Anti-Unification Completeness Analysis in PVS

Mauricio Ayala-Rincón
Universidade de Brasília and Universidade Federal de Goiás
Exact Sciences Institute and Institute of Mathematics and Statistics
Brasília D.F. and Goiânia, Brazil

Thaynara Arielly de Lima
Universidade Federal de Goiás
Institute of Mathematics and Statistics
Goiânia, Brazil

Maria Júlia Dias Lima
Universidade de Brasília
Graduate Program in Informatics
Brasília D.F., Brazil

Temur Kutsia
Johannes Kepler Universität
Research Institute for
Symbolic Computation
Linz, Austria

Marcos Mercandeli-Rodrigues
Universidade de Brasília
Graduate Program in Mathematics
Brasília D.F., Brazil

In syntactic anti-unification, one is concerned with finding the commonalities between terms, while (uniformly) abstracting their differences. The original goal of anti-unification development in the seventies was to automate inductive reasoning. Recent applications of anti-unification techniques include efficiently transforming sequential code into parallel code, detecting code clones, and preventing software failures. Previous work addressed the elements required to verify, in the Prototype Verification System (PVS), termination and soundness of a functional algorithm based on inference rules for syntactic anti-unification. This paper dissects all aspects required to formally establish the completeness of the rule-based algorithm, highlighting the significant differences in the formalizations of anti-unification and unification.

Keywords: Equational Reasoning, Anti-unification, Generalization, Computational Verification, Interactive Theorem Proving, PVS.

1 Introduction

Motivation and Contextualization

Generalization is the scientific principle of drawing broad conclusions from specific observations, aiming to identify patterns or laws that hold beyond the original cases studied. In logic, this idea of generalization is formalized by a concept called *anti-unification*. It consists of finding the common structure and uniformly abstracting over differences when comparing two objects of a certain type, and it was first investigated and developed independently in the 1970s by Plotkin [33] and Reynolds [35].

Given terms s and t, one says that s is *more general* than t (or that t is *more specific* than s) and writes $s \preceq t$ iff there exists a substitution σ such that $s\sigma = t$. The relation $\preceq$ is known as *instantiation preorder* [16]. A *generalizer* for two terms s and t is a term r such that $r \preceq s$ and $r \preceq t$. The $\preceq$-minimal generalizers for s and t are called *least general generalizers* for s and t. Thus, the syntactic anti-unification problem can be formally stated as: For terms s and t, construct their least general generalizers. It is known that syntactic anti-unification is of type unitary [22, 29], i.e., any instance of the problem has a unique solution (up to renaming). For instance, the terms X, $f(X,Y)$, $f(g(X,Y),Z)$, and $f(g(X,Y),X)$ are generalizers for $f(g(c,d),c)$ and $f(g(g(u,v),v),g(u,v))$, but only $f(g(X,Y),X)$ is their least general generalizer.

The first procedural algorithms for solving the syntactic anti-unification problem were presented independently by Plotkin [33] and Reynolds [35] in the 1970s. Their algorithm essentially reads the pair of input expressions, identifies the smallest positions where conflicts occur, and uniformly assigns fresh

Submitted to:
LFMTP 2026

variables to those positions [35]. That allows the construction of the least general generalizer as being the join in a certain complete non-modular lattice whose order is $\succeq$ [35]. Huet presented an algorithm in 1976 in [28] by means of a recursive function λ that acts in the form $\lambda(s,t) = f(\lambda(s_1,t_1),\ldots,\lambda(s_m,t_m))$ if $s = f(s_1,\ldots,s_m)$ and $t = f(t_1,\ldots,t_m)$ and $\lambda(s,t) = \phi(s,t)$ otherwise, where ϕ is a bijection between pairs of terms and variables (ensuring that λ solves conflicts uniformly). A good exposition of that algorithm is presented by Lassez et al. in [29].

Rule-based procedures for anti-unification were presented by Pfenning [32] for the Calculus of Constructions, by Alpuente et al. [3] for order-sorted generalization, by the same authors for anti-unification modulo associativity and commutativity [4], and by Baumgartner et al. [20] for simply-typed lambda-terms in η-long β-normal forms, among others. Alpuente et al.'s proofs of completeness were non-constructive and relied on the anti-unification type; in the syntactic case, for instance, on the fact that syntactic anti-unification is unitary. A more preferred proof of completeness for rule-based algorithms would be constructive, as the one presented in this paper.

Real-world applications of anti-unification are, for instance, the identification of regularities in sequential code to transform it into efficient parallel code [18]; preventing failures and detecting errors in software [30]; repairing software bugs [17,24,39]; detecting code cloning and plagiarism [21,38]; maintaining mathematical or software libraries [27,34]; detecting similarities among chemical compounds to infer their carcinogenicity [5]. An open-source library of anti-unification algorithms (implemented in Java) for first- and second-order unranked terms, higher-order patterns, and nominal terms was presented in [19]. The paper [22] surveys the state-of-the-art in theory and applications of anti-unification.

Surprisingly, the current interest in the practical applications of anti-unification techniques has not yet been followed by the development of formal, mechanically verified certificates in proof assistants. That situation contrasts with the case for unification, where one aims to identify two expressions. For instance, successful work related to the formalization of unification and matching in interactive theorem provers are the ones developed in [23] for AC-matching (in Coq), in [8] for nominal C-matching through unification with protected variables (also in Coq), in [10] for nominal unification, in [13] for nominal C-unification, in [11] for nominal AC-matching, and in [14] for AC-unification (all in PVS). Advancing this line of research to provide formal certificates for proof assistants for anti-unification techniques and to strike a balance between theoretical development and practical application is of utmost importance. The only formalization of an anti-unification algorithm known to date is due to [9] for syntactic anti-unification. In that work, an algorithm for syntactic anti-unification is specified in PVS, and its termination and soundness are mechanically verified. Only the formal certificate for completeness remained to be constructed, and providing such a construction is the goal of the current work. Completing that work is of utmost importance, for it will allow the extraction of certified executable code of a sound and complete algorithm for syntactic anti-unification.

This paper formally studies the additional requirements that are essential for proving the completeness of algorithms based on the standard anti-unification inference rules (see Figure 1) that appear in [9]. The main motivation of this study is to understand in detail the greater complexity of proofs required for anti-unification compared to those for formalizations of unification algorithms (e.g., [6, 36, 37]). In [9], for formalizing soundness, important differences were highlighted, particularly in the cases of detection of *solved* and *syntactically equal* problems. In anti-unification, two crucial aspects concern the treatment of such problems. The former corresponds to equational questions between terms headed by different (function) symbols. In contrast, the latter corresponds to trivial (i.e., syntactically equal atomic) problems in which the terms are either the same constant or the same variable. Surprisingly, most of the formalization work (more than 90%, as quantified in [9]) for verifying the anti-unification algorithm was devoted to these two kinds of apparently straightforward cases. For obtaining a formal proof, the rigor-

ous analysis of these cases requires more elaboration than the one usually devoted to the completeness of anti-unification in pen-and-paper proofs (e.g., [1, 28, 33, 35]).

In unification, if a problem is what we call a *solved* problem above in the context of anti-unification, one has a *failure* because terms with different head function symbols can not be unified; and syntactically equal problems are trivially resolved using the identity substitution. In anti-unification, the situation is completely different. A solved equation does not mean a failure, but the detection of a difference in the structure of the terms being compared. That difference is recorded in the computed substitution of the current configuration. In fact, unlike unification, anti-unification never fails. To solve the general case of anti-unification, one should proceed by further checking for other occurrences of the same difference, thereby detecting all possible regularities.

In anti-unification, the decomposition of the problem is guided by mutually distinct labels associated with subproblems of the input problem and by a substitution that records the problem's structure. As soon as a solved subproblem is detected, it is stored separately. The essential elements for addressing solved and syntactic equations for formalizing soundness in [9] include preservation properties for the sets of labels of the unsolved and solved subproblems, and for the variables in the domain and the image of the computed substitution.

Main Contribution

- For the formalization of completeness, we discuss why the preservation properties developed for soundness together with the adjustments of generalizer notions (Subsection 3.2) are enough to address the inference rules for decomposition.
- Although the required notions for dealing with the cases of solved and syntactic rules in the soundness proof were much more elaborate than those required for unification, these notions were not enough for formalizing anti-unification completeness. We introduce the required additional elements in notions such as arbitrary generalizers so that a series of invariance and preservation properties are guaranteed; among them, properties that assure that the steps of the algorithm do not change anti-unification problems, and that the partially computed solutions can always be refined into generalizers that are less general than any arbitrary generalizer (Subsection 3.1). Based on these properties, a restricted notion of generalizer is introduced (Subsection 3.2).
- Finally, using the restricted notion of generalizer, we formalize (Subsection 3.3) the completeness theorem (stated as Theorem 20) and obtain as a corollary the main result that states the completeness of the algorithm regarding arbitrary generalizers (Corollary 21).

Organization

Section 2 presents the required background following standard nomenclature on anti-unification, which is the one used in the formalization [9]. Section 3 is the kernel of the paper. Subsection 3.1 presents the additional notions required to record preservation and invariance properties that compile the history of the algorithm's computation. Subsection 3.2 presents the adjustments required to address completeness; then, Subsection 3.3 discusses how solved and syntactic subproblems are treated. Subsection 3.4 discusses the analysis of decomposition rules. Finally, Section 4 concludes and briefly discusses future work. Several points in the paper include links to the specification (PDF). An extended version of this paper presents quantitative information on a direct approach for dealing with the decomposition inference rules.

2 Background

The background related to terms, substitutions, and configurations is present in [9] and will be restated here to provide the proper vocabulary. The set of *terms* is generated by the standard nominal grammar $s,t ::= c \mid X \mid () \mid (s,t) \mid f\,s$ adapted from [12], where c stands for *constants*, X stands for *variables*, s and t stand for terms, (s,t) stands for *pairs* of terms, $()$ stands for the *unit*,[1] f stands for *function symbols*, and $f\,s$ stands for *functional applications*. It is specified in PVS by means of the abstract data type (ADT) `first_order_term`, requiring types for constants, function symbols, and variables as parameters. A *basic substitution* is a binding $(X \mapsto t)$, where X is a variable and t is a term. A *substitution* is a finite list of basic substitutions (the *identity substitution* is the empty list ι). Those are specified in PVS in the theory `first_order_substitution` using pairs and lists of these pairs. The action of a substitution on a term is standard. The domain of a substitution σ is denoted by $\texttt{dom}(\sigma)$. The set of variables in its range is denoted by $\texttt{rvars}(\sigma)$. The substitutions recognized by the predicate `nice?` defined in the previous theory are the most important and widely employed in the specification.

Example 1 (Niceness)**.** *A substitution* $(X_1 \mapsto t_1)\cdots(X_m \mapsto t_m)$ *is* nice *iff* $\texttt{vars}(t_i) \cap \{X_i,\ldots,X_m\} = \emptyset$ *and* $X_i \neq X_j$ *for all* $i,j = 1,\ldots,m$ *with* $i \neq j$*. For instance,* $(X \mapsto (Y,Z))\,(W \mapsto fX)$ *is nice, while* $(X \mapsto fX)$ *is not.*

An anti-unification problem is encoded by an *anti-unification triple* (*AUT*), which is an expression of the form $s \triangleq_X t$, where s and t are terms, called its *left-* and *right-hand sides*, respectively, and X is a fresh variable for s and t called its *label*. That structure is specified in PVS by means of a record type `AUT`, where record accessors for label and left- and right-hand sides are present. The expression eq_X will be used to denote an AUT whose label is the variable X. Its left- and right-hand sides will be denoted by $\texttt{lhs}(eq_X)$ and $\texttt{rhs}(eq_X)$, respectively. To solve an anti-unification problem, one must compare two terms and decompose their common structure. That motivates an *AUT classification* that allows no superposition: An AUT eq_X is *decomposable* iff $\texttt{lhs}(eq_X)$ and $\texttt{rhs}(eq_X)$ are either both pairs or both functional applications headed by the same function symbol; it is *trivial* iff $\texttt{lhs}(eq_X) = \texttt{rhs}(eq_X)$ is the unit, a constant or a variable; and it is *solved* in any other case. That AUT classification guides the design of the inference rules presented in Figure 1 and, in PVS, it is specified by means of unary predicates `match_DecF?`, `match_DecP?`, `match_Synt?`, and `match_Sol?` that checks whether or not an AUT matches the application of a rule.

During the decomposition, the labels are the most general solutions to anti-unification (sub)problems. Since those subproblems occur in different positions in the common structure of terms, it makes sense to label new subproblems with fresh labels to distinguish them. The decomposition produces a finite number of AUTs that are described as a finite collection. In PVS, that collection is specified as a list of AUTs. To a finite collection of AUTs A, one associates two sets whose members are variables that play completely distinct roles in the anti-unification algorithm. The first one is its *set of labels* $\texttt{lbls}(A)$, which collects all the labels of the members of A. The second one is its *set of variables* $\texttt{vars}(A)$, which collects all the variables that occur in the left- and right-hand sides of its members. To refer to a collection of subproblems obtained from a problem consistently, it is necessary to define a *valid set of AUTs* as being a finite set A of AUTs such that $\texttt{lbls}(A) = |A|$ and $\texttt{vars}(A) \cap \texttt{lbls}(A) = \emptyset$. In PVS, valid sets of AUTs were specified by means of valid lists of AUTs, which are recognized by the unary predicate `valid_AUTs?`.

[1] The unit is useful for term flattening and for efficiently encoding finitary and variadic functions in the presence of pairs. For instance, considering addition $+\colon \mathbb{N} \to \mathbb{N}$, one can define $\sum\colon \texttt{list}(\mathbb{N}) \to \mathbb{N}$ recursively by $\sum() := 0$ and $\sum(m,L) := m + \sum L$.

As decomposition proceeds, it may be necessary to address an anti-unification problem that was previously handled but at a distinct position in the common structure. Such problems are essentially the same but encoded by AUTs that differ only in their labels. Those are called *repeated AUTs*. That can be extended to sets of AUTs: An AUT eq_X is *repeated in* a set A of AUTs iff there exists an AUT eq_Y in A such that eq_X and eq_Y are repeated. The binary predicates `repeated_AUT?` and `AUT_repeated_in?` are specified to recognize such repetitions.

$$\text{Decompose-Function (DecF)}\ \frac{\langle f\,s \triangleq_X f\,t, U \mid S \mid \sigma\rangle}{\langle s \triangleq_Y t, U \mid S \mid \sigma\,(X \mapsto f\,Y)\rangle}$$

$$\text{Decompose-Pair (DecP)}\ \frac{\langle (s,u) \triangleq_X (t,v), U \mid S \mid \sigma\rangle}{\langle s \triangleq_Y t, u \triangleq_Z v, U \mid S \mid \sigma\,(X \mapsto (Y,Z))\rangle}$$

$$\text{Solve-Repeated (SolR)}\ \frac{\langle s \triangleq_X t, U \mid S \mid \sigma\rangle}{\langle U \mid S \mid \sigma\,(X \mapsto X')\rangle}\ \text{if } s \triangleq_X t \text{ is solved and } s \triangleq_{X'} t \in S$$

$$\text{Solve-Non-Repeated (SolNR)}\ \frac{\langle s \triangleq_X t, U \mid S \mid \sigma\rangle}{\langle U \mid s \triangleq_X t, S \mid \sigma\rangle}\ \text{if } s \triangleq_X t \text{ is solved and not repeated in } S$$

$$\text{Syntactic (Synt)}\ \frac{\langle s \triangleq_X s, U \mid S \mid \sigma\rangle}{\langle U \mid S \mid \sigma\,(X \mapsto s)\rangle}\ \text{if } s \triangleq_X s \text{ is trivial}$$

Figure 1: Standard anti-unification inference rules.

During the decomposition of the common term structure, one must understand the current state and the states visited to reach it. In that way, the problems already computed and those remaining to be computed will be identified. That is encoded by means of a *valid configuration*, which is an expression of the form $\langle \mathcal{C}_{\mathtt{Uns}} \mid \mathcal{C}_{\mathtt{Sol}} \mid \mathcal{C}_{\mathtt{Sub}}\rangle$, where $\mathcal{C}_{\mathtt{Sub}}$ is a substitution, $\mathcal{C}_{\mathtt{Uns}}$ and $\mathcal{C}_{\mathtt{Sol}}$ are valid sets of AUTs called its *computed substitution* and *unsolved* and *solved* parts, respectively, and each of those three components satisfy the following constraints:

1. The sets $\mathcal{C}_{\mathtt{Uns}}$ and $\mathcal{C}_{\mathtt{Sol}}$ are disjoint and $\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}$ is a valid set of AUTs;

2. The set $\mathcal{C}_{\mathtt{Sol}}$ contains only solved AUTs that are not repeated in $\mathcal{C}_{\mathtt{Sol}}$;

3. The sets $\mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}})$, $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}})$, and $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$ are pairwise disjoint. The sets $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$ and $\mathtt{rvars}(\mathcal{C}_{\mathtt{Sub}})$ are also disjoint.

The inference rules in Figure 1 induce a reduction relation over valid configurations, which is denoted as $\Rightarrow$ using standard rewriting notation [15]; thus, $\Rightarrow^*$, $\Rightarrow^+$, and $\Rightarrow^n$ denote the reflexive transitive closure, transitive closure, and the n-reduction steps derivation relations obtained from $\Rightarrow$, respectively. In specific derivations, superscripts with the names of the applied rules are added to the relation symbol $\Rightarrow$. For the example in the introduction, we have the following derivation:

$$\langle f\,(g\,(c,d),c) \triangleq_X f\,(g\,(g\,(u,v),v),g\,(u,v)) \mid \emptyset \mid \iota\rangle \xrightarrow{\texttt{DecF},\texttt{DecP}}$$
$$\langle g\,(c,d) \triangleq_Y g\,(g\,(u,v),v),c \triangleq_Z g\,(u,v) \mid \emptyset \mid X \mapsto f\,(Y,Z)\rangle \xrightarrow{\texttt{DecF},\texttt{DecP}}$$
$$\langle c \triangleq_{Y_1} g\,(u,v),d \triangleq_{Y_2} v,c \triangleq_Z g\,(u,v) \mid \emptyset \mid X \mapsto f\,(g\,(Y_1,Y_2),Z)\rangle \xrightarrow{\texttt{SolNR},\texttt{SolNR}}$$
$$\langle c \triangleq_Z g\,(u,v) \mid c \triangleq_{Y_1} g\,(u,v),d \triangleq_{Y_2} v \mid X \mapsto f\,(g\,(Y_1,Y_2),Z)\rangle \xrightarrow{\texttt{SolR}}$$
$$\langle \emptyset \mid c \triangleq_{Y_1} g\,(u,v),d \triangleq_{Y_2} v \mid X \mapsto f\,(g\,(Y_1,Y_2),Y_1)\rangle$$

A configuration is specified in PVS by means of `Configuration`, which is a record type with record accessors for unsolved, solved, and computed substitution (always a nice one). Among configurations, the valid ones are recognized by the unary predicate `validConfiguration?`. The intuition for each component of a valid configuration is as follows. Its unsolved part encodes the problems to be computed, its solved part encodes the problems where the common term structure differs and which were detected for the first time, and its computed substitution encodes the common term structure already decomposed. Thus, a valid configuration encodes the entire history of the decomposition up to that point.

The decomposition of the term structure is carried out by the inference rules presented in Figure 1. The algorithm `Antiunify` is specified in [9] as a recursive unary function that maps valid configurations into valid configurations, and it consists of the exhaustive application of the rules to a valid input configuration. Since the unsolved part of a valid configuration is specified by a list of AUTs, the rules are always applied to the first member of that list. The size of the unsolved part of a configuration, which is the sum of the `sizes` of its members, was the chosen metric for ensuring *termination* in [9].

The rules presented in Figure 1 are specified in [9] by means of unary functions `DecF`, `DecP`, `Synt`, and `Solve` that require an input valid configuration that matches the application of the rule and returns an output configuration according to their (dependent) type. That type encodes, among other properties, that the output configuration is a valid configuration with a *smaller size* than the input configuration. PVS generated a *type correctness condition* (TCC) for each function, and each one of those TCCs required a manual proof. The advantage of this approach is that PVS generates a termination TCC for the specification of `Antiunify`, and the prover can automatically prove it using the information encoded in the (dependent) types of the input-output configurations of the inference rules. Thus, an exhaustive application of the rules terminates in a configuration with an empty unsolved part. Those are called *normal configurations* (or *final configurations*) and are recognized in PVS by means of the predicate `normal_configuration?`.

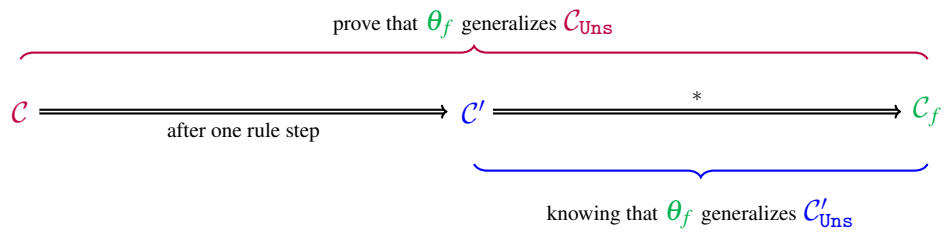

Figure 2: Inductive step in the soundness theorem presented in [9].

The background was enough to prove `antiunif_is_sound` in [9]. The argument was constructed by induction on the size of the unsolved part of a valid configuration together with a case analysis concerning the employed rule in the inductive step, which is presented in Figure 2. The effort required

was measured in [9] by the number of proof commands (for the proof and also its dependencies) and the amounts are 2.09% for DecF, 4.56% for DecP, 61.70% for Synt, 29.60% for SolR, and 2.05% for SolNR. Surprisingly, the rules Synt and SolR together make up 91.10% of the entire effort and algorithmically, going from $\mathcal{C}$ to $\mathcal{C}'$, they only add to $\mathcal{C}_{\mathtt{Sub}}$ a new basic substitution $(X \mapsto a)$, where X is a label in $\mathcal{C}_{\mathtt{Uns}}$ and a is an atomic term. While that transformation seems trivial in a pen-and-paper proof, a considerable amount of work is required to analyze its effects in a mechanized and formally verified proof for the case where a is a variable. In that case, the branches for those two rules relied on dependencies (lemmas) that stated the invariance and preservation properties concerning the action of θ_f on the variable a. These properties explained why all that effort was required.

3 Dissection Towards a Mechanized Completeness Proof

Using the formalization elements developed for the proof of soundness in [9], one can try to address the completeness proofs for the decomposition rules (DecF and DecP). Indeed, for an inductive proof like the one presented in Figure 2 using a notion `r_generalizer` of arbitrary generalizer that restrict its domain and range avoiding occurrences of the variables used by the algorithm, the completeness case analysis of these two rules will amount more than 2500 and 3500 PVS proof commands for the DecF and DecP rules, respectively (see the extended version of this paper). Such an approach is sufficient to guarantee the completeness of the algorithm for the subclass of anti-unification problems that do not require applications of the rules SolR and Synt, but not for arbitrary problems. An important class of such problems comprises those that do not repeat variables or constants. For those problems, the decomposition rules will only lead to solved AUTs that cannot appear more than once. Since no variable nor constant appears repeatedly, no application of the rules SolR or Synt is possible.

Surprisingly, following the same inductive schema with the same notion of generalizer did not work for the SolR, SolNR, and Synt rules, as this section explains. The problem arises in the inductive step: given an arbitrary generalizer γ for the problem encoded by a configuration $\mathcal{C}$, to apply the inductive hypothesis, one needs to build an associated generalizer γ' for the configuration $\mathcal{C}'$; and using the hypothesis that it is more general than the computed solution given by θ_f, in the final configuration $\mathcal{C}_f$, prove that γ is also more general than θ_f (see Figure 3).

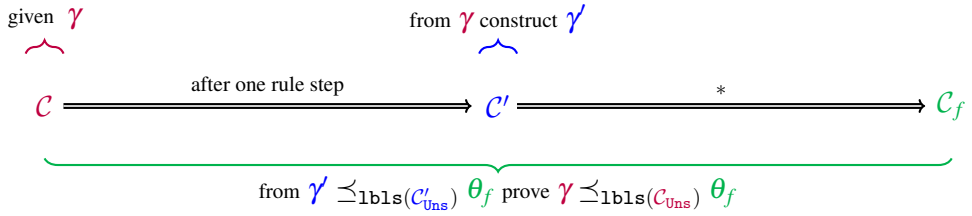

Figure 3: Inductive step in the experiments towards a completeness proof.

Initially, Subsection 3.1 explains new notions that are required to compile configuration preservation properties for the analysis of the Synt, SolR, and SolNR rules. These notions are used to record invariance properties throughout the history of computations. Afterward, Subsection 3.2 presents crucial adjustments required to address completeness for those rules; finally, Subsection 3.3 discusses how these adjusted notions are used in the formalization of the inductive proof. Although the branches of the decomposition rules can be derived from the elements developed in [9], for insertion into the case analysis of this new inductive schema, they require straightforward adaptations, as explained in Subsection 3.4.

3.1 New Notions Required for the Completeness Analysis

An *initial configuration* is a valid configuration with an empty solved part and identity computed substitution, and it is recognized in PVS by means of the unary predicate `initial_configuration?`. In PVS, a single application of any of the rules is recognized by the binary predicate `Antiunify?` specified from the binary predicates `DecF?`, `DecP?`, `Synt?`, and `Solve?`. These predicates check whether a configuration is derived from another configuration by applying the respective inference rule. The binary predicate `RTC(Antiunify?)`, constructed using the reflexive-transitive closure of abstract reduction relations, `RTC(R)`, from the NASAlib theory of rewriting (see, e.g., [25, 31]). For brevity, in this section, we will also use standard rewriting notation for the specified predicate `Antiunify?`; thus, $\Rightarrow$ and $\Rightarrow^*$ will denote `Antiunify?` and `RTC(Antiunify?)`, respectively.

All information pertaining to the derivation of a valid configuration from an initial configuration is stored in the former. All labels that were used during its derivation encode positions in the common term structure where subproblems were detected. Labels are transformed at each step taken: New labels are generated, labels are moved from the unsolved part to the solved part, to the domain of the computed substitution, or to its range. That motivates the following definition:

Definition 2 (Labels of a Valid Configuration)**.** *The set of labels of a valid configuration $\mathcal{C}$ is defined as* $\mathtt{lbls}(\mathcal{C}) := \mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}}) \cup \mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}}) \cup \mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$.

The information encoded by labels is essential if one aims for completeness, as labels encode positions. One important preservation property is stated in the next lemma and proved by induction on the number of steps in a derivation. Intuitively, all information encoded by the labels of a valid configuration is inherited by every valid configuration derived from it. In particular, if $\mathcal{C} \Rightarrow \mathcal{C}'$ by means of `SolR`, `SolNR`, or `Synt`, then $\mathtt{lbls}(\mathcal{C}') = \mathtt{lbls}(\mathcal{C})$.

Lemma 3 (Preservation of Labels)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then* $\mathtt{lbls}(\mathcal{C}) \subseteq \mathtt{lbls}(\mathcal{C}')$.

Another important class of variables in a valid configuration derived from an initial configuration are those which appear on the left- and right-hand sides of the AUTs of the latter. In the decomposition of the common term structure, one can detect a trivial AUT whose both sides are the same variable, and the rule `Synt` transforms that AUT accordingly, storing the following information at the computed substitution: The label that encodes the position where the problem was encountered must be mapped to that variable that occurs at that position. That is the only possible transformation regarding those types of variables, and it motivates the following definition:

Definition 4 (Protected Variables of a Valid Configuration)**.** *The set of protected variables of a valid configuration $\mathcal{C}$ is defined as* $\mathtt{prtd}(\mathcal{C}) := \mathtt{vars}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}) \cup (\mathtt{rvars}(\mathcal{C}_{\mathtt{Sub}}) \setminus \mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}))$.

A variable that occurs in the left- or right-hand side of an AUT of $\mathcal{C}_0$ can occur in a subsequent derived configuration $\mathcal{C}$ in $\mathtt{vars}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}) \cup \mathtt{rvars}(\mathcal{C}_{\mathtt{Sub}})$. How they appear in those sets is understood by the action of the inference rules employed in the derivation. One important invariance property is presented in the following lemma, which is proved by induction on the number of steps of a derivation. A crucial step in that proof is understanding how the set of variables of the computed substitution of a valid configuration changes after applications of the standard inference rules.

Lemma 5 (Invariance of Protected Variables)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then* $\mathtt{prtd}(\mathcal{C}) = \mathtt{prtd}(\mathcal{C}')$.

An important consequence of the previous lemma is the following: the set of protected variables of any configuration derived from an initial configuration is exactly the set of variables of the unsolved part of the latter. Also, it is important to notice that $\mathtt{lbls}(\mathcal{C}) \cap \mathtt{prtd}(\mathcal{C}) = \emptyset$ for any valid configuration $\mathcal{C}$. Another type of invariant present throughout the computation is the problem being solved. To discuss that, one requires the proper vocabulary:

Definition 6 (Lateral Substitutions)**.**

1. *The* left *(resp.* right*)* substitution associated with *a valid set A of AUTs is the substitution ρ_L^A (resp. ρ_R^A) with* $\mathtt{dom}(\rho_L^A) = \mathtt{lbls}(A)$ *(resp.* $\mathtt{dom}(\rho_R^A) = \mathtt{lbls}(A)$*) such that* $X\rho_L^A = \mathtt{lhs}(eq_X)$ *(resp.* $X\rho_R^A = \mathtt{rhs}(eq_X)$*) for every X in* $\mathtt{lbls}(A)$*;*
2. *The* left *and* right substitutions associated with *a valid configuration $\mathcal{C}$ are:*

$$\rho_L^{\mathcal{C}} := \rho_L^{\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}} \qquad \text{and} \qquad \rho_R^{\mathcal{C}} := \rho_R^{\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}}.$$

By definition, one can see that $\mathtt{rvars}(\rho_L^{\mathcal{C}}) \cup \mathtt{rvars}(\rho_R^{\mathcal{C}}) \subseteq \mathtt{prtd}(\mathcal{C})$, thus the lateral substitutions associated with $\mathcal{C}$ are idempotent. Also, their restriction to any part of $\mathcal{C}$ is the corresponding lateral substitution associated with that part. Lateral substitutions play an important role in reconstructing AUTs that encode anti-unification problems. The invariance result hinted at before can be stated as the next lemma. It is proved by induction on the length of the derivation, paying attention to how each inference rule modifies the lateral and computed substitutions associated with a configuration, as well as to the role of the label of the AUT transformed by the inference rule.

Lemma 7 (Problem Invariance)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then $X\mathcal{C}'_{\mathtt{Sub}}\rho_L^{\mathcal{C}'} = X\mathcal{C}_{\mathtt{Sub}}\rho_L^{\mathcal{C}}$ and $X\mathcal{C}'_{\mathtt{Sub}}\rho_R^{\mathcal{C}'} = X\mathcal{C}_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$ for every X in* $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$*.*

Since all information is preserved in a derivation, one can think intuitively about Lemma 7 by employing a jigsaw puzzle: One breaks the entire picture (an AUT of the unsolved part of a valid configuration) into small pieces (AUTs obtained from that AUT by applications of inference rules to configurations) in a way that, if all of them are put together (the compositions of the computed and lateral substitutions), then one obtains the original picture again. That is the *problem invariance*: No matter how one breaks a problem into subproblems, the problem remains the same throughout the computation.

All the preservation properties presented so far can be used to completely describe the history of computation in the derivation of a valid configuration from an initial one. For instance, an argument by induction is sufficient for proving the following lemma:

Lemma 8. *[History of the Solved Part] Let $\mathcal{C}_0$ be an initial configuration and $\mathcal{C}$ be a valid configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}$. For every AUT eq_X in $\mathcal{C}_{\mathtt{Sol}}$, there exist two valid configurations $\mathcal{C}'$ and $\mathcal{C}''$ such that eq_X belongs to $\mathcal{C}'_{\mathtt{Uns}} \cap \mathcal{C}''_{\mathtt{Sol}} \setminus \mathcal{C}'_{\mathtt{Sol}}$ and $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow \mathcal{C}'' \Rightarrow^* \mathcal{C}$, where the step $\mathcal{C}' \Rightarrow \mathcal{C}''$ uses* SolNR*.*

Another important description of the history of computation during a derivation concerns the domain of the computed substitution. That history is presented in the next lemma, which can be proved by induction using Lemma 7:

Lemma 9. *[History of the Computed Substitution] Let $\mathcal{C}_0$ be an initial configuration and $\mathcal{C}$ be a configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}$. For every X in* $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$*, exactly one of the following holds:*

1. *$X\mathcal{C}_{\mathtt{Sub}}$ is a variable and exactly one of the following holds:*
   - **1.1.** *$X\mathcal{C}_{\mathtt{Sub}}$ is the label of the AUT*
   $$X\mathcal{C}_{\mathtt{Sub}}\rho_L^{\mathcal{C}} \triangleq_{X\mathcal{C}_{\mathtt{Sub}}} X\mathcal{C}_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$$
   that belongs to $\mathcal{C}_{\mathtt{Sol}}$;
   - **1.2.** *$X\mathcal{C}_{\mathtt{Sub}}$ is a member of* $\mathtt{prtd}(\mathcal{C})$ *and the AUT*
   $$X\mathcal{C}_{\mathtt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X X\mathcal{C}_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$$
   is the trivial AUT
   $$X\mathcal{C}_{\mathtt{Sub}} \triangleq_X X\mathcal{C}_{\mathtt{Sub}}$$
   that belongs to $\mathcal{C}'_{\mathtt{Uns}}$ for some valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow^* \mathcal{C}$;

2. *$XC_{\mathtt{Sub}}$ is either the unit or a constant and the AUT*

$$XC_{\mathtt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X XC_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$$

is the trivial AUT

$$XC_{\mathtt{Sub}} \triangleq_X XC_{\mathtt{Sub}}$$

that belongs to $\mathcal{C}'_{\mathtt{Uns}}$ for some valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow^* \mathcal{C}$;

3. *$XC_{\mathtt{Sub}}$ is either a functional application or a pair, and the AUT*

$$XC_{\mathtt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X XC_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$$

is decomposable and belongs to $\mathcal{C}'_{\mathtt{Uns}}$ for some valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow^* \mathcal{C}$.

3.2 Adjustment of Generalizer Notions

At this point, one has sufficient grounds to provide the building blocks of a definition that encodes what a generalizer for a valid configuration is. Lemma 7 guarantees the consistency of the following definition:

Definition 10 (Extended Set of AUTs)**.** *The* extended set of AUTs *of a valid configuration $\mathcal{C}$ is defined as:*

$$\mathcal{C}_{\mathtt{Ext}} \coloneqq \left\{ XC_{\mathtt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X XC_{\mathtt{Sub}}\rho_R^{\mathcal{C}} \,\middle|\, X \in \mathtt{dom}(\mathcal{C}_{\mathtt{Sub}}) \right\}.$$

A close inspection of the previous definition and the definition of lateral substitutions ensures that the set just introduced is a valid set of AUTs. Suppose $\mathcal{C} \Rightarrow \mathcal{C}'$. If `SolNR` is the employed rule, then $\mathcal{C}'_{\mathtt{Ext}} = \mathcal{C}_{\mathtt{Ext}}$. If any other rule is the employed one, then $\mathcal{C}'_{\mathtt{Ext}} = \mathcal{C}_{\mathtt{Ext}} \cup \{ eq_X \}$, where eq_X is the AUT in $\mathcal{C}_{\mathtt{Uns}}$ that is transformed by the rule. Thus, an inductive argument together with Lemma 7 provides the following natural result:

Lemma 11 (Preservation of Extended Set)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then $\mathcal{C}_{\mathtt{Ext}} \subseteq \mathcal{C}'_{\mathtt{Ext}}$.*

Notice that in the set of labels of a valid configuration, the labels of the unsolved and solved parts are obtained directly from the AUTs that are already present in the configuration. The labels in the domain of the substitution refer to AUTs that were already transformed, and the extended set of AUTs is a very natural way to recover them. Thus, one can define:

Definition 12 (Total Set of AUTs)**.** *The* total set of AUTs *of a valid configuration $\mathcal{C}$ is defined by*

$$\mathcal{C}_{\mathtt{Tot}} \coloneqq \mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}} \cup \mathcal{C}_{\mathtt{Ext}}.$$

A close inspection of the previous definition ensures that the set just introduced is a valid set of AUTs. Moreover, the identity $\mathtt{lbls}(\mathcal{C}_{\mathtt{Tot}}) = \mathtt{lbls}(\mathcal{C})$ holds. Suppose $\mathcal{C} \Rightarrow \mathcal{C}'$. If `SolNR`, `SolR`, or `Synt` is the employed rule, then $\mathcal{C}'_{\mathtt{Tot}} = \mathcal{C}_{\mathtt{Tot}}$. If any decomposition rule is the employed one, then $\mathcal{C}'_{\mathtt{Ext}} = \mathcal{C}_{\mathtt{Ext}} \cup A$, where A is the set of new AUTs introduced by the employed rule. Thus, an inductive argument provides the following natural result:

Lemma 13 (Preservation of Total Set)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then $\mathcal{C}_{\mathtt{Tot}} \subseteq \mathcal{C}'_{\mathtt{Tot}}$.*

Given two terms s and t, a term g is a generalizer for s and t iff there exist substitutions σ and τ such that $g\sigma = s$ and $g\tau = t$. Thus, a generalizer for an AUT is a generalizer for both its sides. In the case of a valid configuration, one requires a notion of a generalizer that *uniformly generalizes* all the AUTs associated with that configuration. Thus, a generalizer for a valid configuration must be a substitution, and any comparison between generalizers must rely on the instantiation preorder restricted to the set of labels of the configuration. A natural definition is the following one:

Definition 14 (Total Generalizer for a Valid Configuration)**.**

1. *Let A be a valid set of AUTs. A substitution γ is a* total generalizer for *A iff there exist* left *and* right cohesion substitutions τ_L^A *and* τ_R^A*, respectively, such that*

$$X\gamma\tau_L^A = \mathtt{lhs}(eq_X) \qquad \text{and} \qquad X\gamma\tau_R^A = \mathtt{rhs}(eq_X)$$

   for every eq_X in A;

2. *A substitution γ is a* total generalizer for *a valid configuration $\mathcal{C}$ iff γ is a total generalizer for $\mathcal{C}_{\mathtt{Tot}}$. Its left and right cohesion substitutions are denoted by $\tau_L^{\mathcal{C}}$ and $\tau_R^{\mathcal{C}}$, respectively.*

That is the notion of a generalizer for a valid configuration required for proving completeness. The identity ι is a total generalizer for every valid configuration $\mathcal{C}$ and the lateral substitutions of $\mathcal{C}_{\mathtt{Tot}}$ are its cohesion substitutions. That mirrors the fact that a variable generalizes any term. As a bonus, the new notions presented so far are already enough to prove soundness in a way that is different from the ones presented in [9] or in [4]. The new proof of soundness relies on the preservation properties stated in Lemmas 7 and 13, and it is done by induction on the number of steps of the derivation.

The pen-and-paper proof of completeness that is going to be presented here is constructive and completely different from the one presented in [4], because no results on confluence or the type of syntactic anti-unification will be assumed. On the contrary, that type will be a consequence of termination, soundness, and completeness. Before presenting the proof, certain properties of generalizers for valid configurations must be proved. The first one is a natural definitional corollary:

Lemma 15 (Total Generalizer Coherence)**.** *If A is a valid set of AUTs, γ is a total generalizer for it, and eq_X is a member of A, then $X\gamma$ is a total generalizer for eq_X. More precisely:*

1. *If eq_X is a decomposable AUT, then:*
   - **1.1.** *If $\mathtt{lhs}(eq_X)$ is a functional application, then $X\gamma$ is either a variable or a functional application starting with the same root symbol of $\mathtt{lhs}(eq_X)$;*
   - **1.2.** *If $\mathtt{lhs}(eq_X)$ is a pair, then $X\gamma$ is either a variable or a pair;*
2. *If eq_X is a trivial AUT, then $X\gamma$ is the unit, a constant or a variable;*
3. *If eq_X is a solved AUT, then $X\gamma$ is a variable;*

A particularly important preservation result concerning total generalizers for valid configurations is the following one. Despite being a natural definitional corollary, it essentially informs that a total generalizer cannot associate the same solution to problems encoded by AUTs of distinct classes:

Lemma 16 (Preservation of Classification)**.** *Let A be a valid set of AUTs and γ be an arbitrary total generalizer for A. For all eq_X and eq_Y in A, if $X\gamma = Y\gamma$, then eq_X and eq_Y are repeated.*

The key step in the proof of the previous lemma is the existence of cohesion substitutions associated with the total generalizer γ. Under the hypothesis $X\gamma = Y\gamma$, they force the left and right-hand sides of the AUTs eq_X and eq_Y to be respectively the same terms. A particularly important consequence of the previous lemma is the following lemma, which deals with syntactic and solved AUTs:

Lemma 17 (Separation)**.** *Let A be a valid set of AUTs, γ be an arbitrary total generalizer for A, and eq_X and eq_Y be arbitrary members of A.*

1. *If eq_X and eq_Y are trivial AUTs and $\mathtt{lhs}(eq_X) \neq \mathtt{lhs}(eq_Y)$, then $X\gamma \neq Y\gamma$;*
2. *If all members of A are not repeated in A and $X \neq Y$, then $X\gamma \neq Y\gamma$.*

An important application of Lemmas 15 and 17 is the following. If $\mathcal{C}$ is a valid configuration, then the members of $\mathcal{C}_{\mathtt{Sol}}$ are not repeated in it. Thus, a total generalizer for $\mathcal{C}$ maps distinct labels in $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}})$ to distinct variables. In other words, a total generalizer for a valid configuration is able to separate non-repeated solved (resp. non-repeated trivial) AUTs that appear in a derivation ending in that same configuration. This property will play an important role in a step of the proof of completeness. Another important property of total generalizers for valid configurations is presented in the following lemma:

Lemma 18 (Choice of Total Generalizer)**.** *Let $\mathcal{C}$ be a valid configuration and $\mathbb{A}$ be a set of variables. For every total generalizer γ for $\mathcal{C}$ there exists a total generalizer γ' for $\mathcal{C}$ such that $\mathtt{rvars}(\gamma') \cap \mathbb{A} = \emptyset$ and $\gamma' \simeq_{\mathtt{lbls}(\mathcal{C})} \gamma$.*

To prove the previous lemma, one needs to enumerate all the variables in $\mathtt{rvars}(\gamma) \cap \mathbb{A}$ (if there are any) and rename all of them into distinct fresh variables using a renaming α. New cohesion substitutions are constructed using α and the old ones. Lemma 18 serves as inspiration for providing the following definition:

Definition 19 (Restricted Total Generalizer)**.** *Let $\mathcal{C}$ be a valid configuration and $\mathcal{C}_f$ be a final configuration such that $\mathcal{C} \Rightarrow^* \mathcal{C}_f$. A total generalizer γ for $\mathcal{C}$ is a* restricted total generalizer for $\mathcal{C}$ relative to $\mathcal{C}_f$ *iff $\mathtt{rvars}(\gamma) \cap (\mathtt{lbls}(\mathcal{C}_f) \cup \mathtt{prtd}(\mathcal{C}_f)) = \emptyset$ and $\gamma\gamma = \gamma$.*

The specification of that type of total generalizer is crucial in the proof of completeness. Lemma 18 shows that if a valid configuration admits a total generalizer, then it admits a restricted total generalizer that is equivalent to the original one on the labels of that configuration. That means that one can restrict their attention to that class of more well-behaved total generalizers for valid configurations. Also, since equivalence holds, completeness is not lost by such a restriction. That is the path followed in the PVS specification. We are ready to state completeness. All the preservation properties, invariants, and (restricted) total generalizer properties presented so far will naturally guide the construction of the proof. It will be dismembered into parts. `Synt`, `SolNR`, and `SolR` rules are presented in Subsection 3.3, while `DecF` and `DecP` are presented in Subsection 3.4.

Theorem 20 (Strong Completeness for Restricted Total Generalizers)**.** *Let $\mathcal{C}_0$ be an initial configuration, $\mathcal{C}_f$ be a final configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}_f$, and θ_f be the computed substitution of $\mathcal{C}_f$. For every valid configuration $\mathcal{C}$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow^* \mathcal{C}_f$ and every restricted total generalizer γ for $\mathcal{C}$ relative to $\mathcal{C}_f$ it is the case that $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$.*

The proof of Theorem 20 is done by induction on the number of steps taken in $\mathcal{C} \Rightarrow^* \mathcal{C}_f$. In the base case, one has $\mathcal{C} = \mathcal{C}_f$. Let S_f be the solved part of $\mathcal{C}_f$. By the definition of total generalizer for $\mathcal{C}_f$, one can always assume that $\mathtt{dom}(\gamma) \cap \mathtt{prtd}(\mathcal{C}_f) = \emptyset$. Since $\mathtt{rvars}(\theta_f) \subseteq \mathtt{lbls}(S_f) \cup \mathtt{prtd}(\mathcal{C}_f)$, one employs the definition of restricted total generalizer and Lemmas 8, 9, 15, and 17 to conclude that $\gamma \preceq_{\mathtt{lbls}(\mathcal{C}_f)} \theta_f$.

For the inductive step, one assumes the result for derivations of length m and supposes that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow^{m+1} \mathcal{C}_f$. Thus, there exists a valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow \mathcal{C}' \Rightarrow^m \mathcal{C}_f$. The analysis now proceeds by considering the rule employed in the step $\mathcal{C} \Rightarrow \mathcal{C}'$. One must start with an arbitrary restricted total generalizer γ for $\mathcal{C}$ relative to $\mathcal{C}_f$ and use it to construct a restricted total generalizer γ' for $\mathcal{C}'$ relative to $\mathcal{C}_f$ to use the induction hypothesis (i.h.). That construction must be so that one can push the comparison using the instantiation preorder back to γ. The situation is presented in Figure 4. Once the proof is finished, one can employ Lemma 18 to derive the following corollary from which one concludes that the type of syntactic anti-unification is unitary:

Corollary 21 (Strong Completeness)**.** *Let $\mathcal{C}_0$ be an initial configuration, $\mathcal{C}_f$ be a final configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}_f$, and θ_f be the computed substitution of $\mathcal{C}_f$. For every valid configuration $\mathcal{C}$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow^* \mathcal{C}_f$ and every total generalizer γ for $\mathcal{C}$ it is the case that $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$.*

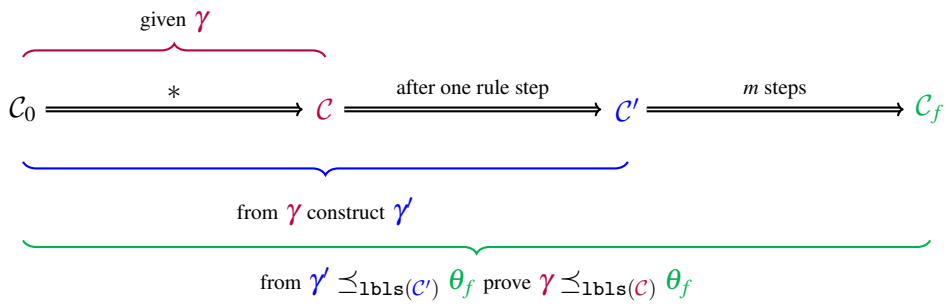

Figure 4: Inductive step in the proof of completeness.

3.3 Completeness Analysis for Syntactic and Solve Rules

This subsection is concerned with the branches of the proof of completeness where the step $\mathcal{C} \Rightarrow \mathcal{C}'$ is achieved by means of the rules `Synt`, `SolNR`, or `SolR`. Let γ be an arbitrary restricted total generalizer for $\mathcal{C}$ relative to $\mathcal{C}_f$. The information concerning both γ and the history of the computation is required for constructing a restricted total generalizer γ' for $\mathcal{C}'$ relative to $\mathcal{C}_f$ that allows the use of the reasoning illustrated in Figure 4.

1. If `Synt` is the employed rule, then there exists a trivial AUT eq_X in $\mathcal{C}_{\mathtt{Uns}}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}}$, and $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}}(X \mapsto \mathtt{lhs}(eq_X))$. By Lemma 16, one knows that $X\gamma$ is different from every variable that is assigned by γ to the labels of non-trivial AUTs. The problem encoded by the trivial AUT eq_X may appear in different positions in the common term structure. The information about it may be already stored in $\mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$ or in $\mathcal{C}_{\mathtt{Sub}}$.
2. If `SolNR` is the employed rule, then there exists a solved AUT eq_X in $\mathcal{C}_{\mathtt{Uns}}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}} \cup \{eq_X\}$, and $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}}$. In other words, this is the first time that the problem encoded by the solved AUT eq_X is encountered. From Lemma 17, it follows that $X\gamma$ is different from $Y\gamma$ for every Y in $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}})$. By Lemma 16, one knows that $X\gamma$ is different from every variable that is assigned by γ to the labels of non-solved AUTs. Notice that no information about the problem is already stored in $\mathcal{C}_{\mathtt{Sub}}$. Nonetheless, information about it may already be stored in $\mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$.
3. If `SolR` is the employed rule, then there exist solved AUTs eq_X in $\mathcal{C}_{\mathtt{Uns}}$ and $eq_{X'}$ in $\mathcal{C}_{\mathtt{Sol}}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}}$, and $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}}(X \mapsto X')$. In other words, the problem encoded by the solved AUT $eq_{X'}$ is encountered again. From Lemma 17, it follows that $X\gamma$ and $X'\gamma$ are different from $Y\gamma$ for every Y in $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}}) \setminus \{X'\}$. Since eq_X and $eq_{X'}$ are repeated AUTs, it is not necessarily the case that $X\gamma = X'\gamma$. By Lemma 16, one knows that $X\gamma$ is different from every variable that is assigned by γ to the labels of non-solved AUTs. The problem encoded by the solved AUT eq_X may appear in different positions in the common term structure, and the information about it is already stored in $\mathcal{C}_{\mathtt{Sol}}$.

That explains why the elements provided by [9] are not sufficient for addressing those rules and why it is crucial to require in the specification of a total generalizer for a valid configuration that it must also generalize both its solved part and the domain of its computed substitution together with its unsolved part. Only requiring that for the latter does not necessarily provide all the needed information regarding the history of the computation to compare the restricted total generalizer and the final computed substitution in those branches of the proof. Now, both the definition of total generalizer and the preservation properties

previously discussed come to the rescue. Since the employed rule is Synt, SolNR, or SolR, one knows by Lemma 13 that $\mathcal{C}'_{\mathtt{Tot}} = \mathcal{C}_{\mathtt{Tot}}$ and $\mathtt{lbls}(\mathcal{C}') = \mathtt{lbls}(\mathcal{C})$. In other words, all the information required for the comparison is already present in $\mathcal{C}$ and is preserved from $\mathcal{C}$ to $\mathcal{C}'$. Thus, one concludes that γ is already a restricted total generalizer for $\mathcal{C}'$ relative to $\mathcal{C}_f$ and $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$ follows immediately from the induction hypothesis. Although the pen-and-paper proof seems straightforward, much of the effort in the verification will be related to its dependencies.

3.4 Completeness Analysis for Decomposition Rules

This subsection is concerned with the branches of the proof of completeness where the step $\mathcal{C} \Rightarrow \mathcal{C}'$ is achieved by means of either DecF or DecP. Let γ be an arbitrary restricted total generalizer for $\mathcal{C}$ relative to $\mathcal{C}_f$.

1. If DecF is the employed rule, then there exist a decomposable AUT $eq_X = ft_L \stackrel{\triangle}{=}_X ft_R$ in $\mathcal{C}_{\mathtt{Uns}}$ and a variable Y fresh for $\mathcal{C}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \cup \{ eq_Y \} \setminus \{ eq_X \}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}}$, $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}} (X \mapsto fY)$, and $eq_Y = t_L \stackrel{\triangle}{=}_Y t_R$. By Lemma 15, the term $X\gamma$ is either a variable or ft for some term t.

1.1. Suppose $X\gamma = ft$ for some term t. Define:

$$\gamma' \coloneqq \gamma|_{\mathtt{dom}(\gamma)\setminus\{Y\}} (Y \mapsto t).$$

That is a restricted total generalizer for $\mathcal{C}'$ relative to $\mathcal{C}_f$. By the i.h., $\gamma' \preceq_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. There exists a substitution δ' such that $\gamma'\delta' =_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. Letting

$$\delta \coloneqq (Y \mapsto t)\, \delta',$$

one concludes that $\gamma\delta =_{\mathtt{lbls}(\mathcal{C})} \theta_f$. Indeed,

$$X\gamma\delta = ft\delta' = fY\gamma'\delta' = fY\theta_f = X\theta_f,$$

and $Z\gamma\delta = Z\gamma'\delta'$ for Z in $\mathtt{lbls}(\mathcal{C}) \setminus \{X\}$. Thus, $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$;

1.2. Suppose $X\gamma = U$ is a variable. There exist cohesion substitutions $\tau_L^{\mathcal{C}}$ and $\tau_R^{\mathcal{C}}$ of γ. There exists a substitution δ'' such that $\gamma\delta'' =_{\mathtt{lbls}(\mathcal{C})} \theta_f\gamma$. Let W be fresh for $\mathcal{C}_f$, γ, $\tau_L^{\mathcal{C}}$, $\tau_R^{\mathcal{C}}$, and δ'. Define:

$$\gamma' \coloneqq \gamma|_{\mathtt{dom}(\gamma)\setminus\{Y\}} (U \mapsto fW)\,(Y \mapsto W).$$

That is a restricted total generalizer for $\mathcal{C}'$ relative to $\mathcal{C}_f$. By the i.h., $\gamma' \preceq_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. There exists a substitution δ' such that $\gamma'\delta' =_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. Letting

$$\delta \coloneqq (U \mapsto fW)\,(Y \mapsto W)\,\delta',$$

one concludes that $\gamma\delta =_{\mathtt{lbls}(\mathcal{C})} \theta_f$. Indeed,

$$X\gamma\delta = fW\delta' = fY\gamma'\delta' = fY\theta_f = X\theta_f,$$

and $Z\gamma\delta = Z\gamma'\delta'$ for Z in $\mathtt{lbls}(\mathcal{C}) \setminus \{X\}$. Thus, $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$;

2. The analysis for DecP is similar and will be omitted.

Crucial steps in verifying the previous arguments rely on the constraints imposed by the definition of restricted total generalizers. For instance, the constraint $\mathtt{rvars}(\gamma) \cap (\mathtt{lbls}(\mathcal{C}_f) \cup \mathtt{prtd}(\mathcal{C}_f)) = \emptyset$ is vital for the construction of the substitution γ'. Moreover, the fact that a total generalizer for a valid configuration generalizes the unsolved part of that configuration is crucial for applying the induction step in verifying these branches of the proof.

4 Conclusion

This paper presented a rigorous pen-and-paper proof of the completeness theorem for syntactic anti-unification and the discussion clarified the formal elements required to mechanically verify the theorem inductively in PVS (and other proof assistants as well). In particular, the paper highlighted important differences that make that mechanization exercise much more complex than verifying the algorithmic solutions to unification, for which the cases of solved and syntactically equal subproblems are resolved immediately, leading to failures and trivial solutions. For formalizing the proof of the completeness theorem in PVS, we addressed these problems with the necessary theoretical rigor. The greater complexity of mechanizing verification for anti-unification algorithms lies in how to record the history of decomposing problems into solved and syntactic subproblems, whose regularities (repetitions) are further encoded by storing repeated solved and syntactic subproblems via the substitution solution under construction. The precise notions that guarantee the construction of a solution more specific than any arbitrary generalizer involve elaborate invariant properties related to the preservation of the problem (Lemma 7), history of the computed solution (Lemma 9), preservation of the extended problem and coherence of the partial solution generated during the computation (Lemmas 13, and 15). In addition, by definition of a restricted notion of arbitrary generalizers (Definition 19), the main completeness result is obtained as a corollary (Corollary 21) of the completeness for such a class of generalizers (Theorem 20). All results presented in Subsection 3.1 are already completely specified and formally verified in PVS.

Future work

Work in progress includes the formalization in PVS of the definitions and lemmas presented in Section 3.2 to obtain a mechanically and formally verified proof of completeness for the functional rule-based algorithm `Antiunify`, from which certified executable code can be extracted. Additional work includes the extension of the formalization for verifying algorithms to anti-unification modulo various algebraic properties widely used in mathematics, such as commutativity, associativity [1, 2], absorption [7, 26], and their combinations.

Acknowledgments

This work was partially supported by the Austrian Science Fund (FWF) project P 35530, the Rustaveli National Science Foundation of Georgia under project FR-25-5957, the Brazilian Research Council CNPq Universal grant 407461/2025-6, and Research productivity grants 313290/2021-0 and 6/2026-7, and a PDSE scholarship of the Brazilian Higher Education Council (CAPES) Finance Code 001 to the last author.

References

[1] María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. ACUOS: A System for Modular ACU Generalization with Subtyping and Inheritance. In *European Conference on Logics in Artificial Intelligence*, volume 8761 of *Lecture Notes in Computer Science*, pages 573–581, 2014.

[2] María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. Order-Sorted Equational Generalization Algorithm Revisited. *Annals of Mathematics and Artificial Intelligence*, 90(5):499–522, 2022.

[3] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. Order-Sorted Generalization. In *17th International Workshop on Functional and (Constraint) Logic Programming*, volume 246 of *Electronic Notes in Theoretical Computer Science*, pages 27–38. Elsevier, 2008.

[4] María Alpuente, Santiago Escobar, Javier Espert, and José. Meseguer. A Modular Order-Sorted Equational Generalization Algorithm. *Information and Computation*, 235:98–136, 2014.

[5] Eva Armengol. Usages of Generalization in Case-Based Reasoning. In *Case-Based Reasoning Research and Development*, volume 4626 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.

[6] Andréia Borges Avelar, André Luiz Galdino, Flávio Leonardo Cavalcanti de Moura, and Mauricio Ayala-Rincón. First-order Unification in the PVS Proof Assistant. *Logic Journal of the IGPL*, 22(5):758–789, 2014.

[7] Mauricio Ayala-Rincón, David M. Cerna, Andrés Felipe González Barragán, and Temur Kutsia. Equational Anti-Unification over Absorption Theories. In *Automated Reasoning - 12th International Joint Conference, Part II*, volume 14740 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2024.

[8] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. Formalising Nominal C-Unification Generalised with Protected Variables. *Mathematical Structures in Computer Science*, 31(3):286–311, 2021.

[9] Mauricio Ayala-Rincón, Thaynara Arielly de Lima, Maria Júlia Dias Lima, Mariano Miguel Moscato, and Temur Kutsia. Verification of an Anti-Unification Algorithm in PVS. In *NASA Formal Methods - 17th International Symposium*, volume 15682 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2025.

[10] Mauricio Ayala-Rincón, Maribel Fernández, and Ana Cristina Rocha Oliveira. Completeness in PVS of a Nominal Unification Algorithm. In *10th Workshop on Logical and Semantic Frameworks with Applications*, volume 323 of *Electronic Notes in Theoretical Computer Science*, pages 57–74. Elsevier, 2015.

[11] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, Temur Kutsia, and Daniele Nantes-Sobrinho. Nominal AC-Matching. In *16th International Conference on Intelligent Computer Mathematics*, volume 14101 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2023.

[12] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, Temur Kutsia, and Daniele Nantes-Sobrinho. Certified First-Order AC-Unification and Applications. *Journal of Automated Reasoning*, 68(25):1–48, 2024.

[13] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. A Certified Functional Nominal C-Unification Algorithm. In *29th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 12042 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2020.

[14] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes Sobrinho. A Certified Algorithm for AC-Unification. In *7th International Conference on Formal Structures for Computation and Deduction*, volume 228 of *Leibniz International Proceedings in Informatics*, pages 8:1–8:21, 2022.

[15] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[16] Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning*. Elsevier, 2001.

[17] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to Fix Bugs Automatically. *Proceedings of the ACM on Programming Languages*, 3:159:1–159:27, 2019.

[18] Adam David Barwell, Cristopher Brown, and Kevin Hammond. Finding Parallel Functional Pearls: Automatic Parallel Recursion Scheme Detection in Haskell Functions via Anti-Unification. *Future Generation Computer Systems*, 79:669–686, 2018.

[19] Alexander Baumgartner and Temur Kutsia. A Library of Anti-Unification Algorithms. In *European Conference on Logics in Artificial Intelligence*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557, 2014.

[20] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A Variant of Higher-Order Anti-Unification. In *24th International Conference on Rewriting Techniques and Applications*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 113–127, 2013.

[21] Peter. E. Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Anti-Unification Algorithms and Their Applications in Program Analysis. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009.

[22] David M. Cerna and Temur Kutsia. Anti-Unification and Generalization: A Survey. In *32nd International Joint Conference on Artificial Intelligence*, pages 6563–6573. International Joint Conferences on Artificial Intelligence Organization, 2023.

[23] Evelyne Contejean. A Certified AC Matching Algorithm. In *Rewriting Techniques and Applications, 15th International Conference*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[24] Reudismam Rolim de Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning Quick Fixes from Code Repositories. In *35th Brazilian Symposium on Software Engineering*, pages 74–83. ACM, 2021.

[25] André Luiz Galdino and Mauricio Ayala-Rincón. A Formalization of the Knuth-Bendix(-Huet) Critical Pair Theorem. *Journal of Automated Reasoning*, 45(3):301–325, 2010.

[26] Andrés Felipe González Barragán. *Anti-Unification in Absorptive Theories*. PhD thesis, University of Brasília, 2025. Available at `http://repositorio.unb.br/handle/10482/54148`.

[27] Fabian Huch. Supporting Maintenance of Formal Mathematics with Similarity Search. In *18th International Conference on Intelligent Computer Mathematics*, volume 16136 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2025.

[28] Gerard Huet. Résolution d'Équations dans des Langages d'ordre $1, 2, \ldots, \omega$. Thèse d'État, Université de Paris VII, 1976.

[29] Jean-Louis Lassez, Michael J. Maher, and Kim Marriott. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[30] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, Balasubramanyan Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation*, pages 435–448. USENIX Association, 2020.

[31] Ana Cristina Rocha Oliveira, André Luiz Galdino, and Mauricio Ayala-Rincón. Confluence of Orthogonal Term Rewriting Systems in the Prototype Verification System. *Journal of Automated Reasoning*, 58(2):231–251, 2017.

[32] Frank Pfenning. Unification and Anti-Unification in the Calculus of Constructions. In *6th Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society, 1991.

[33] Gordon David Plotkin. A Note on Inductive Generalization. *Machine Intelligence*, 5(1):153–163, 1970.

[34] Daniel Ramos, Catarina Gamboa, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. SPELL: Synthesis of Programmatic Edits using LLMs. *CoRR*, abs/2602.01107, 2026.

[35] John Charles Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. *Machine Intelligence*, 5(1):135–151, 1970.

[36] José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo, and Francisco-Jesús Martín-Mateos. Mechanical Verification of a Rule-Based Unification Algorithm in the Boyer-Moore Theorem Prover. In *1999 Joint Conference on Declarative Programming*, pages 289–304, 1999.

[37] José-Luis Ruiz-Reina, Franciso-Jesús. Martín-Mateos, José-Antonio Alonso, and María-José. Hidalgo. Formal Correctness of a Quadratic Unification Algorithm. *Journal of Automated Reasoning*, 37(1-2):67–92, 2006.

[38] Wim Vanhoof and Gonzague Yernaux. Generalization-Driven Semantic Clone Detection in CLP. In *Logic-Based Program Synthesis and Transformation - 29th Int. Symposium, Revised Selected Papers*, volume 12042 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2019.

[39] Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Óskar Haraldsson, John R. Woodward, Serkan Kirbas, Etienne Windels, Olayori McBello, Abdurahman Atakishiyev, Kevin Kells, and Matthew W. Pagano. Towards Developer-Centered Automatic Program Repair: Findings from Bloomberg. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1578–1588. Association for Computing Machinery, 2022.