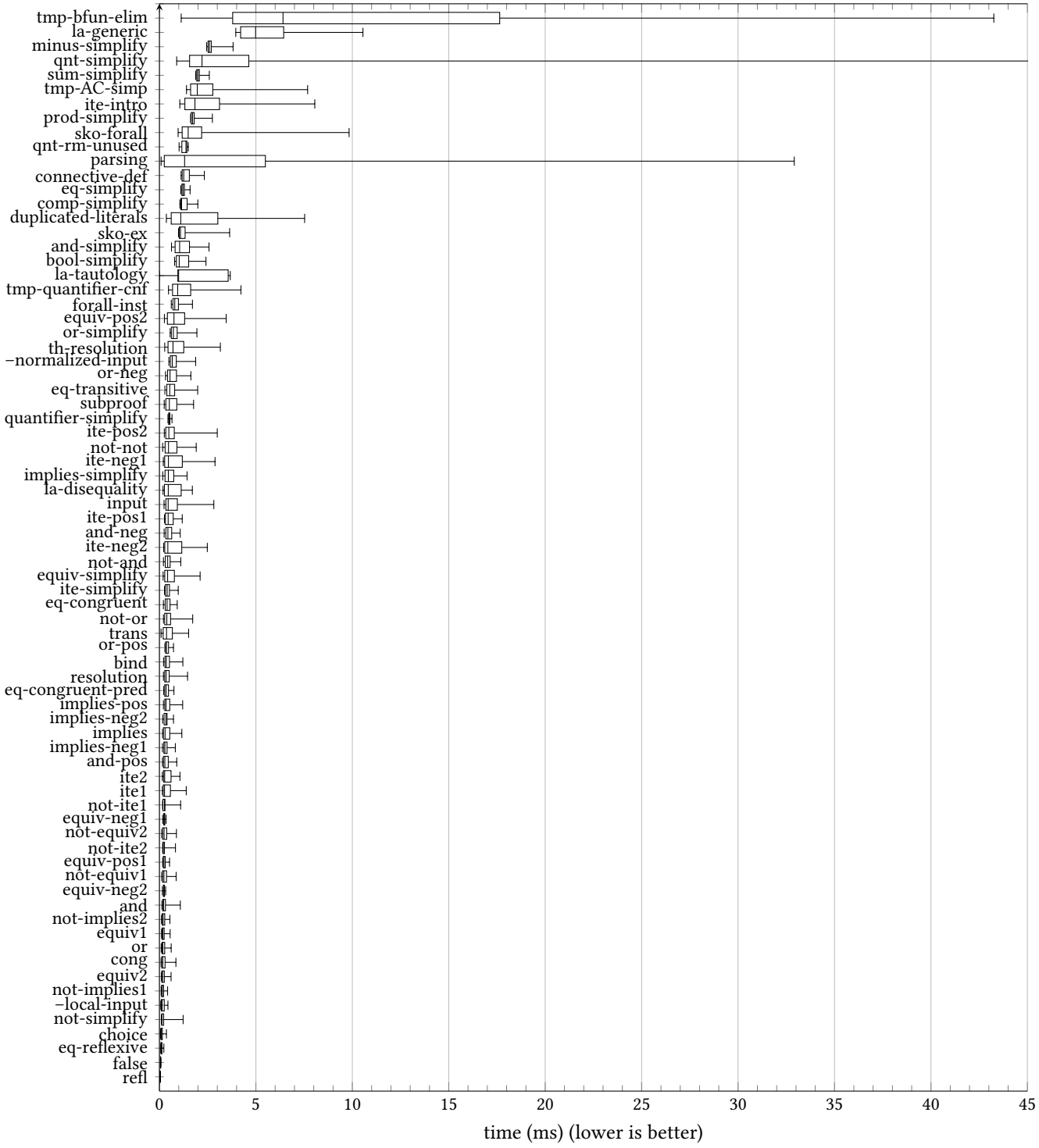

Figure 5. Timing of veriT rules sorted by median. From left to right, the lower whisker marks the 5th percentile, the lower box line the first quartile, the middle of the box the median, the upper box line the third quartile, and the upper whisker the 95th percentile. qnt_simplify's 95th percentile is 240 ms.

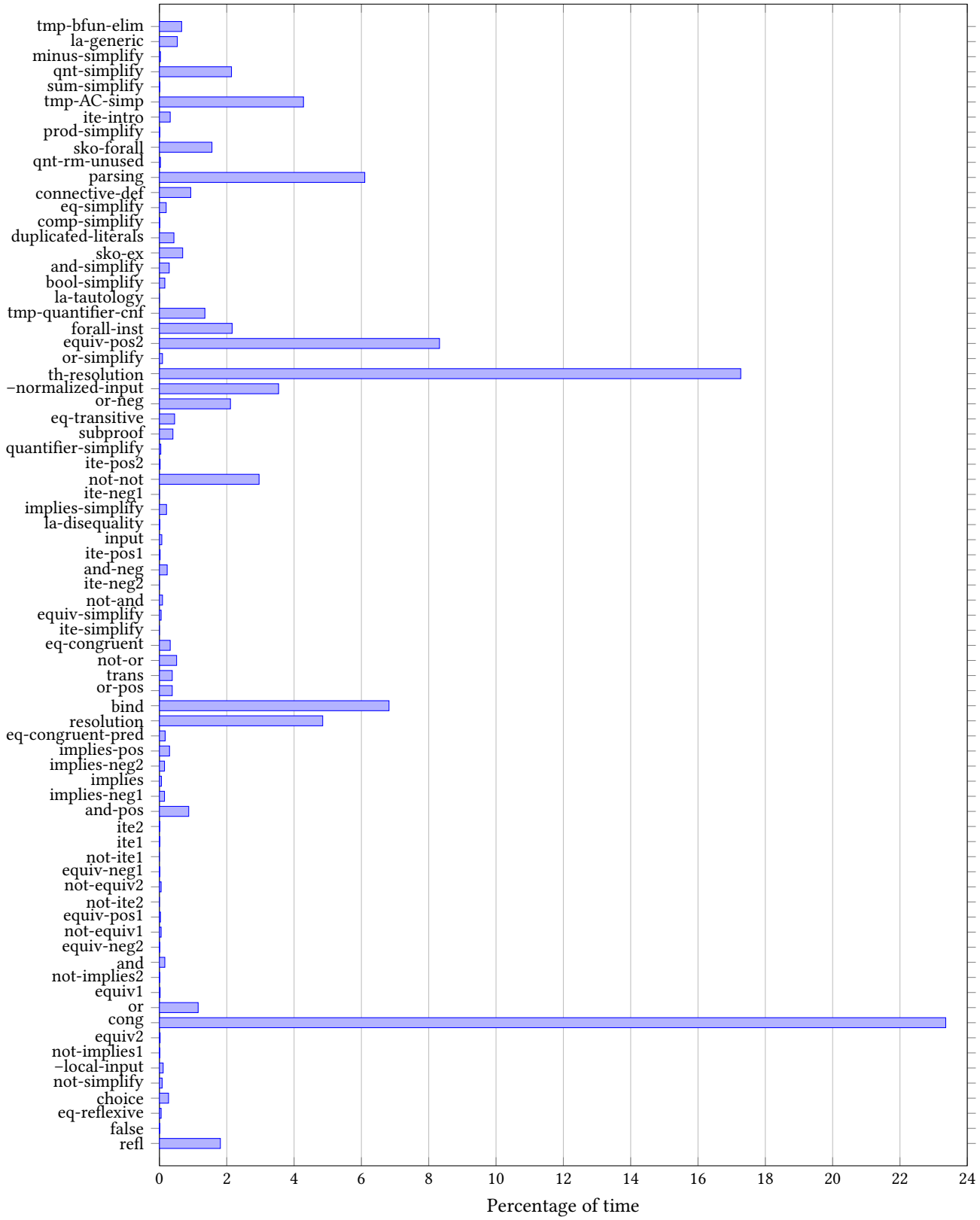

Figure 6. Percentage of the total time spent per rule for the SMT solver veriT.