Boosting SMT Quantifier Instantiation with Deep Unit Processing

Pascal Fontaine[1,2] and Hans-Jörg Schurr[1]

[1] University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
hans-jorg.schurr@inria.fr
[2] Université de Liège, Belgium
pascal.fontaine@uliege.be

Abstract. SMT solvers excel at reasoning on ground formulas and rely on instantiation for quantifier reasoning. Heuristics are used to generate ground consequences of the quantified formulas until the solver is able to derive a contradiction at the ground level. Current instantiation heuristics often fail to quickly generate the right instances, resulting in a failure to solve the problem. We here tackle this issue by using asserted quantified formulas to eliminate nested quantified formulas. We implemented the method in the veriT solver in the preprocessing phase and show it is effective on benchmarks from the SMT-LIB, enabling the solver to prove more formulas, faster.

Keywords: satisfiability modulo theories · quantifier instantiation · theorem proving

1 Introduction

Satisfiability modulo theories (SMT) solvers are successfully used as back-ends for formal method applications, within interactive proof assistants or verification platforms. While SMT solvers based on the CDCL($\mathcal{T}$) calculus [5] excel at handling quantifier-free problems with theories, they are also often used on problems with quantifiers. The main approach for those solvers to handle quantifiers is *quantifier instantiation*. This approach separates the quantified assertions from the ground part of the problem. Whenever the solver finds a model for the ground part of the problem, it generates new ground instances of the quantified formulas. This is repeated until the ground solver determines that the ground problem is unsatisfiable. When done fairly, this approach is refutationally complete for many theories. The main challenge is to find the right instances. SMT solvers use multiple instantiation strategies to find these instances (Section 2).

If the problem contains a quantified lemma also occurring, e.g., as guard for another formula, the common instantiation methods often fail to quickly produce the correct instances. This structure is quite typical of problems generated within interactive theorem provers. The following running example illustrates this issue:

Example 1.

$$\forall x.\, P(x) \to P(f(x, c)) \tag{1}$$
$$\forall y.\, (\forall z.\, P(z) \to P(f(z, y))) \to \neg P(y) \tag{2}$$
$$P(c) \tag{3}$$

This problem is trivially unsatisfiable: when y is set to c, assertion 1 occurs as the guard of the implication in assertion 2, so $\neg P(c)$ is a direct consequence of the first two assertions, in contradiction with the third. As described in Section 2.3, all major instantiation techniques fail to directly produce the correct instances for this problem. Because SMT solvers usually do not fully normalize quantified formulas during preprocessing and also because they do inspect quantified formulas deeply, the instantiation methods fail to recognize and exploit the fact that assertion 1 and the guard in assertion 2 are so similar. Since the instantiation methods do not produce the correct instances early, the SMT solver will need multiple instantiation rounds to solve the problem. As a consequence, one can observe some kind of *butterfly effect*: if the instantiation methods are unlucky, the solver might be misguided to explore a large set of irrelevant instances and reach the solving timeout.

Quantified assertions in CDCL($\mathcal{T}$) are considered black boxes, and are abstracted as propositional variables which occur as unit clauses in the propositional abstraction of the input formula. These units are generally of no value to the ground solver. We here make use of them to simplify larger formulas. To solve the example above we identify the occurrence of the unit assertion 1 within assertion 2. By using unification we can eliminate this quantified subformula. The result is the ground formula $\neg P(c)$ which trivially contradicts the ground part of the formula. In the general case, we use unit clauses to soundly simplify nested formulas and handle strong quantified variables without explicit Skolemization. Hence, we call our method *deep unit processing*. We also propose multiple generalizations of the core rule (Section 3).

We implemented deep unit processing in the SMT solver veriT [6]. To ensure the process is fast, we use a standard term index and unification algorithm which we extended to handle the presence of weakly quantified variables (Section 4).

Our evaluation on SMT-LIB benchmarks shows that deep unit processing allows veriT to solve benchmarks not solved by any strategy before. When applicable, deep unit processing often allows veriT to solve problems within a short timeout. The different variants of deep unit processing are useful within a strategy schedule (Section 5).

2 CDCL($\mathcal{T}$) and Quantifier Instantiation

Figure 1 shows the operation of a typical SMT solver when refuting a problem. It first preprocesses the input problem. (Section 2.2). Then two procedures together refutes the problem: the ground solver either refutes the problem on the ground level, or finds a ground model. If a model is found the instantiation procedure creates new ground lemmas (Section 2.3).

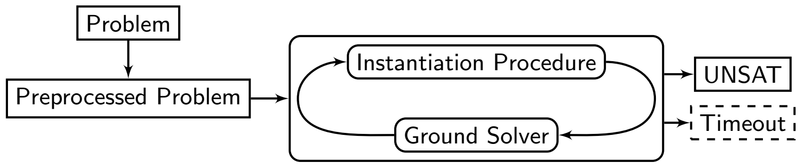

Fig. 1. The instantiation loop of an SMT solver refuting a problem.

2.1 Preliminaries

We use the many-sorted first-order logic with equality as defined in the SMT-LIB standard [4] and assume the reader is familiar with the notions of signature, term, free and bound variable, quantified and ground formula, literal, and substitution. We use x, y, z to denote variables; s, t to denote terms; φ, ψ to denote formulas (i.e., terms of sort Bool); P, Q to denote predicates (i.e., functions with codomain sort Bool); and c to denote constants. To denote the substitution which replaces a variable x with a term t we write $[t/x]$. As usual, σ stands for a substitution. We write $\bar{t}$ for the sequence of terms $t_1, \dots, t_n$ for an unspecified $n \in \mathbb{N}^+$ that is either irrelevant or clear from the context. We write $\mathbf{free}(t)$ to denote the free variables of a term t. The set $\mathcal{T}(S)$ is the set of all subterms of the terms in S. We omit sorts when they are clear from the context and assume that sort constraints are always respected, e.g., substitutions only use terms of the same sort as the substituted variable.

In accordance with the SMT-LIB standard, the signature Σ always contains a dedicated sort Bool, two constants $\top$ and $\bot$, the usual Boolean connectives, and a family of predicate symbols $(\approx : \tau \times \tau \to \mathsf{Bool})$ interpreted as equality for each sort τ. The *polarity* $\mathbf{pol}(\varphi) \in \{+, -\}$ of a formula φ is $+$ (*positive*) if the number of leading negations of φ is even and $-$ (*negative*) if it is odd. A trimmed formula is the generalization of the notion of atom to arbitrary formulas: $\mathbf{trim}(\varphi)$ is the formula φ after removing all leading negations.

We write $t[\,]$ for a term with a hole and $t[u]$ for the term where the hole has been replaced by u. A term has at most one hole. We borrow the notions of weak and strong quantifiers [1]: since we are working in a refutation context, a positive occurrence of a quantifier $\exists\bar{x}.\,\varphi$ or a negative occurrence of a quantifier $\forall\bar{x}.\,\varphi$ is strong and a negative occurrence of a quantifier $\exists\bar{x}.\,\varphi$ or a positive occurrence of a quantifier $\forall\bar{x}.\,\varphi$ is weak. The *matrix* of a formula $Q\bar{x}.\psi$ is ψ. Without loss of generality, we assume that all quantified variables are unique.

To handle strong quantifiers we will later use the Skolemization operator $\mathbf{sk}$. For a formula $Q\bar{x}.\psi$ where $Q \in \{\forall, \exists\}$, it is defined as

$$\mathbf{sk}(Q\bar{x}.\psi, \bar{y}) := \psi[s_1(\bar{y})/x_1] \dots [s_n(\bar{y})/x_n]$$

where each s_i is a fresh function symbol of appropriate arity.

2.2 Preprocessing

Given an input formula $\mathcal{P}$, i.e., a term of sort Bool, a CDCL($\mathcal{T}$) solver performs multiple preprocessing steps before it starts solving. This produces an equisatisfiable problem $\mathcal{P}'$. To make efficient use of the ground solver the problem is clausified. Hence, $\mathcal{P}'$ will have the form $(\ell_1^1 \vee \cdots \vee \ell_{n_1}^1) \wedge \cdots \wedge (\ell_1^m \vee \cdots \vee \ell_{n_m}^m)$. Clausification considers quantified formulas as black boxes. Through this text the disjunctions $\ell_1^i \vee \cdots \vee \ell_{n_i}^i$ are called clauses, and the formulas ℓ_j^i are called literals. A clause with only one literal is a unit clause. Notice that literals can have a first-order structure: they might be predicate applications, but they might also be quantified formulas (or negations thereof).

Some light form of rewriting is applied on quantified formulas. In veriT, most rewriting steps apply on constants. For example, the term $f(5{+}c_1{+}3, c_2{*}(3{-}3))$ is simplified to $f(8{+}c_1, 0)$. Such rewriting is also performed on Boolean connectives. For example, $(\bot \rightarrow \varphi_1) \rightarrow \varphi_2$ is replaced by φ_2. Rewriting also ensures that certain global invariants of veriT are met: for instance, all bound variables are renamed to be distinct, and quantifiers over Boolean variables are removed.

Another common preprocessing step is Skolemization. The CDCL($\mathcal{T}$) calculus does not impose how Skolemization is applied, so this is essentially implementation dependent. In their default configuration, CVC4 [3] and veriT only Skolemize outermost strong quantifiers. The SMT solver Z3 [12], however, has a builtin tactic called *nnf* that fully applies Skolemization. To Skolemize the outermost strong quantifiers, the rewriter of veriT replaces the subformula $\mathbf{trim}(\ell)$ of a literal ℓ by $\mathbf{sk}(\mathbf{trim}(\ell))$ if the subformula has the form $Q\bar{x}.\,\varphi$ with $Q = \exists$ if $\mathbf{pol}(\ell) = +$ or $Q = \forall$ if $\mathbf{pol}(\ell) = -$.

Preprocessing does not fully Skolemize, put in prenex form, or clausify quantified formulas. Hence, some literals of $\mathcal{P}'$ might start with a weak quantifier and contain complicated terms. Without loss of generality, we assume quantified literals have positive polarity and are therefore universally quantified. All other literals in $\mathcal{P}'$ are ground.

2.3 Instantiation Techniques

The instantiation loop (Figure 1) starts with the *ground solver* searching for a ground model of the preprocessed problem $\mathcal{P}'$. It considers the quantified literals as black boxes. The ground solver either determines that the ground literals are unsatisfiable or produces a *ground model* $\mathcal{M}$. If the ground problem is unsatisfiable, then $\mathcal{P}$ is unsatisfiable. The ground model $\mathcal{M}$ is a set of literals $\mathcal{G} \cup \mathcal{Q}$ where $\mathcal{G}$ are ground literals, and $\mathcal{Q}$ are quantified literals. $\mathcal{M}$ propositionally entails $\mathcal{P}'$ and $\mathcal{G}$ is consistent with respect to the used theories. The instantiation procedure will then generate lemmas of the form $(\forall \bar{x}.\, \varphi) \rightarrow \varphi\sigma$ where $(\forall \bar{x}.\, \varphi) \in \mathcal{Q}$ and σ is a substitution of the variables to ground terms. The generated lemmas are then added conjunctively to $\mathcal{P}'$ and the ground solver is called again.

We now give an overview of common instantiation techniques and illustrate why they cannot tackle Example 1 quickly. These techniques are presented in the order they are used by veriT: it first tries conflict driven instantiation. If this fails, it will try trigger-based instantiation. Should this also produce no instances, it will fall back to enumerative instantiation. model-based quantifier instantiation is not implemented within veriT: it is crucial for satisfiability but veriT mainly focuses on proving unsatisfiability.

Conflict-Driven Instantiation. This method tries to find an instance which contradicts the ground model $\mathcal{M}$ in the theory of equality and uninterpreted functions (EUF) [2, 14]. Hence, it searches for a literal $\forall \bar{x}.\, \varphi \in Q$ and a substitution σ, such that $\mathcal{G} \wedge \varphi\sigma \models_{\text{EUF}} \perp$. It returns the instance $\varphi\sigma$ or fails. The algorithm used to find conflicting instances can solve multiple constraints $\mathcal{G} \wedge \psi_1\sigma \models_{\text{EUF}} \perp, \ldots, \mathcal{G} \wedge \psi_n\sigma \models_{\text{EUF}} \perp$ simultaneously. Hence, φ can be of the form $\psi_1 \vee \cdots \vee \psi_n$, but if any ψ_i is itself quantified, the method just fails.

This technique is very helpful, since it only generates instances that are immediately useful. It forces the ground solver to find new models and eliminates spurious models from the search space.

Since assertion 2 of Example 1 contains a quantifier, it cannot be instantiated by conflict driven instantiation. Conflict driven instantiation also fails for assertion 1, because initially there is no ground formula that would be in conflict with an instance of $P(f(x, c))$. Even if the second assertion was Skolemized, conflict-driven instantiation would fail: since there is no ground instance of the Skolem term, no conflicting instance can be found.

Trigger-Based Instantiation. This instantiation scheme works by matching *triggers* with the current ground model. Triggers associate with every literal $\forall \bar{x}.\, \varphi \in \mathcal{Q}$ one or more lists of quantifier-free terms $t_1, \ldots, t_n$ such that $\mathbf{free}(t_1) \cup \cdots \cup \mathbf{free}(t_n) = \{\bar{x}\}$. To construct instances of φ, trigger-based instantiation searches for substitutions σ and terms $g_1, \ldots, g_n \in \mathcal{T}(\mathcal{G})$ such that $\mathcal{G} \models_{\text{EUF}} t_i\sigma \approx g_i$. If the search is successful, it returns the instance $\varphi\sigma$. The process of matching terms within the theory of equality and uninterpreted functions is called *E-matching.* [7, 8, 11]

The triggers are either provided by the user as annotations or heuristically inferred during preprocessing. Due to the heuristic nature of trigger-based instantiation, the generated instances might not be useful to solve the problem. Instead they can slow down or mislead the solver.

In the case of Example 1, a trigger $P(x)$ on assertion 1 would produce the useless instance $P(c) \rightarrow P(f(c, c))$ and a trigger $P(f(x, c))$ can initially not match anything. The trigger $P(y)$ on assertion 2 would produce the instance $(\forall z.\, P(z) \rightarrow P(f(z, c))) \rightarrow \neg P(c)$. This instance is a step towards solving the problem: the strong variable z is no longer below a quantifier and will be Skolemized to create the formula $P(s_1) \rightarrow P(f(s_1, c)) \rightarrow \neg P(c)$ where s_1 is a fresh constant. During the next instantiation round the trigger $P(x)$ on assertion 1 generates the instance $P(s_1) \rightarrow P(f(s_1, c))$ which leads to the contradiction.

This technique is very sensitive to the availability of the right ground terms in the ground model. In the above example, if the formula contained $\forall x.\, P(x)$ instead of $P(c)$, trigger-based instantiation would have been helpless.

Enumerative Instantiation. While conflict driven instantiation is guided by the ground model it tries to contradict, and trigger-based instantiation is guided by the triggers, enumerative instantiation is unguided [13]. For a literal with the form $\forall \bar{x}.\, \varphi \in \mathcal{Q}$ it creates all substitutions $[\bar{t}/\bar{x}]$ where the terms $\bar{t}$ are ground terms from $\mathcal{T}(\mathcal{M})$. To limit the number of generated instances the procedure only uses the ground terms minimal with respect to so some term order and does not return instances already implied by the ground model (i.e., it only returns $\varphi\sigma$ if $\mathcal{G} \not\models_{\mathrm{EUF}} \varphi\sigma$). Enumerative instantiation ensures the theoretical completeness of the SMT solver for the theory of uninterpreted functions. It can also find out the small ground terms that are sometimes necessary to enable the two previous techniques to work, and is thus a useful fallback strategy.

For Example 1, enumerative instantiation also needs at least two rounds. First, the variable y of assertion 2 is instantiated with c. Then, after Skolemization, assertion 1 can be instantiated with the generated Skolem constant. Eventually, the cooperation of enumerative instantiation and the above techniques would succeed. However, in presence of many ground terms of the same sort as c, enumerative instantiation might have needed a lot of time to find the right instance.

Model-Based Quantifier Instantiation. Finally, model-based quantifier instantiation [9] builds a first-order model $\mathfrak{M}$ from the ground model $\mathcal{M}$. If there exist a literal of the form $\forall \bar{x}.\, \varphi \in \mathcal{Q}$ and a substitution σ of $\bar{x}$ with terms from $\mathcal{T}(\mathcal{G})$ such that $\mathfrak{M} \models \neg\varphi\sigma$, then $\mathfrak{M}$ is not a true model of $\mathcal{P}'$ and the ground instance $\varphi\sigma$ is produced. If no such literal and substitution can be found, then $\mathfrak{M}$ is a model of $\mathcal{P}'$ and the input problem is satisfiable.

For Example 1, model-based quantifier instantiation fails for the same reason that trigger-based and enumerative instantiation fail: it might instantiate assertion 2 with c for y, but other rounds of instantiation will still be required to reach a contradiction.

3 Deep Unit Processing

The essence of our technique is to simplify formulas by replacing a quantified subformula with the Boolean constant $\top$ or $\bot$. This can be done if the matrix of this quantified subformula can be unified with the matrix of a unit clause.

Example 2. On our running Example 1, the first assertion serves as a unit clause, whose matrix is unifiable with the matrix of a quantified subformula in the second assertion. As a result, the quantified subformula can be reduced to the Boolean constant $\top$, for some instance of the second formula:

$$\frac{\forall x.\, P(x) \to P(f(x,c)) \quad \forall y.\, (\forall z.\, P(z) \to P(f(z,y))) \to \neg P(y)}{\top \to \neg P(c)}$$

The rewriter simplifies the formula $\top \to \neg P(c)$ to $\neg P(c)$. Notice that, in this example, the variable z must be Skolemized because its quantifier is strong.

The above derivation will be formalized in the DUP rule (Section 3.1), which can be integrated in the preprocessing phase of an SMT solver (Section 3.2). In Section 3.3, we propose several variants of the base rule with different tradeoffs.

3.1 The Deep Unit Processing Rule

The deep unit processing rule replaces a quantified subformula of a quantified formula with a Boolean constant. To be able to do so, the rule unifies the matrix of the subformula with a unit clause using a substitution. The Boolean constant depends on the polarities of the matrices: if they have the same polarity the subformula is replaced by $\top$, if they have different polarity it is replaced by $\bot$. The conclusion of the rule is the *pre-simplified* formula and will be fully simplified by the rewriter.

Definition 1 (DUP Rule).

$$\frac{\forall x_1, \dots, x_n.\, \psi_1 \quad \forall x_{n+1}, \dots, x_m.\, \varphi[Q\bar{y}.\, \psi_2]}{\forall x_{k_1}, \dots, x_{k_j}.\, \varphi[b]\sigma}\ \text{DUP}$$

where $Q \in \{\exists, \forall\}$, $Q\bar{y}.\, \psi_2$ appears only below the outermost universal quantifier, and σ is a substitution. The rule is subject to the conditions,

1. $\mathbf{trim}(\psi_1)\sigma = \mathbf{trim}(\psi_2)\sigma$*, if* $Q\bar{y}.\, \psi_2$ *is weak;*
2. $\mathbf{trim}(\psi_1)\sigma = \mathbf{trim}(\mathbf{sk}(\psi_2, x_{n+1} \dots x_m))\sigma$*, if* $Q\bar{y}.\, \psi_2$ *is strong;*
3. *the bound variables of the conclusion* $\{x_{k_1}, \dots, x_{k_j}\}$ *are exactly* $\mathbf{free}(\varphi[b]\sigma)$*;*
4. $b = \top$ *if* $\mathbf{pol}(\psi_1) = \mathbf{pol}(\psi_2)$ *or* $b = \bot$ *if* $\mathbf{pol}(\psi_1) \neq \mathbf{pol}(\psi_2)$*.*

Notice that this rule generates a simpler version of the right premise, but in general, this simpler version does not subsume the original premise. So the premise cannot be removed from the original formula. Adding the right redundant formulas help the SMT instantiation procedures to find the appropriate instances.

Example 3. In the running example the subformula $\forall z.\, P(z) \to P(f(z, y))$ appears with negative polarity. Since $Q = \forall$, the formula must be Skolemized (Condition 2):

$$\mathbf{sk}(\forall z.\, P(z) \to P(f(z, y)), y) = P(s_1(y)) \to P(f(s_1(y), y))$$

Hence, the unifier used in Example 2 is $\sigma = [x/s_1(c)][y/c]$.

Example 4. Ignoring Skolemization (Condition 2) leads to unsoundness:

$$\frac{\forall x.\, P(x, x) \quad \forall y.\, \neg(\forall z.\, P(y, G(z)))}{\neg\top}$$

The result of Skolemization $\mathbf{sk}(\forall z.\, P(y, G(z)), y)) = P(y, G(s_1(y)))$ is not unifiable with $P(x, x)$. The rule is not applicable.

Example 5. The conclusion can contain variables from both premises. Here the unifier is $\sigma = [y_1/G(x)][z/c]$.

$$\frac{\forall x.\, P(G(x), c) \quad \forall y_1, y_2.\, (\forall z.\, P(y_1, z)) \land P(y_1, y_2)}{\forall x, y_2.\, \top \land P(G(x), y_2)}\ \text{DUP}$$

Example 6. The above examples were cases where $\mathbf{pol}(\psi_1) = \mathbf{pol}(\psi_2)$. This one illustrates the other case:

$$\frac{\forall x.\, \neg P(x, x) \quad \forall y.\, G(c) \land (\forall z.\, P(y, z))}{G(c) \land \bot}\ \text{DUP}$$

3.2 The Simplification Loop

```
   $I \leftarrow \emptyset$
   $Q$ is an empty queue.
   **for** each clause $\mathcal{C}$ in $\mathcal{P}'$ **do**
       **if** $\mathcal{C}$ is a unit clause with the literal $\ell$ **then**
5          $I \leftarrow I \cup \{\ell\}$
       **if** $\mathcal{C}$ contains a literal of the form $\forall \bar{x}.\, \varphi$ **then**
7          push$(Q, \mathcal{C})$
   **while** $Q$ is not empty **do**
       $\ell_1 \lor \cdots \lor \ell_n \leftarrow$ pop$(Q)$
10     **if** there is $\psi \in I$ and $\ell_i$ such that $\dfrac{\psi \quad \ell_i}{\ell'}\ \text{DUP}$ **then**
11         $\ell' \leftarrow$ rewrite$(\ell')$
12         $\mathcal{C}' \leftarrow \ell_1 \lor \cdots \lor \ell_{i-1} \lor \ell' \lor \ell_{i+1} \lor \cdots \lor \ell_n$
13         append $\mathcal{C}'$ to $\mathcal{P}'$
           **if** $\mathcal{C}'$ contains a literal of the form $\forall \bar{x}.\, \varphi$ **then**
15             push$(Q, \mathcal{C}')$
```

Fig. 2. The simplification algorithm of deep unit processing.

In an SMT solver deep unit processing is performed after preprocessing finishes and before the ground solver starts. The pseudocode in Figure 2 provides the outer loop of deep unit processing. It first iterates over the preprocessed problem $\mathcal{P}'$ to build a set I of unit clauses (Line 5) which can be used to simplify quantified subformulas. At the same time, this loop collects in a queue Q all clauses containing quantified literals (Line 7). Then the algorithm takes a clause from the queue and tries to simplify a literal of it. To do so it uses the DUP rule. If this succeeds, the conclusion is the pre-simplified formula. The algorithm then uses the rewriter to finish the simplification. The resulting simplified formula is added conjunctively to the problem (Lines 10 to 13). If the simplified formula is not ground, the clause is pushed back onto the queue (Line 15).

3.3 Variants of Deep Unit Processing

As the experimental evaluation (Section 5) shows, the above version of deep unit processing — that we will, starting now, call the *normal variant* — solves more instances at little cost. Nevertheless, we also developed several variants with different tradeoffs.

Aggressive Simplification. Since quantified subformulas block the instantiation procedures from creating the right instances quickly, the DUP rule is restricted to only simplify quantified subformulas. This restriction, however, can be removed to generate more simplified formulas.

The unrestricted rule can be applied on any subformula not below an extra quantifier. On the one hand, it corresponds to deriving some general consequences of unit clauses in full first-order logic, but on the other hand, it will generate many more new formulas which potentially slow down the solver and can misguide the solver.

Solitary Variable Heuristic. Instead of eliminating quantified subformulas, variables can also be eliminated from the outermost quantifier of the pre-simplified formula. A variable is removed from the pre-simplified formula if it is *solitary*: it appears in the subformula to be simplified, but not in any other subformula of ψ_2. Hence, for example, in the case $\varphi = t_1 \vee \cdots \vee t_i \vee \cdots \vee t_n$ we try to simplify t_i if there is a variable $x \in \mathbf{free}(t_i)$ such that $x \notin \mathbf{free}(t_1 \vee \cdots \vee t_{i-1} \vee t_{i+1} \vee \cdots \vee t_n)$.

This is a compromise between the normal variant of deep unit processing and the aggressive variant. The resulting formula will produce fewer potentially misleading instances.

Deletion of Simplified Formulas. Another way to restrict the number of newly created instances is to delete original formulas after they have been simplified by deep unit processing. While this is no longer complete, it can guide the solver towards solving the refutation problem. Especially, within a strategy schedule this can be a valuable strategy.

This variant can be combined with the three other variants. Overall, this results in six variants of deep unit processing. The amount of formulas deleted depends on the activity of the simplification variant used. Especially in the case of aggressive simplification with deletion many input assertions will be deleted.

4 Implementation

The DUP rule, as presented above, explicitly uses Skolemization. Skolemization however involves the creation of fresh function symbols, and since the rule is used a lot, mostly to fail, Skolemization would generate a lot of garbage symbols. Our implementation is amended to take the strong variables into account without Skolemization and avoid the creation of those unnecessary symbols. To realize the lookup of the first premise of the DUP rule, we implemented an non-perfect term

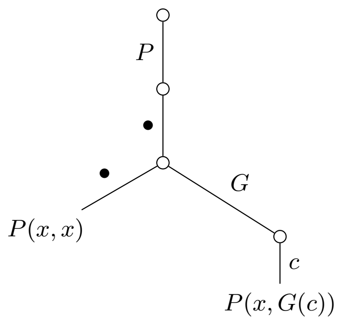

Fig. 3. A discrimination tree indexing two terms.

index (Section 4.1). Since the index is non-perfect, a full unification algorithm is still necessary after lookup (Section 4.2).

The implementation in veriT also does not apply the simplification of clauses everywhere, but focuses on unit clauses only: in other words, Q will be populated by unit clauses only. It indeed appears that quantifiers range most often over full disjunctions, in our benchmarks, and not on disjuncts.

4.1 Indexing

A key element to execute deep unit processing as shown in the algorithm in Figure 2 is the lookup of the formula $\forall \bar{x}.\, \psi_1$ from the index I. The trimmed matrix of this formula must be unifiable with the trimmed matrix of the quantified subformula $Q\bar{y}.\, \psi_2$. To implement the search for unifiable formulas efficiently, we use a term index. Such an index allows the efficient lookup of terms unifiable with a query term. We use non-perfect discrimination trees [15]. Non-perfect means that the lookup is an over-approximation: some returned terms are not unifiable with the query term and must be removed by a full unification step.

For all unit clause $\forall \bar{x}.\, \psi_1$ (of I in the algorithm in Figure 2) the index stores $\mathbf{trim}(\psi_1)$ together with $\mathbf{pol}(\psi_1)$. For each possible subformula $Q\bar{y}.\, \psi_2$ the implementation uses $\mathbf{trim}(\psi_2)$ as a query term and retrieves unification candidates and their polarity. Afterwards, it performs a full unification to construct the substitution σ when possible. If, however, the quantifier of $Q\bar{y}.\, \psi_2$ is strong, the subformula should be Skolemized. To take this into account, our implementation uses a slightly enhanced lookup process.

Discrimination trees work similarly to a string trie. Indexed and query terms are translated into strings by preorder traversal. Variables are replaced by a placeholder. For example, $P(x, G(c))$ becomes the string $[P, \bullet, G, c]$. The edges of the tree represent the individual characters of the indexed strings and the leaves store the index terms. Figure 3 shows a discrimination tree with the indexed terms $P(x, x)$ and $P(x, G(c))$. During lookup the tree is traversed depth-first. If the query term or the indexed term contain variable placeholders, backtracking is used to collect all matches. Since variables are replaced by placeholders, the collected terms might not be unifiable if the query or indexed term contain repeated variables. Hence, a full unification algorithm still has to be used, but most unification pairs that eventually fail are filtered out.

To handle variables that would be Skolemized, the construction of the query string is slightly enhanced: it does not replace the implicitly Skolemized variables $\bar{y}$ with the variable placeholder ($\bullet$). Instead the variables act like constants that can no longer match other constants and functions. Only variables in the indexed term can capture these. Since Skolem terms start with fresh function symbols which can never mach indexed terms, this embeds Skolemization on the fly into indexing. For example, the query term $P(x, G(y))$ matches both terms stored in the tree in Figure 3, but if y should be Skolemized it will only match $P(x, x)$. However, as Example 4 above shows, $P(x, x)$ is also a false positive.

4.2 Unification without Skolemization

After the index returns a filtered set of possible premises $\mathbf{trim}(\psi_1)$ with their polarities $\mathbf{pol}(\psi_1)$ from the index I, the implementation must use full unification [10] to build the unifier σ and to eliminate false positives. It has to solve the unification problem between $\mathbf{trim}(\psi_1)$ and $\mathbf{trim}(\psi_2)$, where $\mathbf{trim}(\psi_2)$ can contain variables that must be Skolemized.

The unification algorithm iteratively builds a substitution σ by solving unification constraints of the form $t_1 = t_2$ by recursion on the term structure. If $f_1(\bar{t}) = f_2(\bar{u})$, unification fails if $f_1 \neq f_2$, otherwise it solves the new constraints $t_i\sigma = u_i\sigma$. If $x = t$, σ is extended with the substitution $[t/x]$ if x does not appear in t. The condition that x does not appear in t is the *occurs check*.

To handle Skolemized variables, our implementation of unification deviates from the standard version in two ways. First, it handles Skolemized variables as constants, similarly to what we described for the term index. Then, during the occurs check, it considers a Skolemized variable as an occurrence of all the variables its Skolem term would depend on.

Example 7. In Example 4 the algorithm tries to unify $P(x, x)$ with $P(y, G(z))$ where Skolemization would replace z with the Skolem term $s_1(y)$. Unification proceeds as follows:

1. to solve $P(x, x) = P(y, G(z))$ the constraints $x = y$ and $x = G(z)$ are added;
2. since $x = y$, the substitution is set to $\sigma = [y/x]$;
3. to solve $x\sigma = G(z)\sigma$, that is, $y = G(z)$, the algorithm performs an occurs check for y in $G(z)$. Since z implicitly stands for $s_1(y)$, y appears in $G(z)$ and unification fails.

The resulting substitution σ cannot substitute a Skolem term into the quantified variables $x_{n+1}, \ldots, x_m$ of the formula that is simplified. Hence, the conclusion $\varphi[b]\sigma$ is free of any Skolem terms, and no Skolem term has ever to be constructed. Overall, restricting deep unit processing to not simplify formulas below multiple nested quantifiers allows for this elegant implementation.

5 Evaluation

This section presents an empirical evaluation of deep unit processing and its variants as implemented in veriT.[3] The default variant of deep unit processing solves more benchmarks than the default configuration of veriT, while loosing few benchmarks. This justifies the activation of deep unit processing in the default configuration. Almost all other variants also solve more benchmarks. veriT exposes a wide range of options to fine-tune the instantiation module. A specific configuration is a *strategy*. Deep unit processing solves benchmarks not solved by any veriT strategy without this technique (Section 5.1).

To fully benefit from the strategies available, veriT can use strategy schedules. We generated strategy schedules with and without deep unit processing and evaluated their performance. The strategies with deep unit processing are an integral component of the generated schedules and increase the number of solved benchmarks. They are especially useful for short timeouts (Section 5.2).

We performed the experiments on the benchmarks form the SMT-LIB benchmark release 2020 [4]. Since deep unit processing works with first-order formulas, we used the SMT-LIB logics supported by veriT which use quantifiers, uninterpreted functions, or arrays. Those are the SMT-LIB logics UF, UFLRA, UFLIA, UFIDL, ALIA, AUFLIA, and AUFLIRA. Since veriT is purely refutational, we removed benchmarks known to be satisfiable from the analysis.[4] Overall, SMT-LIB contains 41 208 benchmarks using these logics. Of those, 1241 benchmarks are known to be satisfiable. This leaves 39 967 relevant benchmarks.

To interpret the numbers, the reader should also keep in mind that veriT has no array solver. It considers the functions of the SMT-LIB theory of arrays as uninterpreted functions. Since veriT is restricted to refute benchmarks, this approach is sound. Nevertheless, veriT can fail to solve easy benchmarks that require array reasoning.

All experiments have been performed on computers with two Intel Xeon Gold 6130 processors with a total of 64 cores and 192 GiB RAM. We ran one instance of veriT per available core and used a memory limit of 6 GiB per instance.

5.1 Baseline Comparison

Table 1 shows the number of benchmark solved within a timeout of 180 s in comparison to the default strategy. The standard version of deep unit processing is denoted *N*, aggressive simplification is denoted *A*, and the solitary variable heuristic is denoted *S*. An suffix *d* denotes the deletion of simplified formulas. Benchmarks are "Gained" if they are not solved by the default strategy and "Lost" if they are solved by the default strategy, but not by the variant. The column "Total" reports the union of the benchmarks solved by all variants.

[3] Note to reviewers: An archive of the experimental data and other material is available at `https://schurr.io/CADE-dup/`.

[4] Since satisfiable benchmarks can identify soundness problems of the implementation, we did not remove them from the experiment run.

Table 1. Comparison of deep unit processing with the default strategy and the theoretical best solver on 39 967 benchmarks.

vs. Default (solves 31 768)	N	A	S	Nd	Ad	Sd	Total
Solved	+**234**	+83	+50	+49	−10 278	+48	+418
Gained	282	**316**	294	293	115	292	432
Lost	**48**	233	244	244	10 393	244	14
vs. Theoretical Best (solves 32 709)							
Gained	82	80	**86**	**86**	32	**86**	**118**

The normal variant shows the biggest improvement by solving 238 benchmarks more. Most other variants solve more benchmarks not solved by the default strategy, but also lose many more. While the normal variant does not have the highest gain, the small loss justifies enabling it in the default strategy of future veriT releases. The huge number of lost benchmarks for the variant with aggressive simplification and formula deletion is not surprising: since most input assertions can be simplified in some way, formula deletion removes much of the original problem. The result is often an unsolvable problem.

Compared to the union of benchmarks solved by *any* existing veriT strategy (Theoretical Best) deep unit processing shows good improvement. We used a list of 43 strategies which are also used by veriT in the SMT competition.[5] The default configuration of veriT is part of this list. Overall, the deep unit processing variants are able to solve 118 benchmarks that veriT could not solve before. While aggressive simplification solves only 80 more, 18 of those are not solved by any other deep unit processing variant. Here aggressive simplification with formula deletion is somewhat redeemed: it solves eight benchmarks not solved by the theoretical best solver and the other deep unit processing variant.

To execute deep unit processing, veriT does not need much time: for the normal variant, we measured a median runtime of 0.5 ms and mean of 3 ms.

5.2 Strategy Scheduling

Since quantifier instantiation relies on heuristics, veriT exposes parameters that can be set by the user. A specific choice of values for the exposed parameter is a strategy. Most benchmarks are solved by an appropriate strategy within a short timeout. Hence, it is sensible to execute many strategies for short time intervals one after another in a schedule.

To evaluate deep unit processing within a strategy schedule, we generated schedules with and without strategies extended with deep unit processing. veriT schedules are automatically generated from a hand crafted list of strategies and timeouts. An optimal schedule is the set of strategy–timeout pairs which solves the most benchmarks. We use an integer programming system to solve this optimization problem. veriT itself uses the logic of the input problem to select a schedule.

[5] Competition website: `https://smt-comp.github.io/`

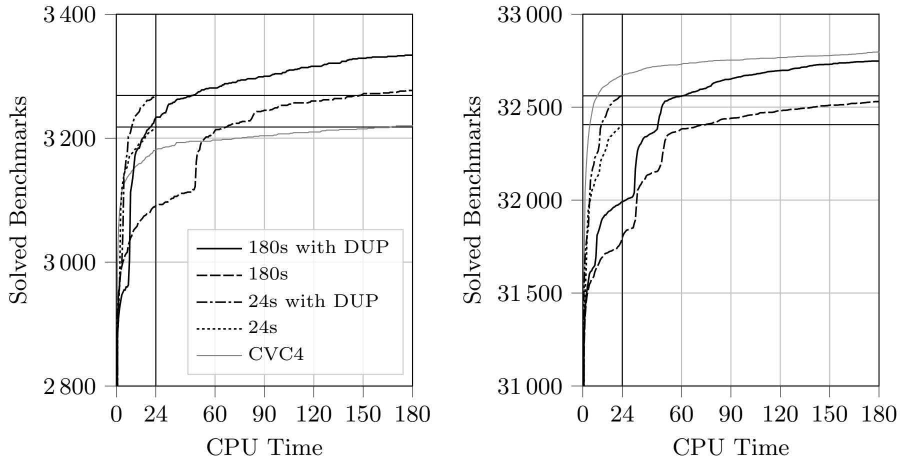

Fig. 4. CDF of different schedules on UF only and all logics.

To build strategies with deep unit processing, we picked from the strategy list of 43 strategies the one that solved the most benchmarks. We also selected two strategies which complement this strategy on all logics and on first-order logic with equality (UF) alone. Finally, we added the default strategy. Overall, this resulted in six strategies. We then added the six variants of deep unit processing to those strategies. This results in 36 new strategies.

We generated schedules optimized for timeouts of 180 s and 24 s. The short 24 s timeout allows us to evaluate the value of of deep unit processing for applications such as interactive theorem provers, which require a short timeout. It corresponds to the timeout used by the SMT competition to evaluate solvers for this purpose.

Figure 4 shows the number of benchmarks solved within a time limit on UF alone and on all logics. On all logics, the schedule with deep unit processing solved 155 benchmarks more after 24 s than the original 24 s schedule. For the 180 s timeout, the 180 s schedule with deep unit processing solve 218 more than the one without. The 24 s schedule with deep unit processing solves 31 benchmarks more than the 180 s schedule after 180 s. Hence, deep unit processing is very useful for short timeouts. Since the form of quantified lemmas that deep unit processing eliminates appear in problems generated by interactive theorem provers, it is most especially useful for this application.

As a reference, the grey line shows the benchmarks solved by the state-of-the-art SMT solver CVC4. We used the official built of version 1.8, and discarded all "satisfiable" results. Since CVC4 has no scheduler optimized for 24 s or 180 s, we ran the default strategy.[6] Overall, it solves 110 benchmarks more than veriT with deep unit processing after 24 s and 49 after 180 s. veriT with deep unit processing

[6] By using the command line arguments: `-L smt2.6 --no-incremental --no-type-checking --no-interactive --full-saturate-quant`

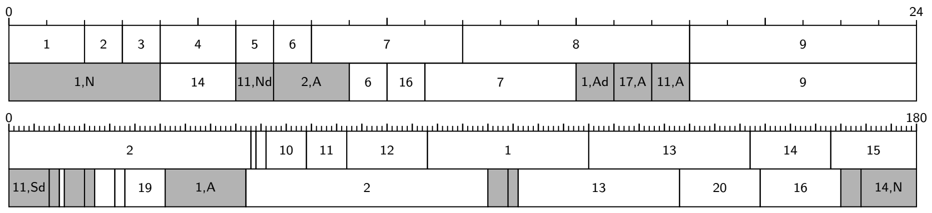

Fig. 5. Visualization of optimized UF schedules. The bottom rows are the schedules with deep unit processing. The numbers denote the base strategies.

after 180 s solves 581 benchmarks not solved by CVC4, of which 95 are also not solved by veriT without deep unit processing. Remember that, contrary to veriT, CVC4 fully supports arrays.

Figure 5 visualizes the schedules for the logic UF. Cells highlighted in grey are strategies which use a variant of deep unit processing. Cells with the same number use the same base strategy. The majority of the new strategies are used during the first half of the schedule. Some base strategies appear both in the schedules with and without deep unit processing. For the 24 s schedule the first strategy is extended with the normal variant of deep unit processing and used for longer. The 180 s schedule with deep unit processing starts with several strategies extended with deep unit processing. Afterwards it uses the strategy 2 that the schedule without deep unit processing uses right at the start.

6 Conclusion

Deep unit processing is a new simplification technique for instantiation-based SMT solvers. Its design is motivated by limitations of modern instantiation methods, and it is efficient. Problems where formulas can be simplified are often solved much faster, despite the simplification creating new quantified formulas. We plan to enable deep unit processing by default in the next veriT release. The release will also produce machine-checkable proofs for simplifications performed by deep unit processing.[7]

We believe that the technique implemented here within veriT can be ported easily into any instantiation-based SMT solver, and we are confident that it would also enable mainstream solvers to tackle problems outside of reach with other current strategies. We will investigate its potential in other solvers.

So far, techniques inspired by resolution-based theorem provers are under represented in SMT solvers. Our method is a step towards using such techniques. It is currently only used as a preprocessing technique, but we plan to investigate novel quantifier instantiation techniques which can directly handle nested strong quantifiers.

[7] Note to reviewers: see Section 7.1 in the appendix.

Acknowledgments. We are grateful to Haniel Barbosa, Jasmin Blanchette, Daniel El Ouraoui, Mathias Fleury, and Martin Riener for many fruitful discussions and suggestions to improve the text. The second author has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

1. Baaz, M., Egly, U., Leitsch, A., Goubault-Larrecq, J., Plaisted, D.: Chapter 5 - normal form transformations. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 273–333. Handbook of Automated Reasoning, North-Holland, Amsterdam (2001). https://doi.org/10.1016/B978-044450813-3/50007-2
2. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 214–230. Springer Berlin Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_13
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
5. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
6. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 22. LNCS, vol. 5663, pp. 151–156. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102
8. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 21. pp. 167–182. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_12
9. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. pp. 306–320. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
10. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems **4**(2), 258–282 (Apr 1982). https://doi.org/10.1145/357162.357169

11. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 21. pp. 183–198. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
12. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-89963-3_7
14. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in smt. In: FMCAD 2014. pp. 195–202. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987613
15. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term Indexing, p. 1853–1964. Elsevier Science Publishers B. V., Amsterdam, Netherlands (2001). https://doi.org/10.5555/778522.778535

7 Appendix

7.1 Proof Production

Deep unit processing integrates nicely into the calculus used by veriT to generate proofs. This is not fully implemented and tested, but we are confident that this will be done in the very near future, without issue. This appendix gives an overview of how deep unit processing fits into the proof framework.

The veriT solver produces detailed proofs for unsatisfiable formulas, to be checked or reconstructed by third party tools. The generated proofs contain fine-grained steps for preprocessing. Hence, deep unit processing must also produce a proof of the simplified formula. To support proofs for deep unit processing no specialized rule is necessary. Instead the existing framework for fine-grained preprocessing steps can be used.

A proof of the applying deep unit processing consists four components:

1. A proof of applying σ to $\mathbf{trim}(\psi_1)$ to get $\mathbf{trim}(\psi_1)\sigma$.
2. A subproof starting with the application of σ to φ. During this subproof $\mathbf{trim}(\psi_2)\sigma$ will be proven which is equal to $\mathbf{trim}(\psi_1)\sigma$.
3. A lift step, which introduces $\mathbf{trim}(\psi_1)\sigma \leftrightarrow \top$ into the subproof by using the conclusion of the first component. This enables the subproof to prove the pre-simplified formula.
4. An application of the proof producing rewriter to simplify the pre-simplified formula.

There is currently no rule which can express the lift step, but such a rule is a simple and natural addition. Furthermore, to proof the application of the substitution σ a fine-grained instantiation rule is necessary. The current instantiation rule is quite restrictive: it forces all variables to be instantiated. Then, contrary to the rest of the proof, the instantiation rule does not provide a fine-grained proof of the substitution of the variables with ground terms.

The generalized quantifier instantiation rule is:

$$\frac{\Gamma, x_{l_1}, \dots, x_{l_m}, x_{k_1} \mapsto t_1, \dots, x_{k_{n-m}} \mapsto t_{k_{n-m}} \rhd \varphi \leftrightarrow \varphi'}{\Gamma \rhd \forall x_1, \dots, x_n.\, \varphi \rightarrow \forall x_{l_1}, \dots, x_{l_m}.\, \varphi'}\ \text{inst}$$

This rule uses a context. Contexts are used by the proof format to express fine grained proofs of term manipulations below quantifiers. The context is essentially a short notation for a lambda term. For example, in $x \mapsto s \rhd t_1 \approx t_2$ the context $(x \mapsto s)$ is used to denote $(\lambda x.t_1)\, s \approx t_2$.

Example 8. Applying this process to the simplification of Example 2 we first get:

$$\dfrac{\dfrac{\forall x.\, P(x) \rightarrow P(f(x,c)) \qquad \dfrac{\dfrac{\dots}{\begin{array}{c} x \mapsto z \rhd (P(x) \rightarrow P(f(x,c))) \\ \leftrightarrow (P(z) \rightarrow P(f(z,c))) \end{array}}}{\begin{array}{c} \forall x.\, P(x) \rightarrow P(f(x,c)) \\ \rightarrow \forall z.\, P(z) \rightarrow P(f(z,c)) \end{array}}\ \text{inst}}{\forall z.\, P(y) \rightarrow P(f(z,c))}\ \dots}{\dagger}$$

Then we perform the simplification on the second assertion:

$$\dfrac{\begin{array}{c} \forall y.\, (\forall z.\, P(z) \rightarrow P(f(z,y))) \\ \rightarrow \neg P(y)) \end{array} \qquad \dfrac{\dfrac{\dfrac{}{y \mapsto c \rhd (\forall z.\, P(z) \rightarrow P(f(z,y))) \leftrightarrow \top}\ \text{lift } \dagger}{\begin{array}{c} y \mapsto c \rhd (\forall z.\, P(z) \rightarrow P(f(z,y))) \rightarrow \neg P(y) \\ \leftrightarrow \top \rightarrow \neg P(c) \end{array}}\ \text{cong}}{\begin{array}{c} \forall y.\, (\forall z.\, P(z) \rightarrow P(f(z,y))) \rightarrow \neg P(y) \\ \leftrightarrow \top \rightarrow \neg P(c) \end{array}}\ \text{inst}}{\top \rightarrow \neg P(c)}\ \dots$$

Here the rule "lift †" lifts the result of the first substitution into the context of the second substitution. To be valid the variables in the context cannot be free in the lifted term.