Walking through the Forest: a Fast EUF Proof-Checking Algorithm

Frédéric Besson, Pierre-Emmanuel Cornilleau, Ronan Saillard
Inria Rennes – Bretagne Atlantique, France

Abstract

The quantifier-free logic of equality with uninterpreted function symbols (EUF) is at the core of SMT solvers. However, there exist several competing proof formats to validate EUF proofs. As EUF proof, we advocate for the *proof forest* that is the artifact proposed by Nieuwenhuis and Oliveras to extract efficiently EUF unsatisfiable cores. An advantage of this proof format is that it can be generated by the SMT solver for almost free. Our preliminary experiments show that our proof forest verifier outperforms other EUF verifiers and that *proof forests* appear to be more concise than existing EUF proofs.

1 Introduction

SMT solvers (*e.g.*, Z3 [11], Yices [12], CVC3 [3], veriT [7]) are the cornerstone of software verification tools (*e.g.*, Dafny [14], Why/Krakatoa/Caduceus [13], VCC [8]). 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 [6, 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 are several competing proof formats generated by SMT solvers. Several of those formats have already been validated by proof-assistants. For instance, Z3 proofs can be reconstructed in Isabelle/HOL and HOL4 [6] and 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 consider several proof formats for EUF and compare their efficiency in terms of i) generation time; ii) checking time; iii) and proof size. Our comparisons are empiric 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 behaviour experimentally over families of handcrafted benchmarks. The contributions of the paper are efficient Coq verifiers for two novel EUF proof format and their comparison with existing ones. The code of the verifiers is available [17]. The most original verifier validates directly the proof forest that is the artifact proposed by Nieuwenhuis and Oliveras to extract efficiently unsatisfiable cores [16]. For the sake of comparison, we have also implemented a Coq verifier for the EUF proof format of Z3 and compared with the existing Coq verifier for the EUF proof format of veriT [2]. The result of our preliminary experiments is that the running time of all the verifiers is negligible *w.r.t* the parsing/type-checking of the textual representation of the EUF proofs. Another result of our experiments is that the proof forest verifier is very fast and that proof forests appear to be more concise that existing EUF proofs. Another advantage is that proof forests should be generated for almost free by SMT solvers.

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 signature Σ. For EUF, an atomic formula is a term 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:

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

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 [15]. An unsatisfiable core is a (minimum) set of literals which conjunction is not satisfiable. Unsatisfiable cores are crucial for the efficiency of modern SMT solvers. Nieuwenhuis and Oliveras have shown how to efficiently generate unsatisfiable cores by instrumenting the congruence closure algorithm with a proof forest [16] 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 currified *i.e.,* are of the form x or $f(x)$ where x is a constant. 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 [16]. Their main contribution is an efficient Explain operation which 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 in which each edge $a \to b$ 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 $a_2 = b_2$.

A recursive version of Explain is given Figure 1. The NearestCommonAncestor and Parent functions return (if it exists) respectively the nearest common ancestor of two nodes in the

```
let Explain (a,b) :=
   let c := NearestCommonAncestor (a,b) in
   let c := HighestNode c in
      ExplainAlongPath (a,c) ;
      ExplainAlongPath (b,c)

let ExplainAlongPath (a,c) :=
   a := HighestNode a
   if a = c then return
   else
      let b := Parent(a) in
      if edge a -> b is labelled with (a = b) then
         Output (a = b)
      else /* edge is labelled with a1(a2)=a and b1(b2)=b */
         Output a1(a2)=a and b1(b2)=b ;
         Explain (a1,b1) ;
         Explain (a2,b2)
      Union a b ;
      ExplainAlongPath (b,c)
```

Figure 1: Recursive Explain algorithm

proof forest and the parent of a node in the proof forest. A union-find structure is also used: Union a b merges the equivalence classes of a and b; HighestNode c is the highest node (in the proof forest) belonging to the equivalence class of c.

SMT solvers such as Z3 [10] and veriT [7] use different techniques to turn the Explain algorithm into a proof-producing procedure for EUF. In Section 3.2, we show how to instrument Explain to generate a list of proof commands to be checked by a specialised proof verifier. In Section 3.3, we show how the proof forest itself can be used as a EUF proof. In that case, the verifier is a variant of the Explain algorithm performing extra checks.

3.2 Command verifier

Our proof language is made of a list of commands reminiscent of our previous format [4]. Each command derives new equalities from initial equalities or already derived equalities. The key commands correspond to the following deduction rules.

$$\textit{Symmetry}\,\frac{a = b}{b = a} \qquad \textit{Transitivity}\,\frac{a = b \quad b = c}{a = c}$$

$$\textit{Congruence}\,\frac{a = a_1(a_2) \quad b = b_1(b_2) \quad a_1 = b_1 \quad a_2 = b_2}{a = b}$$

The commands are obtained by running a version of Explain where the union-find structure has been replaced by a hash-table keeping track of already derived equalities. Each call Explain a b appends a *Transitivity* and a *Symmetry* command to the commands obtained by the two successive calls to ExplainAlongPath a c and ExplainAlongPath b c. Each call ExplainAlongPath a c generates a list of commands justifying the equality $a = c$. An edge $a \xrightarrow{a=b} b$ is justified by a command *Hyp* checking that $a = b$ is indeed an input equation. An edge $a \xrightarrow[a=a_1(a_2)]{b=b_1(b_2)} b$ is justified by a *Congruence* command appended to the result of the recursive calls to Explain a1 b1 and Explain a2 b2 which generate commands justifying the equalities $a_1 = b_1$ and $a_2 = b_2$. The recursive call to ExplainAlongPath b c generates commands justifying the equality $b = c$. The complete list of commands is then obtained by adding a *Transitivity* command to prove $a = c$ from $a = b$ and $b = c$. The verification of such a proof then consists in executing in order each command to derive the wanted equality checking that each rule is correctly applied.

3.3 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 $a \xrightarrow[a=a_1(a_2)]{b=b_1(b_2)} b$, 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 this 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 the EUF proof that requires the least extra-work from the SMT solver.

Unlike more traditional proof verifiers, this verifier needs more sophisticated data-structures such as an auxiliary union-find. Our opinion is that the slight complication of the soundness proof is overweight by the simplicity of the proof generation.

4 Implementation and Experiments

4.1 EUF verifiers in Coq

Our verifiers are implemented using the native version of Coq [5] which features *persistent* arrays [9]. 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.

Compared to other languages, a constraint imposed by Coq is that all programs must be terminating. The command-based verifier (see Section 3.2) is trivially terminating. Termination of the proof-forest verifier is more intricate because the Explain algorithm (see Figure 3.3) 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. An interesting observation is that the original Explain algorithm [16, Section 3.4] 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 Explain algorithm of Figure 1 does not terminate. 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 [10, Section 3.4.2]. Unlike our verifier described in Section 3.2, Z3 proofs are using explicit boolean reasoning and *modus ponens*. As a consequence formulae do not have a constant size. As already noticed by others [6, 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 non-satisfiability. For all our benchmarks the running time of the verifiers is negligible especially compared to the time spent parsing/type-checking the textual representation of the different EUF proofs. Moreover, the proof size is linear in the size of the formulae.

Figure 2 shows our experimental results for a family of formulae of the general form

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

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 CPU T9600 (2.80GHz) and 4GB of Ram. The Figure on the left shows the time needed to construct and compile Coq proof terms. The Figure on the right shows the size of the compiled proof terms. For all our benchmarks, the proof

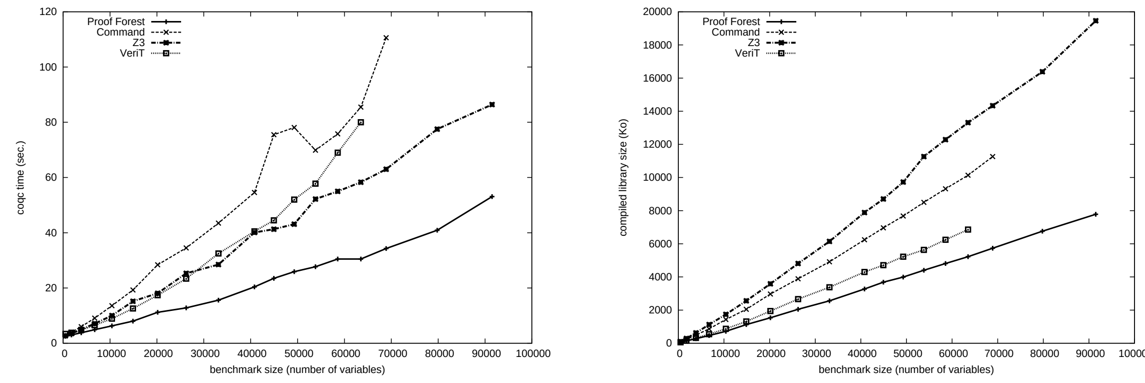

Figure 2: Comparison of generation time and size of compiled proofs

forest verifier 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. This is an impressive result knowing that veriT produces sometimes traces that can be more than two orders of magnitude bigger. In the timings, the pre-processing needed to perform the 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, the veriT SMT solver fails to generate proofs. This is also the case for our proof generator for the command verifier. The Z3 verifier is more scalable despite being slightly but constantly overrun by the proof forest verifier. As all the verifiers are equally optimised we are confident that proof forests are responsible for smaller EUF proofs in general and that the proof forest verifier scales remarkably well.

5 Conclusion

We have compared empirically different proof verifiers for EUF proofs that can be generated by SMT solvers. The conclusion of our experiments is that a proof forest verifier outperforms existing verifiers for EUF proofs. The proof forest can be generated by the SMT solver with little overhead, the proof is succinct and the proof checking is fast. Compared to other verifiers, the soundness of the proof forest verifier is a little more intricate. We are currently completing the proofs of the different verifiers. As future work, we intend to integrate the proof forest verifier into the SMT proof verifier developed by several of the current authors [4] 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. Barrett and C. Tinelli. CVC3. In *Proc. of CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

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

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

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

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