Certified Interpolant Generation for EUF

Andrew Reynolds
The University of Iowa

Cesare Tinelli
The University of Iowa

Liana Hadarean
New York University

Abstract

Logical interpolants have found a wide array of applications in automated verification, including symbolic model checking and predicate abstraction. It is often critical to these applications that reported interpolants exhibit desired properties, correctness being first and foremost. In this paper, we introduce a method in which interpolants are computed by type inference within the trusted core of a proof checker. Interpolants produced this way from a proof of the joint unsatisfiability of two formulas are certified as correct by construction. We focus our attention to the quantifier-free theory of equality and uninterpreted functions (EUF) and present an interpolant generating proof calculus that can be encoded in the LFSC proof checking framework with limited reliance upon computational side conditions. Our experimental results show that our method generates certified interpolants with small overhead with respect to solving.

1 Introduction

Given a logical theory T, a T-interpolant for a pair of formulas A, B that are jointly unsatisfiable in T is a formula φ over the symbols of T and the free symbols common to A and B such that (i) $A \models_T \varphi$ and (ii) $B, \varphi \models_T \bot$, where $\models_T$ is logical entailment modulo T. For certain theories, interpolants can be generated efficiently from a refutation of $A \wedge B$.

Interpolants have been used successfully in a variety of contexts, including symbolic model checking [9] and predicate abstraction [8]. In many applications, it is critical that formulas computed by interpolant-generation procedures be indeed interpolants, that is, exhibit the defining properties above. For example, in [9] McMillan shows how interpolants can be used to produce a sound and complete method for checking safety properties of finite state systems based on a fixed-point computation that over-approximates the set of reachable states. Using as interpolants formulas that violate property (ii) above makes the method incomplete, as it leads to spurious counterexamples that do not contribute to the overall progress of the main algorithm. Even worse, using interpolants formulas that violate property (i) above may cause the algorithm to reach a fixed point prematurely, thereby reporting a unsound result. While there are known methods to generate interpolants efficiently [4, 3], none of them do so in a verified way. This is an obstacle to the use of interpolation-based model checking in applications of formal methods that require independently checkable proof certificates. We present a way of addressing this deficiency by means of a method for generating *certified interpolants*.

We describe how T-interpolants can be efficiently produced using LFSC, a framework for defining logical calculi that extends the Edinburgh Logical Framework [14] with computational side conditions. In our approach, a proof system for the theory T of interest is encoded declaratively as a set of types and typing rules in a user-defined *signature*, separate from the core of the proof checker. The interpolant itself is computed by *type inference* as a side effect of proof checking. Because the interpolant is computed using the trusted core of the proof checker, it is certified as correct by construction.

Interpolant-generating calculi exist for many theories, including quantifier-free linear integer arithmetic [2] and linear real arithmetic [10]. In this work, we focus on the quantifier-free (fragment of the) theory of equality and uninterpreted functions (EUF). Our approach, however, applies to other quantifier-free theories. Extending previous work [6], we present an intuitive interpolant-generating calculus for EUF that is more flexible than previous approaches, and can be implemented in LFSC in a natural way. We have encoded this calculus in LFSC, and instrumented the SMT-solver Cvc3 [1] to output proofs in this format to verify the viability of our approach.

Related Work. In [10], McMillan gives a calculus for interpolant generation in EUF that is implemented within an interpolating theorem prover, intended for the purposes of interpolation-based model checking and predicate refinement. Fuchs *et al.* give an alternative approach in [6] for interpolant generation in EUF in terms of an algorithmic procedure. An approach for efficient interpolant generation in SMT is given by Cimatti *et al.* in [4] and is implemented within the MathSAT SMT solver.

Contributions. The preliminary work discussed here has a two-fold contribution. Firstly, we develop a general framework for generating interpolants in a certified manner via type checking. Secondly, we provide a novel calculus for interpolant generation in EUF based on the procedure of [6] that can be encoded within this framework. We present an implementation of our approach and discuss comparative performance results in various configurations that provide experimental evidence of practical feasibility. We are working on extending this work to arbitrary quantifier-free formulas, as opposed to only conjunction of literals.

Paper Outline. Section 2 gives an overview of the LFSC framework, in particular, how a proof signature can incorporate proof rules that carry additional information. Section 3 contains a description of our interpolant generating calculus for EUF, and details on how it is encoded. A detailed proof of soundness and (relative) completeness can be found in a longer version of this paper [13]. Finally, experimental results from our implementation are given in Section 4.

1.1 Formal Preliminaries

We work in the context of first-order logic with equality, and use standard notions of signature, term, literal, formula, clause, Horn clause, entailment, satisfiability, and so on. We use the symbol $\approx$ to denote the equality predicate in the logic. We abbreviate $\neg(s \approx t)$ as $s \not\approx t$. We identify finite sets of formulas with the conjunction of their elements. For terms or formulas we use "ground", i.e., variable-free, and "quantifier-free" interchangeably since for our purposes free variables can be always seen as free constants. For convenience, we follow Nieuwenhuis and Oliveras [11] in considering only *Curried signatures*, signatures with no function symbols of positive arity except for a distinguished infix binary symbol $\cdot$. Then, EUF can be defined as the (empty) theory of $\cdot$. This is without loss of generality since a ground formula over any signature can be converted into an equisatisfiable ground formula over the signature above [11]. For example, the formula $g(a) \approx f(a, g(a))$ can be converted into $g \cdot a \approx (f \cdot a) \cdot (g \cdot a)$ with f, g and a all treated as constant symbols.

Throughout the paper we will work with two ground EUF formulas A and B. Let Σ_A and Σ_B be the sets of non-logical symbols (i.e., variables/constants) that occur in A and B, respectively. Terms, literals and formulas over Σ_A are *A-colorable*, and those over Σ_B are *B-colorable*. Such expressions are *colorable* if they are either A- or B-colorable, and are *AB-colorable* if they are both.

$$\frac{t_1 \approx t_2 \quad t_2 \approx t_3}{t_1 \approx t_3}\ \textsf{trans}$$

```
term : type
formula : type
eq : (! t1 term (! t2 term  formula))
proof : (! f formula type)
```

```
trans : (! t1 term (! t2 term (! t3 term
        (! p1 (proof (eq t1 t2)) (! p2 (proof (eq t2 t3))
          (proof (eq t1 t3)))))))
```

$$\frac{t_1 \approx t_2\ [l_1] \quad t_2 \approx t_3\ [l_2]}{t_1 \approx t_3\ [l_1 + t_2 :: l_2]}\ \textsf{trans\_aug}$$

```
...
term_list : type
proof_aug : (! f formula
            (! tl term_list  type))
```

```
trans_aug : (! t1 term (! t2 term (! t3 term
            (! tl1 term_list (! tl2 term_list
              (! p1 (proof_aug (eq t1 t2) tl1)
              (! p2 (proof_aug (eq t2 t3) tl2)
                (proof_aug (eq t1 t3) (concat tl1 t2 tl2)))))))))
```

Figure 1: Example of two proof systems in LFSC, the second being an augmented version of the first. In LFSC's LISP-like concrete syntax, the symbol ! represents LF's Π binder, for the dependent function space.

2 Interpolant Generation via Type Inference

Edinburgh LF is a framework for defining logics by means of a dependently-typed lambda calculus, the $\lambda\Pi$-calculus [7]. LFSC (Logical Framework with Side Conditions) extends LF by adding support for computational side conditions, i.e., functional programs used to test logical conditions on proof rules [14]. As in LF, a proof system can be defined in LFSC as a list of typing declarations, called a *signature*. Each proof rule is encoded as a constant symbol whose type represents the inference allowed by the rule. An example of a minimal proof system in LFSC is given in the upper half of Figure 1, which shows how the transitivity rule for equality can be encoded in LFSC syntax. In mathematical notation, the LFSC signature in the figure declares eq, for example, as a symbol of type $\Pi\, x{:}\mathsf{term}.\, \Pi\, y{:}\mathsf{term}.\, \mathsf{formula}$, or, written in more conventional form, of type $\mathsf{term} \to \mathsf{term} \to \mathsf{formula}$. The symbol trans, encoding the transitivity proof rule, has type $\Pi\, t_1{:}\mathsf{term}.\, \Pi\, t_2{:}\mathsf{term}.\, \Pi\, t_3{:}\mathsf{term}.\, (\mathsf{proof}\ (\mathsf{eq}\ t_1\ t_2) \to \mathsf{proof}\ (\mathsf{eq}\ t_2\ t_3) \to \mathsf{proof}\ (\mathsf{eq}\ t_1\ t_3))$, indexed by three terms. Intuitively, for any terms t_1, t_2, t_3, trans produces a proof of the equality $t_1 \approx t_3$ from two subproofs p_1 and p_2 of $t_1 \approx t_2$ and $t_2 \approx t_3$, respectively.

The LF metalanguage provides the user with the freedom to choose how to represent logical constructs in a signature. In particular, we may augment proof rules to carry additional information, through the use of suitably modified types. The lower half of Figure 1 shows a modified version of the aforementioned calculus. In this version, the modified proof rule takes as premises *annotated* equalities and produce annotated equalities. The annotation consists of a list of terms occurring in equalities used to prove the overall equality. The corresponding proof judgment in LFSC, encoded as the type proof_aug, is used by the trans_aug rule, which combines a chain of equalities and concatenates their respectively lists with the intermediate term t_2.[1]

One can write proof terms in this calculus where every subterm of type term_list is a (type) variable. Concretely, this is done by using a distinguished *hole* symbol _ , each occurrence of which stands for a different variable. Then, the lists of intermediate terms can be computed by an LFSC type checker by type

[1] Although not shown here, the list concatenation constant concat, of type $\mathsf{term\_list} \to \mathsf{term} \to \mathsf{term\_list} \to \mathsf{term\_list}$, can be given as a logical definition in LFSC, and does not need to be implemented algorithmically.

$$\frac{}{t \approx t}\ \textsf{refl} \qquad \frac{t_2 \approx t_1}{t_1 \approx t_2}\ \textsf{sym} \qquad \frac{s_1 \approx s_2 \quad t_1 \approx t_2}{s_1 \cdot t_1 \approx s_2 \cdot t_2}\ \textsf{cong}$$

$$\frac{t_1 \approx t_2 \quad t_2 \approx t_3}{t_1 \approx t_3}\ \textsf{trans} \qquad \frac{t_1 \approx t_2 \quad \neg(t_1 \approx t_2)}{\bot}\ \textsf{contra}$$

Figure 2: A standard proof calculus for EUF literals. We assume that all terms are in Curried form, and so the only function symbol of non-zero arity is $\_ \cdot \_$, denoting function application.

inference when type checking the proof term. In this example, type checking a proof term P against a type of the form (proof_aug (eq t_1 t_2) _) for some terms t_1 and t_2 will cause the LFSC checker to verify that P has type (proof_aug (eq t_1 t_2) l) for some list of terms l, and if successful, output the computed value of l.

In this example, an account of the steps taken for the proven equality is recorded as specificational data, in the rule's annotation. One can follow the same approach to capture information useful to compute interpolants. In general, one may augment a proof signature to operate on judgments carrying additional terms that satisfy some specific invariant. We will use the common notion of a *partial interpolant* in the next section, which can be encoded as an LF type declaration in a natural way.

Our approach allows for two options for obtaining certified interpolants. First, an LFSC proof term P can be tested against the type (interpolant F) for some formula F. In this option, the interpolant F is explicitly provided as part of the proof, and if proof checking succeeds, then both the proof P and the interpolant F are certified to be correct. Note that the user and the proof checker must agree on the exact form for the interpolant F. Alternatively, P can be tested against the type (interpolant _). If the proof checker verifies that P has type (interpolant F) for some formula F, it will output the computed value of F. In this approach, interpolant generation comes as a side effect of proof checking, via type inference, and the returned interpolant F is correct by construction.

3 Interpolant Generation in EUF with Partial Interpolants

In this section we provide an *interpolating calculus* for EUF whose rules can be used to build refutations in EUF but are also annotated with information for generating interpolants from these refutations. For simplicity, we restrict ourselves to the core case of input sets of formulas containing only literals. A calculus for general quantifier-free formulas is not substantially harder. It can be developed along similar lines, relying on existing methods for extending in a uniform way any interpolation procedure for sets of literals to sets of arbitrary qffs [10]; but this is left to future work.

Our interpolating calculus is an annotated version of the basic calculus for EUF, shown in Figure 2. To compute an interpolant for two jointly unsatisfiable sets of literals A and B, we take a refutation of $A \cup B$ in the basic calculus and *lift* it to a refutation in our interpolating calculus. The lifting is straightforward and consists in essence in (i) annotating the literals of A and B with a suitable *color* (see later) and (ii) replacing each rule application by a corresponding set of rule applications in the interpolating calculus. Every rule in the latter calculus is such that the annotation of the rule's conclusion is derived from the annotations of its premises. Thus, these annotations can be left unspecified in the proof, and derived by type inference in LFSC during proof checking.

Note that since both the proof generation and the proof lifting steps are done before proof checking, neither of them needs to be trusted. This allows us to handle various complications outside the trusted

$$\frac{\{\ t \text{ is } A\text{-colorable}\ \}}{t \approx t\,[\top, \top, A]}\ \mathsf{refl\_A} \qquad \frac{t_1 \approx s_1\,[\varphi_1, \psi_1, A] \qquad t_2 \approx s_2\,[\varphi_2, \psi_2, A]}{t_1 \cdot t_2 \approx s_1 \cdot s_2\,[\varphi_1 \wedge \varphi_2, \psi_1 \wedge \psi_2, A]}\ \mathsf{cong\_A}$$

$$\frac{t_1 \approx t_2\,[A]}{t_1 \approx t_2\,[\top, \top, A]}\ \mathsf{base\_A} \qquad \frac{t_1 \approx t_2\,[\varphi, \psi, A] \quad t_1 \not\approx t_2\,[A]}{\bot\,[\varphi \wedge \neg\psi]}\ \mathsf{contra\_A}$$

$$\frac{t_1 \approx t_2\,[\varphi_1, \psi_1, c] \qquad t_2 \approx t_3\,[\varphi_2, \psi_2, c'] \qquad \{\ t_1, t_3 \text{ are } A\text{-colorable}\ \}}{t_1 \approx t_3\,[\varphi_1 \wedge \varphi_2, \psi_1 \wedge \psi_2, A]}\ \mathsf{trans\_A}$$

$$\frac{\{\ t \text{ is } B\text{-colorable}\ \}}{t \approx t\,[\top, t \approx t, B]}\ \mathsf{refl\_B} \qquad \frac{t_1 \approx s_1\,[\varphi_1, \psi_1, B] \qquad t_2 \approx s_2\,[\varphi_2, \psi_2, B]}{t_1 \cdot t_2 \approx s_1 \cdot s_2\,[\varphi_1 \wedge \varphi_2, t_1 \cdot t_2 \approx s_1 \cdot s_2, B]}\ \mathsf{cong\_B}$$

$$\frac{t_1 \approx t_2\,[B]}{t_1 \approx t_2\,[\top, t_1 \approx t_2, B]}\ \mathsf{base\_B} \qquad \frac{t_1 \approx t_2\,[\varphi, \psi, B] \quad t_1 \not\approx t_2\,[B]}{\bot\,[\varphi]}\ \mathsf{contra\_B}$$

$$\frac{t_1 \approx t_2\,[\varphi_1, \psi_1, c_1] \qquad t_2 \approx t_3\,[\varphi_2, \psi_2, c_2] \qquad \{\ t_1, t_3 \text{ are } B\text{-colorable}\ \}}{t_1 \approx t_3\,[\varphi_1 \wedge \varphi_2 \wedge (\psi_1 \wedge \psi_2) \rightarrow (t_1 \approx t_3), t_1 \approx t_3, B]}\ \mathsf{trans\_B}$$

Figure 3: A-Prover and B-Prover Partial Interpolant Rules. The text in braces denote computational side conditions. Note that the trans rules are actually a summary of multiple rules for cases of colors c and c'. In some of these cases, formulas in the annotation of conclusions may be simplified to respect colorability constraints.

core of LFSC, such as applications of congruence between uncolorable terms. In the end, our interpolating calculus can be encoded in 203 lines of LFSC type declarations. The core rules contain just one kind of computational side conditions, which test for term colorability.

3.1 An Interpolating Calculus for EUF

For the rest of this section, we fix two sets A and B of literals such that $A \cup B$ is unsatisfiable in EUF. Without loss of generality, we assume that A and B are disjoint.

An interpolant-generating calculus for EUF can be thought of as one that explicitly records the communication between two provers, a *A-prover* and a *B-prover*, whose initial assumptions are the sets A and B, respectively [6]. In our calculus, this communication is achieved through *proof judgments*. The rules of the calculus derive or use as premises one of three kinds of judgments, of the following forms:

$$A, B \vdash L\,[c], \qquad A, B \vdash t_1 \approx t_2\,[\varphi, \psi, c], \qquad A, B \vdash \bot\,[\varphi]$$

where L is an equational literal, s, t are terms, φ, ψ are quantifier-free formulas, and c is an element of a binary set of *colors*. For convenience, we name these colors A and B.[2] Although the judgments are parameterized by the literal sets A and B, these sets do not change within a proof. So from now on, we will omit $A, B \vdash$ from proof judgments. Each judgment will consist of a literal annotated with additional information (the information enclosed in square brackets). The calculus starts with the set of judgments

$$\{L\,[A] \mid L \in A\} \cup \{L\,[B] \mid L \in B\},$$

and derives new judgments according to the proof rules defined in Figure 3.

[2]It will be clear from context whether A (B) refers to the input clause set or its associated color.

Many of the rules in Figure 3 are annotated versions of those in the basic calculus of Figure 2. In addition, the rules base_A and base_B are used to provide an equality with the proper annotation. The only side condition our calculus requires is a test for whether a particular term is colorable. Although not shown here, symmetry may be applied to judgments of the form $t_1 \approx t_2\,[c]$ to conclude $t_2 \approx t_1\,[c]$. This was done for simplicity, and does not impact the relative completeness of the calculus. The A and B provers communicate through judgments of the form $t_1 \approx t_2\,[\varphi, \psi, c]$, where $t_1 \approx t_2$ is the overall equality that A and B have cooperated in proving; φ contains interpolation information provided by the A-prover for the benefit of the B-prover; ψ contains interpolation information provided in turn by the B-prover for the A-prover, possibly using information provided by the A-prover. By construction, φ is a conjunction of Horn clauses and ψ a conjunction of literals.

In addition to its limited dependence on computational side conditions, a clear advantage of our calculus for EUF is its flexibility. In particular, the user has the option of applying either the trans_A or trans_B rules for equalities between AB-colorable terms. This choice produces different interpolants, with different size and logical strengths.

To show that our calculus is interpolating, we will use the fact that all derived judgments of the form $s \approx t\,[\varphi, \psi, c]$ are partial interpolants in the sense below.

Definition 1. *A judgment J of the form $t_1 \approx t_2\,[\varphi, \psi, c]$ is a* partial interpolant *if the following hold:* (1) $A \models \varphi$; (2) $B, \varphi \models \psi$; (3) $A, \psi \models t_1 \approx t_2$; (4) t_i *is c-colorable for* $i = 1, 2$; (5) *either*

a. J is $t_1 \approx t_2\,[\varphi, \psi, A]$, and φ and ψ are AB-colorable, or

b. J is $t_1 \approx t_2\,[\varphi, t_1 \approx t_2, B]$ and φ is AB-colorable.

In the definition above, when J is $t_1 \approx t_2\,[\varphi, \psi, A]$, the formula $\varphi \wedge \neg\psi$ is an interpolant for $(A \wedge t_1 \not\approx t_2, B)$. Similarly, when J is $t_1 \approx t_2\,[\varphi, t_1 \approx t_2, B]$, the formula φ is an interpolant for $(A, B \wedge t_1 \not\approx t_2))$.

Our calculus is sound and complete for interpolation in EUF in the following sense.

Theorem 2. *The following hold:*

a. For all derivable judgments $\bot\,[I]$ we have that (i) $A \models I$, (ii) $B, I \models \bot$ and (iii) I is AB-colorable.

b. For every jointly unsatisfiable set of ground EUF literals A and B, there exists a derivation of the judgment $\bot\,[I]$, for some formula I.

The proof of soundness above uses fairly standard arguments. That of completeness relies on the fact that EUF is equality interpolating [15], i.e,, for all colorable terms s, t such that $A \wedge B \models s \approx t$, there exists an AB-colorable term u such that $A, B \models s \approx u \wedge u \approx t$. The term u, and its subterms, which may not occur in $A \cup B$, can be introduced as needed in a proof in our calculus by the refl_A and refl_B rules.

3.2 Encoding Into LFSC

Our calculus for interpolant generation in EUF can be encoded in an LFSC signature with limited reliance on computational side conditions. The encoding of the judgements and the rules is relatively straightforward. An excerpt of the encoding is provided in Figure 4. Looking at it salient features, we encode color as a base type in our signature, with two nullary term constructors, A and B. Interpolant judgments $\bot\,[\varphi]$ are encoded as the type (interpolant φ). Partial interpolants $t_1 \approx t_2\,[\varphi, \psi, c]$ are encoded as the type (p_interpolant t_1 t_2 φ ψ c). In proof terms, we specify whether an input literal L occurs in the set A or B by introducing (local) lambda variables of type (colored L c), where c is the color A or the color B, respectively. Since formulas in the literal sets A and B can be inferred from the types of the free variables in our proof terms, the sets do not need to be explicitly recorded as part of the proof judgment types.

```
formula : type          colored : (! f formula (! c color type))
term : type             interpolant : (! f formula type)
color : type            p_interpolant : (! t1 term (! t2 term
A: color                                  (! f1 formula (! f2 formula
B : color                                  (! c color type)))))
```

Figure 4: An excerpt of the LFSC signature that encodes the interpolation calculus.

3.3 Side Conditions for Testing Colorability

Proving colorability for expressions (that is, proving that the non-logical symbols in an expression occur all in A or all in B) is a common requirement for interpolant generating calculi. To prove such facts a purely declarative calculus would require $\Omega(n)$ proof rule applications for an expression E, where n is the number of symbols in E. Since LFSC's side conditions can be used for this purpose, we decided in this work to verify colorability through them.

Side conditions in LFSC are expressed in a simply typed functional programming language with minimal imperative features, and are intended to be simple enough to be verified by manual inspection. As described in [14], LFSC contains limited support for computational tests run on terms with a mutable state. In particular, each lambda variable introduced in a proof term contains 32 bit fields that are accessible to the writer of the signature. The semantics of LFSC's side condition language provides constructs for toggling (markvar C) and scrutinizing (ifmarked C C_1 C_2) these bit fields, where in both cases C is a code term that evaluates to an LF variable.

We may enforce a scheme for testing colorability using two of these bit fields, one for denoting A-colorable and the other for denoting B-colorable. Whenever a variable of type (colored L c) is introduced in our proof, we use a side condition to traverse L and mark the field specified by color c for all variables. Since LFSC supports term sharing of variables, each mark applies globally for all occurrences of that variable within our proof. To test the c-colorability of a term t, we use another side condition that traverses t and succeeds if and only if the field specified by color c has been marked for all variables occurring in t. In total, the two side condition functions account for 21 lines of functional side condition code.

4 Experimental Results

To evaluate the feasibility of our approach for producing certified interpolants, we used a version of the Cvc3 SMT solver instrumented to produce LFSC proofs from its refutations of EUF formulas. We ran experiments on a set of 25,246 unsatisfiable EUF benchmarks, all of which were a conjunction of equational literals. The benchmarks were extracted from the set of the quantifier-free EUF benchmarks in the SMT-LIB repository as follows.

In its native proof generation mode, Cvc3 produces unsatisfiability proofs with a two-tiered structure, where a propositional resolution style skeleton is filled with theory-specific subproofs of *theory lemmas*, that is, valid (ground) clauses. First, we selected all unsatisfiable EUF benchmarks that Cvc3 could solve in less than 60 seconds. We reran Cvc3 on these benchmarks with native proof generation enabled, and examined all theory lemmas within all proofs. Each theory lemma produced by Cvc3 for EUF is an equational Horn clause. For each theory lemma $e_1 \wedge \cdots \wedge e_n \rightarrow e$ we encountered, we created a new EUF benchmark consisting of the unsatisfiable formula $e_1 \wedge \cdots \wedge e_n \wedge \neg e$. We only considered unique[3] theory lemmas and

[3]Benchmarks were passed through multiple filters for recognizing duplication. It was infeasible to verify that certain theory

		Solving + Pf Gen + Pf Conv (sec)				Proof Size (KB)			Pf Check Time (sec)	
Benchmark	#	cvc	cvcpf	euf	eufi	cvcpf	euf	eufi	euf	eufi
eq_diamond	28	0.07	0.07	0.08	0.08	56.1	37.9	50.9	0.01	0.014
NEQ	2185	4.00	4.75	4.34	5.15	2765.8	2276.3	3873.8	0.27	0.524
PEQ	2252	4.65	6.27	5.81	6.91	4901.2	4256.3	7458.9	0.55	1.044
QG-loops6	2854	5.01	5.64	4.90	5.80	2446.0	1872.9	3052.2	0.21	0.418
QG-qg5	5337	9.05	10.08	10.67	9.60	4189.2	3415.2	5514.4	0.39	0.762
QG-qg6	1789	3.16	3.55	3.43	3.62	1970.1	1493.8	2669.7	0.172	0.368
QG-qg7	7860	16.96	22.77	23.07	25.96	19161.2	18843.2	35527.7	2.544	5.024
SEQ	2941	6.04	7.61	7.11	7.86	5517.8	4315.4	6926.6	0.52	0.948
	25246	48.95	60.74	59.40	64.98	41007.4	36511.1	65074.2	4.666	9.102

Figure 5: Cumulative Results for average of Runs $1 \ldots 5$, grouped by benchmark class. Columns 3 through 6 give Cvc3's (aggregate) runtime for each of the four configurations. Columns 7 through 9 show the proof sizes for each of the three proof-producing configurations. Columns 10 and 11 show LFSC proof checking times for the **euf** and **eufi** configurations.

whose corresponding proof contained at least five deduction steps.

We collected runtimes for the following four configurations of our instrumented version of Cvc3:

cvc: Default, solving benchmarks but with no proof generation.
cvcpf: Solving with proof generation in Cvc3's native format.
euf: Solving with proof generation, translation to LFSC format for EUF.
eufi: Solving with proof generation, translation to LFSC format for interpolant generation in EUF.

In each configuration, Cvc3's decision procedure for EUF was used when solving. We ran each configuration five times and took the average for all runs. For each benchmark used by configuration **eufi**, the sets A and B were determined by randomly placing $\frac{k}{6}$ of the benchmarks into set A and the rest in B on run k for $k = 1 \ldots 5$. This did not affect the difficulty of solving, and we believe provided a sufficient measurement of the effectiveness of our interpolant generation scheme.

We measured total time to solve all benchmarks grouped by benchmark class. These results are shown in Figure 5. In this data set, proof generation came at a 25% overhead with respect to solving. Converting proofs to the LFSC format required very little additional overhead. In fact, LFSC proofs in the **euf** configuration were generally smaller in size (number of bytes) than Cvc3's native proofs, thus slightly reducing proof generation times. LFSC proofs in the interpolanting calculus were nearly twice as large as non-interpolating proofs. The difference in proof size can be attributed to differences in syntax between the two calculi, as well as additional information added to the header of interpolanting proofs for specifying A and B.

As expected from previous work [12], proof checking times using the LFSC checker were very small with respect to solving times. Proof checking times without interpolant generation were about an order of magnitude faster than solving times. Since additional information is inferred within judgments in the **eufi** configuration, proof checking took approximately twice as long with interpolant generation as without. Proof checking times in the **eufi** configuration were a factor of 5 faster than solving times.

We estimate the time to produce uncertified interpolants by measuring solving and proof generation without proof checking (configuration **cvcpf**). Overall, configuration **eufi** shows a 22% overhead with

lemmas were symbolic permutations of others. However, by visual inspection, we believe that such cases were rare.

respect to **cvcpf**, indicating that the generation of certified interpolants is practicably feasible with high-performance SMT solvers.

5 Conclusion and Future Work

We have introduced a method of generating interpolants by type inference within the trusted core of a proof checker for LFSC. Our experiments show that this method is efficient and practical for use with high performance solvers. Overall, interpolant generation in LFSC can be performed 5.38 times faster than solving time on average. Our method is based on a novel calculus for interpolant generation in EUF. By performing certain colorability tests during a proof lifting phase, we are able to simplify the amount of side conditions required during proof checking.

We plan to extend our method to ground EUF formulas with an arbitrary Boolean structure by incorporating an interpolating version of the propositional resolution calculus. Future work includes instrumenting Cvc4, the forthcoming successor of Cvc3, to output interpolating proofs for arbitrary ground EUF formulas as well as in combination with linear real arithmetic. Cvc4 currently supports the theories of EUF and arithmetic, and preliminary work has been planned for a proof generating infrastructure that aims at minimizing performance overhead.

The flexibility of our proof lifting phase allows the user to make various decisions when assigning colors to a congruence graph. In previous work [6], a coloring strategy is used that aims to minimize interpolant size. We plan to explore other strategies with desired properties in mind, including logical strength, which has been explored in recent work for the propositional case [5]. In applications such as interpolation-based model checking [9], logical strength is desirable for an interpolant since a stronger interpolant may produce tighter over-approximations of the reachability relation.

We also plan to explore other applications of this framework related to automated verification, including interpolant generation procedures where additional constraints are considered, such as relative strength with respect to other interpolants. By encoding more restrictive calculi in the LFSC framework, other properties of produced interpolants may be certified by construction.

References

[1] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, Lecture Notes in Computer Science. Springer, 2007.

[2] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *Proceedings of IJCAR'10*, volume 6173 of *LNCS*, pages 384–399. Springer, 2010.

[3] R. Bruttomesso, S. Rollini, N. Sharygina, and A. Tsitovich. Flexible interpolation with local proof transformations. In *Proceedings of ICCAD'10*, pages 770–777. IEEE, 2010.

[4] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *Proceedings of TACASA'08*, Lecture Notes in Computer Science. Springer, 2008.

[5] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proceedings of VMCAI*, pages 129–145, 2010.

[6] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground interpolation for the theory of equality. In S. Kowalewski and A. Philippou, editors, *Proceedings of TACAS'09*, volume 5505 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2009.

[7] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[8] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proceedings of TACAS'06*, pages 459–473, 2006.

[9] K. McMillan. Interpolation and SAT-based model checking. In W. A. H. Jr. and F. Somenzi, editors, *Proceedings of CAV'03*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[10] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[11] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In J. Giesl, editor, *Proceedings of RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[12] A. Reynolds, L. Hadarean, C. Tinelli, Y. Ge, A. Stump, and C. Barrett. Comparing proof systems for linear real arithmetic with LFSC. In A. Gupta and D. Kroening, editors, *Proceedings of SMT'10*, 2010.

[13] A. Reynolds, C. Tinelli, and L. Hadarean. Certified interpolant generation for EUF. Technical report, Department of Computer Science, The University of Iowa, June 2011.

[14] A. Stump. Proof Checking Technology for Satisfiability Modulo Theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

[15] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2005.