Walking through the Forest: Fast EUF Proof-Checking Algorithms

Frédéric Besson, Pierre-Emmanuel Cornilleau and Ronan Saillard

Inria Rennes – Bretagne Atlantique, France

Abstract

The quantifier-free logic of Equality with Uninterpreted Function symbols (EUF) is at the core of Satisfiability Modulo Theory (SMT) solvers. There exist several competing proof formats for EUF proofs. We propose original proof formats obtained from *proof forests* that are the artifacts proposed by Nieuwenhuis and Oliveras to extract efficiently EUF unsatisfiable cores. Our proof formats can be generated by SMT solvers for almost free. Moreover, our preliminary experiments show that our novel verifiers outperform other existing EUF verifiers and that our proof formats appear to be more concise than existing EUF proofs.

1 Introduction

SMT solvers (*e.g.*, Z3 [15], Yices [16], CVC3 [4], veriT [10]) are the cornerstone of software verification tools (*e.g.*, Dafny [18], Why/Krakatoa/Caduceus [17], VCC [11]). They are capable of discharging proof obligations of impressive size and make possible verification tasks that would otherwise be unfeasible. Scalability matters, but when it comes to verifying critical software, soundness is mandatory. SMT solvers are based on highly optimised decision procedures and proving the correctness of their implementation is probably not a viable option. To sidestep this difficulty, several SMT solvers are generating proof witnesses that can be validated by external verifiers. In order to minimise the Trusted Computing Base (TCB), an ambitious approach consists in validating SMT proofs using a general purpose proof verifier *i.e.,* a proof-assistant such as Isabelle/HOL or Coq. Recent works [9, 2] show that SMT proofs are big objects and that the bottleneck is usually the proof-assistant. Ideally, SMT proofs should be i) generated by SMT solvers with little overhead; ii) verified quickly by proof assistants.

Even for the simplest logic such as the quantifier-free logic of equality with uninterpreted function symbols (EUF) there exist competing proof formats generated by SMT solvers. Several of those formats have been validated by proof-assistants: Z3 proofs can be reconstructed in Isabelle/HOL and HOL4 [9]; veriT proofs can be reconstructed in Coq [2]. Even though the results are impressive, certain SMT proofs cannot be verified because they are too big.

We propose novel proof formats for EUF and compare their efficiency in terms of i) generation time; ii) checking time; iii) and proof size. Our comparisons are empirical and do not compare the formats in terms of proof-complexity. Instead, we have implemented the verifiers using the functional language of Coq and compared their asymptotic behaviours experimentally over families of handcrafted benchmarks. The contributions of the paper are efficient Coq verifiers for two novel EUF proof formats and their comparison with existing verifiers. The code of the verifiers is available [21]. Our first proof format is made of proof forests that are the artifact proposed by Nieuwenhuis and Oliveras to extract efficiently unsatisfiable cores [20]. Our second proof format is a trimmed down version of the proof forest reduced only to the edges responsible for triggering a congruence. For the sake of comparison, we have also implemented a Coq verifier for the EUF proof format of Z3 [14] and compared with the existing Coq verifier for the EUF proof format of veriT [2]. The main result of our preliminary experiments is that the running time of all the verifiers is negligible *w.r.t.* the type-checking of the EUF proofs. Another result of our experiments is that proof forests verifiers are very fast and that (trimmed down) proof forests appear to be more concise than existing EUF proofs. Another advantage is that proof forests can be generated for almost free by SMT solvers based on proof forests.

The rest of the paper is organised as follows. Section 2 provides basic definitions and known facts about the EUF logic. Section 3 recalls the nature of proof forests and explains the workings of our novel EUF proof verifiers. Section 4 is devoted to experiments. Section 5 concludes.

2 Background

We write $\mathcal{T}(\Sigma, \mathcal{V})$ the smallest set of terms built upon a set of variables $\mathcal{V}$ and a ranked alphabet Σ of uninterpreted function symbols. For EUF, an atomic formula is of the form $t = t'$ for t, $t' \in \mathcal{T}(\Sigma, \emptyset)$ and a literal is an atomic formula or its negation. Equality (=) is reflexive, symmetric, transitive and closed under congruence:

$$\frac{}{x = x} \qquad \frac{x = y}{y = x} \qquad \frac{x = y, y = z}{x = z} \qquad \frac{x_1 = y_1, \ldots, x_n = y_n}{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)}$$

Ackermann has proved using a reduction approach that the EUF logic is decidable [1]. The reduction consists in introducing for each term $f(\vec{x})$ a boolean variable $f_{\vec{x}}$ and encoding the congruence rule by adding the boolean formula $x_1 = y_1 \wedge \ldots \wedge x_n = y_n \Rightarrow f_{\vec{x}} = f_{\vec{y}}$ for each pair of terms $f(\vec{x})$, $f(\vec{y})$. This encoding is responsible for a quadratic blow up of the formula. Nelson has proposed an efficient decision procedure for deciding the satisfiability of a set of EUF literals using a congruence closure algorithm [19]. Nieuwenhuis and Oliveras have shown how to efficiently generate unsatisfiable cores by instrumenting the congruence closure algorithm with a proof forest [20] gathering all the necessary information for generating proofs (see Section 3.1).

3 Walking through the proof forest

In the following, we assume *w.l.o.g.* that EUF terms are flat and curried [20, Section 3.1] *i.e.,* are of the form a or $f(a_1, a_2)$ where f represents an explicit application symbol and a, a_1 and a_2 are constants. In the rest, we drop the f and write $a_1(a_2)$ for $f(a_1, a_2)$. These transformations can be performed in linear time and simplify the decision procedure.

3.1 Proof forest

Nieuwenhuis and Oliveras have proposed a *proof-producing* congruence closure algorithm for deciding EUF [20]. Their main contribution is an efficient Explain operation that outputs a (small) set of input equations E needed to deduce an equality, say $a = b$. If $a \neq b$ is also part of the input, $E \cup a \neq b$ is a (small) unsatisfiable core that is used by the SMT solver for backtracking. As a result, SMT solvers using congruence closure run a variant of the Explain algorithm – whether or not they are proof producing.

The Explain algorithm is based on a specific data structure: the proof forest. A proof forest is a collection of trees and each edge $a \rightarrow b$ in the proof forest is labelled by a *reason* justifying why the equality $a = b$ holds. A reason is either an input equation $a = b$ or a pair of input equations $a_1(a_2) = a$ and $b_1(b_2) = b$. For the second case, there must be justifications in the forest for $a_1 = b_1$ and $a_2 = b_2$.

We give in Figure 1 a modified version of the original Explain algorithm. The NearestAncestor and Parent functions return (if it exists) respectively the nearest common ancestor of two nodes

```
let ExplainAlongPath(a,c) :=
    a:=HighestNode(a);
    c:=HighestNode(c);
    if a = c then return
    else
      b:=Parent(a);
      if edge has form $a \xrightarrow{a=b} b$
      then  Output (a = b)
      else{ /*edge has form $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$*/
          Output a1(a2)=a and b1(b2)=b ;
          Explain(a1,b1); Explain(a2,b2)};
      Union(a,b); ExplainAlongPath(b,c)
```

(a) ExplainAlongPath

```
let Explain(a,b) :=
  c:=NearestAncestor(a,b);
  ExplainAlongPath(a,c);
  ExplainAlongPath(b,c)
```

(b) Explain

Figure 1: Recursive Explain algorithm

in the proof forest and the parent of a node in the proof forest. A union-find data structure [23] is also used: Union(a,b) merges the equivalence classes of a and b and Find(a) returns the current representive of the class of a. The HighestNode(c) operation returns the highest node (*i.e.,* the node with minimal depth in the proof forest) belonging to the equivalence class of c.

Contrary to the original version, our algorithm is recursive. The Union operation has also been moved. As a consequence output equations are ordered differently. This will ease our futur proof production. Another difference is that the original algorithm always terminates but does not detect certain ill-formed proof forests *e.g.,* $a \xrightarrow[a=f(a)]{b=f(b)} b$. In this case, the recursive algorithm does not terminate (this issue is dealt with in section 4.1).

3.2 Proof forest verifier

The Explain algorithm of Figure 1 can be turned into a EUF proof verifier. The verifier is a version of Explain augmented with additional checks to ensure that the edges obtained from the SMT solver correspond to a well-formed proof forest. For instance, the verifier checks that edges are only labelled by input equations. Moreover, for edges of the form $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$, the recursive calls to Explain ensure that $a_1 = b_1$ and $a_2 = b_2$ have proofs in the proof forest *i.e.,* a_1 (resp. a_2) is connected with b_1 (resp. b_2) by some valid path in the proof forest. For efficiency and simplicity, the *least common ancestors* are not computed by the verifier but used as untrusted hints. The soundness of the verifier does not depend on the validity of this information as the proposed *least common ancestor* is just used to guide the proof. If the return node is not a common ancestor, the verifier will simply fail.

For the verifier, a EUF proof is a pruned proof forest corresponding to the edges walked through during a preliminary run of Explain. As the SMT solver needs to traverse the proof forest to extract unsatisfiable core, we argue that the proof forest is a EUF proof that requires no extra-work from the SMT solver.

```
let UFchecker input edges (x,y) :=
  for (a = b) ∈ input do Union(a,b) done
  for $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$ in edges do
    if (a1(a2)=a) ∈ input & (b1(b2)=b) ∈ input
              & isEqual(a1,b1) & isEqual(a2,b2)
    then Union(a,b)  else fail
  done
  return isEqual(x,y)
```

Figure 2: Verifier algorithm for trimmed forests

3.3 A verifier using trimmed forests

To avoid traversing the same edge several times, the Explain algorithm and its verifier are using a union-find data structure. Therefore, the Explain verifier implicitly embeds a *decision procedure* for the theory of equality. Our optimised verifier fully exploits this observation and starts by feeding all the input equalities of the form $a = b$ into its union-find. For the decision procedure, new equalities are obtained by applying the congruence rule and efficiency crucially depends on a clever indexing of the equations. The verifier does not require this costly machinery and takes as argument a trimmed down proof forest reduced to the list of edges of the forest of form $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$. The edge labels indicate the equations that need to be paired to derive $a = b$ by the congruence rule. The algorithm of the verifier is given in Figure 2 where the predicate isEqual(a,b) checks whether a and b have the same representative in the union-find *i.e.*, Find(a) = Find(b). Once again, a preliminary run of Explain is sufficient to generate a proof for this optimised verifier.

4 Implementation and experiments

4.1 EUF verifiers in Coq

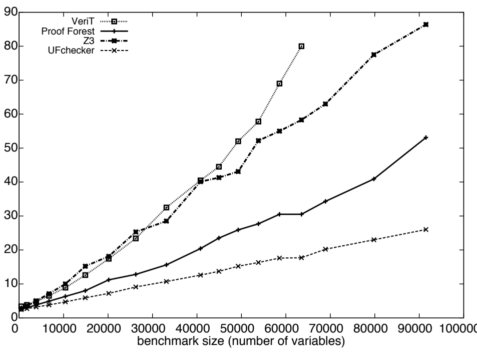
Our verifiers are implemented using the native version of Coq [8] which features *persistent* arrays [13]. Persistent arrays are purely functional data-structures that ensure constant time accesses and updates of the array as soon as it is used in a monadic way. For maximum efficiency, all the verifiers make a pervasive use of those arrays that allow for an efficient union-find implementation: union and find have their logarithmic asymptotic complexity.

Compared to other languages, a constraint imposed by Coq is that all programs must be terminating. The UFchecker (see Section 3.3) is trivially terminating. Termination of the proof-forest verifier is more intricate because the Explain algorithm (see Figure 3.2) does not terminate if the proof forest is ill-formed *e.g.*, has cycles. However, if the proof forest is well-formed, an edge is only traversed once. As a result, at each recursive call, our verifier decrements an integer initialised to the size of the proof forest. In Coq, the verifier fails after exhausting the maximum number of allowed recursive calls.

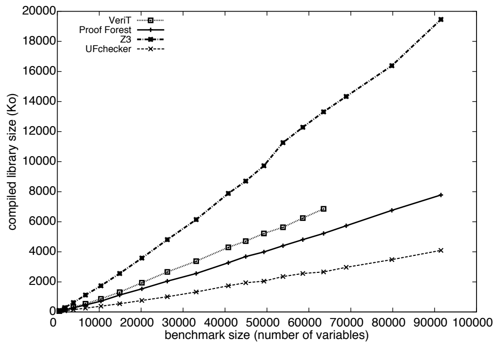
For the sake of comparison, we have also implemented the EUF proof format of Z3. Z3 refutations are also generated using Explain [14, Section 3.4.2]. Unlike our verifiers, Z3 proofs use explicit boolean reasoning and *modus ponens*. As a consequence formulae do not have a constant size. As already noticed by others [9, 2], efficient verifiers require a careful handling of sharing. Our terms and formulae are *hash-consed*; sharing is therefore maximum and comparison of terms or formulae is a constant-time operation.

4.2 Benchmarks

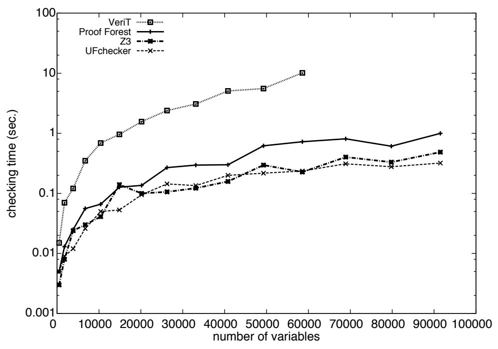
We have assessed the efficiency of our EUF verifiers on several families of handcrafted conjunctive EUF benchmarks. The benchmarks are large and all the literals are necessary to prove unsatisfiability. The formulae are not representative of conflict clauses obtained from SMT-LIB [3] benchmarks which are usually very small [7] and would not stress the scalability of the verifiers. For all our benchmarks the running time of the verifiers is negligible especially compared to the time spent type-checking the EUF proofs and the proof size is linear in the size of the formulae.



(a) Total running time



(b) Size of proofs



(c) Running time of the verifier

Figure 3: Comparison of the verifiers

Figure 3 shows our experimental results for a family F of formulae of the general form

$$F_j = \left\{ \begin{array}{ll} x_0 = x_1 \qquad x_0 \neq x_{(j+1)\cdot j} & \\ f(x_{i\cdot j}, x_{i\cdot j}) = x_{i\cdot j+1} = \ldots = x_{i\cdot j+j} & \text{for} \quad i \in \{0 \ldots j\} \end{array} \right.$$

The benchmarks are indexed by the number of EUF variables and the results are obtained using a Linux laptop with a processor Intel Core 2 Duo T9600 (2.80GHz) and 4GB of Ram.

Figure 3a shows the time needed to construct and compile Coq proof terms. Figure 3b shows the size of the compiled proof terms. Figure 3c is focusing on the running time of the verifiers excluding the time needed to construct and dump proof terms.

For all our benchmarks, the UFchecker shows a noticeable advantage over the other verifiers. We can also remark that its behaviour is more predicable. The veriT verifier [2] is using proofs almost as small as proof forests but the traces generated by veriT are sometimes two orders of magnitude bigger. In the timings, the pre-processing needed to perform this impressive proof reduction is accounted for and might explain why the veriT verifier gets slower as the benchmark size grows. Remark also that for the biggest benchmarks, veriT fails to generate proofs.

Our Z3 verifier scales well despite being slightly but constantly outrun by the verifiers based on proof-forests. The running time of the different verifiers is given Figure 3c using a logarithmic scale. This emphases the fact that, except for the veriT verifier, the running time of the all the verifiers is below the second and is therefore not the limiting factor of the verification. Not surprisingly, the UFchecker requires smaller proofs and therefore its global checking time is also smaller. For big benchmarks, its running time also tends to be smaller than the running time of other verifiers.

5 Conclusion

Proof generating SMT solvers such as Z3 [15], CVC3 [4] or veriT [10] routinely generate EUF proofs in various formats. Those formats have in common that proof-checking is essentially linear in the size of the proof. In theory, our EUF proofs cannot be checked in linear time and the verifiers need to embed a union-find algorithm. In practice, our experiments conclude that our UFchecker verifier outperforms existing verifiers for EUF proofs and that succinct EUF proofs are the key for scalability. Embedding union-find in EUF proofs was previously proposed by Conchon *et al.,* [12]. However, in their approach, union-find computations are interleaved with logic inferences that are responsible for an overhead that is absent from our verifiers. The importance of succinct proofs has been recognised by Stump [22] and its Logical Framework with Side Conditions (LFSC). Unlike LFSC proofs, our proofs are less flexible but purely computational and use the principle of proof by *reflection* [5, Chapter 16]. As future work, we intend to integrate the UFchecker into the SMT proof verifier developed by several of the current authors [6] and extend its scope to the logic of constructors.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.

[3] C. Barret, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0, 2010.

[4] C. Barrett and C. Tinelli. CVC3. In *Proc. of CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, 2004.

[6] F. Besson, P-E. Cornilleau, and D. Pichardie. Modular SMT Proofs for Fast Reflexive Checking Inside Coq. In *CPP*, volume 7086 of *LNCS*, pages 151–166. Springer, 2011.

[7] F. Besson, P-E. Cornilleau, and D. Pichardie. A nelson-oppen based proof system using theory specific proof systems. In *PxTP 2011*, 2011.

[8] M. Boespflug, M. Dénès, and B. Grégoire. Full Reduction at Full Throttle. In *CPP*, volume 7086 of *LNCS*, pages 362–377. Springer, 2011.

[9] S. Böhme and T. Weber. Fast LCF-style Proof Reconstruction for Z3. In *Proc. of ITP 2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[10] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Proc. of CADE 2009*, LNCS. Springer, 2009.

[11] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[12] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Lightweight integration of the ergo theorem prover inside a proof assistant. In *Proceedings of the second workshop on Automated formal methods*, AFM '07, pages 55–59. ACM, 2007.

[13] S. Conchon and J-C. Filliâtre. Semi-persistent Data Structures. In *ESOP*, volume 4960 of *LNCS*, pages 322–336. Springer, 2008.

[14] L. M. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In *Proc. of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants*, volume 418. CEUR-WS.org, 2008.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[16] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[17] J-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.

[18] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[19] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[20] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In *Proc. of RTA 2005*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.

[21] R. Saillard. EUF Verifiers in Coq. `http://www.irisa.fr/celtique/ext/euf`.

[22] A. Stump. Proof checking technology for satisfiability modulo theories. *Electron. Notes Theor. Comput. Sci.*, 228:121–133, January 2009.

[23] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.