

# Documentation of VERIPB and CAKEPB for the SAT Competition 2026

Markus Anders      Bart Bogaerts      Benjamin Bogø      Arthur Gontier  
Wietze Koops      Ciaran McCreesh      Magnus O. Myreen      Jakob Nordström  
Adrián Rebola-Pardo      Andy Oertel      Yong Kiam Tan

March 20, 2026

## Abstract

This is the documentation for the pseudo-Boolean proof checker VERIPB together with its formally verified backend CAKEPB as proposed for usage in the SAT competition 2026. If there are questions regarding corner cases not covered by this documentation, or regarding how to use pseudo-Boolean proof logging to certify correctness of different forms of reasoning, inquiries are welcome and may be directed to `jn@di.ku.dk`.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Quickstart Guide for Boolean Satisfiability (SAT) Proof Logging</b>	<b>2</b>
2.1	Running the Proof Checkers	2
2.2	Proof Format	3
<b>3</b>	<b>Pseudo-Boolean and Proof Checker Preliminaries</b>	<b>4</b>
3.1	Pseudo-Boolean Notation and Terminology	4
3.2	Assignments, Substitutions, Slack, and Unit Propagation	5
3.3	Syntax for Pseudo-Boolean Inequalities in Proofs	6
3.4	General Principles for the Proof Checker	6
3.5	General Principles for the Proof Checker Syntax	6
<b>4</b>	<b>Overall Proof Structure</b>	<b>7</b>
4.1	Proof Sections	7
4.2	Output Section	7
4.3	Conclusions Section	9
<b>5</b>	<b>Derivation Rules</b>	<b>11</b>
5.1	Manipulation of Constraints Database	12
5.2	Implicational Rules	12
5.2.1	Implicational Rules in Kernel Format	12
5.2.2	Syntactic Implication Rules in Augmented Format	14
5.3	Orders	14
5.4	Strengthening Rules	19
5.4.1	Redundance-Based Strengthening	19
5.4.2	Dominance-Based Strengthening	21
5.5	Subproofs	23
5.5.1	Autoproven Proof Goals in Kernel Format	25

5.5.2	Autoproven Proof Goals in Augmented Format	25
5.6	Deletion Rules	26
5.6.1	Deletion Rules in Kernel Format	26
5.6.2	Deletion Rules in Augmented Format	26
5.6.3	Semantics for Mixed Deletion by Reference and Specification	27
<b>6</b>	<b>Formally Verified Proof Checking</b>	<b>27</b>
6.1	Summary of Kernel Format	27
6.2	Verified Correctness Theorem for CAKEPB_CNF	28
6.3	Complexity	29

## 1 Introduction

The pseudo-Boolean proof format used for the proof checker VERIPB [Ver] supports proof logging for decision, enumeration, and optimization problems, as well as problem reformulations, all in a unified format. So far, VERIPB has been used for proof logging of enhanced SAT solving techniques [GN21, BGMN23], constraint programming [EGMN20, GMN22, MM23, MMN24], pseudo-Boolean CDCL-based solving [GMNO22], subgraph solving [GMN20, GMM<sup>+</sup>20, GMM<sup>+</sup>24], presolving in 0–1 integer linear programming [HOGN24], dynamic programming and decision diagrams [DMM<sup>+</sup>24], and MaxSAT solving [VDB22, BBN<sup>+</sup>23, IOT<sup>+</sup>24, BBN<sup>+</sup>24], and this list of applications is expected to keep growing. In this technical documentation paper we present a recently revised version of the proof format, focusing on how it can be used to certify unsatisfiability of CNF formulas in the SAT competition 2026.

The full proof format makes it possible to specify *different types of proofs*, where it might only be known towards the end of the proof what kind of proof was produced (e.g., if a preprocessor did not just reformulate a problem but actually solved it). However, in this document we focus on the restricted version of the format that is proposed to be supported in the SAT competition 2026.

## 2 Quickstart Guide for Boolean Satisfiability (SAT) Proof Logging

This section contains the bare minimum of information needed to use VERIPB and CAKEPB as proof checkers for Boolean satisfiability (SAT) solvers with pseudo-Boolean proof logging. A good way to learn more (in addition to reading this document) might be to study the example files in the directory `tests/integration_tests/correct/` in the repository [Ver] and run VERIPB with the option `--trace`, which will output detailed information about the proofs and the proof checking.

### 2.1 Running the Proof Checkers

If a SAT solver with pseudo-Boolean proof logging has solved the instance `input.cnf`, the generated proof `input.pbp` can be checked by VERIPB and CAKEPB by running the following commands:

```
# Translate to kernel format proof
veripb -u --cnf --elaborate translated.pbp input.cnf input.pbp
# Check the kernel proof
cake_pb_cnf input.cnf translated.pbp
```

The first command recompiles the pseudo-Boolean proof `input.pbp` into a more restricted “kernel-format” proof `translated.pbp` using VERIPB, after which the kernel proof is checked using CAKEPB. In case of successful recompilation, VERIPB will output:

```
# Running veripb as shown above
...
s VERIFIED UNSATISFIABLE
```

Upon successful proof checking, CAKEPB will report success on the standard output stream:

```
# Running cake_pb_cnf as shown above
s VERIFIED UNSATISFIABLE
```

All errors are reported on standard error.

## 2.2 Proof Format

The syntactic format of a pseudo-Boolean proof of unsatisfiability for a CNF formula as expected by the version of VERIPB proposed for the SAT competition 2026 is

```
pseudo-Boolean proof version 3.0
[f <N> ;]
<derivation section>
output NONE;
conclusion UNSAT [: <id> ];
end pseudo-Boolean proof;
```

where square brackets [...] denote optional parts of the syntax. When specified,  $\langle N \rangle$  must be the number of clauses in the formula. The  $\langle \text{derivation section} \rangle$  should contain the actual proof which derives contradiction. The constraint ID  $\langle id \rangle$  specifies which pseudo-Boolean constraint in the  $\langle \text{derivation section} \rangle$  that is a contradiction. In the kernel format, the line  $f \langle N \rangle ;$  and the constraint ID  $\langle id \rangle$  are mandatory.

Although this document focuses on the VERIPB 3.0 format, we also still support the older VERIPB 2.0 format as supported in the SAT competition 2023 [BMM<sup>+</sup>23]. However, new features will only be supported in the VERIPB 3.0 format.

In pseudo-Boolean format, a disjunctive clause like

$$x_1 \vee \bar{x}_2 \vee x_3 \tag{2.1a}$$

is represented as the inequality

$$x_1 + \bar{x}_2 + x_3 \geq 1 \tag{2.1b}$$

claiming that at least one of the literals in the clause is true (i.e., takes value 1), and this inequality is written as

```
+1 x1 +1 ~x2 +1 x3 >= 1
```

in the OPB format [RM16] used by VERIPB (except that we do not terminate constraints with a semicolon). The proof checker can also read CNF formulas in the DIMACS and WDIMACS formats used for SAT solving and MaxSAT solving, respectively. For such files, VERIPB will parse a clause

```
1 -2 3 0
```

to be identical to the constraint (2.1b), and the variables should be referred to in the pseudo-Boolean proof file as  $x_1$ ,  $x_2$ ,  $x_3$ , et cetera.

DRAT proofs [WHH14] can be transformed into valid VERIPB proofs by simple syntactic manipulations. Most of the proof resulting from a CDCL SAT solver is the ordered sequence of clauses learned during conflict analysis. Since all such clauses are guaranteed to be *reverse unit propagation* (RUP) clauses, the easiest way to provide pseudo-Boolean proof logging for a learned clause (2.1a) would be to write

```
rup +1 x1 +1 ~x2 +1 x3 >= 1;
```

in the derivation section of the pseudo-Boolean proof (as explained in more detail in Section 5.2.2).

If instead the clause (2.1a) is a *resolution asymmetric tautology* (RAT) clause [JHB12, HHW13] that is RAT on the literal  $x_1$ , then this is written as

```
red +1 x1 +1 ~x2 +1 x3 >= 1 : x1 -> 1;
```

in the pseudo-Boolean proof using the more general *redundance-based strengthening* rule (discussed in Section 5.4.1). And if the RAT literal would instead have been  $\bar{x}_2$ , this would have been indicated by ending the proof line above by  $x_2 \rightarrow 0$  instead.

Finally, in order to delete the clause (2.1a), the deletion command

```
del spec +1 x1 +1 ~x2 +1 x3 >= 1;
```

is issued. (This and other deletion rules are covered in Section 5.6.) An important difference from DRAT proofs is that deletion is made also for unit clauses, i.e., clauses containing only a single literal—DRAT proof checkers typically ignore such deletion commands [AR20]. Another crucial difference is that all clauses learned during CDCL execution need to be written down in the proof log, including unit clauses. If unit clauses are missing in a DRAT proof, the proof checkers will typically be helpful and silently infer and add the missing clauses. No such patching of formally incorrect proofs is offered by VERIPB.

It should be noted, though, that if all the reasoning performed by some particular SAT solver can efficiently be captured by standard DRAT proof logging, then there is no real reason to use pseudo-Boolean proof logging for that solver. Pseudo-Boolean proof logging becomes relevant only if the solver uses more advanced techniques such as, for instance, cardinality reasoning, Gaussian elimination, or symmetry breaking. We refer the reader to [GN21] and [BGMN23], respectively, for detailed descriptions of how to do efficient pseudo-Boolean proof logging for the latter two techniques.

### 3 Pseudo-Boolean and Proof Checker Preliminaries

In this section, we briefly review some pseudo-Boolean preliminaries and general principles for how pseudo-Boolean proof checking works. We refer to the survey chapter [BN21] for more details on the cutting plane proof system.

#### 3.1 Pseudo-Boolean Notation and Terminology

A *literal*  $\ell$  over a Boolean variable  $x$  is  $x$  itself or its negation  $\bar{x} = 1 - x$ , where variables take values 0 (false) or 1 (true). A *pseudo-Boolean (PB) inequality* is a 0–1 linear inequality

$$C \doteq \sum_i a_i \ell_i \geq A, \tag{3.1}$$

where  $a_i$  and  $A$  are integers (and where we write  $\doteq$  to denote syntactic equality). We can assume without loss of generality that pseudo-Boolean constraints are *normalized*; i.e., that all literals  $\ell_i$  are over distinct variables and that the *coefficients*  $a_i$  are non-negative. This is how constraints are represented internally in the proof checker, but most of the time there is no need to worry about this, and the proof checker accepts constraints written in non-normalized form.

A *pseudo-Boolean formula* is a conjunction  $F \doteq \bigwedge_j C_j$  of pseudo-Boolean inequalities, which we can also think of as the set  $\bigcup_j \{C_j\}$  of inequality constraints in the formula. Since a (*disjunctive*) clause  $\ell_1 \vee \dots \vee \ell_k$  is equivalent to the pseudo-Boolean constraint  $\ell_1 + \dots + \ell_k \geq 1$ , formulas in *conjunctive normal form (CNF)* can be viewed as special cases of pseudo-Boolean formulas.

To introduce some further convenient notation, we write equality  $\sum_i a_i \ell_i = A$  as syntactic sugar for the pair of pseudo-Boolean inequalities  $\sum_i a_i \ell_i \geq A$  and  $\sum_i -a_i \ell_i \geq -A$ . The *negation*  $\neg C$  of the constraint  $C$  in (3.1) can be represented as the pseudo-Boolean inequality

$$\neg C \doteq \sum_i -a_i \ell_i \geq -A + 1, \tag{3.2}$$

and the fact that the set of pseudo-Boolean inequalities is closed under negation is quite convenient for proof logging purposes. If  $z$  is a Boolean variable and  $\sum_i a_i \ell_i \geq A$  is a pseudo-Boolean inequality in normalized form with  $\sum_i a_i = M$ , then we write

$$z \Rightarrow \sum_i a_i \ell_i \geq A \doteq A \cdot \bar{z} + \sum_i a_i \ell_i \geq A \tag{3.3a}$$

to denote the *right reification* and

$$z \Leftarrow \sum_i a_i \ell_i \geq A \doteq (M - A + 1) \cdot z + \sum_i a_i \bar{\ell}_i \geq M - A + 1 \quad (3.3b)$$

for the *left reification* of  $\sum_i a_i \ell_i \geq A$ . As the notation suggests, the constraints (3.3a)–(3.3b) enforce that  $z$  is true if and only if  $\sum_i a_i \ell_i \geq A$  holds.

A constraint  $C$  is *weakly syntactically implied* by  $D$  if  $C$  can be derived from  $D$  by adding literal axioms. A constraint  $C$  is *syntactically implied* by  $D$  if  $C$  can be derived from  $D$  by adding literal axioms, doing a saturation step, and again adding literal axioms.

### 3.2 Assignments, Substitutions, Slack, and Unit Propagation

A (*partial*) *assignment*  $\rho$  is a (partial) function from variables to  $\{0, 1\}$ ; a *substitution*  $\omega$  can also map variables to literals. These are extended from variables to literals in the natural way by respecting the meaning of negation. We also identify a partial assignment  $\rho$  with the set of literals set to true by  $\rho$ , so that  $\ell \in \rho$  if and only if  $\rho(\ell) = 1$ . We can write  $x \mapsto b$  when  $\rho(x) = b$ , for  $b$  a literal or truth value.

We write  $\rho \circ \omega$  to denote the composed substitution resulting from applying first  $\omega$  and then  $\rho$ , i.e.,  $\rho \circ \omega(x) = \rho(\omega(x))$ . As an example, for  $\omega = \{x_1 \mapsto 0, x_3 \mapsto \bar{x}_4, x_4 \mapsto x_3\}$  and  $\rho = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto 0\}$  we have  $\rho \circ \omega = \{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 0\}$ . Applying  $\omega$  to a pseudo-Boolean inequality  $C$  as in (3.1) yields

$$C|_\omega \doteq \sum_i a_i \omega(\ell_i) \geq A, \quad (3.4)$$

substituting literals or values as specified by  $\omega$ . For a formula  $F$  we define  $F|_\omega \doteq \bigwedge_j C_j|_\omega$ . For a list of variables  $\vec{x} = x_1, \dots, x_n$ , we define  $\vec{x}|_\omega = \omega(x_1), \dots, \omega(x_n)$  to be elementwise application of  $\omega$  to  $\vec{x}$ . If  $F$  is a pseudo-Boolean formula over variables  $\vec{x}$ , we can write  $F(\vec{x})$  to make explicit the list of variables  $\vec{x}$  over which  $F$  is defined. For a list of literals or truth values  $\vec{b} = b_1, \dots, b_n$ , the notation  $F(\vec{b})$  is syntactic sugar for  $F|_\omega$  with  $\omega$  denoting the substitution (implicitly) defined by  $\omega(x_i) = b_i$  for  $i = 1, \dots, n$ .

The (normalized) pseudo-Boolean inequality  $C$  in (3.1) is *satisfied* by  $\rho$  if  $\sum_{\ell_i \in \rho} a_i \geq A$ . A pseudo-Boolean formula  $F$  is satisfied by  $\rho$  if all constraints in it are, in which case it is *satisfiable*. If there is no satisfying assignment,  $F$  is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable. We also consider optimisation problems, where in addition to  $F$  we are given an integer linear (or affine) objective function  $f \doteq \sum_i w_i \ell_i + k$  and the task is to find an assignment that satisfies  $F$  and minimizes  $f$ . (To deal with maximization problems we can negate the objective function.) For pseudo-Boolean formulas  $F, F'$  and constraints  $C, C'$ , we say that  $F$  *implies* or *models*  $C$ , denoted  $F \models C$ , if any assignment satisfying  $F$  also satisfies  $C$ , and write  $F \models F'$  if  $F \models C'$  for all  $C' \in F'$ .

The *slack* of a normalized pseudo-Boolean inequality  $C$  as in (3.1) with respect to  $\rho$  measures how far  $\rho$  is from falsifying  $C$ , and is defined formally as

$$\text{slack}(\sum_i a_i \ell_i \geq A; \rho) = \sum_{\bar{\ell}_i \notin \rho} a_i - A. \quad (3.5)$$

The constraint  $C$  is *conflicting* under  $\rho$  if  $\text{slack}(C; \rho) < 0$ . If  $\rho$  does not assign  $\ell_j$  but  $0 \leq \text{slack}(C; \rho) < a_j$ , then  $C$  *propagates*  $\ell_j$  under  $\rho$ , meaning that if the literal  $\ell_j$  is falsified, then the constraint will become conflicting. See Figure 1 for some example calculations of slack. Note that a constraint can be conflicting even though not all variables in it have been assigned.

During *unit propagation* on  $F$  under  $\rho$ , the assignment  $\rho$  is extended iteratively by any propagated literals until an assignment  $\rho'$  is reached under which no constraint  $C \in F$  is propagating, or under which some constraint  $C$  is conflicting as described above. We refer to this latter scenario as a *conflict*, and say that  $\rho'$  *violates* the constraint  $C$  in this case. We say that  $F$  *implies*  $C$  by *reverse unit propagation (RUP)*, and that  $C$  is a *RUP constraint* with respect to  $F$ , if  $F \wedge \neg C$  unit propagates to conflict under the empty assignment. Similarly to RUP clauses in clausal proof logging systems, the concept of RUP constraints will be important in pseudo-Boolean proof logging.

$\rho$	$slack(C; \rho)$	Remark
$\{\}$	8	
$\{\bar{x}_5\}$	3	$C$ propagates $\bar{x}_4$ (coefficient $>$ slack)
$\{\bar{x}_5, \bar{x}_4\}$	3	Propagation does not change slack
$\{\bar{x}_5, \bar{x}_4, \bar{x}_3, x_2\}$	-2	Conflict (slack is negative)

**Figure 1:** Example slack calculations for the constraint  $C \doteq x_1 + 2\bar{x}_2 + 3x_3 + 4\bar{x}_4 + 5x_5 \geq 7$ .

Finally, we introduce a special constraint. If for an optimization problem with objective function  $\sum_i w_i \ell_i$  to be minimized the solver finds a solution  $\alpha$ , then we refer to

$$\sum_i w_i \ell_i \leq -1 + \sum_i w_i \cdot \alpha(\ell_i) \quad (3.6)$$

as an *objective-improving constraint* enforcing solutions yielding a strictly better value of the objective function during the rest of the search.

### 3.3 Syntax for Pseudo-Boolean Inequalities in Proofs

VERIPB expects pseudo-Boolean inequalities to be written in a format similar to the OPB format [RM16] except that constraints do not end with a semicolon when used in rules. Some examples are provided in Section 2.1. The proof checker also supports the usage of arbitrary variable names instead of only  $x_1, x_2, x_3$ , et cetera. Variable names in this extended format should:

- start with an underscore  $_$  or an ASCII letter in A-Z or a-z;
- continue with characters in A-Z, a-z, 0-9, or  $[\ ] \{ \} \_ \wedge$  (square and curly brackets, underscore, caret);
- contain at least two characters.

Variable names cannot contain spaces. Support for additional characters in variable names may be added but is implementation-specific. Unsupported characters will generate an error upon parsing and can never generate erroneous verification results.

### 3.4 General Principles for the Proof Checker

At any step in a proof the proof checker maintains a (multi-)set  $\mathcal{C}$  of *core constraints* and a (multi-)set  $\mathcal{D}$  of *derived constraints*, where all constraints are pseudo-Boolean inequalities. At the outset,  $\mathcal{C}$  contains the constraints in the input formula. All derived constraints are added to  $\mathcal{D}$ , but constraints can be moved from  $\mathcal{D}$  to  $\mathcal{C}$ . New constraints are derived from  $\mathcal{C} \cup \mathcal{D}$  using the *cutting planes* proof system as described more formally in [BN21] together with the *redundance-based strengthening* and *dominance-based strengthening* rules discussed in [BGMN23, GN21]. The syntactic form of the derivation rules and their exact semantic meaning is explained in Section 5.

A general design principle behind VERIPB is that the core set  $\mathcal{C}$  should be equivalent to the input formula when it comes to satisfiability (for a decision problem) or optimal value of the objective function (for an optimization problem), and therefore a constraint  $C$  should only be deleted from  $\mathcal{C}$  if it can be proven that  $C$  can be recovered from  $\mathcal{C} \setminus \{C\}$  (without making the objective function worse, in case there is an objective). A deletion from the core set that violates this is referred to as an *unchecked deletion*. SAT solvers that only use proofs to establish unsatisfiability can perform such unchecked deletions.

### 3.5 General Principles for the Proof Checker Syntax

Finally, we mention a number of general principles used in the VERIPB syntax:

- Every proof line ends with a semicolon (;).
- A single proof line can be split over multiple lines in the file.

- Optional items (e.g., a hint, witness, or subproof) are preceded by a colon (:).
- The percent sign (%) is used for starting comments. Comments run until the end of the line.
- Subproofs and proofgoals are closed off with `qed`.

## 4 Overall Proof Structure

Let us now describe what different types of VERIPB proofs there are, although for the SAT competition only proofs for decision problems are of interest. The most general version of the pseudo-Boolean proof format also allows to compose proofs, but this is not supported for the proof checker in the SAT competition and so we will not discuss it further here.

### 4.1 Proof Sections

A single, atomic VERIPB proof consists of three sections:

1. A **derivation** section, where derivations are performed on the input constraints using the cutting planes and strengthening rules, and where solutions can also be logged.
2. An **output** section, where the constraints currently in the core set  $\mathcal{C}$  of the constraint database can be specified (for verification/debugging purposes or as input to the next stage of the solving process).
3. A **conclusions** section, which specifies what if anything was established by the derivation in terms of satisfiability/unsatisfiability or optimality.

The syntactic format of the proof is

```
pseudo-Boolean proof version 3.0
[f <N> ;]
<derivation section>
output <output section> ;
conclusion <conclusion section> ;
end pseudo-Boolean proof;
```

(where the reason for the final line `end pseudo-Boolean proof;` is to make it very easy for experiment scripts to decide when parsing log files whether proof logging terminated successfully or not). Let us now discuss the different sections in more detail. The first part of the proof is

```
pseudo-Boolean proof version 3.0
[f <N> ;]
<derivation section>
```

where the parameter  $\langle N \rangle$  is the number of pseudo-Boolean inequalities in the input formula. The line with `f <N> ;` is optional and can be omitted. The *<derivation section>* consists of the different cutting planes and strengthening rules as well as solution logging rules explained in Section 5.

### 4.2 Output Section

The output section specifies the output of the proof (if any). The output is a listing of the (potentially multi-set of) core constraints at the end of the proof where every core constraint should be listed according to its multiplicity. The output is specified by stating the guarantee of the output *<output guarantee>*, how the output is given *<output type>* and the actual output content *<output content>*. In terms of syntax, the output section has the format:

```
output <output guarantee> <output type> <output content> ;
```

where  $\langle output\ guarantee \rangle$  determines if  $\langle output\ type \rangle$  needs to be specified and  $\langle output\ type \rangle$  determines if  $\langle output\ content \rangle$  needs to be specified. The output guarantee `NONE` states that the proof has no output:

```
output NONE;
```

such that no further things have to be specified (which is the only supported option for the SAT competition). The remaining output guarantees are:

1. `DERIVABLE` : The listed core set is derivable. This can be used to implement *finalization* [BCH21] by requiring that all constraints in the core set should be listed exactly once and that the derived set should be empty (which forces the solver to prove that it knows exactly what it has derived).
2. `EQUISATISFIABLE` : The listed core set is satisfiable if and only if the input problem at the start of this (atomic) proof is satisfiable. This option might only be relevant for decision problems, and it requires that the proof uses no unchecked deletion steps.<sup>1</sup>
3. `EQUIOPTIMAL` : The optimal solution for the listed core set with respect to the objective function has the same value as the optimal solution for the input problem at the start of the proof, except if the derivation section of the proof has happened to log an optimal solution, in which case the core set is guaranteed to be unsatisfiable. This option is relevant for optimization problems, and requires that the proof uses no unchecked deletion steps.

All output guarantees except `NONE` require to also specify an output type  $\langle output\ type \rangle$ , where it is possible to specify any of the following options:

1. `FILE` : The output is given as an OPB file [RM16] that is specified as the (optional) third positional argument when running VERIPB. Hence, to run VERIPB with an output section with output type `FILE` on a CNF file `input.cnf` using the proof file `input.pbp` and the OPB file `output.opb` as output, use:

```
veripb --cnf input.cnf input.pbp output.opb
```

The syntax in VERIPB is:

```
output  $\langle output\ guarantee \rangle$  FILE;
```

where  $\langle output\ guarantee \rangle$  is an output guarantee other than `NONE`. Example:

```
output EQUIOPTIMAL FILE;
```

2. `CONSTRAINTS` : The output is given as an OPB file explicitly in the proof. The syntax in VERIPB is:

```
output  $\langle output\ guarantee \rangle$  CONSTRAINTS opb
  * #variable=  $\langle number\ of\ variables \rangle$  #constraint=  $\langle number\ of\ constraints \rangle$ 
   $\langle objective\ (optional) \rangle$ 
   $\langle list\ of\ constraints \rangle$ 
end opb;
```

where  $\langle number\ of\ variables \rangle$  is the number of variables in the OPB file,  $\langle number\ of\ constraints \rangle$  is the number of constraints listed in the OPB file,  $\langle objective\ (optional) \rangle$  is the objective function to maximize or minimize (only for optimization problems) ending with a semicolon, and  $\langle list\ of\ constraints \rangle$  is a list of pseudo-Boolean constraints, each ending with a semicolon. Example:

<sup>1</sup>Note that standard SAT solvers should not necessarily be expected not to use unchecked deletions—they only use the proof to show unsatisfiability, whereas for satisfiable instance the proof log is ignored and instead the proposed solution output directly by the SAT solver is checked.

```
output DERIVABLE CONSTRAINTS opb
* #variable= 3 #constraint= 2
min: +3 x1 +2 ~x2 ;
+1 x1 +1 ~x2 >= 1;
+1 ~x1 +1 ~x3 >= 1;
end opb;
```

3. **IMPLICIT** : The output is (implicitly) the current constraints in the core set. The syntax is:

```
output <output guarantee> IMPLICIT;
```

Example:

```
output EQUISATISFIABLE IMPLICIT;
```

4. **PERMUTATION** : The output is a permutation of the current constraints in the core set. The syntax is:

```
output <output guarantee> PERMUTATION <list of constraint IDs> ;
```

where *<list of constraint IDs>* is a list of constraint IDs separated by whitespace for each constraint in the core. It is required that this list contains each non-deleted ID exactly once, i.e., is a permutation of the list of non-deleted IDs. Example (assuming there are constraints with IDs 1, 2 and 4):

```
output EQUIOPTIMAL PERMUTATION 2 4 1;
```

### 4.3 Conclusions Section

Proof logging is supported for the following types of problems and conclusions:

**Decision problems:** For decision problems without objective function, possible solver conclusions are:

1. **SAT** : The problem instance is satisfiable.
2. **UNSAT** : Problem instance is unsatisfiable/infeasible.
3. **NONE** : Result unknown.

**Optimization problems:** For problems with an objective function, possible solver conclusions are:

1. **BOUNDS** : The optimal solution to the problem instance lies in a specified interval  $[LB, UB]$ . If  $LB = UB$ , then optimality has been proven. If no solution has been found (or if one is only interested in lower bounds), then we can set  $UB = \infty$  (in which case **VERIPB** does not check anything regarding the upper bound). Similarly, we can set  $LB$  equal to the constant term in the (normalized) objective function if no nontrivial lower bound has been proven. If  $LB = \infty$ , then it was shown that the problem instance is infeasible.
2. **NONE** : Result unknown—in particular, no solution or lower bound has been found.

The conclusion section has the format

```
conclusion <conclusion section> ;
end pseudo-Boolean proof;
```

and after the keyword **conclusion** the proof should state what has been established. If there are no conclusions (as for, e.g., a pure reformulation of a problem), then

```
conclusion NONE;
```

signals this. In the syntax descriptions below, square brackets  $[ \dots ]$  denote optional parts of the syntax.

**Conclusion UNSAT** For an unsatisfiability proof for a decision problem, the conclusion section should be

```
conclusion UNSAT [ : <id> ] ;
```

where  $\langle id \rangle$  is an optional reference to a constraint ID with negative slack. If the optional constraint ID is provided, then VERIPB checks if the constraint is a contradiction (i.e., has negative slack), and throws an error if it is not. If no constraint ID is provided, then VERIPB checks whether any constraint in the database has negative slack, and throws an error otherwise. Note that the constraint ID  $\langle id \rangle$  is not needed for efficient verification—if indeed there is such a constraint, then  $0 \geq 1$  is a reverse unit propagation (RUP) constraint, which can easily be checked—but it can be helpful for debugging purposes or to be able to reconstruct the actual proof found by a solver. In the kernel format, stating the  $\langle id \rangle$  is mandatory.

**Conclusion SAT** If a solution to a decision problem is found, the conclusion section should be

```
conclusion SAT [ : <assignment> ] ;
```

where  $\langle assignment \rangle$  is a list of variables with the values they are mapped to. The assignment is provided as a list of literals (e.g.  $x1 \sim x2 \ x4$ ). If the optional assignment is provided, then VERIPB propagates the assignment with respect to the current database and checks if all original constraints are satisfied. If all original constraints are satisfied by the propagated assignment, then the check is accepted. If no assignment is provided, then the check is accepted if a solution has been logged while checked deletion was enabled. In all other cases an error is thrown.

**Conclusion BOUNDS** The conclusion type `conclusion BOUNDS` for optimization problems is not supported for the SAT competition, but is listed for completeness. Its syntactic format is

```
conclusion BOUNDS <LB> [ : <id> ] <UB> [ : <assignment> ] ;
```

for an optional constraint ID  $\langle id \rangle$  and an optional assignment  $\langle assignment \rangle$ . The lower bound  $\langle LB \rangle$  and the upper bound  $\langle UB \rangle$  can also have the value `INF`. If the lower bound is `INF`, then the optimization problem is infeasible/unsatisfiable. In this case, the same checks as for `conclusion UNSAT` are performed and the constraint ID  $\langle id \rangle$  serves as the hint for the constraint with negative slack. In addition, it is checked that no solutions were logged.<sup>2</sup> If the upper bound is `INF`, then no claims regarding the upper bound are made, e.g., this can be used if someone is only interested in a lower bound, or this has to be used if the problem is infeasible/unsatisfiable. No checks will be performed for this case. Hence, even if there is a solution logged, no error is thrown if the upper bound is `INF`.

If the lower bound or the upper bound is an integer (i.e., not `INF`), then the lower or upper bound is this value. If there is a check that does not succeed in the following description, then an error is thrown.

For the lower bound, VERIPB keeps track of the value  $UB_{\text{best}}$  of the best solution logged so far (i.e., the solution with the lowest objective value). A necessary condition for the lower bound to be accepted is that  $LB \leq UB_{\text{best}}$ .<sup>3</sup> If this condition holds, the following check is performed. The constraint ID  $\langle id \rangle$  should refer to a constraint from which  $Obj_{\text{final}} \geq LB$  follows by weak syntactic implication (i.e., by adding literal axioms), where  $Obj_{\text{final}}$  is the objective at the end of the proof, so after possible objective rewrite steps. Alternatively, the constraint ID  $\langle id \rangle$  can refer to any contradictory constraint to justify the lower bound. If no constraint ID is specified, then VERIPB loops over the full database at the end of the proof, and checks for each constraint whether  $Obj_{\text{final}} \geq LB$  follows by weak syntactic implication or whether the constraint is contradictory. In the kernel format, the hint  $\langle id \rangle$  is required.

The assignment  $\langle assignment \rangle$  should be an assignment corresponding to a solution with value  $UB$ . If the optional assignment is provided, then VERIPB propagates the assignment with respect to the current database and checks if the propagated assignment satisfies all original constraints and whether the original

<sup>2</sup>This check is required because otherwise the constraint with negative slack could have been derived from a solution improving constraint, in which case it is not sound to claim unsatisfiability.

<sup>3</sup>Similarly, this check is required because otherwise the proof of the lower bound could use a stronger solution improving constraint than the constraint that could be used when proving the lower bound by contradiction, which is again unsound.

objective evaluates to  $UB$  under the propagated assignment. If so, then the check is accepted. If no assignment is provided, then `VERIPB` checks whether there were solutions logged while checked deletion was activated. If the best objective value of all solutions that were logged while checked deletion was active is equal to the upper bound, then the upper bound is accepted.

## 5 Derivation Rules

In this section we describe the different rules or commands that can appear in a `VERIPB` proof. Every rule in a proof file is terminated by a semicolon (;). Whitespace separates tokens, but is otherwise ignored. Different rules can create different numbers of new constraints (or none).

The `VERIPB` proof checker accepts proofs in what we will refer to as an “augmented” proof format, including a rich collection of rules intended to make proof logging as convenient as possible. The verified proof checker `CAKEPB` only supports a “kernel” subset of these commands, where the intended workflow for formally verified proof checking is to use `VERIPB` as a preprocessor to compile augmented proofs into the kernel format to be formally checked by `CAKEPB`. In this section, we focus on describing the general augmented proof format, but we also mention the restrictions in the kernel format. A summary of the kernel format and the formally verified proof checker is presented in Section 6. We note that in the SAT competition 2026, the only supported option is to run both `VERIPB` and `CAKEPB` (even if the provided proof is already in kernel format).

The expected setting in a `VERIPB` proof is that there is an input formula  $F$  in (linear) pseudo-Boolean form (note that CNF formulas are a special case of this). For a *decision problem*, the proof should establish whether  $F$  has satisfying assignments or is unsatisfiable. For an *optimization problem*, the goal is to minimize a 0–1 integer linear objective function  $f$  subject to the constraints in  $F$ , or at least to prove as tightly matching upper and lower bounds on the objective function as possible. Decision problems can be viewed as a special case of optimization problems by considering the trivial objective function  $f \doteq 0$ .

**Proof checker state** At all times, `VERIPB` maintains a current state with the following information:

- The input formula  $F$  (which is immutable once loaded as described in Section 5.1).
- The objective function  $f$  (which would be the trivial function 0 for a decision problem).
- A constraint database consisting of a (multi-)set  $\mathcal{C}$  of *core constraints* and a (multi-)set  $\mathcal{D}$  of *derived constraints*, where all constraints are pseudo-Boolean inequalities. Each constraint is identified by a unique positive integer referred to as a *constraint ID*.
- A counter of the largest integer `maxId` used for any constraint ID in the proof so far.
- The best objective function value  $v$  achieved for any solution found so far (which is  $\infty$  if no solution has been found).
- Two (possibly empty) sets of pseudo-Boolean inequalities, the *specification*  $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$  and the *definition*  $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ , over ordered sets of variables  $\vec{u}$  and  $\vec{v}$  and a (possibly empty) set of auxiliary variables  $\vec{a}$ . Together, these encode a preorder, i.e., a reflexive and transitive relation.
- A sequence of variables  $\vec{z}$  (usually, but not necessarily, distinct) on which the preorder  $\mathcal{O}_{\preceq}$  is applied.
- The set of all variables that has appeared so far in the proof.
- The current proof level `currLevel`, offered as a convenience to help backtracking solvers keep track of which constraints should be erased when.

**IDs** When a constraint is expected by some proof command, as a general rule a positive integer  $N$  is interpreted as referring to the constraint in the database with constraint ID  $N$ . Also, a negative integer  $-N$  is interpreted in the augmented format as the constraint with identifier  $\text{maxId} + 1 - N$ , i.e., the  $N$ th most recent constraint derived, but this convention is not supported in the kernel format.

## 5.1 Manipulation of Constraints Database

**Input formula size** VERIPB always starts by loading the input formula. If the input formula contains exactly  $N$  pseudo-Boolean inequalities, then these will get constraint IDs  $1, 2, \dots, N$ . The command

```
f <N> ;
```

checks that the input formula contains exactly  $N$  pseudo-Boolean inequalities.

Note that any pseudo-Boolean equality  $\sum_i a_i l_i = A$  in the input formula will be counted as two constraints  $\sum_i a_i l_i \geq A$  and  $\sum_i a_i l_i \leq A$  in this order. As an example, if the OPB file

```
* #variable= 4 #constraint= 2
1 x1 2 x2 >= 1 ;
1 x3 1 x4 = 1 ;
```

is loaded in a pseudo-Boolean proof file then the formula stored in the proof checker core set is

```
1 x1 2 x2 >= 1 ; % Gets ID 1
1 x3 1 x4 >= 1 ; % Gets ID 2
1 ~x3 1 ~x4 >= 1 ; % Gets ID 3
```

For the purpose of the `f` command, equalities in the input count as two constraints.

**Move to core** Constraints can be moved from the derived set to the core set with the commands

```
core id <list of constraint IDs> ;
core range <idStart> <idEnd> ;
```

where the first command variant specifies a list of one or more constraint IDs separated by whitespace and the second variant specifies a range between  $\langle idStart \rangle$  (inclusive) and  $\langle idEnd \rangle$  (exclusive). For `core id`, all constraint IDs that are moved must be valid (i.e., at most  $\text{maxId}$ , and not deleted). For `core range`, deleted IDs are ignored. Moving a constraint to the core that is already in the core has no effect. In the kernel proof format only the `core id` command is supported.

## 5.2 Implicational Rules

Implicational rules derive new pseudo-Boolean inequalities  $C$  that are guaranteed to be semantically implied by the constraint database  $\mathcal{C} \cup \mathcal{D}$ . We write  $\mathcal{C} \cup \mathcal{D} \vdash C$  when a derivation of  $C$  from  $\mathcal{C} \cup \mathcal{D}$  using the implicational derivation rules below can be exhibited.

### 5.2.1 Implicational Rules in Kernel Format

**Reverse polish notation** The reverse polish notation rule

```
pol <sequence of operations in reverse polish notation>;
```

derives a new pseudo-Boolean inequality that receives constraint ID  $\text{maxId} + 1$  (after which  $\text{maxId}$  is incremented). The derivation is specified by a sequence of operations in the cutting planes proof system (as described in [BN21]). The sequence is given in reverse polish notation. That is, all operands in the sequence are pushed on a stack, and operations are performed on the top one or two elements (where  $\langle op1 \rangle$  is below the top element  $\langle op2 \rangle$  of the stack), after which the result is pushed on the stack:

- $\langle op1 \rangle \langle op2 \rangle +$  adds two constraints.

- $\langle op1 \rangle \langle op2 \rangle *$  multiplies the constraint  $\langle op1 \rangle$  by the positive integer  $\langle op2 \rangle$ .
- $\langle op1 \rangle \langle op2 \rangle c$  divides the constraint  $\langle op1 \rangle$  by the positive integer  $\langle op2 \rangle$  (rounding up all coefficients and the right-hand side to the next integer) in variable form, i.e., all literals are positive, which is equivalent to the Chvátal-Gomory cut.
- $\langle op1 \rangle \langle op2 \rangle d$  divides the constraint  $\langle op1 \rangle$  by the positive integer  $\langle op2 \rangle$  (rounding up all coefficients and the right-hand side to the next integer) in normalized form.
- $\langle op1 \rangle \langle op2 \rangle m$  applies the mixed integer rounding cut [MW01, DGN21] with the positive integer divisor  $\langle op2 \rangle$  to the constraint  $\langle op1 \rangle$  in variable form, i.e., all literals are positive.
- $\langle op1 \rangle \langle op2 \rangle n$  applies the mixed integer rounding cut [MW01, DGN21] with the positive integer divisor  $\langle op2 \rangle$  to the constraint  $\langle op1 \rangle$  in the normalized form.
- $\langle op \rangle s$  saturates the constraint  $\langle op \rangle$ .
- $\langle op1 \rangle \langle op2 \rangle w$  weakens the constraint  $\langle op1 \rangle$  by removing the variable  $\langle op2 \rangle$  (assuming that it contributes towards satisfying the constraint, so the right-hand side of  $\langle op1 \rangle$  will be decreased accordingly), where  $\langle op2 \rangle$  should be a variable name without negation.
- $\langle op1 \rangle \langle op2 \rangle -$  decreases the right-hand side of the constraint  $\langle op1 \rangle$  by the positive integer  $\langle op2 \rangle$ .

At the end of a reverse polish notation line the stack should contain a single constraint, which is the result of the `pol` rule application. It is an error if the stack is instead empty or contains more than one element.

As is the case in general for derivation rules, when a constraint is expected by some arithmetic operation a positive integer is interpreted as an absolute constraint ID and a negative integer  $-N$  as referring to the constraint with ID  $\text{maxid} + 1 - N$ . If instead a literal `var` or `~var` is encountered where a constraint is expected, then `var` is interpreted as the literal axiom constraint  $\text{var} \geq 0$  and `~var` is interpreted as the constraint  $\sim\text{var} \geq 0$  (or, equivalently,  $\text{var} \leq 1$ ). Note that since variables must consist of at least two characters, they cannot be confused with arithmetic operations.

**Reverse unit propagation (RUP)** The rule

<code>rup</code> $\langle \text{pseudo-Boolean inequality } C \text{ in OPB format} \rangle$ [ : $\langle \text{list of constraint IDs} \rangle$ ] ;
--

adds the specified pseudo-Boolean inequality  $C$  if it is implied by the constraint database  $\mathcal{C} \cup \mathcal{D}$  by reverse unit propagation. That is, the proof checker temporarily adds  $\neg C$  to the constraint database and performs unit propagation to check that this leads to conflict.

It is also possible to perform *annotated RUP* by specifying a list of constraint IDs such that only those constraint IDs are used for unit propagation. The tilde character `~` may (only in this rule) be used in  $\langle \text{list of constraint IDs} \rangle$  to represent the constraint  $\neg C$  in the RUP check. Annotated RUP has slightly different semantics in the augmented format and the kernel format. Specifically, we consider an annotation in the augmented format to be a *set annotation* which only specifies the set of constraints on which we propagate. The constraint  $\neg C$  (denoted by `~`) is automatically added to this set. On the other hand, an annotation in the kernel format is considered to be a *list annotation*, which specifies which constraints propagate and in which order.

In the augmented format, `VERIPB` performs propagation on the constraints corresponding to the constraint IDs in the hints until conflict, or it can be guaranteed that no further propagations happen from this set of constraints. It is allowed to specify a hint that does not propagate anything. As soon as a conflict is found, the propagation is terminated and the RUP check is accepted, except that it is still checked that all hints are valid constraint IDs.

In the kernel format, `CAKEPB` goes through the hints in the order they appear in the proof file once, and propagates all literals that the constraint corresponding to the hint propagates. The RUP check is accepted with the first constraint in this list of hints that is falsified with respect to the current assignment.

**Proof by contradiction (pbc)** The rule

```

pbc ⟨pseudo-Boolean inequality  $C$  in OPB format⟩ [ : subproof
    ⟨contradiction proof⟩
qed [pbc] [ : ⟨id⟩ ]];

```

derives a constraint  $C$  using a proof by contradiction. The contradiction itself can be derived in several steps, and these steps are organized inside a *subproof*. If a subproof is provided (which is required unless  $C$  is a tautology), then the constraint  $\neg C$  is introduced with constraint ID  $\text{maxid} + 1$ . Within the subproof, further implicational rules can be used that also introduce new IDs. These IDs are only valid within the subproof. The derived constraint  $C$  receives its ID at the `qed` line. Hence, its ID is equal to  $\text{maxid} + 1$ , where  $\text{maxid}$  is the largest ID used within the subproof. Finally, the  $\langle id \rangle$  at the end must (if specified) refer to a contradictory constraint in the subproof (normally, this will be the last constraint, and one can also use the relative ID  $-1$ ). If the ID is not specified, VERIPB will search for a contradictory constraint and throw an error if there is none. In the kernel format, the  $\langle id \rangle$  is mandatory.

As an example, consider the input constraint  $C_1 \doteq 4x_1 + 3x_2 + 2x_3 \geq 5$  which is loaded with ID 1, and the following proof of the constraint  $C \doteq 3x_1 + 3x_2 + 2x_3 \geq 5$ :

```

pbc +3 x1 +3 x2 + 2 x3 >= 5 : subproof
  pol 1 2 +; % gets ID 3
  rup >= 1; % gets ID 4
qed pbc : 4;

```

Then the constraint  $\neg C \doteq 3\bar{x}_1 + 3\bar{x}_2 + 2\bar{x}_3 \geq 4$  receives ID 2. Adding this to  $C_1 \doteq 4x_1 + 3x_2 + 2x_3 \geq 5$  yields  $x_1 \geq 1$ , which receives ID 3. Using this, we propagate to contradiction: now  $\neg C$  propagates  $x_2$  and  $x_3$  to false, which contradicts  $C_1$ . Hence, the `pbc` rule was applied successfully, the constraint  $C$  receives ID 5, and ID 2, 3 and 4 may no longer be used throughout the rest of the proof.

## 5.2.2 Syntactic Implication Rules in Augmented Format

Next, we turn to the syntactic implication rules. The rule

```

i ⟨pseudo-Boolean inequality  $C$  in OPB format⟩ [ : ⟨id⟩ ] ;

```

checks if there is a single constraint in the database that syntactically implies the specified pseudo-Boolean inequality  $C$  (as discussed in [BGMN23]). The optional argument  $\langle id \rangle$  specifies the constraint ID of the syntactically implying constraint and throws an error if it does not syntactically imply  $C$ . If  $\langle id \rangle$  is not specified, then every constraint in the database is checked and only if none syntactically implies  $C$ , then an error is thrown. In the kernel format, the optional argument  $\langle id \rangle$  is required.

Finally, the rule

```

ia ⟨pseudo-Boolean inequality  $C$  in OPB format⟩ [ : ⟨id⟩ ] ;

```

does the same as the `i`-rule and also adds the constraint  $C$  to the derived set.

## 5.3 Orders

Some rules make use of an order  $\mathcal{O}_{\preceq}$ , which can be defined as a set of pseudo-Boolean inequalities. Moreover, the order can make use of an optional *specification*  $\mathcal{S}_{\preceq}$ , which is described further below.

A set of pseudo-Boolean inequalities encoding a preorder is introduced by using the following template, in which square brackets denote the optional parts of the notation:

```

def_order ⟨order name⟩
  vars
    left ⟨ordered set of left variables⟩ ;
    right ⟨ordered set of right variables (of same size)⟩ ;
    [aux ⟨list of auxiliary variables⟩ ; ]
  end [vars] ;
  [spec

```

```

    <list of proof lines, whose corresponding constraints provide the specification  $\mathcal{S}_{\preceq}$  of the order>
end [spec];]
def
    <list of pseudo-Boolean inequalities  $\mathcal{O}_{\preceq}$  providing the definition of the order>
end [def];
[transitivity
  vars
    fresh_right <ordered set of extra variables (of same size)>;
    [fresh_aux_1 <list of extra auxiliary variables (of same size as aux)>;
    fresh_aux_2 <list of extra auxiliary variables (of same size as aux)>; ]
  end [vars];
  proof
    proofgoal #1
      <proof of transitivity of relation>;
    qed [#1] [: <id> ];
    ...
    qed [proof] [: <id> ];
  end [transitivity];]
[reflexivity
  proof
    proofgoal #1
      <proof of reflexivity of relation>;
    qed [#1] [: <id> ];
    ...
    qed [proof] [: <id> ];
  end [reflexivity];]
end [def_order];

```

Let us describe in more detail the different parts of the definition of a preorder  $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$  over sets of variables  $\vec{u}$ ,  $\vec{v}$ , and possibly a set of auxiliary variables  $\vec{a}$ .

**Name** The heading

```
def_order <order name>
```

introduces the preorder and gives it a unique name (with the same naming conventions as those for variables discussed in Section 3.3).

**Variables** The next section

```

vars
  left <ordered set of left variables>;
  right <ordered set of right variables (of same size)>;
  [aux <list of auxiliary variables>; ]
end [vars];

```

specifies the two sets of variables  $\vec{u}$  and  $\vec{v}$  used in  $\mathcal{O}_{\preceq}$ . The list of auxiliary variables  $\vec{a}$  is optional in the augmented format. Note that all auxiliary variables *must* be prefixed with  $\$$  (and continue with one or more characters in A-Z, a-z, 0-9, or [ ] { } \_ ^, as usual). In particular, recall that the  $\$$  prefix is *not allowed* for variables occurring elsewhere. This ensures that auxiliary variables are distinct from all the other variables occurring in the proof. We may refer to variables not prefixed by  $\$$  as *non-auxiliary* variables.

Auxiliary variables are only allowed in very particular places in a proof. Specifically, they are only allowed to be used in the derivation of the specification or in subproofs, where the corresponding specification is a valid premise. This means they are *exclusively* allowed in the following places:

- Anywhere within a `def_order` section, except as a `left`, `right` or `fresh_right` variable.

- In a *scope* within the subproof of the redundancy rule (see Section 5.4.1).
- In a scope within the subproof of the dominance rule (see Section 5.4.2).

Note that in particular, auxiliary variables can neither be used in the domain nor the image of any witness outside of the derivation of a specification. As a consequence, the constraint database can only contain constraints containing auxiliary variables within scopes that are used in rules that define or use an order, but never at the top-level.

**Specification** The next section

```
[spec
  ⟨list of proof lines, whose corresponding constraints provide the specification  $\mathcal{S}_{\preceq}$  of the order⟩
end [spec];]
```

provides the specification  $\mathcal{S}_{\preceq}$  of the order. The intuition is that  $\mathcal{S}_{\preceq}$  can define a circuit over the variables  $\vec{u}$ ,  $\vec{v}$ , and  $\vec{a}$ , which in turn helps us define our order. As mentioned previously, the specification is optional. In particular, when an order does not use auxiliary variables  $\vec{a}$ , there is also no need for a specification.

In the specification, we provide a proof that derives a set of pseudo-Boolean constraints from the empty set of premises. Specifically, each proof line corresponds to one application of a rudimentary *redundance rule*. Formally, the derivation of a specification  $\mathcal{S}_{\preceq}$  is a list

$$(C_1, \omega_1), (C_2, \omega_2), \dots, (C_n, \omega_n)$$

which satisfies the following:

1. For each  $i \in \{1, \dots, n\}$  we have that  $C_i$  can be added by the redundancy rule to  $\bigcup_{j=1}^{i-1} C_j$  with the witness  $\omega_i$ . In other words, it should hold that

$$\bigcup_{j=1}^{i-1} C_j \cup \neg C_i \models \bigcup_{j=1}^i C_j \upharpoonright_{\omega_i}.$$

2. For every witness  $\omega_i$  where  $i \in \{1, \dots, n\}$ , the support of  $\omega_i$  is a subset of  $\vec{a}$ .

The specification is then the set  $\mathcal{S}_{\preceq} := \{C_i \mid i \in \{1, \dots, n\}\}$  of all the derived constraints. For the syntax of the redundancy rule applications we refer to the definition of the general redundancy rule in Section 5.4.1 (and to the example below).

A crucial property of a specification is that we can recover an assignment of the auxiliary variables from the assignment of the non-auxiliary variables. Specifically, if  $\rho$  is *any* assignment over the non-auxiliary variables, we can extend it to an assignment  $\rho'$  over all variables that satisfies  $\mathcal{S}_{\preceq}$ .

**Definition** Next, the section

```
def
  ⟨list of pseudo-Boolean inequalities  $\mathcal{O}_{\preceq}$  defining order⟩
end [def];
```

presents the pseudo-Boolean formula claimed to encode a preorder defined by  $\vec{u} \preceq \vec{v}$  if and only if there exists an assignment  $\rho$  to the auxiliary variables  $\vec{a}$  such that  $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a} \upharpoonright_{\rho}) \wedge \mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a} \upharpoonright_{\rho})$ . (In the case that the specification is empty, this simplifies to:  $\vec{u} \preceq \vec{v}$  if and only if  $\mathcal{O}_{\preceq}(\vec{u}, \vec{v})$ ). Each constraint in the list of pseudo-Boolean inequalities  $\mathcal{O}_{\preceq}$  must end with a semicolon.

**Transitivity and reflexivity** The definition of the preorder is concluded by proofs showing that the defined order is transitive and reflexive (in this order). The transitivity proof has the following structure:

```

transitivity
  vars
    fresh_right <ordered set of extra variables  $\vec{w}$  (of same size as left)>;
    [fresh_aux_1 <list of extra auxiliary variables  $\vec{b}$  (of same size as aux)>;
    fresh_aux_2 <list of extra auxiliary variables  $\vec{c}$  (of same size as aux)>; ]
  end [vars];
  proof
    proofgoal #1
      <proof of transitivity of relation>;
    qed [#1] [: <id> ];
    ...
    qed [proof] [: <id> ];
  end [transitivity];
    
```

This transitivity proof establishes that  $\mathcal{O}_{\preceq}$  defines a transitive relation, i.e., that the implication

$$\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a}) \wedge \mathcal{S}_{\preceq}(\vec{v}, \vec{w}, \vec{b}) \wedge \mathcal{S}_{\preceq}(\vec{u}, \vec{w}, \vec{c}) \wedge \mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a}) \wedge \mathcal{O}_{\preceq}(\vec{v}, \vec{w}, \vec{b}) \models \mathcal{O}_{\preceq}(\vec{u}, \vec{w}, \vec{c})$$

provably holds. The transitivity proof has one proofgoal per constraint in the definition  $\mathcal{O}_{\preceq}$ . To make the premises available, the IDs 1 up to  $3|\mathcal{S}_{\preceq}| + 2|\mathcal{O}_{\preceq}|$  are used at the start of the `proof` section, providing the constraints in  $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ ,  $\mathcal{S}_{\preceq}(\vec{v}, \vec{w}, \vec{b})$ ,  $\mathcal{S}_{\preceq}(\vec{u}, \vec{w}, \vec{c})$ ,  $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ , and  $\mathcal{O}_{\preceq}(\vec{v}, \vec{w}, \vec{b})$ , respectively, in this order. Note that the `def_order` block uses its own constraint database for proving transitivity and reflexivity, starting again from ID 1.

In addition to transitivity, also reflexivity must be shown. In the augmented format, the reflexivity proof is optional if the proofgoals are trivial, i.e., if the negated proofgoals are already a contradiction. In the kernel format, the reflexivity proof is required. The syntax for the reflexivity proof is

```

reflexivity
  proof
    proofgoal #1
      <proof of reflexivity of relation>;
    qed [#1] [: <id> ];
    ...
    qed [proof] [: <id> ];
  end [reflexivity];
    
```

There is one proofgoal for each constraint in the definition of the order. The proofgoal is the constraint that results from the substitution of the variables in `right` with the variables in `left`.

**Examples** As an example, consider the lexicographic order over 3 bits defined by  $(u_1, u_2, u_3) \preceq (v_1, v_2, v_3)$  when  $\vec{u}$  viewed as a binary number is less than or equal to  $\vec{v}$ . We now show how to implement this with or without auxiliary variables. Note that the claim that  $\vec{u}$  is less than or equal to  $\vec{v}$  as a binary number can be expressed by the pseudo-Boolean inequality

$$-4u_1 + 4v_1 - 2u_2 + 2v_2 - u_3 + v_3 \geq 0 \tag{5.1}$$

and can be introduced as a preorder `ternarylex` (without auxiliary variables) as follows:

```

def_order ternarylex
  vars
    left  u1 u2 u3;
    right v1 v2 v3;
  end;
  def
    -4 u1 4 v1 -2 u2 2 v2 -1 u3 1 v3 >= 0 ;
    
```

```

end;
transitivity
  vars
    fresh_right w1 w2 w3;
  end;
  proof
    proofgoal #1;
      pol 1 2 + 3 +;
    qed : -1;
  qed;
end;
end;

```

With auxiliary variables, we introduce the auxiliary variable  $a_1$  in the specification to mean that  $u_1 \geq v_1$  and the auxiliary variable  $a_2$  to mean that  $a_1$  holds and that  $u_2 \geq v_2$ . We can then define the order  $(u_1, u_2, u_3) \preceq (v_1, v_2, v_3)$  as follows: (i) we need  $u_1 \leq v_1$ , (ii) if  $a_1$  holds (which together with  $u_1 \leq v_1$  implies that  $u_1 = v_1$ ), then we also need  $u_2 \leq v_2$ , and (iii) if  $a_2$  holds (which similarly implies that  $u_1 = v_1$  and  $u_2 = v_2$ ), then we also need  $u_3 \leq v_3$ .

The following block now defines this order. We omit some details in the transitivity proof.

```

def_order ternarylex_aux
  vars
    left u1 u2 u3;
    right v1 v2 v3;
    aux $a1 $a2;
  end;
  spec
    % expresses that $a1 ==> (u1 >= v1)
    red +1 ~$a1 +1 u1 +1 ~v1 >= 1 : $a1 -> 0;
    % expresses that $a1 <== (u1 >= v1)
    red +2 $a1 +1 ~u1 +1 v1 >= 2 : $a1 -> 1;
    % expresses that $a2 ==> ($a1 and u2 >= v2)
    red +2 ~$a2 +2 $a1 +1 u2 +1 ~v2 >= 2 : $a2 -> 0;
    % expresses that $a2 <== ($a1 and u2 >= v2)
    red +3 $a2 +2 ~$a1 +1 ~u2 +1 v2 >= 3 : $a2 -> 1;
  end;
  def
    +1 ~u1 +1 v1 >= 1;
    +1 ~$a1 +1 ~u2 +1 v2 >= 1;
    +1 ~$a2 +1 ~u3 +1 v3 >= 1;
  end;
  transitivity
    vars
      fresh_right w1 w2 w3;
      fresh_aux_1 $b1 $b2;
      fresh_aux_2 $c1 $c2;
    end;
    proof
      proofgoal #1
        pol -1 5 + 12 +;
      qed : -1;
      proofgoal #2
        % long proof omitted
      qed : -1;
      proofgoal #3
        % long proof omitted
      qed : -1;
    qed;
  end;
end;

```

```
end;
end;
```

**Loading an order** The `load_order` command loads the named order (which must previously have been successfully defined by a `def_order` command). The command

```
load_order ternarylex x1 x2 x3;
```

loads our example order `ternarylex` and specifies that it will be applied to the variables  $x_1, x_2, x_3$ . Calling `load_order` in this way in the middle of a proof also has the effect of moving all constraints currently in the derived set  $\mathcal{D}$  to the core set  $\mathcal{C}$ .

The `load_order` command can also be called with no arguments, in which case the currently loaded order is unloaded and is replaced by the empty preorder, which we denote by  $\mathcal{O}_\top$ . Derived constraints are not moved to the core for such an empty order command.

## 5.4 Strengthening Rules

Most proof logging steps for a solver trying to minimize  $f$  subject to the constraints in the pseudo-Boolean formula  $F$  (or trying to solve the decision problem  $F$ , in which case we recall that we can consider the objective function  $f \doteq 0$  to be trivial), are expected to be performed using the implicational rules in Section 5.2. However, we also need to allow *strengthening rules* deriving constraints  $C$  that are not semantically implied by the input formula. Adding such constraints  $C$  is in order as long as some optimal solution is maintained, i.e., a satisfying assignment to  $F$  that minimizes  $f$ . This idea was formalized in [BGMN23] by allowing the use of an additional pseudo-Boolean formula  $\mathcal{O}_\preceq(\vec{u}, \vec{v})$  that, together with a sequence of variables  $\vec{z}$ , defines a relation on the set of truth value assignments. As a generalization of this, but following the same ideas, we allow the use of an order with auxiliary variables  $\mathcal{O}_\preceq(\vec{u}, \vec{v}, \vec{a})$  together with a specification  $\mathcal{S}_\preceq(\vec{u}, \vec{v}, \vec{a})$ .

We let  $\mathcal{O}_\preceq$  and  $\mathcal{S}_\preceq$  define a relation  $\preceq$  as follows. For assignments  $\alpha, \beta$  to non-auxiliary variables, we let  $\alpha \preceq \beta$  hold, if and only if there exists an assignment  $\rho$  to the variables  $\vec{a}$ , such that

$$\mathcal{S}_\preceq(\vec{z}\upharpoonright_\alpha, \vec{z}\upharpoonright_\beta, \vec{a}\upharpoonright_\rho) \wedge \mathcal{O}_\preceq(\vec{z}\upharpoonright_\alpha, \vec{z}\upharpoonright_\beta, \vec{a}\upharpoonright_\rho)$$

evaluates to true. Then  $\preceq$  can be combined with the objective function  $f$  to define a preorder  $\preceq_f$  on assignments by

$$\alpha \preceq_f \beta \quad \text{if} \quad \alpha \preceq \beta \text{ and } f\upharpoonright_\alpha \leq f\upharpoonright_\beta, \quad (5.2)$$

and we require that all strengthening rules preserve a solution that is minimal with respect to  $\preceq_f$ .

### 5.4.1 Redundance-Based Strengthening

In this section, we introduce the redundance-based strengthening rule. We make a distinction as to whether the loaded preorder uses a (non-empty) specification or not.

**Without Specification** We first consider the case where the loaded preorder has *no specification* and uses *no auxiliary variables*. In this case, the redundance-based strengthening rule has the format

```
red [pseudo-Boolean inequality C in OPB format] [[: substitution ω] [[: subproof
  subproof showing all proofgoals]
qed [red] [[: id] ]]];
```

where we are again denoting optional parts of the rule with square brackets  $[ \dots ]$ . This rule makes it possible to derive a constraint  $C$  from  $\mathcal{C} \cup \mathcal{D}$  even if  $C$  is not implied, provided that the proof logger establishes that any assignment  $\alpha$  that satisfies  $\mathcal{C} \cup \mathcal{D}$  can be transformed into another assignment  $\alpha' \preceq_f \alpha$  that satisfies both  $\mathcal{C} \cup \mathcal{D}$  and  $C$ . (In case the order is empty, which we write as  $\mathcal{O}_\preceq = \mathcal{O}_\top$ , then the

condition  $\alpha' \preceq_f \alpha$  just means that the inequality  $f|_{\alpha'} \leq f|_{\alpha}$  should hold—note that this is vacuously true for a decision problem). We remark that this rule is a generalized version of the RAT rule in [HHW13]. The redundancy-based strengthening rule in the form we are using it here originated in [GN21], which in turn relies heavily on [HKB17, BT19].

More formally, if  $v$  is the best value for the objective function achieved by any solution so far (or  $\infty$  if no solution has been found), then  $C$  can be derived by redundancy-based strengthening, or just *redundance* for brevity, if there is a substitution  $\omega$  (referred to as the *witness*) such that an explicit derivation

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup C)|_{\omega} \cup \{f|_{\omega} \leq f\} \cup \mathcal{O}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a}) \quad (5.3)$$

can be provided. Intuitively, (5.3) says that if some assignment  $\alpha$  satisfies  $\mathcal{C} \cup \mathcal{D}$  but falsifies  $C$ , then the assignment  $\alpha' = \alpha \circ \omega$  still satisfies  $\mathcal{C} \cup \mathcal{D}$  and also satisfies  $C$ . In addition, the condition  $f|_{\omega} \leq f$  ensures that  $\alpha \circ \omega$  achieves an objective function value that is at least as good as that for  $\alpha$ .

**Witness** In a redundancy rule application, the witness  $\omega$  is presented as a space-separated list

```
var1 -> val1 var2 -> val2 var3 -> val3 ...
```

of variables in the domain of  $\omega$  and what they are mapped to (i.e., truth values or literals). The arrow symbols  $->$  are optional in the augmented format, but not in the kernel format.

**Subproof and proof goals** The  $\langle \text{subproof showing all proofgoals} \rangle$  part contains a derivation of every constraint on the right-hand side of (5.3). One possible way to do this is to derive contradiction directly within the subproof. To indicate this, the  $\langle id \rangle$  should refer to a contradictory constraint. Otherwise, each proof goal needs to be proven separately, except that some proof goals (e.g., RUP constraints and syntactically implied constraints) may be *autoprov*en by the proof checker. How much automatic proof generation the proof checker will provide depends on whether the augmented or the kernel proof format is used, with less generous support provided in the kernel format (see Sections 5.5.1 and 5.5.2 for more details). To show the proof goals, the subproof can also include implicational rules at top level, which can be used within the proofs for each proof goal.

We will discuss subproofs in more detail in Section 5.5, but note here that the constraints on the right-hand of (5.3) for which subproofs are needed, and which are referred to as *proof goals*, are referred by labels constructed as follows:

1. For  $(\mathcal{C} \cup \mathcal{D})|_{\omega}$ , each proof goal  $D|_{\omega}$  is labelled by the constraint ID of  $D$  in the database  $\mathcal{C} \cup \mathcal{D}$ .
2. The remaining constraints have special labels with a distinguishing prefix “#” as follows:
  - (a) Label #1 refers to the proof goal  $C|_{\omega}$  for the constraint  $C$  being derived by the redundancy rule application.
  - (b) Labels #2, #3, ..., # $N + 1$  refer to proof goals for the order  $\mathcal{O}_{\preceq}$  with  $N = |\mathcal{O}_{\preceq}|$ , i.e., there is one goal per constraint in the order (or  $N = 0$  if no order is loaded).
  - (c) Label # $N + 2$  refers to the proof goal for the constraint  $f|_{\omega} \leq f$  saying that the objective function must not get worse, if the problem contains an objective function  $f$ .

We note that proof goals are always proven by contradiction. In particular, providing an explicit proof of a proof goal automatically introduces the negation of that proof goal as a premise.

**With Specification** When an order with a specification  $\mathcal{S}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a})$  is loaded, then the redundancy-based strengthening rule requires an explicit derivation

$$\begin{aligned} &\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a}) \\ &\vdash (\mathcal{C} \cup \mathcal{D} \cup C)|_{\omega} \cup \{f|_{\omega} \leq f\} \cup \mathcal{O}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a}), \end{aligned} \quad (5.4)$$

i.e., the specification  $\mathcal{S}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a})$  is added as an extra premise compared to (5.3). Intuitively, the reason that it is sound to add the specification as an extra premise is that the specification can be derived from an empty set of premises. Moreover, since any variables witnessed over are fresh variables, this derivation is still valid in the presence of larger core and derived sets  $\mathcal{C} \cup \mathcal{D}$ .

When using a specification, the premises corresponding to the specification  $\mathcal{S}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a})$  are not introduced directly, but only upon entering the `scope leq` scope. The `scope leq` scope may only be introduced once in each subproof. We show the precise syntax below. Finally, we note that autoproofing is by default done without the specification constraints, but it is possible to specify a proof goal with an empty proof within the `scope leq` to indicate that that proof goal should be autoproofed with help of the constraints introduced in the scope.

We now illustrate the syntax of a possible subproof:

```

red ⟨pseudo-Boolean inequality C in OPB format⟩ : ⟨substitution ω⟩ : subproof
  % ¬C is introduced here
  pol -1 5 2 * +; % implicational rules are allowed here
  proofgoal #1    % proof goal C|_ω
    % ¬C|_ω is introduced here
    pol -1 -2 +; % add ¬C|_ω and the previous pol line
  qed #1 : -1;   % -1 indicates that previous line was a contradiction
  scope leq
    % introduce all constraints from S_≤(z|_ω, z, a) (each gets a new ID)
    % the scope is typically used for proofgoals from the order,
    % but for redundance, there are no restrictions
    proofgoal #2
      % introduces the negation of the first constraint in O_≤(z|_ω, z, a)
      rup x1 >= 1 : -1;
      pol -1 -3 +;
    qed;
    pol -4 5 +; % implicational rules are still allowed here
    proofgoal #3 qed; % autoprove proofgoal #3 using the scope
  end scope leq;
  % any remaining proof goals will be autoproofed here
qed [red];
    
```

#### 5.4.2 Dominance-Based Strengthening

We now discuss the *dominance-based strengthening* rule. We again make a distinction as to whether the loaded preorder uses a (non-empty) specification or not.

**Without Specification** We first consider the case where the loaded preorder has *no specification* and uses *no auxiliary variables*. In this case, the dominance-based strengthening rule has the format

```

dom ⟨pseudo-Boolean inequality C in OPB format⟩  [ : ⟨substitution ω⟩ [ : subproof
  ⟨subproof showing all proofgoals⟩
qed [dom] [ : ⟨id⟩ ] ] ] ;
    
```

(with optional parts within square brackets).

In order to explain how it works we first make a quick detour to discuss order relations. For any preorder  $\preceq$ , we can define a strict order  $\prec$  by postulating that  $\alpha \prec \beta$  if  $\alpha \preceq \beta$  and  $\beta \not\preceq \alpha$  (which means that, as a special case, the empty preorder yields a “strict order” that does not relate any elements at all). The relation  $\prec_f$  obtained in this way from the preorder (5.2) coincides with what is called a *dominance relation* in [CS15] in the context of constraint optimization, which is the reason for why the dominance-based strengthening rule has received its name.

Just as for the redundance rule, the dominance rule allows to derive a constraint  $C$  from  $\mathcal{C} \cup \mathcal{D}$  even if  $C$  is not implied. A crucial difference, however, is that in the dominance rule, an assignment  $\alpha$  satisfying

$\mathcal{C} \cup \mathcal{D}$  but falsifying  $C$  only needs to be mapped to an assignment  $\alpha'$  that satisfies  $\mathcal{C}$ , but not necessarily  $\mathcal{D}$  or  $C$ . On the other hand, the new assignment  $\alpha'$  should satisfy the strict inequality  $\alpha' \prec_f \alpha$  and not just  $\alpha' \preceq_f \alpha$  as in the redundance rule.

To see that this is sound, we can argue by induction that if these conditions are satisfied, then it must be possible to construct an assignment that satisfies  $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ , and achieves at least as good a value with respect to the objective function  $f$ , by iteratively applying the witness of the dominance rule. Note, however, that the derivation in the proof log does not actually carry out this construction, but just provides an existential proof that such a construction would be possible in principle. We sketch the proof of the soundness of this argument here, referring the reader to [BGMN23] for the missing details.

For the base case, if the assignment  $\alpha'$  obtained from  $\alpha$  satisfies  $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ , we are done. Otherwise, since  $\alpha'$  satisfies  $\mathcal{C}$ , and since  $\mathcal{D}$  has previously been derived from  $\mathcal{C}$ , it can be shown that there exists an assignment  $\alpha''$  that satisfies  $\mathcal{C} \cup \mathcal{D}$  and is such that  $\alpha'' \prec_f \alpha' \prec_f \alpha$  holds. If  $\alpha''$  does not satisfy  $C$ , then this assignment satisfies exactly the same conditions as the assignment  $\alpha$  that we started with, and the whole argument can be repeated to get  $\alpha^{(4)} \prec_f \alpha^{(3)} \prec_f \alpha''$ . Arguing by induction, we get a strictly decreasing sequence of assignments with respect to  $\prec_f$ . Since the set of possible assignments is finite, this sequence has to terminate eventually with an assignment  $\alpha^*$  that satisfies all constraints in  $\mathcal{C} \cup \mathcal{D} \cup \{C\}$  and for which the inequality  $f|_{\alpha^*} \leq f|_{\alpha}$  holds.

More formally, we would like to say that if  $v$  is the best value for the objective function achieved so far (or  $\infty$ ), and if the preorder  $\mathcal{O}_{\preceq}$  has been loaded to be applied to  $\vec{z}$ , then the pseudo-Boolean inequality  $C$  can be derived by dominance-based strengthening given a substitution  $\omega$  such that

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \vdash \mathcal{C}|_{\omega} \cup \mathcal{O}_{\preceq}(\vec{z}|_{\omega}, \vec{z}) \cup \neg \mathcal{O}_{\preceq}(\vec{z}, \vec{z}|_{\omega}) \cup \{f|_{\omega} \leq f\}, \quad (5.5)$$

where  $\mathcal{O}_{\preceq}(\vec{z}|_{\omega}, \vec{z})$  and  $\neg \mathcal{O}_{\preceq}(\vec{z}, \vec{z}|_{\omega})$  taken together imply that  $\alpha \circ \omega \prec \alpha$  for any assignment  $\alpha$ . A technical problem with this proposal is that the pseudo-Boolean formula  $\mathcal{O}_{\preceq}$  may contain multiple constraints, so that the negation of it is not a set of pseudo-Boolean inequalities and thus is not in the correct syntactic format. To get around this, one can divide (5.5) into two separate conditions and move  $\neg \mathcal{O}_{\preceq}(\vec{z}, \vec{z}|_{\omega})$  to the premise of the implication, which eliminates the negation. After this rewriting step, we get the formal definition that  $C$  is derivable by the dominance-based strengthening rule if there is a substitution  $\omega$  such that explicit derivations

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \vdash \mathcal{C}|_{\omega} \cup \mathcal{O}_{\preceq}(\vec{z}|_{\omega}, \vec{z}) \cup \{f|_{\omega} \leq f\} \quad (5.6a)$$

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}|_{\omega}) \vdash \perp \quad (5.6b)$$

can be provided.

As for the redundance rule,  $\omega$  should be given as a list  $var1 \rightarrow val1 \quad var2 \rightarrow val2 \dots$  of variables and what these variables are mapped to by  $\omega$ , with the arrow symbols  $\rightarrow$  being optional in the augmented format.

**Subproof and proof goals** As for the redundance rule, the *(subproof showing all proofgoals)* part contains a derivation for every constraint on the right-hand side of (5.6b)–(5.6b) with the same conventions as in Section 5.4.1. The proof goals for a dominance rule application are labelled as follows:

1. For  $\mathcal{C}|_{\omega}$ , each proof goal  $D|_{\omega}$  is labelled by the constraint ID of  $D$  in the core constraint set  $\mathcal{C}$ .
2. The remaining proof goals get labels with a distinguishing prefix “#” as follows:
  - (a) Labels #1, #2, ..., #N for  $N = |\mathcal{O}_{\preceq}|$  refer to proof goals for the order  $\mathcal{O}_{\preceq}$  with one proof goal per constraint in the order (or  $N = 0$  if no order is loaded).
  - (b) Label #N + 1 refers to the proof goal for the negated order in (5.6b). Note that this proof goal behaves slightly differently from the others in that it directly adds a list of assumptions from which contradiction must be derived.
  - (c) Label #N + 2 refers to the proof goal for the inequality  $f|_{\omega} \leq f$  if applicable, i.e., if the problem contains an objective function  $f$ .

**With Specification** Let us now consider the case where we are using a non-trivial specification  $\mathcal{S}_{\preceq}$  and set of auxiliary variables  $\vec{a}$ . Similarly to the redundancy rule, we now have the specification as an additional premise in our derivations. There is, however, a crucial technical detail. Looking at Equation (5.5), we can see that it contains two instantiations of our pre-order  $\mathcal{O}_{\preceq}$ . The first one for establishing that  $\vec{z}|_{\omega}$  is less than or equal to  $\vec{z}$ , that is, we have  $\mathcal{O}_{\preceq}(\vec{z}|_{\omega}, \vec{z})$ . The second one for establishing that the order is strict, that is we have  $\neg\mathcal{O}_{\preceq}(\vec{z}, \vec{z}|_{\omega})$ . Regarding the specification, it should be clear that these two instantiations can not be established over the same set of auxiliary variables when written in the form of Equation (5.5).

However, in Equation (5.6a) and Equation (5.6b) these two separate instantiations only occur in one of the two, respectively. Hence, the required derivations for the dominance rule become the following:

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a}) \vdash \mathcal{C}|_{\omega} \cup \mathcal{O}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a}) \cup \{f|_{\omega} \leq f\} \quad (5.7a)$$

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}, \vec{z}|_{\omega}, \vec{a}) \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}|_{\omega}, \vec{a}) \vdash \perp. \quad (5.7b)$$

In the proof, we establish access to these different premises of Equation (5.7a) and Equation (5.7b) through the use of the `scope leq` and `scope geq` scopes, respectively. The meaning is that the first set of premises  $\mathcal{S}_{\preceq}(\vec{z}|_{\omega}, \vec{z}, \vec{a})$  is only made available inside the `scope leq` scope. The second set of premises  $\mathcal{S}_{\preceq}(\vec{z}, \vec{z}|_{\omega}, \vec{a})$  is only available in the `scope geq` scope. In the `scope leq` scope, any proof goal except proof goal  $\#N + 1$  can be proven. Conversely, in the `scope geq` scope, only proof goal  $\#N + 1$  may be proven. Proof goals can also be shown outside of both scopes. Each scope may only be introduced once in each subproof, but the scopes can occur in either order. Otherwise, the list of proof goals and in particular the identifiers of the proof goals do not change when using auxiliary variables.

Finally, we give an example of the syntax (where we assume  $N = 3$ ):

```
dom (pseudo-Boolean inequality C in OPB format)  [: <substitution ω>  [: subproof
proofgoal #5          % proof goal f|_ω ≤ f
  % f|_ω > f is introduced here
  pol -1 3 2 * +;
qed #5 : -1;          % -1 indicates that previous line was a contradiction
scope leq
  % introduce all constraints from S_≤(z|_ω, z, a) (each gets a new ID)
  % only proofgoal #1, #2, #3 and #5 would be allowed here
proofgoal #1          % first constraint in O_≤(z|_ω, z, a)
  % introduces the negation of the first constraint in O_≤(z|_ω, z, a)
  pol -1 -2 +;
qed #1 : -1;
end scope leq;
scope geq
  % introduce all constraints from S_≤(z, z|_ω, a) (each gets a new ID)
  % only proofgoal #4 is allowed here
proofgoal #4
  % introduce all N constraints from O_≤(z, z|_ω, a)
  pol -4 -3 +; % add last constraint specification
                % and first constraint in O_≤(z, z|_ω, a)
qed #4 : -1;
end scope geq;
% any remaining proof goals will be autoprovden here
qed [dom] [: <id> ]];
```

## 5.5 Subproofs

The rules for subproofs are arguably somewhat complex. A good way to understand the syntax is to run VERIPB on files such as `tests/integration_tests/correct/dominance/example.pbp` in the repository with the options `--trace`, which will (among other things) display the required proof goals.

If all proof goals for a strengthening rule application can be automatically derived by the proof checker, then there is no need for a list of subproofs and the strengthening rule can be stated as

```
⟨strengthening rule⟩ ⟨derived constraint C⟩ [ : ⟨substitution ω⟩ ] ;
```

on a single line, where  $\langle \text{strengthening rule} \rangle$  is `red` or `dom`. In the general case, however, the proof checker will need to be provided with a list of explicit subproofs, and then the format of the strengthening rules is

```
⟨strengthening rule⟩ ⟨derived constraint C⟩ [ : ⟨substitution ω⟩ [ : subproof
  ⟨list of proofgoals or top-level contradiction proof⟩
qed [⟨strengthening rule⟩] [ : ⟨id⟩ ] ] ] ;
```

The optional constraint ID at the end of the `red` or `dom` rule with subproofs, if such a final constraint ID is provided, refers to a contradiction that is derived in the top-level subproof of these rules. As soon as such a constraint has been derived, there is no need to check any remaining proof goals.

In  $\langle \text{list of proofgoals or top-level contradiction proof} \rangle$  implicational derivation steps can be interleaved with proof goals, where the latter of which are formatted as follows:

```
proofgoal ⟨pid⟩
  [⟨list of implicational steps⟩]
qed [⟨pid⟩] [ : ⟨id⟩ ] ;
```

Each proof goal is labeled with a proof goal ID  $\langle pid \rangle$  which is on the form  $\langle id \rangle$  or  $\#\langle id \rangle$  as explained at the end of Sections 5.4.1 and 5.4.2 for redundancy and dominance, respectively.

Proof checking proceeds through subproofs sequentially, populating the database according to the implicational commands, and checking proof goals with the accumulated database up to that proof goal. This is illustrated below with a commented trace, starting from a constraint database  $\mathcal{S}$  which is the union of the core set and the derived set at this point in the overall proof.

```
% Initial database S
⟨list of implicational steps⟩
% Derived database S' from S following implicational steps
proofgoal ⟨pid⟩
  % Add constraint(s) from proof goal for ⟨pid⟩ to S'
  ⟨list of implicational steps⟩
  % Derived database S'' from S' implicationally
qed ⟨pid⟩ [ : ⟨id⟩ ] ;
% Check if ⟨id⟩ is a contradiction
% Rewind to database S' and continue
...

```

If subproof checking succeeds, the argument at the end of the explicit subproofs specifies a constraint ID at which  $\langle \text{list of proofgoals or top-level contradiction proof} \rangle$  derives contradiction. If such a contradiction is derived, the proof of the redundancy or dominance step is complete. Otherwise, all proof goals for the given redundancy or dominance step are checked to either be explicitly covered by a proof goal in the subproofs or implicitly covered by automatic proof.

**IDs in subproofs** The rules for incrementing `maxId` in subproof checking are as follows.

- All implicational steps increment the global `maxId` count whenever they add a new constraint to the database. This includes both top-level implicational steps as well as steps within each proof goal.
- The `maxId` counter is *not* reset to its original value after finishing a subproof even after the database is rewound, e.g., when rewinding to  $\mathcal{S}'$  in the illustration above.
- For each proof goal with an explicit subproof, there is a constraint  $D$  to be derived from the current constraint database (as described in Section 5.4.1 and Section 5.4.2). The negated constraint  $\neg D$  is added to the constraint database with the current ID `maxId` and `maxId` is incremented.

- The sole exception is the proof goal labeled  $\#N + 1$  for dominance-based strengthening, which adds all constraints from the substituted order  $\mathcal{O}_{\succeq}(\vec{z}, \vec{z}|_{\omega}, \vec{a})$  with IDs  $\text{maxId}$  up to  $\text{maxId} + |\mathcal{O}_{\succeq}| - 1$  to the database. The constraints are ordered as in their specification;  $\text{maxId}$  is incremented accordingly.
- When entering `scope leq`, all constraints from the specification  $\mathcal{S}_{\succeq}(\vec{z}|_{\omega}, \vec{z}, \vec{a})$  are added with IDs  $\text{maxId}$  up to  $\text{maxId} + |\mathcal{S}_{\succeq}| - 1$  to the database (in the same order as they were derived in the specification in the `def_order` block). Similarly, when entering `scope geq`, all constraints from the specification  $\mathcal{S}_{\preceq}(\vec{z}, \vec{z}|_{\omega}, \vec{a})$  are added with IDs  $\text{maxId}$  up to  $\text{maxId} + |\mathcal{S}_{\preceq}| - 1$ .
- All of the above applies to proof goals with explicit subproofs, i.e., no incrementing of  $\text{maxId}$  happens for automatically generated subproofs.

**Scopes** At all times, the checker keeps track of the constraints in the current *scope*. Here, a scope should be interpreted in the broad sense: besides `scope leq` and `scope geq` (which introduce constraints of a specification as explained in Section 5.4), also `subproof` and `proofgoal` blocks define scopes. A general design principle in the proof system is that each ID is valid for a contiguous block in the proof, starting when the constraint is introduced (and hence given its ID) and ending when the scope in which the ID was introduced (or when the ID is deleted, which is only allowed at top level; see Section 5.6).

### 5.5.1 Autoproven Proof Goals in Kernel Format

The kernel format uses the following rules to determine if a proof goal requires explicit proofs (assuming contradiction has not been derived). Let  $C$  be the constraint that is derived using the redundancy or the dominance step and let  $\omega$  be the substitution of the redundancy or the dominance step.

1. For  $\#\langle id \rangle$  proof goals, only limited autoprovng is supported. Let  $D$  be the constraint corresponding to  $\#\langle id \rangle$ . The proof goal  $\#\langle id \rangle$  can be skipped if:
  - (a)  $D$  is a tautology;
  - (b)  $D$  is weakly syntactically implied by  $\neg C$ ;
  - (c) The order proofgoals  $\#2, \#3, \dots, \#N + 1$  of the redundancy rule only can be skipped if the variables over which the order is loaded are untouched by the substitution  $\omega$  (since the proof goal is then implied by reflexivity of the order).
2. For  $\langle id \rangle$  proof goals, let  $D$  be the constraint corresponding to  $\langle id \rangle$  in the database. The proof goal  $\langle id \rangle$  can be skipped if:
  - (a)  $D$  is untouched by the substitution  $\omega$ ;
  - (b)  $\omega$  only assigns literals to true in (the normalized form of)  $D$ ;
  - (c)  $D|_{\omega}$  is weakly syntactically implied by  $D$ ;
  - (d)  $D|_{\omega}$  is weakly syntactically implied by  $\neg C$ ;
  - (e)  $D|_{\omega}$  is contained in the database (at the start of the redundancy or the dominance step);
  - (f)  $D|_{\omega}$  is identical to a proof goal that has already been proven.

Note that the case that  $D|_{\omega}$  is a tautology is a special case of item 2c (or 2d), since any (non-tautological) constraint weakly syntactically implies a tautology.

### 5.5.2 Autoproven Proof Goals in Augmented Format

In addition to what is mentioned in Section 5.5.1, in the augmented format a constraint  $D$  can also be autoprovng if  $D$  can be derived using reverse unit propagation or using syntactic implication from any constraint currently in the database.

## 5.6 Deletion Rules

Deletion is a complex topic, not least because the pseudo-Boolean proof format supports both *deleting by reference* (i.e., by specifying a constraint ID) and *deleting by specification* (i.e., describing the constraint to be deleted). The latter type of deletion is not a good fit for the proof format, but is supported for ease of integration with SAT solvers using standard DRAT-style proof logging.

As a further complication, a pseudo-Boolean constraint  $C$  in the core set  $\mathcal{C}$  should ideally be deleted only if it can be proven that  $C$  can be recovered from  $\mathcal{C} \setminus \{C\}$ , which is referred to as *checked deletion* in [BGMN23].

### 5.6.1 Deletion Rules in Kernel Format

The kernel format only supports deleting constraints by referring to their constraint IDs, and in addition forces the proof logger to specify whether a core set constraint or a derived set constraint is being erased.

**Deletion from derived set** The command

```
del d <list of constraint IDs> ;
```

deletes a list of constraints with specified IDs from the derived set  $\mathcal{D}$ . It is an error if any constraint is instead in the core set  $\mathcal{C}$  or is not in the constraint database at all.

**Deletion from core set** The unchecked core deletion command

```
del c <list of constraint IDs> ;
```

provided for the SAT competition 2026 first checks that all constraint IDs in the specified list identify constraints currently in the core set. Provided that no order (or the empty order) has been loaded, or alternatively that the derived set  $\mathcal{D}$  is currently empty, all the specified constraints are deleted from the core. Otherwise the command generates an error.

Note that the fact that core deletion is unchecked means that the current database  $\mathcal{C} \cup \mathcal{D}$  can turn satisfiable although the input formula is unsatisfiable.

### 5.6.2 Deletion Rules in Augmented Format

The augmented proof format supports a general deletion-by-reference command, where the proof logger is required to know the constraint ID but does not need to be aware of the difference between core set and derived set. It also provides a deletion-by-specification command that provides the encoding of the constraint to be deleted. Deletion by specification should be avoided if possible, but is supported as a convenience for SAT solvers already equipped with DRAT-style proof logging as well as for solvers in more powerful paradigms where the constraints in the proof constraints database might not match well what the solver is keeping track of in its own constraints database.

**Deletion by reference** The command

```
del id <list of constraint IDs> ;
```

has the same effect as checking the type of each constraint  $\langle id \rangle$  in the list and then issuing a delete core command `del c` or delete derived command `del d` depending on the type of the constraint. All constraint IDs must be valid references to constraints currently in the database.

**Deletion by specification** The command

```
del spec <pseudo-Boolean inequality C in OPB format> ;
```

is an error if there is no constraint  $C$  in the database. Otherwise,  $C$  is marked for deletion as per the description in Section 5.6.3.

Note that if `del spec` has been used in the proof, then it is not allowed to specify an output (see Section 4.2). In particular, it is only allowed to specify `output NONE; .`

### 5.6.3 Semantics for Mixed Deletion by Reference and Specification

We implement a multiset deletion semantics for deletion by associating to constraint  $C$  in the database a delete-by-specification kill counter  $K_s(C)$  and two lists  $L_{\mathcal{D}}(C)$  and  $L_{\mathcal{C}}(C)$  containing the constraint IDs with which  $C$  appears in the derived set  $\mathcal{D}$  and core set  $\mathcal{C}$ , respectively.

When deletion by reference for a constraint  $C$  is encountered, the specified constraint ID is removed from the appropriate list  $L_{\mathcal{D}}(C)$  or  $L_{\mathcal{C}}(C)$ . When deletion by specification of  $C$  is encountered,  $K_s(C)$  is incremented by 1. After any of the above updates to  $K_s(C)$ ,  $L_{\mathcal{D}}(C)$ , or  $L_{\mathcal{C}}(C)$ , the following procedure is run:

1. If  $K_s(C) < |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$ , then there are still derived copies of  $C$  left, and no action is taken.
2. Else if  $K_s(C) = |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$ , then as many copies of  $C$  as are available in the database have been deleted. Therefore the constraint  $C$  is completely removed from the database by issuing `deld` commands for all IDs in  $L_{\mathcal{D}}(C)$  and `delc` commands for all IDs in  $L_{\mathcal{C}}(C)$ , after which we set  $L_{\mathcal{D}}(C) = L_{\mathcal{C}}(C) = \emptyset$  and  $K_s(C) = 0$ .
3. The case  $K_s(C) > |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$  is impossible, as the proof checker preserves the invariant  $K_s(C) \leq |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$  (as a result of the semantics in the previous case).

## 6 Formally Verified Proof Checking

The kernel proof checker CAKEPB has been formally verified in the HOL4 theorem prover [SN08] using the CAKEML suite of tools for program verification, extraction, and compilation [TMK<sup>+</sup>19, GMKN17, MO14]. In this section, we present the verification guarantees for CAKEPB\_CNF, a version of CAKEPB equipped with a DIMACS CNF parser frontend for UNSAT proof checking with pseudo-Boolean proof logging.

### 6.1 Summary of Kernel Format

We summarize the kernel format with reference to the derivation rules presented in Section 5. In general, rules in the kernel format have the same semantics on the proof state, but may have additional syntactic restrictions or requirements, e.g., requiring more explicit subproofs.

**Constraint IDs** The kernel format does not support the interpretation of a negative integer  $-N$  as the constraint with ID `maxid + 1 - N`.

**Load formula** The `f` command behaves identically in the kernel format. It must be the first command in the input pseudo-Boolean proof after the header (see Section 2.1 for an example).

**Move to core** Only the `core id` command for moving constraints to the core is supported.

**Implicational rules** Only the `pol`, `rup` and `pb` implicational rules are supported. For `rup`, it is mandatory to annotate which constraints propagate and in which order.

**Orders** The kernel format requires an `aux` line. In addition, there must be explicit proofs for all proof goals in the transitivity proof for the specified order.

**Strengthening rules and subproofs** The kernel format supports both `red` and `dom` strengthening commands, but requires more explicit subproofs. See Section 5.5.1 for the exact explicit subproof requirements in the kernel format.

**Deletion rules** Only the `deld` and `delc` deletion by ID commands are supported in kernel format. Notably, deletion by specification is not supported.

**Conclusions section** In the kernel format, the conclusion section *must* be explicitly given all required IDs (cf. Section 4.3). In particular, for an unsatisfiability proof, the conclusion section

```
conclusion UNSAT : ⟨id⟩ ;
```

must have a constraint ID specifying the contradictory constraint in the database.

$$\begin{array}{l}
 \vdash \text{cake\_pb\_cnf\_run } cl \ fs \ mc \ ms \Rightarrow \quad (6.1) \\
 \left. \begin{array}{l}
 \text{machine\_sem } mc \ (\text{basis\_ffi } cl \ fs) \ ms \subseteq \\
 \text{extend\_with\_resource\_limit} \\
 \{ \text{Terminate Success } (\text{cake\_pb\_cnf\_io\_events } cl \ fs) \} \wedge
 \end{array} \right\} (6.2) \\
 \exists \text{ out err.} \\
 \left. \begin{array}{l}
 \text{extract\_fs } fs \ (\text{cake\_pb\_cnf\_io\_events } cl \ fs) = \\
 \text{SOME } (\text{add\_stdout } (\text{add\_stderr } fs \ err) \ out) \wedge
 \end{array} \right\} (6.3) \\
 \left. \begin{array}{l}
 \text{if } out = \ll s \text{ VERIFIED UNSAT } \backslash n \gg \text{ then} \\
 \text{LENGTH } cl = 3 \wedge \text{inFS\_fname } fs \ (\text{EL } 1 \ cl) \wedge \\
 \exists \text{ fml.} \\
 \text{parse\_dimacs } (\text{all\_lines } fs \ (\text{EL } 1 \ cl)) = \text{SOME } \text{fml} \wedge \\
 \text{unsatisfiable } (\text{interp } \text{fml}) \\
 \text{else } out = \ll \gg
 \end{array} \right\} (6.4)
 \end{array}$$

**Figure 2:** The end-to-end correctness theorem for the CAKEML pseudo-Boolean proof checker with a CNF parser

## 6.2 Verified Correctness Theorem for CAKEPB\_CNF

The end-to-end verified correctness theorem for CAKEPB\_CNF is shown in Figure 2. This theorem can be intuitively understood in four parts, corresponding to the indicated lines (6.1)–(6.4):

- The theorem assumes (6.1) that the CAKEML-compiled machine code for CAKEPB\_CNF is executed in an x64 machine environment set up correctly for CAKEML. The definition of `cake_pb_cnf_run` is shown below, where the first line (`wfcl cl ∧ wfFS fs ∧ ...`) says the command line `cl` and filesystem `fs` match the assumptions of CAKEML’s FFI model. The second line says that the compiled code (`cake_pb_cnf_code`) is correctly set up for execution on an x64 machine.

$$\begin{array}{l}
 \text{cake\_pb\_cnf\_run } cl \ fs \ mc \ ms \stackrel{\text{def}}{=} \\
 \text{wfcl } cl \wedge \text{wfFS } fs \wedge \text{STD\_streams } fs \wedge \text{hasFreeFD } fs \wedge \\
 \text{installed\_x64 } \text{cake\_pb\_cnf\_code } mc \ ms
 \end{array}$$

- Under these assumptions, the CAKEPB\_CNF program is guaranteed to never crash (6.2). However, it may run out of resources such as heap or stack memory (`extend_with_resource_limit ...`). In these cases, CAKEPB\_CNF will fail gracefully and report out-of-heap or out-of-stack on standard error.
- Upon termination, the CAKEPB\_CNF program will output some (possibly empty) strings `out` and `err` to the standard output and standard error streams, respectively (6.3).
- The key verification guarantee (6.4) is that, whenever the string “s VERIFIED UNSAT” is printed to standard output, the input CNF file (first command line argument) parses in DIMACS format to a CNF which is unsatisfiable. No other output is possible on standard output; error strings are always printed to standard error.

Internally, CAKEPB\_CNF transforms input CNF clauses (in DIMACS format) to normalized pseudo-Boolean constraints, as exemplified by (2.1a) and (2.1b). This transformation is formally verified to preserve satisfiability as part of the end-to-end correctness theorem shown in Figure 2.

Note that the CAKEPB\_CNF tool has an essentially identical correctness theorem to an existing verified Boolean unsatisfiability proof checking tool [THM21]. In fact, these tools share exactly the same definitions of DIMACS CNF parsing, Boolean satisfiability semantics, and all of the CAKEML’s standard assumptions.

### 6.3 Complexity

All of the commands in the kernel format are designed to minimize the need to search over the entire constraint database. For example, each implicational and deletion proof step can be performed in linear time with respect to the size of that step.

The only proof steps that scale linearly with respect to the size of the constraint database are redundancy and dominance-based strengthening steps. For either of these steps, the proof checker potentially needs to loop over the entire constraint database to check all the necessary proof goals. However, the maximum size of the database is linear in the size of the input formula and the proof. Therefore, the overall complexity of the verified proof checker is polynomial in the size of the input formula and proof, as required.

### Acknowledgements

We wish to acknowledge the monumental contributions of Stephan Gocht [Goc22], without whom there would not have been any pseudo-Boolean proof checker.

### References

- [AR20] Johannes Altmanninger and Adrián Rebola-Pardo. Frying the egg, roasting the chicken: Unit deletions in DRAT proofs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*, pages 61—70, January 2020.
- [BBN<sup>+</sup>23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.
- [BBN<sup>+</sup>24] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Certifying without loss of generality reasoning in solution-improving maximum satisfiability. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, September 2024.
- [BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.
- [BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.
- [BMM<sup>+</sup>23] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at <https://satcompetition.github.io/2023/checkers.html>, March 2023.
- [BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 233–350. IOS Press, 2nd edition, February 2021.

- [BT19] Samuel R. Buss and Neil Thapen. DRAT proofs, propagation redundancy, and extended resolution. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 71–89. Springer, July 2019.
- [CS15] Geoffrey Chu and Peter J. Stuckey. Dominance breaking constraints. *Constraints*, 20(2):155–182, April 2015. Preliminary version in *CP '12*.
- [DGN21] Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints*, 26(1–4):26–55, October 2021. Preliminary version in *CPAIOR '20*.
- [DMM<sup>+</sup>24] Emir Demirović, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean reasoning about states and transitions to certify dynamic programming and decision diagram algorithms. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, September 2024.
- [EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.
- [GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In Hongseok Yang, editor, *ESOP*, volume 10201 of *LNCS*, pages 584–610. Springer, 2017.
- [GMM<sup>+</sup>20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.
- [GMM<sup>+</sup>24] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.
- [GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.
- [GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.
- [GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.
- [GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.
- [Goc22] Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, June 2022.

## References

- [HHW13] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.
- [HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, August 2017.
- [HOGN24] Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. Certifying MIP-based presolve reductions for 0–1 integer linear programs. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.
- [IOT<sup>+</sup>24] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Certified MaxSAT preprocessing. In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.
- [JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.
- [MM23] Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.
- [MMN24] Matthew McIlree, Ciaran McCreesh, and Jakob Nordström. Proof logging for the circuit constraint. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14743 of *Lecture Notes in Computer Science*, pages 38–55. Springer, May 2024.
- [MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.
- [MW01] Hugues Marchand and Laurence A. Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations Research*, 49(3):325–468, June 2001.
- [RM16] Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at <http://www.cril.univ-artois.fr/PB16/format.pdf>, January 2016.
- [SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.
- [THM21] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake\_lpr: Verified propagation redundancy checking in CakeML. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *TACAS*, volume 12652 of *LNCS*, pages 223–241. Springer, 2021.
- [TMK<sup>+</sup>19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.

- [VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.
- [Ver] VeriPB: Verifier for pseudo-Boolean proofs. <https://gitlab.com/MIAOresearch/software/VeriPB>.
- [WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.