The Cooperating Proof Calculus: Comprehensive Proofs for an SMT Solver

Andrew Reynolds[1(✉)], Hans-Jörg Schurr[1,2,3], Haniel Barbosa[4], Ofec Israel[5], Jibiana Zoe Jakpor[6], Hanna Lachnitt[6], Abdalrhman Mohamed[6], Aina Niemetz[6], Mathias Preiner[6], Yoni Zohar[5], Robert Jones[7], Clark Barrett[6], and Cesare Tinelli[1]

[1] The University of Iowa, Iowa, USA (✉)andrew-reynolds@uiowa.edu
[2] KU Leuven, Leuven, Belgium
[3] Vrije Universiteit Brussel, Brussels, Belgium
[4] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[5] Bar-Ilan University, Ramat Gan, Israel
[6] Stanford University, Stanford, USA
[7] Amazon Web Services, Seattle, USA

Abstract. We present the Cooperating Proof Calculus (CPC), an evolving set of proof rules encompassing all inferences used in the mainstream theories of the SMT solver cvc5. CPC consists of 585 proof rules, which are formalized in 8025 lines of definitions in the logical framework Eunoia. Eunoia proofs are independently checkable by the proof checker Ethos. This paper gives a detailed summary of CPC, surveying its proof rules over its major components. Having instrumented cvc5 to generate CPC proofs in Eunoia, we show that the solver is capable of generating fine-grained CPC proofs, with no proof holes, for all benchmarks in the SMT library except those in logics with floating-point arithmetic, which are currently not supported. This results in more than 900 million proof steps using 427 unique proof rules. We also discuss ongoing work in the proof assistants Lean and Isabelle to verify the correctness of CPC.

1 Introduction

A longstanding challenge in Satisfiability Modulo Theories (SMT) is to efficiently generate fine-grained proofs that capture the entire scope of reasoning performed by modern SMT solvers. These solvers are large,[8] highly complex, and use a multitude of techniques that include preprocessing, rewriting, and specialized subsolvers for a large, and still growing, number of theories.

Ensuring the correctness of such solvers has grown in importance and has become highly critical in many current applications of SMT, including security and safety analysis of software [13]. While it is impractical to directly verify the correctness of modern SMT solvers, a viable alternative is to instrument the solver to generate externally checkable *proof certificates*, which typically consist

[8] The source code of cvc5 is approximately 363k lines of C++.

of text files encoding a formal proof of correctness of the solver's response. Such certificates can then be checked by a separate trusted proof checking tool [2,49].

Largely due to the diversity of techniques used by modern SMT solvers, there is presently no standard proof system that easily captures all those techniques. In this paper, we define an evolving proof calculus, the Cooperating Proof Calculus (CPC), designed to capture the reasoning techniques currently used by cvc5. We rely on the proof checker Ethos [44], which was developed primarily for the purposes of checking proofs emitted by cvc5. Ethos uses the meta-framework *Eunoia*, whose syntax is an extension of SMT-LIB version 2.7.[9] This means that Ethos has no built-in proof rules. Rules are instead defined by users in a Eunoia *signature*, which is provided as input in textual form. As new solving techniques are added or changed, a proof signature may be updated to reflect these changes.

The Cooperating Proof Calculus is formalized as a Eunoia signature and covers all safe features[10] of the SMT solver cvc5. We show in an empirical evaluation on the entire SMT-LIB benchmark library that cvc5 generates fully fine-grained proofs that are checkable by Ethos using this signature. As a result, the trusted computing base (TCB) of cvc5 shrinks from its more than 360k lines of C++ code to one consisting of the external proof checker Ethos (about 10k lines of C++) and the CPC signature (about 8k lines of Eunoia).

Related Work. Proof support for propositional SAT solvers is well developed, and includes formats like DRAT [50], LRAT [20], and variations of them [21,31].

Existing proof formats for SMT include Alethe [47] and LFSC [49], which are generated by SMT solvers such as veriT [14], CVC4 [7], and cvc5 [5,30]. In contrast to Alethe which has a pencil-and-paper formalization, the Cooperating Proof Calculus is mechanized using the logical framework Eunoia. A key advantage is that CPC allows agile development, where proof rules can be added or modified based on the needs of the SMT solver, in contrast Alethe rules evolve more slowly due to its on paper formalization. In contrast to LFSC, the CPC format is based on the concrete syntax of SMT-LIB and has improved built-in support for evaluation on SMT-LIB literals like bit-vectors, strings and variadic operators. Another proof format for SMT is RESOLUTE [24], supported by the SMT solver SMTInterpol [16]. While simpler than Alethe, RESOLUTE also covers a smaller number of background theories. The eDRAT approach [23] focuses on the extension of the DRAT format to include facts derived from theory reasoning, but without justifications, thus requiring proof checking to re-derive these facts. Preprocessing and rewriting are not supported.

In previous work [5], cvc5 was instrumented to produce proofs for many of its theories using a robust and extensible infrastructure. Notably, only 83% of generated proofs had no holes, where some theory solving techniques and preprocessing passes had not yet been made proof producing. This work also included

[9] Ethos and Eunoia are in active development. We describe their current state here.

[10] A *safe feature* is one that is not activated by an option marked as "expert", or is marked as not proof supporting. Note that default builds of cvc5 enable a few techniques that are not safe features, which we detail in Section 5.

an initial corpus of proof rules and an internal proof checker within cvc5's code base. We build upon this work by fully instrumenting cvc5 so 100% of proofs are complete and by providing a formal external definition of these proof rules in the logical framework Eunoia. An external formalization of cvc5's proof system serves as a way of enabling independent audits of cvc's proofs (for instance, by checking them with Ethos), as well as providing the basis for translation of cvc5 proofs into proof assistants such as Lean 4 [34], Isabelle/HOL [36], and Dedukti/Lambdapi [3]. Ongoing work has focused on the translation of cvc5 proofs to Lean [33], Isabelle [30] and Dedukti/LambdaPi [18, 19, 22]. These translations are based on CPC either directly, as the one to Lean, or via a translation to Alethe.

Contributions. This paper focuses on a detailed overview of the Cooperating Proof Calculus. The logical framework Eunoia and the proof checker Ethos are subjects of ongoing work, and are described only at a high-level here. In detail, our contributions are as follows:

- We present the Cooperating Proof Calculus, a set of 585 proof rules encompassing all inferences used in mainstream theories of the SMT solver cvc5. These rules are currently formalized as 8025 lines of Eunoia definitions and can be checked by the proof checker Ethos.
- We provide implementation details on the formalization of CPC, including an efficient algorithm for checking Boolean chain resolution steps, efficient methods for term normalization and extensions of Eunoia to support proof rules over generic datatypes.
- We discuss an experimental evaluation showing that cvc5 efficiently generates fine-grained proofs over all SMT-LIB logics (apart from logics with floating point) without proof holes. In total, cvc5 generates roughly 913 million proof steps over 427 unique proof rules for this set.
- We describe initial verification efforts to show the correctness of the proof rules of CPC.

We overview many of the components of CPC, including its core rules, rules for Boolean resolution, rewrite rules, as well as rules used by theory-specific solving procedures. Throughout, we describe the novel aspects of the implementation and formalization details.

2 Proofs in Eunoia, Checked by Ethos

In this section, we give a high-level overview of Eunoia and the proof checker Ethos. This description is not intended to be comprehensive, but rather to give enough detail to understand the rules of CPC, described later in Section 3.[11]

Eunoia is a logical framework tailored to the specification of theory symbols and proof rules for SMT solvers. Its syntax is an extension of Version

[11] A more thorough description of Eunoia can be found in the Ethos user manual [43].

2.7 of the SMT-LIB standard but is expressed in terms of typed higher-order logic, as opposed to many-sorted first-order logic. It includes commands for declaring dependently-typed symbols (`declare-parameterized-const`), proof rules (`declare-rule`) and computations (`program`). The commands `assume` and `step` are used to specify proofs over the declared rules in the natural deduction style of Alethe proofs [47].

A Eunoia file, or *signature*, can be used to define type constants, term constants, and proof rules for them. Eunoia provides only a minimal set of built-in types and constants: `Bool`, `true`, `false`, `->` (the function type constructor) and `Type` (the type of all types). All other symbols in SMT theories, standard or solver-specific, can be declared by the user in a signature.

Eunoia has also a built-in set of *values*, distinguished literals such as numerals, decimals and so on, with a syntax conforming to SMT-LIB. Correspondingly, it also includes built-in computational operators over those values. Such operators are prefixed by `eo::`, and include operators like addition and multiplication (`eo::add`, `eo::mul`) over arbitrary precision numerals and rationals, as well as binary values. Eunoia also supports common Boolean operators; string-like operators, e.g., concatenation and extract (`eo::concat`, `eo::extract`); as well as conversions between basic types. This rich set of operators is useful for defining user types and proof rules. An example declaration is the CPC definition of the bit-vector concatenation operator, whose type involves `eo::add`:

```
(declare-const Int Type)
(declare-const BitVec (-> Int Type))
(declare-parameterized-const concat ((m Int :implicit) (n Int :implicit))
  (-> (BitVec m) (BitVec n) (BitVec (eo::add m n))))
```

The constant `concat` takes two implicit (i.e., inferrable) integer arguments `m` and `n` and two bit-vector arguments of width `m` and `n`, respectively. Its return type, `(BitVec (eo::add n m))`, is the bit-vector type whose width is the result of adding `m` and `n`. For example, `(concat a b)` has type `(BitVec 7)` if `a` and `b` are of type `(BitVec 2)` and `(BitVec 5)`.

2.1 Proof Rules and Proof Steps

A proof signature uses the `declare-rule` command to declare proof rules. An example proof rule is the following symmetry rule for equality:

```
(declare-parameterized-const = ((T Type :implicit)) (-> T T Bool))
(declare-rule symm ((T Type) (s T) (t T)) :premises ((= s t))
 :conclusion (= t s))
```

The rule is parametrized by a type `T`, and two values `s` and `t` of that type. These parameters are treated as implicit inputs to the proof rule, which expects a premise of the form `(= s t)` (indicated by the keyword `:premises`) and derives the conclusion `(= t s)` (indicated by the keyword `:conclusion`).

When feeding CPC proofs to Ethos, one must first include the CPC proof signature, which contains rules like the one above. The CPC proof proper contains a preamble of constant declarations and definitions, followed by a list of `assume` and `step` commands, which are used for describing initial assumptions and individual steps of the proof respectively. This is illustrated by the example below, where the `include` command loads a signature provided in text file `"Cpc.eo"` that contains the definition of `symm` from above.

```
(include "Cpc.eo")
(declare-const Int Type)
(declare-const a Int) (declare-const b Int)
(assume @p0 (= a b)) ; derives formula (= a b)
(step @p1 (= b a) :rule symm :premises (@p0)) ; derives (= b a) from (= a b)
```

Each `assume` or `step` command takes as first argument a fresh identifier which can be used subsequently to reference the formula derived by that command. The second argument is the derived formula itself. `step` commands additionally require the name of the rule applied by the command as well as a list of premises and any additional arguments for that rule. Proof checking for a `step` command succeeds if (i) the proof rule applied to the given premises and arguments derives a formula, i.e., a term of type `Bool`, and (ii) the formula provided in the command, if any, matches the rule's conclusion. Typical SMT proofs are *refutations*, deriving `false` from a given set of assumptions $A_1, \ldots, A_n$.

2.2 Proofs with Local Assumptions

Eunoia proof rules can be marked as taking an assumption. Such rules are used to consume local assumptions in a CPC proof. The following rule `scope` is part of CPC, and is analogous to the implication introduction rule of natural deduction:

```
(declare-rule scope ((F Bool) (G Bool))
  :assumption F :premises (G) :conclusion (=> F G))
```

Intuitively, from a previously introduced *local* assumption `F` and a premise `G` (possibly derived from `F` and other formulas), it concludes `(=> F G)`. The following proof shows an example usage of `scope`:

```
(declare-const Int Type)
(declare-const a Int) (declare-const b Int)
(assume-push @p0 (= a b))
  (step @p1 (= b a) :rule symm :premises (@p0))
(step-pop @p2 (=> (= a b) (= b a)) :rule scope :premises (@p1))
```

The syntax of the commands `assume-push` and `step-pop` is identical to that of the commands `assume` and `step`. The first command introduces a local assumption in the proof and the second removes it. The command `step-pop` removes the most recently introduced local assumption. This style of proofs is highly useful for representing proofs of theory lemmas in CDCL(T) theory solvers, where a lemma is proved by contradiction by a local proof that derives `false` from an assumption consisting of the negated lemma.

2.3 Computations

Eunoia follows the pragmatic approach of frameworks like Dedukti and LFSC which provide support for the specification of computational side conditions for derivation rules. Similar to LFSC, Eunoia includes a sublanguage of programs to encode side conditions. More generally, it also allows program applications to occur in arbitrary terms, as seen earlier in the declaration of the string `concat` constant, where the output type `(BitVec (eo::add m n))` contains an application of the built-in program `eo::add`.

CPC takes full advantage of this feature: many of its proof rules rely on computations to express side conditions and to construct the conclusions of proof steps. Similarly to Dedukti, a program is a declarative definition by cases of a pure function f. The definition's body is an ordered list of pairs consisting of a *pattern* term p and a corresponding *return* term r over the variables of p. Each pattern term is the application of f to some arguments. The corresponding return term denotes the return value of that application. Pattern variables are declared globally for the entire body in a list typed parameters. A program application $(f\ t_1 \cdots t_n)$ evaluates to the return term of the first pattern in the body that pattern matches $(f\ t_1 \cdots t_n)$. Any parameters occurring in the return term are instantiated by the match. Program evaluation is based on eager application semantics: the arguments of an application are fully evaluated before evaluating the application itself. Here is an example program that computes the sum of all integers from 0 to n:[12]

```
(declare-const Int Type)
(declare-consts <numeral> Int)
(program $sum_n ((n Int)) :signature (Int) Int (
  (($sum_n 0) 0)
  (($sum_n n) (eo::add ($sum_n (eo::add n -1)) n))))
```

In this example, the command `declare-consts` assigns type `Int` to all (signed) numerals, collectively denoted by built-in symbol `<numeral>`. The `program` command defines program `$sum_n` by cases, first specifying a list of parameters to be used as pattern variables, and then providing the program's type signature (introduced by the keyword `:signature`). This is followed by two cases for the body. The first simply states that `($sum_n 0)` evaluates to `0`. The second case specifies that the pattern `($sum_n n)` evaluates to the result of recursively evaluating the application `($sum_n (eo::add n -1))` and then adding the value of `n` to it.

Note that it is possible to define non-terminating programs. In fact, the program above does not terminate for negative numerals. It is up to the user to make sure that defined programs are terminating. It is also possible for a program application to get *stuck*, that is, not to fully evaluate, because no pattern in the body matches the application's arguments or, recursively, because the return term of the matching pattern is itself stuck.

In Eunoia, programs are meant to be evaluated during proof checking. A proof fails if it contains anywhere a computational term that does not fully eval-

[12] We use the convention of prefixing program names by `$`.

uate. A formal semantics of Eunoia defining its type system and the operational semantics of commands and programs is under development.

3 The Cooperating Proof Calculus

This section gives an overview of the Cooperating Proof Calculus (CPC). The definition of CPC currently contains (*i*) a complete definition of the type rules of all 177 functions symbols native to the SMT solver cvc5, and (*ii*) a set of 585 proof rules. The proof rules are comprehensive for the safe features of cvc5. Overall, the definition of CPC is written in 8025 lines of Eunoia.

3.1 Rewrite Rules

Term simplification by (equivalence-preserving) rewriting is a key component of SMT solvers. It is critical for performance as described in several works [38, 42, 46]. The majority of our proof rules can be expressed as rewrite rules.

In previous work, Nötzli et al. [37] defined a domain-specific language called RARE for representing these rules. RARE only allows the definition of purely declarative rules, meaning it is not possible to use arbitrary computational side conditions. The RARE language initially focused on the theory of strings. Further work defined many of the bit-vector rewrites used by cvc5, and proposed a framework for verifying them [29]. We developed a translation from RARE rules to Eunoia, which involved simplifications to the rules to ensure they are expressible in Eunoia, as well as changes to address minor differences between the semantics of RARE and Eunoia.

In total, CPC contains 424 rewrite proof rules over 8 theories[13], divided as follows: 164 for the theory of strings, 135 for the theory of bit-vectors, 38 for the theory of arithmetic, 19 for the theory of finite sets, 6 for the theory of arrays, 8 for arithmetic/bit-vector conversions, 41 for Boolean terms, and 13 for the remaining core theory symbols of SMT-LIB (e.g., `=`, `distinct`, and `ite`).

3.2 Clausal Normal Form and Boolean Reasoning

A key component to preprocessing in SMT solving is the conversion of the input formula to clausal normal form. This typically uses a Tseitin transformation, which introduces new Boolean variables that abstract subformulas of the input. Our rules for CNF conversion do not explicitly introduce new variables, but rather treat the formulas themselves as literals. In total, CPC contains 21 rules for the CNF conversion, covering all of the Boolean connectives in SMT-LIB.

An example is the following rule, which takes an application of `xor` as input and derives the valid clause corresponding to one case of the conversion of `xor`, namely that the `xor` application is true, or its first argument is false, or its second argument is true.

[13] There are no rules specific to the combination of theories. Due to the modularity of Nelson–Oppen theory combination, cvc5 largely reduces combination to equality-splitting lemmas, which have a trivial justification.

```
(declare-rule cnf_xor_neg1 ((F1 Bool) (F2 Bool))
  :args ((xor F1 F2)) :conclusion (or (xor F1 F2) (not F1) F2))
```

CPC has 28 additional rules for Boolean reasoning, which include many of the typical rules for natural deduction such as modus ponens, and-elimination, and-introduction, and so on.

3.3 Binary and Chain Resolution

In total, CPC models three variants of the resolution rule used for propositional reasoning in SMT solvers. The first is the standard rule for binary resolution, taking exactly two premises, a polarity and a pivot formula. Note that since the latter two cannot be inferred from the premises, they need to be provided explicitly as additional arguments (in the attribute `:args`).

```
(declare-rule resolution ((C1 Bool) (C2 Bool) (pol Bool) (L Bool))
  :args (pol L) :premises (C1 C2) :conclusion ($resolve C1 C2 pol L))
```

The conclusion of `resolution` is generated by the program `$resolve` that produces the resolvent of clauses `C1` and `C2` over pivot `L`, provided `L` occurs with polarity `pol` in `C1` and opposite polarity in `C2`.

The remaining two rules, `chain-resolution` and `chain-m-resolution` (i.e. chain multiset resolution), take a list of premises, and a list of pivot formulas and polarities to resolve. The latter rule also takes a target conclusion and verifies that it is equivalent modulo multiset reasoning to the result of computing the resolution over the list of steps, i.e., it allows implicitly removing duplicates.

Modern SAT solvers typically generate proofs using the DRAT and LRAT formats [50], with reverse unit-propagation (RUP) steps capturing without loss of generality both resolution and RAT reasoning. RUP steps translate naturally to `chain-m-resolution`, so these steps from LRAT proofs can be directly translated to CPC. Since modern SAT solvers natively produce only RUP steps [11] and there is a polynomial translation of RAT inferences to resolution [26], CPC can capture the reasoning of modern SAT solvers when they are used as subcomponents of an SMT solver.

Now, chain resolution is often a bottleneck for SMT proof checking, especially for problems with bit-vectors and for large problems using the QF_UF logic of SMT-LIB. For this reason, CPC defines an efficient program for computing the chain resolvent of clauses $C_1, \ldots, C_n$ given pivots $\ell_1 \ldots \ell_{n-1}$. At a high level, the program does not compute intermediate resolvents—obtained by resolving C_1 with C_2 to generate $C_{1,2}$, then resolving $C_{1,2}$ and C_3 and so on. Instead, it computes the contribution of each C_i to the final resolvent, which is the result of removing $\neg\ell_i, \ell_{i+1} \ldots \ell_n$ from C_i. This removal uses an efficient built-in operator in Eunoia implementing multiset difference.[14] For `chain-m-resolution`, we additionally use a built-in operator to determine the equality of two clauses when

[14] The Ethos implementation of this operator is optimized at a low-level, and minimizes the number of intermediate terms that must be allocated.

seen as multisets. In total, the definition of this proof rule and the programs it uses is 66 lines of Eunoia code. This rule effectively performs resolution, factoring and reordering of literals in the resultant clause all at once.

3.4 Equality and Uninterpreted Functions

Congruence closure is a core algorithm in SMT, used to reason about equality and uninterpreted functions. The CPC signature includes rules for reflexivity (`refl`) and transitivity (`trans`) of equality, and for symmetry of (dis)equality (`symm`). The transitivity rule can take an arbitrary number of premise equalities.

The signature contains four variants of the congruence rule. The first variant (`cong`) is used for congruence over a function that expects a fixed number of arguments, which includes all instances of user-declared uninterpreted functions. The second variant (`ho_cong`) is used for higher-order congruence and takes an additional premise `(= f g)` equating the *functions* `f` and `g` in a conclusion of the form `(= (f s) (g t))`. The remaining two variants (`nary_cong` and `pairwise_cong`) handle the cases of congruence over variadic operators (e.g., `or`) and pairwise operators (e.g., `distinct`), respectively.

Distinctness Constraints. The CPC signature has two rules (`distinct_true` and `distinct_false`) for reasoning about applications of the `distinct` predicate symbol, to handle cases where the application rewrites to `true` or `false`, respectively. Another rule (`distinct_elim`) reduces applications of `distinct` to more than two arguments to a conjunction of disequalities.

CPC contains two proof rules for reasoning about distinct *values* of a given type. The rule `distinct_values` concludes the disequality between two terms that denote distinct values of a given type, such as `1` and `4`. More precisely, the values of basic SMT types such as integers, reals, strings, bit-vectors are atomic terms that can be compared for distinctness directly. Container-like types such as sequences, sets, and algebraic datatypes, define distinct values in according to more elaborate criteria such as identity of normalized forms or extensionality. For example, showing two set values to be distinct requires showing that one element term in the first set is not contained in the second (or vice versa). Implemented as a recursive program, this criterion allow us to reason about the distinctness of, e.g., sets of sets of integers. The rule `distinct_card_conflict` concludes that a `distinct` constraint evaluates to `false` based on the cardinality of the types involved. For example, it can be used to conclude that the application of `distinct` to three terms of type `Bool` is equivalent to `false`.

3.5 Real and Integer Arithmetic

The definition of CPC allows arithmetic symbols to be overloaded, namely, many operators such as `+` and `*` are applied to either integer or real terms. Many of the arithmetic rules of CPC apply to arithmetic relations over integer or real terms.

CPC has rules for linear arithmetic which can be used to build proofs that resemble refutations via Farka's lemma. Specifically, we provide `arith_sum_ub` for taking the sum of two arithmetic relations, and rules `arith_mult_pos` and `arith_mult_neg` for multiplying relations by positive and negative coefficients. A simplified version of the third rule is:

```
(program $arith_rel_inv ((a Int) (b Int))
  :signature ((-> Int Int Bool) Int Int) Bool (
  (($arith_rel_inv = a b) (= a b))
  (($arith_rel_inv > a b) (< a b))
  (($arith_rel_inv >= a b) (<= a b))))

(declare-rule arith_mult_neg ((r (-> Int Int Bool)) (a Int) (b Int) (m Int))
  :args (m (r a b))
  :conclusion (=> (and (< m 0) (r a b)) ($arith_rel_inv r (* m a) (* m b))))
```

Additionally, CPC contains rules `int_tight_ub` and `int_tight_lb` for integer reasoning, respectively rounding upper and lower bounds to the nearest integer. Finally, the rule (`arith_trichotomy`) concludes for any two arithmetic terms of the same type that one is either smaller than, greater than, or equal to the other.

A key component of arithmetic rewriting is the normalization of polynomials. Rule `arith_poly_norm` concludes that two terms are equal if they reduce to the same normal form. Rule `arith_poly_norm_rel` concludes that two arithmetic relations are equivalent if they are over terms whose difference normalizes to the same term, possibly multiplied by a positive or non-zero constant. In total, the signature defining the computations that confirm whether two polynomials normalize to the same term is 122 lines of Eunoia. To normalize polynomials, our signature defines a program roughly comparable to insertion sort based on a built-in term ordering operator (`eo::cmp`). The ordering used by `eo::cmp` is established dynamically (but deterministically) based on the the order in which terms are encountered over the course of the proof, often meaning that the sorting program achieves best case complexity.

Non-linear Arithmetic. The non-linear solver in cvc5 uses a portfolio of techniques for non-linear arithmetic, including cylindrical algebraic coverings [27] and incremental linearization [17]. The CPC contains proof rules for the latter only as proofs for the former are a subject of ongoing work.

CPC contains three rules that are specific to incremental linearization. The rule `arith_mult_sign` is used to reason about the signedness of a monomial. The rule `arith_mult_abs_comparison` is used to reason about the relative magnitude of monomials. The rule `arith_mult_tangent` is used to infer a tangent plane for a multiplication term centered on a concrete point, as described in [17]. These rules capture the lemma schemas described by Kremer et al. [28] and Reynolds et al. [45]. Further arithmetic functions (including `/`, `div`, `mod`, `is_int`, `to_int`, `abs`), are handled by rule `arith_reduction` which derives a formula which reduces them to core arithmetic functions.

3.6 Bit-vectors

CPC's subcalculus for bit-vectors is primarily composed of a program for bit-blasting. The proof rule `bv_bitblast_step` defines a single-step reduction of a bit-vector term to a propositional formula by means of a 540-line Eunoia program.

Additionally, the signature for bit-vectors contains five proof rules for performing rewrites that cannot be expressed declaratively. This includes the elimination of operators such as bit-vector `repeat`, which requires a recursively defined side condition. The bit-vector signature also contains two proof rules, `bv_poly_norm` and `bv_poly_norm_eq`, that are analogous to polynomial normalization for arithmetic, and can be used to normalize bit-vector terms over arithmetic operators.

3.7 Arrays

The array theory solver in cvc5 is based on a procedure that lazily instantiates read-over-write axioms and reasons about the extensionality of arrays. The CPC signature for the theory of arrays contains four rules. The first three are variants of the read-over-write axiom for arrays. The final rule defines arrays extensionality:

```
(declare-rule arrays_ext ((T Type) (U Type) (a (Array T U)) (b (Array T U)))
  :premises ((not (= a b)))
  :conclusion (not (= (select a (@diff a b)) (select b (@diff a b)))))
```

Rule `arrays_ext` witnesses the disequality of two arrays with the *difference* term `(@diff a b)`[15] which denotes an index, if any, at which the arrays `a` and `b` are different, or is uninterpreted otherwise.

3.8 Strings

To solve problems in the theory of strings and regular expressions, cvc5 relies on multiple techniques in addition to using 164 rewrite rules (as discussed in Section 3.1), the proof signature for this theory has 40 proof rules that cover a diverse set of techniques, making its proof signature one of the most extensive among all theories supported by cvc5.

The core procedure in cvc5 for solving string equations is based on constructing a normal form for concatenation terms, while exploring possible length arrangements between component strings [9,32]. It is captured by nine rules. The rules `string_length_pos` and `string_length_non_empty` are used to infer the positivity of string length. The rule `concat_eq` is used to strip away common prefixes (alternatively, suffixes) of strings on either side of an equality. The rule `concat_unify` is used to infer an equality between a prefix (alternatively, suffix) of equal length from both sides of a string equality. The rules `concat_lprop` and `concat_cprop` infer when a prefix (alternatively, suffix) is known to be longer than the prefix on the other side of the equality. The rules `concat_csplit` and `concat_split` infer a splitting arrangement between the prefix (alternatively, suffix) of a string equality.

[15] We use the convention that internal function symbols are prefixed by `@`.

Finally, the rule `string_decompose` is used to infer that a string term is equal to the concatenation of two of its parts based on a given position.

cvc5 extends reasoning about word equations to regular expressions using a lazy unfolding of positive regular expression membership (`re_unfold_pos`), and a reduction to bounded quantifiers for negative regular expression membership (`re_unfold_neg`). An optimized version of the latter rule is included for regular expressions having a fixed length (`re_unfold_neg_concat_fixed`). Two additional rules are used to combine membership constraints (`re_inter` and `re_concat`).

CPC contains an efficient evaluator for regular expressions based on symbolic derivatives used in the definition of proof rule `str-in-re-eval` [25]. The evaluator implements Brzozowski's derivatives [15], which can be seen as a lazy on-the-fly matching of a string against a regular expression. Several optimizations are included when computing derivatives, in the form of algebraic simplification rules that are applied during the construction, e.g., flattening and normalizing union and intersection, as well as flattening concatenation.

CPC furthermore contains a proof rule `str-in-re-consume` for partially evaluating regular expression memberships. This rule uses a side condition that simplifies a membership predicate of the form $s \in R$, based on partially consuming a prefix or suffix of the string s, in particular when the derivative of the regular expression does not introduce case splitting.

Similar to the reduction rule for arithmetic, CPC also has a `string_reduction` rule for handling all extended string functions (e.g. `str.contains`, `str.replace`, `str.indexof`) supported by cvc5. In short, this proof rule defines a predicate for each extended function whose semantics is defined in terms of simpler functions. This includes all SMT-LIB standard functions, as well as some non-standard extensions, e.g. `str.to_lower`, `str.to_upper`, `str.rev`.

CPC also contains two proof rules to reason about entailed bounds on arithmetic terms in the theory of strings, as described by Reynolds et al. [42] (Section 3). In detail, rule `arith-string-pred-entail` is used to reason about the positivity of an arithmetic term in the theory, based on the assumption that strings have non-negative length. Rule `arith-string-pred-safe-approx` uses approximation techniques from Reynolds et al. [42] to show that a predicate over arithmetic terms from the theory of strings is entailed.

The solver for the theory of strings uses several sophisticated rewrite rules for strings. While 164 of these are expressed in RARE, the proof signature for strings contains an additional 13 rewrite rules that require computational definitions to perform operations like eliminating regular expression loop terms, reasoning about multiset containment between strings [42], as well as defining evaluators for extended regular expression functions from SMT-LIB, e.g. `str.replace_re_all`.

3.9 Datatypes

For reasoning about inductive datatypes, CPC defines 10 proof rules based on cvc5's decision procedures [8, 40] and applying generally to any user-defined datatype. For example, the rule `dt-split` states that the top symbol of a datatype term is one of the constructors of that datatype, which depends on the list of

constructors for the datatype in question. We use the feature of Eunoia for extracting the list of constructors (`eo::dt_constructors`) for a given datatype. A simplified version of this rule is defined recursively over the list of constructors:

```
(program $mk_dt_split ((D Type) (x D) (T Type) (c T) (xs eo::List :list))
  :signature (eo::List D) Bool (
  (($mk_dt_split eo::List::nil x) false)
  (($mk_dt_split (eo::List::cons c xs) x) (or (is c x) ($mk_dt_split xs x)))))

(declare-rule dt_split ((D Type) (x D))
  :args (x) :conclusion ($mk_dt_split (eo::dt_constructors (eo::typeof x)) x))
```

In detail, given a datatype term x, we extract the list of its constructors using `eo::dt_constructors`, which returns a Eunoia list (`eo::List`). If x is not of datatype type, this call will fail. The recursive side condition `$mk_dt_split` traverses this list, constructing disjunctions of applications of *tester* predicates `(is c x)`, i.e. the predicate that is true if and only if x has c as its top symbol. For example, this proof rule proves `(or (is node x) (or (is leaf x) false))` where x is a datatype of type whose constructors are `node` and `leaf`.

In addition to the above rule for splitting on the top symbol of a datatype term (`dt-split`), CPC contains a rule for relating datatype testers with constructor terms (`dt-inst`), a rule for reasoning about injectivity of constructors (`dt-cons-eq`), and a rule for reasoning about the acyclicity of datatype terms (`dt-cycle`). The remaining six rules are used to collapse or eliminate selectors, testers, and updater terms.

The proof signature handles parametric datatypes as well as tuples, which can be seen as a subclass of datatypes with a single constructor.

3.10 Sets

The CPC contains definitions of all set operators supported in the safe version of cvc5, including set intersection, union, difference, subset and so on. cvc5 uses a procedure based on upwards and downwards propagation of set membership [4]. These propagation steps can be expressed with 19 rewrite rules.

These rules are included in the set of rewrites described in Section 3.1. the proof signature for the theory of sets also contains rules that cannot be described as declarative rewrite rules, mainly the injectivity of singleton sets (`sets_singleton_inj`), a rule for evaluating set functions on concrete set values (`sets-eval-op`), and the introduction of a witness for set disequalities (`sets_ext`). For the latter rule, we introduce a witness term using a distinguished indexed function symbol, which is analogous to the rule for extensionality for arrays `arrays_ext` in Section 3.5.

3.11 Quantifiers

In CPC, the quantifiers `forall` and `exists` are each represented by an uninterpreted constant that expects two arguments. The first argument is a sequence

of variables (the variables to bind), and the second is a formula (the body). This representation is convenient since manipulations of quantified formulas are handled analogously to manipulations of quantifier-free terms. However, as a downside, this representation allows variable capturing in certain operations like substitutions (of variables by terms) which must be defined manually. Thus, special care must be given to side conditions on quantified formulas.[16]

Specifically, Eunoia provides a class of atomic terms called *variables* that are identified by their name and type, i.e., there is a single unique variable of type `Int` with the name `x`. In CPC, a formula like $\forall x, y{:}\mathsf{Int}.\exists\, z{:}\mathsf{Int}.P(x, y, z)$ is then written as `(forall ((x Int) (y Int)) (exists ((z Int)) (P x y z)))`. Eunoia adopts a lexical scoping discipline for variables. For example, in the body of `(forall ((x Int)) (exists ((x Int)) (Q x)))`, variable `(x Int)` is bound by the existential quantifier which then shadows `(x Int)` from the universal one. To ensure proof rules are sound under this semantics, we define several utilities in the CPC signature for manipulating quantified formulas, including a version of substitution that guards for variable capture and is implemented by about 30 lines of Eunoia.

Overall, the signature includes a proof rule `instantiate` to instantiate universally quantified formulas, and a proof rule `skolemize` to Skolemize existentially quantified formulas. The proof rule `alpha_equiv` is used to infer that two formulas are equivalent up to variable renaming. The remaining rules handle rewriting on quantified formulas, including three rules for miniscoping, a rule `quant-var-elim-eq` for eliminating solved variables, and rules for prenexing `quant-merge-prenex`, variable reordering `quant_var_reordering`, and dropping unused variables `quant-unused-vars`. Finally, the elimination of existential quantification by a reduction to universal quantification is handled by `exists-elim`.

3.12 Theory Independent Rules

CPC contains an additional set of core rules that are applicable to (a subset of) all theory symbols or types from its signature.

The proof rule `aci_norm` reasons about theory symbols that have been declared as having one or more of the following properties: associativity, commutativity, idempotency, and having an identity element (such as `0` for `+` and `false` for `or`). Given an input term, this proof rule normalizes the term based on flattening applications of associative operators, sorting commutative operators, dropping identity elements, and dropping duplicate elements from idempotent operators. In total, the program computing this normalization handles ten theory symbols and is about 100 lines of Eunoia.

The rule `absorb` reasons about theory symbols that have an absorbing element (e.g., `false` for `and`). This rule concludes that a nested application of a theory symbol that contains its absorbing element is equal to the absorbing element.

[16] Note that this treatment of quantified formulas coincides with cvc5's internal treatment of such formulas.

The program used to check this rule handles seven theory symbols and is roughly 57 lines of Eunoia.

Finally, the rule `evaluate` is used to implement constant folding of all theory symbols over atomic types. This rule expects a term whose AST has only atomic values as leaves (e.g., concrete integer, rational, bit-vector or string literals), and concludes that the given term is equal to the constant it evaluates to. In total, the program used to evaluate terms handles cases for 78 theory symbols. This program and its dependencies are roughly 540 lines of Eunoia.

4 Verification of CPC

It is important to note that even if a CPC proof successfully checks with Ethos, there may still be bugs in (*i*) the implementation of Ethos itself, or (*ii*) the correctness of the 585 proof rules of CPC. In this section, we focus on ongoing efforts to address the latter challenge.

Verification in Lean. A complementary line of work makes use of the CPC proof rules to prove Lean theorems by reimplementing their semantics inside the Lean proof assistant [34] and checking the resulting proofs with Lean's trusted kernel. In particular, the lean-smt tactic [33] translates CPC proofs into equivalent Lean proofs using a shallow embedding of SMT-LIB into Lean. These proofs are then checked by the Lean kernel, which verifies every inference step against its small trusted core. While this approach does not constitute a full formal verification of CPC itself, it provides strong evidence for the correctness of its proof rules within the supported SMT fragments.

The translation currently supports 180 CPC proof rules covering linear integer and real arithmetic (including incremental linearization rules), uninterpreted functions, and quantifiers. Of these, approximately 115 proof and rewrite rules are directly translated into Lean theorems. Normalization rules (e.g., `aci_norm`, `absorb`, and `arith_poly_norm`) are verified by reflection into certified Lean programs. The remaining rules are handled via specialized tactics that discharge the corresponding proof obligations in Lean.

Due to the reconstruction performed by lean-smt, its performance is significantly slower (about 20x) than native CPC checking in Ethos. Nevertheless, it can be used to independently validate CPC proofs in the supported fragments. This makes it a practical tool for cross-checking proof rules and increases confidence in their soundness through Lean's kernel-level verification.

Verification in Isabelle/HOL. CPC rules phrased in RARE are amenable to verification via IsaRare [29], a tool that automatically translates RARE rules into Isabelle/HOL [36] lemmas. When we prove such a lemma in Isabelle/HOL, we gain high confidence that the original RARE rule is sound. Another benefit of this process that unsound rules can be exposed via Isabelle/HOL's counterexample generator Nitpick [12]. Building on the initial work by Lachnitt et al. [29], which

verified 217 rules, we have increased the number of verified rules to 338, while also identifying 6 faulty rules that were then amended. Our focus has been on string rewrite rules, which are particularly intricate.

Similarly to the corroboration of CPC's correctness provided through the reconstruction into Lean described above, CPC proofs can be reconstructed inside Isabelle/HOL. Currently, this is done via a translation into the Alethe proof format, implemented in cvc5 [30], to capitalize on previous work that added support for Alethe in Isabelle/HOL. While this does not directly entail correctness of the translated CPC rules, it shows that the resulting translations into Alethe proofs can be successfully reconstructed into valid Isabelle/HOL proofs [30, 48]. The fragment of CPC supported by the effort described in [30] comprises rewriting (via IsaRare), clausification and Boolean reasoning, equality and uninterpreted functions, linear arithmetic, and quantifiers. Thus CPC proofs in these fragments can also be independently validated through Isabelle.

Verification Using a Translation Back to SMT. As a more general alternative to verification, we have developed a translation from each Eunoia proof rule to an SMT-LIB 2 verification condition corresponding to whether a proof rule is sound, which can be checked by an SMT solver. While this does not necessarily decrease the trusted core (since an SMT solver is used for verification), it provides an important pragmatic way of cross checking the correctness of our proof rules. In practice, it was instrumental in discovering several soundness bugs.

The translation from Eunoia to SMT-LIB involves a formal definition of the semantics of Eunoia and of the model semantics of SMT-LIB 2. For the former, we wrote a signature to bootstrap the semantics of non-core Eunoia primitives in terms of ordinary Eunoia programs, as well as references to SMT-LIB operators for evaluating values. At a high level, Eunoia terms (including the embedding of SMT-LIB terms) are translated to constructors of an SMT-LIB datatype `eo.Term`, and Eunoia programs are translated to uninterpreted functions over this datatype, possibly with recursive definitions.

This translation is implemented as an extension of Ethos and is approximately 3.5k lines of C++ and roughly 1.3k lines of Eunoia. Using this methodology, we were able to generate verification conditions for 571 proof rules. The remaining 14 CPC rules involved internal symbols whose model semantics could not be easily defined. We ran the SMT solvers z3 [35] (version 4.13.4) and cvc5 (version 1.3.2) on these verification conditions with a 60 second timeout. Of these 571 verification conditions, 139 were solved by z3 and 238 were solved by cvc5 (99 uniquely). Note that 41 of the verification conditions involved symbols that were specific to cvc5 and hence are out of scope for z3.

In total, 238 proof rules (40.6%) could be automatically validated using this method; for the others, we expect we will need inductive reasoning [41] or invariants on recursive side conditions. Another possibility is to use different translations for operators that currently require Eunoia programs to capture their semantics, such as bitvector-and (`bvand`), whose semantics can more naturally be encoded via parametric bit-vectors [10].

Conversely, as done in Isabelle for translated rewrite rules thanks to Nitpick, this methodology can be used as a way of searching for counterexamples showing where proof rules are unsound. We compiled the verification conditions in the same manner described above, but instead of checking for satisfiability, we generated syntax-guided synthesis (SyGuS) [1] queries that search for an input to a proof rule that indicates the rule is unsound. In particular, the SyGuS query asks for a term in which the premises of a proof rule are satisfied and the conclusion is not satisfied. Using cvc5's syntax-guided synthesis solver [39], we discovered counterexamples to 10 proof rules released in an earlier version of CPC, including to one of the core string proof rules. Among these, 4 of the 10 counterexamples[17] could be *confirmed* in Isabelle, indicating that this methodology is capable of finding genuine bugs not found by other methods. All 10 proof rules were subsequently fixed in the current release of CPC.

As ongoing work, we plan to port this general methodology to a higher-assurance setting, e.g., in Lean or Isabelle/HOL. This will require a formal definition of the semantics of Eunoia and an automated or semi-automated translation from the CPC signature to these environments.

5 Evaluation

We evaluated cvc5's ability to generate CPC proofs on the SMT-LIB benchmark library [6]. We considered all logics that are supported by cvc5's safe configuration, which includes all logics of SMT-LIB that do not contain the theory of floating point numbers (FP). FP benchmarks are not included in our evaluation because cvc5 currently does not produce full proofs for that theory. What is missing are subproofs for cvc5's word-blasting module SymFPU which reduces FP constraints to bit-vector constraints. Since that component is quite complex, writing Eunoia rules beforehand is a somewhat futile exercise because, especially in this case, formalization and proof-instrumentation need to go hand in hand.

The goals of the evaluation were to (*i*) measure the overhead of proof production in cvc5 and proof checking in Ethos, (*ii*) confirm that the generated proofs are complete and every step is justified by a CPC rule. All experiments were run on Intel Xeon E5-2686 v4 CPUs running at 2.6 GHz with 252 GB of RAM We use a development version of cvc5 version 1.3.2, running in the *safe configuration* that disables all expert features and those that are not proof producing.

For (*i*), a summary of the performance of cvc5 is given in Table 1. We considered all 59 logics of SMT-LIB that do not contain FP operators. We divide the benchmarks in these logics into 6 categories. The category `QF+UF` contains all benchmarks in quantifier-free logics that do not contain arithmetic, strings, or bit-vectors. This includes quantifier-free logics with uninterpreted functions, arrays, and datatypes. We then consider three additional categories of quantifier-free benchmarks, for those containing bit-vectors (`QF+BV`), strings (`QF+Str`) and arithmetic but not strings or bit-vectors (`QF+Arith`). We additionally consider

[17] The other 6 counterexamples were out of scope for the current translation to Isabelle.

Table 1. Runtime in seconds of solving with and without proof generation and checking, overhead ratio and average proof size (steps); by SMT-LIB benchmark category.

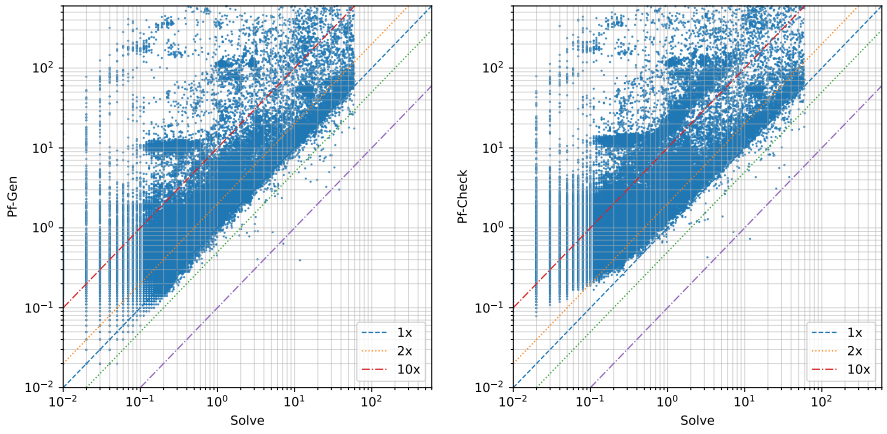
		Solve		**+Proof**			**+Check**			
Category	Total	№	Time	№	Time	Ratio	№	Time	Ratio	Size
QF+UF	16,457	8,971	6.8k	8,968	27.6k	4.12	8,941	88.7k	15.85	22.7k
QF+Arith	60,041	14,515	49.4k	14,422	121.2k	2.58	14,246	172.7k	4.04	19.0k
QF+BV	59,057	27,840	46.7k	27,239	440.0k	11.94	26,707	447.0k	16.28	9.9k
QF+Str	69,908	26,793	5.0k	26,793	8.7k	1.75	26,793	13.9k	2.77	0.5k
Q+UF	92,457	65,015	40.8k	64,918	95.6k	2.44	64,848	114.8k	3.12	2.2k
Q-UF	12,910	10,054	2.7k	10,042	8.6k	3.38	10,036	16.0k	6.37	1.1k
Overall	310,830	153,188	151.5k	152,382	702.9k	5.11	151,571	852.1k	7.11	6.1k

quantified benchmarks containing either UF, datatypes or arrays (Q+UF) and finally all remaining quantified benchmarks (Q-UF).

For each category, the second column gives the total number of benchmarks in that category. The **Solve** columns give the number of benchmarks cvc5 determines to be unsatisfiable within a 60 s timeout and the cumulative time for solved benchmarks in that category. In total, 153k benchmarks were solved in a 60 s timeout. We then ran cvc5 with proofs enabled on these 153k benchmarks. The columns **+Proof** give the number of benchmarks cvc5 can solve and generate a CPC proof within a 600 s timeout and the cumulative time for solved benchmarks; the column *Ratio* gives the ratio of runtime between solving and solving with proof generation on commonly solved instances. The columns **+Check** give the number of benchmarks cvc5 can solve and generate a CPC proof for, with the proof checkable by Ethos within 600 s. Again, we give the cumulative time on successful benchmarks and the runtime ratio in relation to pure solving. The final column gives the average number of proof steps for proofs that were successfully generated (*Size*).

First, cvc5 was able to successfully generate CPC proofs within 600 s for 152.3k out of the 153.1k benchmarks it could solve in 60 s, giving an overall success rate of 99.4% (similarly, 98.9% for proof generation and checking). In the vast majority of unsuccessful cases, cvc5 timed out.[18] These timeouts can partially be attributed to the instability of cvc5's solving procedures when using a different set of options (i.e., enabling proofs). When considering commonly solved instances, the average overhead of proof generation is 5.11x and the average overhead of proof generation and checking in Ethos is 7.11x. This overhead varies significantly based on the category of benchmarks. Notably, proof generation and checking is generally efficient (e.g., for quantifier-free string benchmarks and benchmarks with quantifiers) but is slower for quantifier-free bit-vectors and quantifier-free logics with UF.

[18] For two benchmarks, we generated a proof that failed to check due to certain features of SMT-LIB being incompatible with Ethos.



(a) Solve vs. proof generation (b) Solve vs. proof checking

Fig. 1. Performance comparison of cvc5 solving vs. proof generation and proof generation+checking on unsatisfiable benchmarks.

Table 1 additionally gives the average size of CPC proofs for benchmarks in the various categories. The proof generation and checking overhead is typically correlated with proof size, where for instance proofs in `QF+Strings` is quite small.

Figure 1 gives log-log scatter plots comparing solve time against +**Proof** and +**Check**. It shows shows the performance overhead for generating proofs is within an order of magnitude for a majority of benchmarks beyond those solved in less than one second; the performance overhead of generating and checking proofs is slightly more than an order of magnitude in a number of cases but also has many cases with little overhead.

For every problem it solved, cvc5 generated fully fine-grained proofs. In total, 913.4 million proof steps were generated (including `assume` steps). Over these proof steps, 427 of 585 proof rules were used. The remaining rules either correspond to rules for non-standard SMT-LIB theories (e.g. sets), non-standard operators supported by cvc5, or rules over standard operators that SMT-LIB does not cover. Amongst the 427 rules, the most commonly used rules include transitivity (12.8%) and congruence (12.6%) rules, heavily used in proofs from congruence closure and input preprocessing by rewriting; `chain_m_resolution` (9.8%), for proofs from the underlying SAT solver; constant evaluation `evaluate` (1.6%); arithmetic polynomial normalization `arith_poly_norm` (1.5%); and ACI normalization `aci_norm` (1.4%).

Performance Impact of Safe Configuration. As mentioned, our evaluation considers the safe configuration in cvc5, which disables a small number of features otherwise enabled by default, including symmetry breaking for `QF_UF`, the use of the CaDiCaL SAT solver [11] for bit-vector solving, and cylindrical algebraic (CA) coverings for non-linear arithmetic. Proof production for some of

these features involve open research challenges. For example, the CA coverings procedure lacks a mature proof calculus in the literature. Similarly, symmetry breaking relies on reasoning steps that preserve *equisatisfiability* in the background theories rather than equivalence, for which proof production has not been explored in sufficient depth in the context of SMT.

We point out that disabling these techniques leads to degraded performance in certain domains, notably quantifier-free UF, bit-vectors and non-linear real arithmetic. In total, cvc5's default settings solve 156,407 unsat benchmarks in SMT-LIB logics without floating point compared to 153,188 with its safe configuration enabled, a 2.1% increase. Its safe configuration is 17.1% slower on average than the default one on the unsatisfiable instances solved by both. This is mostly due to quantifier-free bit-vector logics where CaDiCaL is disabled. Excluding quantifier-free benchmarks from bit-vector logics, cvc5's safe configuration is 9.9% slower on average. Work to extend that configuration to include these high performance techniques is ongoing.

6 Conclusion

We have presented the Cooperating Proof Calculus, an evolving set of proof rules capturing the reasoning capabilities of the SMT solver cvc5. We consider this work an important milestone, since all mainstream theories and techniques of an industrial strength SMT solver now produce complete and externally checkable proofs. As a result, the trusted computing base of cvc5 can now be reduced to a much smaller trusted computing base consisting of a Eunoia proof signature and the source code of a Eunoia proof checker. We also believe it is important that the specification of CPC is flexible and can evolve as new solving techniques are added to cvc5.

We plan to pursue formal verification of CPC as well as a verified proof checker for CPC as an alternative to Ethos. For the former, we presented preliminary work in this direction in Section 4, which we plan to port to an interactive theorem prover such as Lean or Isabelle. For the latter, a verified proof checker has been shown effective in some logics. However, we expect a verified proof checker for CPC to be significantly slower than Ethos.

Acknowledgments. We thank the anonymous reviewers for their feedback on this paper. This work was supported in part by the Stanford Center for Automated Reasoning, a gift from Amazon Web Services, the Binational Science Foundation (BSF) grant number 2024049, the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-24-2-1001, the European Union (ERC, CertiFOX, 101122653), and Fonds Wetenschappelijk Onderzoek — Vlaanderen (project G070521N). Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA, the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement

The experimental artifact supporting the results in this paper is publicly available on Zenodo: https://zenodo.org/records/19867026. The artifact contains the implementation, benchmarks, scripts, and data used to reproduce the experimental evaluation reported in the paper.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.6679385
2. Andreotti, B., Lachnitt, H., Barbosa, H.: Carcara: An efficient proof checker and elaborator for SMT proofs in the alethe format. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I. Lecture Notes in Computer Science, vol. 13993, pp. 367–386. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_19
3. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory. CoRR **abs/2311.07185** (2023). https://doi.org/10.48550/ARXIV.2311.07185
4. Bansal, K., Reynolds, A., Barrett, C.W., Tinelli, C.: A new decision procedure for finite sets and cardinality constraints in SMT. In: Olivetti, N., Tiwari, A. (eds.) Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9706, pp. 82–98. Springer (2016). https://doi.org/10.1007/978-3-319-40229-1_7
5. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.W.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 15–35. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_3
6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
7. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
8. Barrett, C.W., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. J. Satisf. Boolean Model. Comput. **3**(1-2), 21–46 (2007). https://doi.org/10.3233/SAT190028
9. Barrett, C.W., Tinelli, C., Deters, M., Liang, T., Reynolds, A., Tsiskaridze, N.: Efficient solving of string constraints for security analysis. In: Scherlis, W.L.,

Brumley, D. (eds.) Proceedings of the Symposium and Bootcamp on the Science of Security, Pittsburgh, PA, USA, April 19-21, 2016. pp. 4–6. ACM (2016). https://doi.org/10.1145/2898375.2898393

10. Berger, Z., Zohar, Y., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-precise reasoning with parametric bit-vectors. In: Berg, J., Nordström, J. (eds.) 28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 341, pp. 4:1–4:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.SAT.2025.4
11. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: CaDiCaL 2.0. In: Computer Aided Verification (CAV), Part I. p. 133–152. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-65627-9_7
12. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6172, pp. 131–146. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_11
13. Bouchet, M., Cook, B., Cutler, B., Druzkina, A., Gacek, A., Hadarean, L., Jhala, R., Marshall, B., Peebles, D., Rungta, N., Schlesinger, C., Stephens, C., Varming, C., Warfield, A.: Block public access: trust safety verification of access control policies. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 281–291. ACM (2020). https://doi.org/10.1145/3368089.3409728
14. Bouton, T., Oliveira, D.C.B.D., Déharbe, D., Fontaine, P.: verit: An open, trustable and efficient smt-solver. In: Schmidt, R.A. (ed.) Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5663, pp. 151–156. Springer (2009). https://doi.org/10.1007/978-3-642-02959-2_12
15. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM (1964)
16. Christ, J., Hoenicke, J., Nutz, A.: Smtinterpol: An interpolating SMT solver. In: Donaldson, A.F., Parker, D. (eds.) Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings. pp. 248–254. Lecture Notes in Computer Science, Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19
17. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 58–75 (2017). https://doi.org/10.1007/978-3-662-54577-5_4
18. Coltellacci, A., Andreotti, B., Barbosa, H., Dowek, G., Merz, S.: Reconstruction of SMT proofs with Lambdapi. Acta Informatica **63**(1), 8 (2026). https://doi.org/10.1007/S00236-025-00515-W
19. Coltellacci, A., Merz, S.: Checking linear integer arithmetic proofs in Lambdapi. In: Thiemann, R., Weidenbach, C. (eds.) Frontiers of Combining Systems (FroCoS). Lecture Notes in Computer Science, vol. 15979, pp. 367–385. Springer (2025). https://doi.org/10.1007/978-3-032-04167-8_20

20. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_14
21. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 10205, pp. 118–135 (2017). https://doi.org/10.1007/978-3-662-54577-5_7
22. Dunne, C., Burel, G.: Automatically translating proof systems for SMT solvers to the lambda pi calculus. In: Biere, A., Lutz, C., Negri, S. (eds.) Automated Reasoning - 13th International Joint Conference, IJCAR 2026. Lecture Notes in Computer Science, Springer (2026)
23. Hitarth, S., Codel, C.R., Lachnitt, H., Dutertre, B.: Extending DRAT to SMT. In: Narodytska, N., Rümmer, P. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024. pp. 1–11. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_8
24. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022. CEUR Workshop Proceedings, vol. 3185, pp. 54–70. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3185/paper9527.pdf
25. Israel, O., Zohar, Y., Singh, H., Dutertre, B., Reynolds, A., Barrett, C., Tinelli, C.: Checking regular expressions in cvc5 proofs. In: Biere, A., Lutz, C., Negri, S. (eds.) Automated Reasoning - 13th International Joint Conference, IJCAR 2026. Lecture Notes in Computer Science, Springer (2026)
26. Kiesl, B., Rebola-Pardo, A., Heule, M.J.H.: Extended resolution simulates DRAT. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) International Joint Conference on Automated Reasoning (IJCAR). Lecture Notes in Computer Science, vol. 10900, pp. 516–531. Springer (2018)
27. Kremer, G., Ábrahám, E., England, M., Davenport, J.H.: On the implementation of cylindrical algebraic coverings for satisfiability modulo theories solving. In: Schneider, C., Marin, M., Negru, V., Zaharie, D. (eds.) 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2021, Timisoara, Romania, December 7-10, 2021. pp. 37–39. IEEE (2021). https://doi.org/10.1109/SYNASC54541.2021.00018
28. Kremer, G., Reynolds, A., Barrett, C.W., Tinelli, C.: Cooperating techniques for solving nonlinear real arithmetic in the cvc5 SMT solver (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 95–105. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_7
29. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: Isarare: Automatic verification of SMT rewrites in Isabelle/HOL. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. Lecture

Notes in Computer Science, vol. 14570, pp. 311–330. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_17

30. Lachnitt, H., Fleury, M., Barbosa, H., Jakpor, J., Andreotti, B., Reynolds, A., Schurr, H., Barrett, C.W., Tinelli, C.: Improving the SMT proof reconstruction pipeline in Isabelle/HOL. In: Forster, Y., Keller, C. (eds.) 16th International Conference on Interactive Theorem Proving, ITP 2025, Reykjavik, Iceland, September 28 - October 1, 2025. LIPIcs, vol. 352, pp. 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). https://doi.org/10.4230/LIPICS.ITP.2025.26
31. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 10395, pp. 237–254. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_15
32. Liang, T., Reynolds, A., Tinelli, C., Barrett, C.W., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 646–662. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_43
33. Mohamed, A., Mascarenhas, T., Khan, H., Barbosa, H., Reynolds, A., Qian, Y., Tinelli, C., Barrett, C.W.: lean-smt: An SMT tactic for discharging proof goals in Lean. In: Piskac, R., Rakamaric, Z. (eds.) Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15933, pp. 197–212. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_11
34. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
35. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 337–340. Lecture Notes in Computer Science, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
36. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9
37. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 65–74. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_12
38. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C.W., Tinelli, C.: Syntax-guided rewrite rule enumeration for SMT solvers. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 279–297. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_20

39. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 74–83. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_5
40. Reynolds, A., Blanchette, J.C.: A decision procedure for (co)datatypes in SMT solvers. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9195, pp. 197–213. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_13
41. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8931, pp. 80–98. Springer (2015). https://doi.org/10.1007/978-3-662-46081-8_5
42. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 23–42. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_2
43. Reynolds, A., Schurr, H.J., Soldevila, M., Tinelli, C.: The ethos user manual (2026), https://github.com/cvc5/ethos/blob/main/user_manual.md
44. Reynolds, A., Schurr, H.J., Soldevila, M., Barbosa, H., Barrett, C., Tinelli, C.: Ethos: A fast proof checker for the Eunoia logical framework. In: Biere, A., Lutz, C., Negri, S. (eds.) Automated Reasoning - 13th International Joint Conference, IJCAR 2026. Lecture Notes in Computer Science, Springer (2026)
45. Reynolds, A., Tinelli, C., Jovanovic, D., Barrett, C.W.: Designing theory solvers with extensions. In: Dixon, C., Finger, M. (eds.) Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10483, pp. 22–40. Springer (2017). https://doi.org/10.1007/978-3-319-66167-4_2
46. Reynolds, A., Woo, M., Barrett, C.W., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 453–474. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_24
47. Schurr, H., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). In: Keller, C., Fleury, M. (eds.) Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021. EPTCS, vol. 336, pp. 49–54 (2021). https://doi.org/10.4204/EPTCS.336.6
48. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 450–467. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_26

49. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods Syst. Des. **42**(1), 91–118 (2013). https://doi.org/10.1007/S10703-012-0163-3
50. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 8561, pp. 422–429. Springer (2014)