A Simple Proof Format for SMT

Jochen Hoenicke, Tanja Schindler

Department of Computer Science, University of Freiburg, Germany

Abstract

We present a simple resolution-based proof format targeted for SMT. The proof format uses a syntax similar to SMT-LIB terms. The resolution rule is its only proof rule with premises; all other rules are axioms proving tautological clauses. The format aims to be solver-independent by only including a minimal complete set of axioms while still being able to express proofs succinctly. Most of its axioms are purely syntactic; only for arithmetic reasoning, some axioms with side conditions are used for succinct reasoning with linear (in)equalities. The format can be extended with solver-specific rules, which can then either be treated as trusted axioms, or, better, replaced by their low-level proof. The format has been implemented in the solver SMTInterpol and the solver produces proofs for all benchmarks it can solve in the combinations of the theories of equality, linear arithmetic, arrays and datatypes. There is also a stand-alone proof checker for this format.

Keywords

SMT, proofs, proof format, resolution

1. Introduction

SMT solvers have become an integral part of many modern verification techniques. They are used to prove the absence of errors in complex systems that cannot be verified by hand. This begs the question whether the solvers themselves are correct.

Many SMT solvers can support their answer on the satisfiability problem of a formula by providing either a model for a satisfiable formula, or a proof for an unsatisfiable formula. The SMT-LIB standard [1] already prescribes the form of models. Models are relatively easy to verify, at least for quantifier-free formulas, by evaluating the formula using the model's function definitions. The annual competition of SMT solvers, the SMT-COMP, includes a model validation track since 2019, where models are checked by an independent tool.[1] For unsatisfiable formulas, many solvers can already produce proofs in some solver-specific format, but no standardized proof format exists so far.

In this paper we present a very simple but expressive proof format that we propose as a candidate for such a standard. Our proof format uses the resolution rule known from SAT proofs and combines it with axioms for SMT theories. We propose to use a minimal complete set of axioms, and the idea is that a solver needs to explain its theory lemmas using only these axioms. An important idea is that even the meaning of the built-in logical symbols like `not` and

SMT 2022: Satisfiability Modulo Theories, August 11–12, 2022, Haifa, Israel

hoenicke@informatik.uni-freiburg.de (J. Hoenicke); schindle@informatik.uni-freiburg.de (T. Schindler)

0000-0002-6314-1041 (J. Hoenicke); 0000-0002-7462-8445 (T. Schindler)

[1] https://smt-comp.github.io/2019/

`or` is given by axioms that correspond to the Tseitin transformation. Thus, even the conversion into conjunctive normal form (CNF) can be explained in this format. By implementing the proof format in SMTInterpol, we have shown that despite the minimality of the axiom set, the overhead of generating proofs is manageable (it is linear in the run-time of the solver).

We claim that our proof format solves the following goals: (1) it is easy and fast to verify, (2) it avoids solver-specific rules, (3) it supports several reasoning techniques, from equality rewriting to symmetry breaking, (4) it enables short proofs that are linear in the number of reasoning steps the solver took.

2. Basic Notation

We use the SMT-LIB [1] syntax for terms. SMT-LIB is based on sorted first-order logic with the sort `Bool` representing Boolean values. A function symbol `f` has an arity $\tau_1 \times \cdots \times \tau_n \to \tau$ (determined by the theory or by a function declaration) and if $t_1, \ldots, t_n$ are terms of type $\tau_1, \ldots, \tau_n$ respectively, then `(f` t_1 $\ldots t_n$`)` is an (SMT-LIB) term of type τ. SMT-LIB formulas are terms of type `Bool`. The logical operators are built-in function symbols `and`, `or`, `=>`, etc. of arity `Bool` $\times$ `Bool` $\to$ `Bool`. Similarly, the symbols `true` and `false` are function symbols of arity `Bool` (no arguments) and thus also terms of type `Bool`.

We use typewriter font for literal SMT-LIB terms, t stands for an SMT-LIB term and p for an SMT-LIB formula (term of type `Bool`). We use the symbol `f` for an (uninterpreted or theory) function symbol, the symbol x for variables, and τ for SMT-LIB sorts. We use prf for proof terms. In linear arithmetic, we use g for a polynomial.

3. Proof Format

Our proof format is a resolution proof that shows unsatisfiability by deriving the empty clause. The leaves of the proof are input formulas and axioms. The only proof rule is the resolution rule that proves from two given clauses a resolvent.

$$\frac{\{p\} \cup C_1 \qquad \{\neg p\} \cup C_2}{C_1 \cup C_2} \tag{res}$$

A clause is a set of literals, which is interpreted as the disjunction of these literals. A literal is a positive or negated atomic formula. The core idea of our proof format is that atomic formulas are exactly the SMT-LIB formulas, i.e., SMT-LIB terms of type `Bool`.

As far as the proof format is concerned, the atomic formulas are opaque and even the built-in SMT-LIB functions `and`, `or`, `not` have no predefined meaning. Instead the meaning is given by axioms that can be seen as a set of theory lemmas for the core theory. For each SMT-LIB formula p, there are the axioms `(not+` p`)` : $\{$`(not` p`)`$, p\}$[2] and `(not-` p`)` : $\{\neg$`(not` p`)`$, \neg p\}$. Similarly, the axioms `(or-` p_0 p_1`)` : $\{\neg$`(or` p_0 p_1`)`$, p_0, p_1\}$, `(or+ 0` p_0 p_1`)` : $\{$`(or` p_0 p_1`)`$, \neg p_0\}$, and `(or+ 1` p_0 p_1`)` : $\{$`(or` p_0 p_1`)`$, \neg p_1\}$ define the meaning of `(or` p_0 p_1`)`.

The syntax of proof terms is similar to the syntax of SMT-LIB terms. The simplest proof term is an axiom, e.g., `(not+` p`)`. Moreover, for each assertion `(assert` p`)` in the SMT-LIB

[2] We write $prf : C$ to denote that prf is an axiom stating the validity of the clause C.

```
(set-option :produce-proofs true)
(set-logic QF_UF)
(declare-fun q1 () Bool)
(declare-fun q2 () Bool)
(assert (or (not q1) q2))
(assert q1)
(assert (not q2))
(check-sat)
(get-proof)
```

```
unsat
(res q1 (assume q1) (res q2
  (res (not q1)
    (res (or (not q1) q2)
      (assume (or (not q1) q2))
      (or- (not q1) q2))
    (not- q1))
  (res (not q2) (assume (not q2))
    (not- q2))))
```

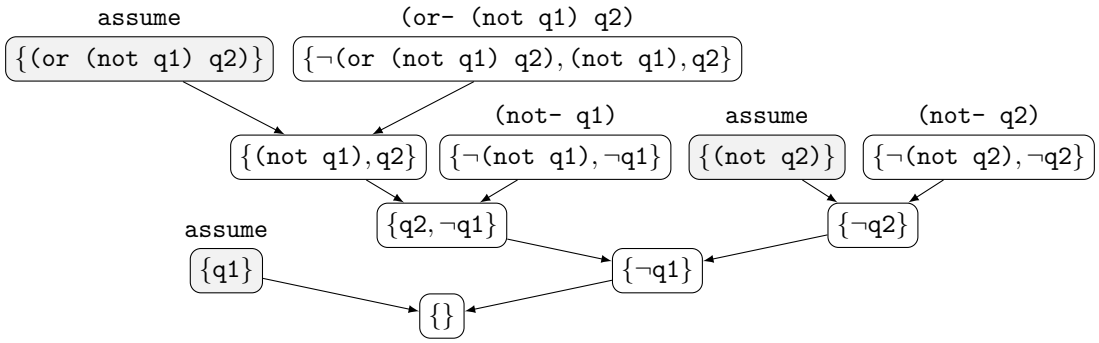

Figure 1: Example proof for the unsatisfiability of the three assertions `(or (not q1) q2)`, `q1`, `(not q2)`. The text on the left shows the SMT-LIB input, the text on the right shows the output produced by a (hypothetical) SMT solver on that input. The output is a proof term and is visualized by the proof tree in the bottom part of the figure.

input, the proof term `(assume` p`)` proves the unit clause $\{p\}$. The concrete syntax for the resolution rule is `(res` p prf_1 prf_2`)` where p is the pivot atom and prf_1 and prf_2 are the proof terms for the subproofs. Figure 1 shows a simple proof for the unsatisfiabilty of the assertions `(or (not q1) q2)`, `q1`, `(not q2)`, where `q1`, `q2` are constants of type `Bool`. Each proof term proves a clause and the whole proof of unsatisfiability is correct if it proves the empty clause.

Sharing Proofs and Subterms. Proofs often use the same intermediate clause several times. For a succinct representation of such proofs, repeated subproofs may be bound to variables, e.g., the syntax

`(let-proof ((C (res (not q) (assume (not q)) (not- q)))) `prf`)`

binds the proof of $\{\neg q\}$ to the variable `C`. This variable may then be used in prf, i.e., it may be given as argument to `res` applications whenever a resolution with this clause should be performed. Note that the syntax is similar to the `let`-syntax for SMT-LIB terms. Furthermore the proof format also supports the `let` binder with the identical syntax as for SMT-LIB terms to bind terms to variables that are used in axioms or as pivot atoms multiple times. The latter is important for keeping the proof size down, because large subterms may naturally appear

several times, for example in proofs where a large input formula is converted into CNF. In fact, in most proofs produced by our solver, every subterm of every asserted term is bound to a variable because it is used multiple times in the proof. In general, proofs grow exponentially if `let` terms are expanded.

In our proof format, `let` and `let-proof` are syntactic sugar and equivalent to their expanded form where every variable is replaced by its definition. As an example,

```
(let ((x (not q2))) (res x (assume x) (not- q2)))
```

is a valid subproof because `x` is implicitly expanded to `(not q2)` and the pivot atom matches the negated atom in the clause `(not- q2)` : $\{\neg$`(not q2)`$, \neg$`q2`$\}$. There is no proof rule for let expansion as both the expanded and contracted term are seen as the same term. Internally, our proof checker expands all let terms, using a DAG representation to avoid exponential blow-up.

The two different binders `let-proof` for proofs and `let` for terms simplify parsing. If `let` would be used for both, then symbols like `res`, `and+`, etc. would need to become predefined symbols and must not be used as user-defined symbols in benchmarks. By using two different binders, it is clear from the context whether a term is a proof term or an SMT-LIB term.

Defining Functions. In proofs it may be desirable to have prover-defined auxiliary functions that do not appear in the input problem. Our proof format uses the syntax

$$\texttt{((define-fun f ((}x_1\ \tau_1\texttt{) ...(}x_n\ \tau_n\texttt{)) }t\texttt{) }prf\texttt{)}$$

where `f` is a function symbol and t the defining term that uses the variables $x_1, \ldots, x_n$ of type $\tau_1, \ldots, \tau_n$. The whole proof term proves the same clause as its subproof prf proves. The subproof prf may use `f` as if it was a user-defined function. In particular, prf can contain the axioms `cong` and `expand` from the core theory.

Oracles. Sometimes it is useful to state a fact without proving it, e.g., because a rigorous proof requires new axioms (when extending the format to a new theory) or because a simplification step should be expressed in proofs before reducing it to the axioms of the theory at a later time. Our proof format supports an oracle axiom with the following syntax.

$$\texttt{(oracle }\pm\ p_1\ \ldots\ \pm\ p_n\ \ \mathit{Attributes}\texttt{)}$$

Here $\pm$ is `+` (positive literal) or `-` (negated literal) and $Attributes$ is a user-defined list of key-value pairs of the form `:key value`, where `:key` is an identifier prefixed by a colon and `value` an arbitrary s-expression or empty. The oracle itself proves the clause $\{(\neg)\, p_1, \ldots, (\neg)\, p_n\}$ where the literal p_i is negated if and only if p_i was prefixed by `-` in the axiom.

4. Axioms

To provide meaning for the built-in SMT-LIB functions of the core theory, several axioms are defined, see Figure 2 for a complete list. For each logical operator there is an introduction axiom, e.g., `and+`, and an elimination axiom, e.g., `and-`. These can be used to eliminate or

$$
\begin{array}{ll}
\texttt{true+}: \{\texttt{true}\} & \texttt{false-}: \{\neg\texttt{false}\} \\
\texttt{not+}: \{(\texttt{not}\ p_0), p_0\} & \texttt{not-}: \{\neg(\texttt{not}\ p_0), \neg p_0\} \\
\texttt{and+}: \{(\texttt{and}\ p_0\ \ldots\ p_n), \neg p_0, \ldots, \neg p_n\} & \texttt{and-}: \{\neg(\texttt{and}\ p_0\ \ldots\ p_n), p_i\} \\
\texttt{or+}: \{(\texttt{or}\ p_0\ \ldots\ p_n), \neg p_i\} & \texttt{or-}: \{\neg(\texttt{or}\ p_0\ \ldots\ p_n), p_0, \ldots, p_n\} \\
\texttt{=>+}: \{(\texttt{=>}\ p_0\ \ldots\ p_n), (\neg)\, p_i\} & \texttt{=>-}: \{\neg(\texttt{=>}\ p_0\ \ldots\ p_n), \neg p_0, \ldots, p_n\} \\
\texttt{=+1}: \{(\texttt{=}\ p_0\ p_1), p_0, p_1\} & \texttt{=-1}: \{\neg(\texttt{=}\ p_0\ p_1), p_0, \neg p_1\} \\
\texttt{=+2}: \{(\texttt{=}\ p_0\ p_1), \neg p_0, \neg p_1\} & \texttt{=-2}: \{\neg(\texttt{=}\ p_0\ p_1), \neg p_0, p_1\} \\
\texttt{xor+}: \{(\texttt{xor}\ l_1), (\texttt{xor}\ l_2), \neg(\texttt{xor}\ l_3)\} & \texttt{xor-}: \{\neg(\texttt{xor}\ l_1), \neg(\texttt{xor}\ l_2), \neg(\texttt{xor}\ l_3)\}
\end{array}
$$

where l_1,l_2,l_3 are lists of formulas and each formula occurs an even number in total

$$
\begin{array}{l}
\texttt{forall+}: \{(\texttt{forall}\ ((x\ \tau))\ p), \neg(\texttt{let}\ ((x\ (\texttt{choose}\ (x\ \tau)\ (\texttt{not}\ p))))\ p)\} \\
\texttt{forall-}: \{\neg(\texttt{forall}\ ((x\ \tau))\ p), (\texttt{let}\ ((x\ t))\ p)\} \\
\texttt{exists+}: \{(\texttt{exists}\ ((x\ \tau))\ p), \neg(\texttt{let}\ ((x\ t))\ p)\} \\
\texttt{exists-}: \{\neg(\texttt{exists}\ ((x\ \tau))\ p), (\texttt{let}\ ((x\ (\texttt{choose}\ (x\ \tau)\ p)))\ p)\}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{refl}: \{(\texttt{=}\ t\ t)\} & \texttt{symm}: \{(\texttt{=}\ t_0\ t_1), \neg(\texttt{=}\ t_1\ t_0)\} \\
\texttt{trans}: \{(\texttt{=}\ t_0\ t_n), \neg(\texttt{=}\ t_0\ t_1), \ldots, \neg(\texttt{=}\ t_{n-1}\ t_n)\} & \\
\texttt{cong}: \{(\texttt{=}\ (\texttt{f}\ t_0\ \ldots\ t_n)\ (\texttt{f}\ t'_0\ \ldots\ t'_n)), \neg(\texttt{=}\ t_0\ t'_0), \ldots, \neg(\texttt{=}\ t_n\ t'_n)\} & \\
\texttt{=+}: \{(\texttt{=}\ t_0\ \ldots\ t_n), \neg(\texttt{=}\ t_0\ t_1), \ldots, \neg(\texttt{=}\ t_{n-1}\ t_n)\} & \\
\texttt{=-}: \{\neg(\texttt{=}\ t_0\ \ldots\ t_n), (\texttt{=}\ t_i\ t_j)\} &
\end{array}
$$

$$
\begin{array}{l}
\texttt{distinct+}: \{(\texttt{distinct}\ t_0\ \ldots\ t_n), (\texttt{=}\ t_0\ t_1), \ldots, (\texttt{=}\ t_0\ t_n), \ldots, (\texttt{=}\ t_{n-1}\ t_n)\} \\
\texttt{distinct-}: \{\neg(\texttt{distinct}\ t_0\ \ldots\ t_n), \neg(\texttt{=}\ t_i\ t_j)\}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{ite1}: \{(\texttt{=}\ (\texttt{ite}\ p_0\ t_1\ t_2)\ t_1), \neg p_0\} & \texttt{ite2}: \{(\texttt{=}\ (\texttt{ite}\ p_0\ t_1\ t_2)\ t_2), p_0\} \\
\texttt{del!}: \{(\texttt{=}\ (\texttt{!}\ t\ \ldots)\ t)\} &
\end{array}
$$

$$
\texttt{expand}: \{(\texttt{=}\ (\texttt{f}\ t_0\ \ldots\ t_n)\ (\texttt{let}\ ((x_0\ t_0)\ldots(x_n\ t_n))\ t))\}
$$

where f is defined by $(\texttt{define-fun f}\ ((x_0\ \tau_0)\ldots(x_n\ \tau_n))\ t)$

Figure 2: The axioms for the core theory. Each axiom is a valid clause whose literals are positive or negated SMT-LIB formulas. All axioms need to obey the SMT-LIB type-checking rules.

introduce the corresponding operator into the formula by applying the resolution rule. The axioms are written as application terms that take the instantiated terms as arguments, e.g., $(\texttt{and+}\ p_0\ p_1) : \{(\texttt{and}\ p_0\ p_1), \neg p_0, \neg p_1\}$. The axioms `and-`, `or+`, `=>+`, `=-`, `distinct-` take as additional argument the value i (and j), a literal number between 0 and n, e.g., $(\texttt{and-}\ 0\ p_0\ p_1) : \{\neg(\texttt{and}\ p_0\ p_1), p_0\}$.

Boolean equality in SMT-LIB corresponds to logical equivalence. We define corresponding introduction and elimination axioms `=+1` to `=-2`. These rules may only be used on terms of type `Bool` and this has to be verified by the proof checker.

Quantifiers are supported using the SMT-LIB 3.0 `choose` operator, which is Hilbert's epsilon operator. For brevity, the figure shows the axioms only for one variable, but they are also

supported for multiple variables, e.g., the axiom `exists-` is the following.

```
(exists- ((x1 τ1) ... (xn τn)) p) :
 {¬(exists ((x1 τ1) ... (xn τn)) p),
  (let ((x1 (choose (x1 τ1) (exists ((x2 τ2) ... (xn τn)) p))))
  (let ((x2 (choose (x2 τ2) (exists ((x3 τ3) ... (xn τn)) p))))
  ⋮
  (let ((xn (choose (xn τn) p))) p)···))}
```

The axioms from `refl` to `distinct-` are the axioms for the theory of equality. They can also be used for Boolean equality (logical equivalence) and `cong` can also be used on built-in function symbols like `and` or even `=` itself. The axioms `ite1` and `ite2` can be used for Boolean and non-Boolean `ite` terms. The axiom `del!` states that adding annotations to an SMT-LIB term does not change its meaning. Finally, the axiom `expand` can be used to expand user-defined and prover-defined (see Section 3) function symbols.

4.1. Axioms for Linear Arithmetic

For linear arithmetic, purely syntactic rules come to their limit, for example, when proving arithmetic operations on rationals like {`(= (+ (/ 1.0 3.0) (/ 1.0 6.0)) (/ 1.0 2.0))`}. To ease the reasoning about arithmetic, our proof format has some built-in axioms with side conditions that use addition and/or multiplication of rational numbers and polynomials.

We use the SMT-LIB canonical form for numeric constants, with a slight alteration for rationals. In SMT-LIB, the number 1 of type `Real` is represented as `1` in the logic LRA and as `1.0` in the logic LIRA. We use `1.0` in both logics. Similarly, we always represent fractions as `(/ 1.0 3.0)` or `(/ (- 2.0) 3.0)` (like in SMT-LIB, fractions need to be reduced).

We define a *polynomial* as a finite map from multisets of SMT-LIB terms to non-zero rational coefficients. Each entry $\{t_1, \ldots, t_n\} \mapsto c$ of this map is a monomial, which is represented as `(* c t1 ... tn)` in SMT-LIB. In this representation, c can be omitted if $c = 1$ and $n > 0$. The multiplication symbol is omitted if it has only one argument. The terms $t_1, \ldots, t_n$ must not be applications of `+` or `*` and they must not be numeric constants in canonical form. A polynomial is either of type `Real` or `Int` and all terms in all multiset must be of the same type and all coefficients must be integral if the polynomial is of type `Int`. The term representing a polynomial is `(+ m1 ... mn)` where $m_1, \ldots, m_n$ are the monomial representations of the entries of the map. A polynomial with a single entry in the map is represented by m_1 and the empty map is represented by `0` or `0.0`. As an example, the polynomial $\{\{\mathtt{x}, \mathtt{x}\} \mapsto 2, \{\mathtt{x}\} \mapsto 1, \{\} \mapsto 1\}$ with `x` of type `Int` is represented by the SMT-LIB term `(+ (* 2 x x) x 1)`.

A polynomial can have several representations that differ in the order of the terms, but each SMT-LIB term can only be the representation of at most one polynomial. The *sum* $g_1 + g_2$ of two polynomials g_1, g_2 is built by conjoining the maps, adding the rational constants if the same multiset appears in both polynomials, and removing the entries whose sum of the constants is zero. The *product* $g_1 \cdot g_2$ is built by computing the sum of the products for each pair of monomials. The product of two monomials is just the union of the multisets mapped to the product of the rational constants. A polynomial of type `Int` can be *cast to real* by surrounding

each term in each multiset with a `to_real` application and converting the coefficients to real by appending `.0` to the constant. A polynomial *is a constant*, if it is zero (the empty map) or a singleton map whose multiset is empty. Note that the representation of a constant polynomial is the constant term in canonical form.

Figure 3 shows the axioms for linear arithmetic. The first two axioms `poly+` and `poly*` can be used to simplify arithmetic terms in the input formula and have a side condition that requires polynomial addition or multiplication, respectively. Note that the resulting polynomial has to be given in the proof, as its SMT-LIB representation is not unique, e.g., the axioms `(poly+ (+ x y x) (+ (* 2 x) y))` and `(poly+ (+ x y x) (+ y (* 2 x)))` are both valid and prove different clauses. The prover can choose the representation and the proof checker checks that g is indeed a valid representation of the polynomial that is the sum or product of $g_1, \ldots, g_n$. However, `(poly+ (+ x y x) (+ x y x))` is not well-formed (a polynomial can only have one entry for the multiset $\{x\}$).

The heart of our arithmetic proof is the axiom `farkas` that proves the unsatisfiability of the conjunction of several literals, whose weighted sum yields a contradiction. A variant of Farkas' lemma states that a set of inequalities $Ax \leq b$ is unsatisfiable if and only if there is a vector $y = (c_1, \ldots, c_n)$ with $y \geq 0$, $A^T y = 0$, and $b^T y < 0$. The axiom provides the coefficients $c_1, \ldots, c_n$, e.g., `(farkas 1 (<= x (* 2 y))) 2 (< y z) 1 (= (+ (* 2 z) 5) x))` is a valid instance proving $\{\neg$`(<= x (* 2 y))`$, \neg$`(< y z)`$, \neg$`(= (+ (* 2 z) 5) x)`$\}$, because the inequalities sum up to the contradiction $5 < 0$ and `x`, `y`, `z` cancel out. It is allowed to use both integer and real literals in the same instance of the axiom `farkas` and in that case the weighted sum is computed after implicitly casting all integer polynomials to real.

We found that `farkas`, `trichotomy`, `total`, and `total-int` together with the core axioms for equality are complete for linear arithmetic in the sense that any valid disjunction of possibly negated equalities and inequalities of linear polynomials can be proved using just these axioms. The axiom `farkas` can prove clauses containing negated literals and the other three axioms can be used to rewrite negated literals into their positive form. The axiom `total-int` can be used to introduce arbitrary integer cuts, since t_1 can be an arbitrary term.

If an input term is not already a polynomial, the rules `poly*` and `poly+` together with the axiom `cong` can be used to rewrite it into a polynomial. To support arbitrary SMT-LIB input formulas, we need a few more axioms to rewrite other operators like `>`, `>=`, `-`, `/`, `div`, `mod`, etc. See the appendix for a complete list.

5. Proofs in SMTInterpol

Our solver SMTInterpol produces proofs in the proof format presented in this paper. The reader is invited to experiment using our web interface for proof production and checking.[3]

The solver itself proves a benchmark in three phases. In the first phase, the input formula is simplified using equality rewriting and several solver-specific rewriting rules. The second phase converts the input formula into conjunctive normal form (CNF), creating an internal representation for every literal and adding auxiliary clauses. The third phase runs the DPLL($\mathcal{T}$) algorithm and generates theory lemmas, building the final resolution proof.

[3] See https://ultimate.informatik.uni-freiburg.de/smtinterpol/online/proof.html

$(\texttt{poly+}\ (\texttt{+}\ g_1 \dots g_n)\ g) : \{(\texttt{=}\ (\texttt{+}\ g_1 \dots g_n)\ g)\}$ where $g = g_1 + \cdots + g_n$

$(\texttt{poly*}\ (\texttt{*}\ g_1 \dots g_n)\ g) : \{(\texttt{=}\ (\texttt{*}\ g_1 \dots g_n)\ g)\}$ where $g = g_1 \cdots g_n$

$(\texttt{farkas}\ c_1\ (\texttt{<=?}\ g_1\ g_1')\ \dots\ c_n\ (\texttt{<=?}\ g_n\ g_n')) : \{\neg(\texttt{<=?}\ g_1\ g_1'), \dots, \neg(\texttt{<=?}\ g_n\ g_n')\}$

where c_i are positive integer constants, the weighted sum $c_1(g_1 - g_1') + \cdots + c_n(g_n - g_n')$ is a non-negative constant. <=? stands for <, <=, or =, and if the weighted sum is zero, at least one of the literals must be <.

$(\texttt{trichotomy}\ t_1\ t_2) : \{(\texttt{<}\ t_1\ t_2), (\texttt{=}\ t_1\ t_2), (\texttt{<}\ t_2\ t_1)\}$

$(\texttt{total}\ t_1\ t_2) : \{(\texttt{<=}\ t_1\ t_2), (\texttt{<}\ t_2\ t_1)\}$

$(\texttt{total-int}\ t_1\ c) : \{(\texttt{<=}\ t_1\ c), (\texttt{<=}\ c+1\ t_1)\}$

where t_1 has type `Int`, c is an integer constant in canonical form and $c+1$ the same integer constant increased by one in canonical form.

Figure 3: The main axioms for linear integer and real arithmetic. The side conditions of some axioms require arithmetic on polynomials.

In the first phase, the input term t is inductively rewritten into a simplified term t'. If proof production is enabled, the solver builds a proof for $\{(\texttt{=}\ t\ t')\}$ at the same time it inductively computes t'. The proof mainly uses the `cong` axiom for the induction step and a set of solver-specific rewrite rules, such as `(oracle +(= (ite t1 true t2) (or t1 t2)) :rewrite iteBool3)`. These rewrite rules can later be easily replaced by their proof.

In the second phase, the formula is split and converted to CNF. If the top-level formula is a conjunction of smaller components, the formula is split into the components and the proof for each component is tracked. The proofs are already generated in the final form, e.g., by resolving the input formula with the `and-` axiom that corresponds to the component. If the formula is a disjunction, the subproof is resolved with the `or-` axiom to split the formula into its literals. The literals may be further rewritten into their internal form, which is again explained by rewrite rules in form of oracles. These rewrite steps are integrated into the proof of the clause. Finally the clause is added to the DPLL($\mathcal{T}$) engine and the clause keeps a reference to its proof. To avoid exponential blow-up of the CNF transformation, a nested `and` term below a disjunction is treated as a Boolean variable. A named literal is created and new clauses are created using the Plaisted–Greenbaum algorithm. These new clauses are annotated with their proof, which is an instance of the `and-` or `and+` axiom.

In the third phase, producing resolution proofs with the DPLL($\mathcal{T}$) algorithm is straightforward. Theory lemmas are added by theory-specific decision procedures. These lemmas are added as clauses to the DPLL($\mathcal{T}$) engine and are annotated with an oracle proof and explained later when the final proof is computed.

An earlier version of SMTInterpol used the rewrite rules and theory lemmas as its axioms and implemented a proof checker that checked these solver-specific rules. However, we found that writing a proof checker for custom rules leaves room for errors. Soundness errors in the proof checker are never caught, as testing is usually only done with correct proofs. While it is possible to write unit tests that specifically create invalid instances of proof rules, this is itself a

tedious task with no reward and was never done.

Therefore, we created a proof translator that replaces each solver-specific oracle rule with a corresponding proof using only the axioms. This gives much more confidence, as a wrong rule cannot be explained by a valid proof. The proof translator from the solver-specific rules to the minimal rules is roughly the same size as the old proof checker. The new proof checker only has to handle the much simpler and smaller set of axioms and is less than a third the size of the old proof checker. More importantly, with the new proof format the trusted core does not have to be changed when new rewrite rules are added to the solver. Only the proof translator is affected, which is producing a certificate of correctness.

6. Proof Techniques

Our proof format optimizes simplicity of the proof checker over simplicity of the prover. Nonetheless, we think that the language is expressive enough to also allow for advanced proof techniques that exponentially decrease the proof size. In this section we investigate this in detail.

These techniques are made possible by the expressiveness of SMT-LIB, which we use for literals, and by having a complete set of axioms. Normally, resolution is only refutation-complete, i.e., if p is valid, one can derive a contradiction from $\neg p$. Our format can prove the unit clause $\{p\}$ from the axioms if p is valid: The idea is to prove the clause $\{$`(! `p` :x)`$, \neg p\}$ from the axioms `(del! `p` :x)` and `(=-1 (! `p` :x) p))`. Using refutation completeness, a proof for $\{$`(! `p` :x)`$\}$ can be constructed, and then for $\{p\}$ using the axioms `del!` and `=-2`.

Logging Proofs is a technique where provers output their proofs on the fly into a file. The main reason for doing this is to keep the memory overhead low. The proofs can be forgotten after writing them to the proof file. Our proof format supports this technique using the `let-proof` command. Instead of remembering the proof for every clause, one can directly write the line

$$\texttt{(let-proof ((}C\ \ prf_C\texttt{))}$$

into the proof file, where C is a unique identifier for the clause and prf_C the proof for this clause. If C was proved by resolution from other clauses, prf_C can use the identifiers of the other clauses and combine them with the rule `res`. Deleted clauses can be indicated by assigning a trivial clause, e.g., `true+` to the previously used clause identifier.

To avoid repeating shared subterms, they can also be assigned to an identifier using `let` when they are first used. The prover just needs to give every subterm a unique identifier and subsequently use this identifier to refer to the subterm. To ensure that the parentheses match, the prover only needs to count the number of opening parentheses to add the same number of closing parentheses at the end.

Extended Resolution [2] introduces a new literal that stands for the conjunction of two existing literals together with corresponding clauses. This technique can be exponentially more succinct than using plain resolution proofs [3] and it can simulate DRAT proofs [4]. Extended resolution is already built into our proof format, since the conjunction of two literals can be

expressed as an SMT-LIB formula and the added clauses are just instances of the `and+` and `and-` axioms. Thus, an extended resolution proof can be expressed as follows.

```
(let ((p (and p1 p2))) (let-proof ((C1 (and+ p1 p2))
      (C2 (and- 0 p1 p2)) (C3 (and- 1 p1 p2))) ...))
```

Quantifier Introduction is usually done by a rule `all-intro` that takes a subproof prf_p for a formula p containing a free variable `x` and returns a proof for `(forall ((x` τ`))` p`)`. In our proof format, there is no such rule and also free variables are not allowed in proofs. Nonetheless it is possible to express this rule using resolution, `let` binders and the `forall+` axiom.

`(let ((x (choose (x` τ`) (not` p`)))) (res` p prf_p `(forall+ ((x` τ`))` p`)))`

Symmetry Breaking is a technique that can greatly reduce the search space by exploiting symmetry [5, 6]. The idea is to detect symmetry with respect to certain input variables, i.e., find a renaming of the constants such that the input formula is equivalent to itself after renaming. For example, consider a formula $F(c_1, c_2)$ with two uninterpreted constants c_1 and c_2. Assume that F is symmetric, i.e., $F(c_1, c_2)$ is equivalent to $F(c_2, c_1)$, e.g., it differs only in the order of the terms in commutative operators. If there is a clause `(or (p` c_1`) (p` c_2`))` in this formula, then w.l.o.g. one can assume that `(p` c_1`)` holds: if adding `(p` c_1`)` to F is unsatisfiable, it is clear by symmetry that adding `(p` c_2`)` to F is also unsatisfiable, hence F is unsatisfiable. Adding `(p` c_1`)` to F is called symmetry breaking, since after adding it, the formula is no longer symmetric with respect to this renaming. The reason why this is sound is because if `(p` c_1`)` does not hold, then `(p` c_2`)` holds. In this case we can swap c_1 and c_2 by defining $c'_1 := c_2$ and $c'_2 := c_1$. By symmetry now $F(c'_1, c'_2)$ still holds and additionally `(p` c'_1`)` holds.

This can be formally expressed in our proof format. First define $c'_1 :=$ `(ite (p` c_1`)` c_1 c_2`)` and $c'_2 :=$ `(ite (p` c_1`)` c_2 c_1`)`. These definitions can be introduced using the `let` keyword. Then show that $F(c_1, c_2) = F(c'_1, c'_2)$ holds by case distinction over `(p` c_1`)` and exploiting the symmetry if `(p` c_1`)` is false. Thus one can prove F' from F where F' is the formula containing c'_1 and c'_2. Now one can show from `(or (p` c_1`) (p` c_2`))` that `(p` c'_1`)` holds, again by case distinction over `(p` c_1`)`. The remaining proof is now done on the renamed formula, using c'_1 and c'_2, ignoring their definition but treating them as uninterpreted constants.

The proof for $F = F'$ is linear in the size of F and for each of the two cases proceeds by induction over F using `ite1`/`ite2` for the leaves, and an instance of `cong` for every subterm. Commutativity of logical operators needs to be explicitly proved. The proof of `(p` c'_1`)` is simple and requires `ite1`, `ite2`, congruence over `p`, and equivalence reasoning. An example proof using this technique is provided with the online proof checker on our website.

7. Related Work

Böhme and Weber [7] published guidelines for proof formats. Proofs should be human-readable (i.e., not in binary format), but easy to parse. They should include sufficient detail for the proof checker but avoid redundancy, e.g., clauses that were never used. The proof rules should be simple and canonical, and avoid special cases. Proofs should explicitly contain the proved facts.

Our proof format follows these guidelines in most aspects. For the last point, our format allows the prover to annotate each proof term with the clause it proves, but this is not mandatory.

For SAT solvers a standardized proof format is DRAT [8]. A DRAT proof consists of a list of clauses in such a way that each clause can be derived from the previous clauses by well-defined rules. However, DRAT assumes that the input is already in CNF and there is no support for theory lemmas needed in SMT. More importantly, the RAT property in each proof step must be checked against all (not yet deleted) input clauses. This means all used theory lemmas must be introduced in the beginning, otherwise the proof is unsound.

CVC5 uses LFSC [9, 10] for its proofs. LFSC can be seen as a language for writing proof checkers and provides a syntax for proof rules and a functional language for expressing non-syntactical side conditions. The rules and side conditions written in this language are part of the trusted core of the proof checker. CVC5 uses about 100 rules for core theory and arithmetic with about 40 subroutines to check side conditions. Some of the rules are similar to ours, e.g., they have a rule for resolution, where the resolvent is computed by the side condition. Overall, they have more rules and some of them are specific to the way their solver works.

Alethe[4] [11] is also based on resolution, but has a few rules following natural deduction style. Facts are represented as clauses similar to us. In contrast to our format, they require to give the proved clause for every intermediate step, but they allow for more complicated reasoning like hyper-resolution to combine several resolutions into one proof step. The Alethe specification mentions that the theoretical complexity of checking their resolution rule is NP-complete. Compared to our format, Alethe contains more rules and axioms. Alethe has a rule for let expansion but expansion of defined function (and named terms) is automatic, which is the opposite of what our format does. Our feeling is that automatic function expansion is problematic, because substituting function arguments can lead to exponential blow-up in the worst case. Also `let` enables us to express the `forall-` axiom syntactically.

The proof format of OpenSMT [12] is more a collection of proof formats. They use certificates to explain CNF conversion using Tseitin and De Morgan rules. The theory lemmas are explained by certificates in a theory-specific format. Each certificate is specific to the rules used by the solver, and so is the certificate checker. The remaining proof is purely propositional and uses the DRAT format.

Quip[5] is an experimental proof format for first-order and higher-order logic. It aims at proofs that are easy to produce by providing high-level proof rules and redundant rules (e.g. resolution and unit resolution). In return, the proof checker needs to perform more complex reasoning, e.g. congruence closure and other theory-specific reasoning. The format is resolution-based and also uses clauses with SMT-LIB formulas, but it also supports hyperresolution steps. Shared terms are abbreviated by the top-level command `deft`, which corresponds to `let` in our format.

The Z3 proof format [13] is based on natural deduction. Interestingly, it does not have a generic resolution rule. Instead it has a lemma rule for RUP steps that adds all literals negated as hypothesis and then only uses unit resolution on these hypotheses and existing clauses. The format uses mainly proof rules, e.g. for symmetry, transitivity, and congruence, using the hypothesis rule for decided literals that end up negated in the proved clause. For input formulas

[4]Specification available at https://verit.loria.fr/documentation/alethe-spec.pdf
[5]See https://github.com/c-cube/quip and https://c-cube.github.io/quip-book

it uses a rewrite rule whose side condition is that the rewritten term is equal to the original (they are similar to our oracle axioms in that they are not proved). For NNF encoding they use quoted formulas, which means they use SMT-LIB formulas as literals like we do. The paper does not mention how `let`-terms are handled, but the proof format uses them in the same way as our format and, like our `let-proof`, to abbreviate common subproofs. Proofs are SMT-LIB terms of sort `Proof`.

8. Conclusion and Discussion

We presented a simple proof format for SMT based on the resolution rule. Its main feature is its restriction to only a few basic axioms. In total we currently define one rule (resolution) and 60 axioms for the core theory, arithmetic, arrays, and datatypes, which is on average two axioms per theory-defined function. By implementing this proof format in our SMT solver, we have shown that despite its simplicity, the format can succinctly express the reasoning steps done by our solver. We have also shown that the format can handle more advanced reasoning techniques like symmetry breaking.

Every single proof format may favour some solvers while making it more difficult for other solvers to express their proofs. Our format is based on the resolution rule, which favours solvers based on DPLL($\mathcal{T}$). Moreover, the axioms for linear arithmetic, especially the axiom `farkas`, are inspired by the theory lemmas that our solver produces. However, other proof formats like CVC5's LFSC, Alethe, and Quip are also based on resolution and Alethe has a rule `la_generic` that is almost identical to `farkas`. In fact, most of our axioms are also contained in Alethe in the same or a similar form. Our experience with SMTInterpol gives us confidence that proofs from SMT solvers can be translated into this format with moderate overhead.

In our format, rules with assumptions are expressed as axioms, e.g., modus ponens is expressed as `=>-` where the premises are negated literals and the conclusion a positive literal. The axiom form is more flexible, e.g., introduction rules can also be used to eliminate negated operators. Also the axioms are needed in this form to explain the Tseitin encoding.

The main reason we do not use the SMT-LIB operators `or`/`not` to represent clauses or negated literals is to make a clean separation between the meta-language (resolution proofs) and the language in which the formulas are written. This avoids ambiguity, whether a term `(or` p_1 p_2`)` represents a Tseitin literal or a clause. It also avoids problems with double negation; other proof formats interpret terms starting with `not` as negated literal but double negated literals are seen as positive. Note that the only place we use a concrete syntax for clauses is the oracle rule.

Implicitly expanding `let` terms seems to go against the goal of a low-level proof format, but this is essential. First, a way of defining names for repeatedly used terms or clauses is needed to avoid exponential blow-up. Second, our solver and proof checker represent formulas and proofs internally using directed acyclic graphs (DAGs). In a way, `let` is just a concrete syntax to represent DAGs in textual form. Finally, `let` is the SMT-LIB syntax for substituting variables with terms, which is used in the quantifier elimination and introduction axioms and the expand axiom for function definitions. Thus, we also use it to avoid inventing a new syntax for substitutions.

Acknowledgments

This work is supported by the German Research Council (DFG) under HO 5606/1-2.

References

[1] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Technical Report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[2] G. S. Tseitin, On the complexity of derivation in propositional calculus, in: Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970, Springer, 1983, pp. 466–483. URL: https://doi.org/10.1007/978-3-642-81955-1_28. doi:10.1007/978-3-642-81955-1_28.

[3] S. A. Cook, A short proof of the pigeon hole principle using extended resolution, SIGACT News 8 (1976) 28–32.

[4] B. Kiesl, A. Rebola-Pardo, M. J. H. Heule, Extended resolution simulates DRAT, in: IJCAR, volume 10900 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 516–531.

[5] H. Katebi, K. A. Sakallah, I. L. Markov, Symmetry and satisfiability: An update, in: SAT, volume 6175 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 113–127.

[6] D. Déharbe, P. Fontaine, S. Merz, B. W. Paleo, Exploiting symmetry in SMT problems, in: CADE, volume 6803 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 222–236.

[7] S. Böhme, T. Weber, Designing proof formats: A user's perspective, in: PxTP, 2011, pp. 27–32.

[8] M. J. H. Heule, The DRAT format and DRAT-trim checker, CoRR abs/1610.06229 (2016).

[9] A. Stump, D. Oe, Towards an SMT proof format, in: SMT '08/BPR '08, ACM, 2008, p. 27–32. URL: https://doi.org/10.1145/1512464.1512470. doi:10.1145/1512464.1512470.

[10] A. Stump, D. Oe, A. Reynolds, L. Hadarean, C. Tinelli, SMT proof checking using a logical framework, Formal Methods Syst. Des. 42 (2013) 91–118.

[11] H. Schurr, M. Fleury, H. Barbosa, P. Fontaine, Alethe: Towards a generic SMT proof format (extended abstract), in: PxTP, volume 336 of *EPTCS*, 2021, pp. 49–54.

[12] R. Otoni, M. Blicha, P. Eugster, A. E. J. Hyvärinen, N. Sharygina, Theory-specific proof steps witnessing correctness of SMT executions, in: DAC, IEEE, 2021, pp. 541–546.

[13] L. M. de Moura, N. Bjørner, Proofs and refutations, and Z3, in: LPAR Workshops, volume 418 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2008.

A. Axioms of the Core Theory

Axioms for the logical operators:

$$\texttt{true+} : \{\texttt{true}\}$$
$$\texttt{false-} : \{\neg\texttt{false}\}$$
$$\texttt{(not+}\ p_0\texttt{)} : \{\texttt{(not}\ p_0\texttt{)}, p_0\}$$
$$\texttt{(not-}\ p_0\texttt{)} : \{\neg\texttt{(not}\ p_0\texttt{)}, \neg p_0\}$$
$$\texttt{(and+}\ p_0 \ldots p_n\texttt{)} : \{\texttt{(and}\ p_0\ \ldots\ p_n\texttt{)}, \neg p_0, \ldots, \neg p_n\}$$
$$\texttt{(and-}\ i\ p_0 \ldots p_n\texttt{)} : \{\neg\texttt{(and}\ p_0\ \ldots\ p_n\texttt{)}, p_i\}$$
$$\texttt{(or+}\ i\ p_0 \ldots p_n\texttt{)} : \{\texttt{(or}\ p_0\ \ldots\ p_n\texttt{)}, \neg p_i\}$$
$$\texttt{(or-}\ p_0 \ldots p_n\texttt{)} : \{\neg\texttt{(or}\ p_0\ \ldots\ p_n\texttt{)}, p_0, \ldots, p_n\}$$
$$\texttt{(=>+}\ i\ p_0 \ldots p_n\texttt{)} : \{\texttt{(=>}\ p_0\ \ldots\ p_n\texttt{)}, (\neg)\, p_i\} \qquad (p_i \text{ is negated for } i = n)$$
$$\texttt{(=>-}\ p_0 \ldots p_n\texttt{)} : \{\neg\texttt{(=>}\ p_0\ \ldots\ p_n\texttt{)}, \neg p_0, \ldots, p_n\}$$
$$\texttt{(=+1}\ p_0\ p_1\texttt{)} : \{\texttt{(=}\ p_0\ p_1\texttt{)}, p_0, p_1\}$$
$$\texttt{(=-1}\ p_0\ p_1\texttt{)} : \{\neg\texttt{(=}\ p_0\ p_1\texttt{)}, p_0, \neg p_1\}$$
$$\texttt{(=+2}\ p_0\ p_1\texttt{)} : \{\texttt{(=}\ p_0\ p_1\texttt{)}, \neg p_0, \neg p_1\}$$
$$\texttt{(=-2}\ p_0\ p_1\texttt{)} : \{\neg\texttt{(=}\ p_0\ p_1\texttt{)}, \neg p_0, p_1\}$$
$$\texttt{(xor+ (}l_1\texttt{) (}l_2\texttt{) (}l_3\texttt{))} : \{\texttt{(xor}\ l_1\texttt{)}, \texttt{(xor}\ l_2\texttt{)}, \neg\texttt{(xor}\ l_3\texttt{)}\}$$
$$\texttt{(xor- (}l_1\texttt{) (}l_2\texttt{) (}l_3\texttt{))} : \{\neg\texttt{(xor}\ l_1\texttt{)}, \neg\texttt{(xor}\ l_2\texttt{)}, \neg\texttt{(xor}\ l_3\texttt{)}\}$$

where l_1,l_2,l_3 are lists of terms with each term occurring an even number in total and if l_i is only one term, `(xor` l_i`)` stands for the term l_i.

Axioms for quantifiers:

$$\texttt{(forall+ ((}x_1\ \tau_1\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{)} :$$
$$\{\texttt{(forall ((}x_1\ \tau_1\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{)},$$
$$\neg\texttt{(let ((}x_1\ \texttt{(choose (}x_1\ \tau_1\texttt{) (forall ((}x_2\ \tau_2\texttt{) ... (}x_n\ \tau_n\texttt{)) (not }p\texttt{)))))}$$
$$\texttt{(let ((}x_2\ \texttt{(choose (}x_2\ \tau_2\texttt{) (forall ((}x_3\ \tau_3\texttt{) ... (}x_n\ \tau_n\texttt{)) (not }p\texttt{)))))}$$
$$\vdots$$
$$\texttt{(let ((}x_n\ \texttt{(choose (}x_n\ \tau_n\texttt{) (not }p\texttt{)))) }p\texttt{)}\cdots\texttt{))}\}$$
$$\texttt{(forall- ((}x_1\ t_1\texttt{) ... (}x_n\ t_n\texttt{)) }p\texttt{)} :$$
$$\{\neg\texttt{(forall ((}x_1\ \tau_1\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{)}, \texttt{(let ((}x_1\ t_1\texttt{)...(}x_n\ t_n\texttt{)) }p\texttt{)}\}$$
$$\texttt{(exists+ ((}x_1\ t_1\texttt{) ... (}x_n\ t_n\texttt{)) }p\texttt{)} :$$
$$\{\texttt{(exists ((}x_1\ \tau_1\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{)}, \neg\texttt{(let ((}x_1\ t_1\texttt{)...(}x_n\ t_n\texttt{)) }p\texttt{)}\}$$
$$\texttt{(exists- ((}x_1\ \tau_1\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{)} :$$
$$\{\neg\texttt{(exists ((}x_1\ \tau_1\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{)},$$
$$\texttt{(let ((}x_1\ \texttt{(choose (}x_1\ \tau_1\texttt{) (exists ((}x_2\ \tau_2\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{))))}$$
$$\texttt{(let ((}x_2\ \texttt{(choose (}x_2\ \tau_2\texttt{) (exists ((}x_3\ \tau_3\texttt{) ... (}x_n\ \tau_n\texttt{)) }p\texttt{))))}$$
$$\vdots$$
$$\texttt{(let ((}x_n\ \texttt{(choose (}x_n\ \tau_n\texttt{) }p\texttt{))) }p\texttt{)}\cdots\texttt{))}\}$$

In the axioms `forall-` and `exists+` the sorts τ_i are automatically determined from the type of the expressions t_i.

Axioms for equality:

$$\texttt{(refl } t\texttt{)} : \{\texttt{(= } t\ t\texttt{)}\}$$
$$\texttt{(symm } t_0\ t_1\texttt{)} : \{\texttt{(= } t_0\ t_1\texttt{)}, \neg\texttt{(= } t_1\ t_0\texttt{)}\}$$
$$\texttt{(trans } t_0 \ldots t_n\texttt{)} : \{\texttt{(= } t_0\ t_n\texttt{)}, \neg\texttt{(= } t_0\ t_1\texttt{)}, \ldots, \neg\texttt{(= } t_{n-1}\ t_n\texttt{)}\}$$
$$\texttt{(cong (f } t_0 \ldots t_n\texttt{) (f } t'_0 \ldots t'_n\texttt{))}: \{\texttt{(= (f } t_0\ \ldots\ t_n\texttt{) (f } t'_0\ \ldots\ t'_n\texttt{))}, \neg\texttt{(= } t_0\ t'_0\texttt{)}, \ldots, \neg\texttt{(= } t_n\ t'_n\texttt{)}\}$$
$$\texttt{(=+ } t_0 \ldots t_n\texttt{)} : \{\texttt{(= } t_0\ \ldots\ t_n\texttt{)}, \neg\texttt{(= } t_0\ t_1\texttt{)}, \ldots, \neg\texttt{(= } t_{n-1}\ t_n\texttt{)}\}$$
$$\texttt{(=- } i\ j\ t_0 \ldots t_n\texttt{)} : \{\neg\texttt{(= } t_0\ \ldots\ t_n\texttt{)}, \texttt{(= } t_i\ t_j\texttt{)}\} \quad \text{where } i \neq j$$
$$\texttt{(distinct+ } t_0 \ldots t_n\texttt{)} : \{\texttt{(distinct } t_0\ \ldots\ t_n\texttt{)}, \texttt{(= } t_0\ t_1\texttt{)}, \ldots, \texttt{(= } t_0\ t_n\texttt{)}, \ldots, \texttt{(= } t_{n-1}\ t_n\texttt{)}\}$$
$$\texttt{(distinct- } i\ j\ t_0 \ldots t_n\texttt{)} : \{\neg\texttt{(distinct } t_0\ \ldots\ t_n\texttt{)}, \neg\texttt{(= } t_i\ t_j\texttt{)}\} \quad \text{where } i \neq j$$

Axioms for `ite`, annotations and `define-fun`:

$$\texttt{(ite1 } p_0\ t_1\ t_2\texttt{)} : \{\texttt{(= (ite } p_0\ t_1\ t_2\texttt{) } t_1\texttt{)}, \neg p_0\}$$
$$\texttt{(ite2 } p_0\ t_1\ t_2\texttt{)} : \{\texttt{(= (ite } p_0\ t_1\ t_2\texttt{) } t_2\texttt{)}, p_0\}$$
$$\texttt{(del! } t\ \ldots\texttt{)} : \{\texttt{(= (! } t\ \ldots\texttt{) } t\texttt{)}\}$$
$$\texttt{(expand (f } t_0 \ldots t_n\texttt{))} : \{\texttt{(= (f } t_0\ \ldots\ t_n\texttt{) (let ((}x_0\ t_0\texttt{)}\ldots\texttt{(}x_n\ t_n\texttt{)) } t\texttt{))}\}$$

where `f` is defined by `(define-fun f ((`$x_0\ \tau_0$`)...(`$x_n\ \tau_n$`)) t)`

Here are two examples for the `xor` axioms:

$$\texttt{(xor+ (}p_1\ p_2\ p_3\texttt{) (}p_2\texttt{) (}p_1\ p_3\texttt{))} : \{\texttt{(xor } p_1\ p_2\ p_3\texttt{)}, p_2, \neg\texttt{(xor } p_1\ p_3\texttt{)}\}$$
$$\texttt{(xor- (}p\ p\texttt{) (}p\ p\texttt{) (}p\ p\texttt{))} : \{\neg\texttt{(xor } p\ p\texttt{)}\}$$

B. Axioms of Linear Arithmetic

$$\texttt{(poly+ (+ } g_1 \ldots g_n\texttt{) } g\texttt{)} : \{\texttt{(= (+ } g_1 \ldots g_n\texttt{) } g\texttt{)}\} \quad \text{where } g = g_1 + \cdots + g_n$$
$$\texttt{(poly* (* } g_1 \ldots g_n\texttt{) } g\texttt{)} : \{\texttt{(= (* } g_1 \ldots g_n\texttt{) } g\texttt{)}\} \quad \text{where } g = g_1 \cdots g_n$$

$$\texttt{(farkas } c_1\ \texttt{(<=? } g_1\ g'_1\texttt{)}\ \ldots\ c_n\ \texttt{(<=? } g_n\ g'_n\texttt{))} : \{\neg\texttt{(<=? } g_1\ g'_1\texttt{)}, \ldots, \neg\texttt{(<=? } g_n\ g'_n\texttt{)}\}$$

where c_i are positive integer constants, the weighted sum $c_1(g_1 - g'_1) + \cdots + c_n(g_n - g'_n)$ is a non-negative constant. `<=?` stands for `<`, `<=`, or `=`, and if the weighted sum is zero, at least one of the literals must be `<`.

$$\texttt{(trichotomy } t_1\ t_2\texttt{)} : \{\texttt{(< } t_1\ t_2\texttt{)}, \texttt{(= } t_1\ t_2\texttt{)}, \texttt{(< } t_2\ t_1\texttt{)}\}$$
$$\texttt{(total } t_1\ t_2\texttt{)} : \{\texttt{(<= } t_1\ t_2\texttt{)}, \texttt{(< } t_2\ t_1\texttt{)}\}$$
$$\texttt{(total-int } t_1\ c\texttt{)} : \{\texttt{(<= } t_1\ c\texttt{)}, \texttt{(<= } c+1\ t_1\texttt{)}\}$$

where t_1 has type `Int`, c is an integer constant in canonical form

and $c+1$ the same integer constant increased by one in canonical form.

(to_real g) : {(= (to_real g) g')}
where g' is the result of casting the integer polynomial g to real.

(>def t_1 t_2) : {(= (> t_1 t_2) (< t_2 t_1))}
(>=def t_1 t_2) : {(= (>= t_1 t_2) (<= t_2 t_1))}
(/def t_1 t_2 ... t_n) : {(= t_1 (* t_2 ... t_n (/ t_1 t_2 ... t_n))), (= t_2 0), ..., (= t_n 0)}
(-def t_1) : {(= (- t_1) (* (- 1) t_1))}
(-def t_1 t_2 ... t_n) : {(= (- t_1 t_2 ... t_n) (+ t_1 (* (- 1) t_2)) ... (* (- 1) t_n))}
(to_int-low t_1) : {(<= (to_real (to_int t_1)) t_1)}
(to_int-high t_1) : {(< t_1 (+ (to_real (to_int t_1)) 1.0))}
(abs-def t_1) : {(= (abs t_1) (ite (< t_1 0) (- t_1) t_1))}
(div-low t_1 t_2) : {(<= (* t_2 (div t_1 t_2)) t_1), (= t_2 0)}
(div-high t_1 t_2) : {(< t_1 (+ (* t_2 (div t_1 t_2)) (abs t_2))), (= t_2 0)}
(mod-def t_1 t_2) : {(= (mod t_1 t_2) (- t_1 (* t_2 (div t_1 t_2)))), (= t_2 0)}
(divisible-def c t_1) : {(= ((_ divisible c) t_1) (= t_1 (* c (div t_1 c))))}

C. Axioms of Arrays

We support an extension of the array theory with `@diff` for returning the elements where two arrays differ and `@const` for the array that stores the same element for all indices. In these axioms a stands for a term of array type, i for a term of index type and v for a term of element type.

(selectstore1 a i v) : {(= (select (store a i v) i) v)}
(selectstore2 a i_1 i_2 v) : {(= (select (store a i_1 v) i_2) (select a i_2)), (= i_1 i_2)}
(extdiff a_0 a_1) : {(= a_0 a_1), ¬(= (select a_0 (@diff a_0 a_1))
(select a_1 (@diff a_0 a_1)))}
(const v i) : {(= (select (@const v) i) v)}

D. Axioms of Data Types

In the following axioms, $c_1 \ldots, c_n$ are the constructors $s_{i1}, \ldots s_{im_i}$ are the selectors of c_i.

(dt_project s_{ij} $t_1 \ldots t_{m_i}$) : {(= (s_{ij} (c_i $t_1 \ldots t_{m_i}$)) t_j)}
(dt_cons c_i t) : {¬((_ is c_i) t), (= (c_i (s_{i1} t) ... (s_{im_i} t)) t)}
(dt_test+ c_i (c_i $t_1 \ldots t_{m_i}$)) : {((_ is c_i) (c_i $t_1 \ldots t_{m_i}$))}
(dt_test- c_j (c_i $t_1 \ldots t_{m_i}$)) : {¬((_ is c_j) (c_i $t_1 \ldots t_{m_i}$))} where $j \neq i$
(dt_exhaust t) : {((_ is c_1) t), ..., ((_ is c_n) t)}

`(dt_acyclic` t_0 `(`$i_1 \ldots i_n$`))` :{¬`(=` t_0 t_n`)`}

where $n \geq 1$, t_j is the i_j-th argument of t_{j-1} and t_{j-1} is an application of a constructor for $j = 1, \ldots, n$. Hence, t_n appears nested inside the constructor term t_0.

`(dt_match (match` t `...))` :{`(= (match t ((`c_i $x_1 \ldots x_{m_i}$`)` t_i`) ...)`
`(ite ((_ is` c_i`) t)`
`(let ((`x_1 `(`s_{i1} t`))...(`x_{m_i} `(`s_{im_i} t`))) `t_i`) ...)`}