Verification of an Anti-Unification Algorithm in PVS

Mauricio Ayala-Rincón[1], Thaynara Arielly de Lima[2], Maria Júlia Dias Lima[3], Mariano Miguel Moscato[4], and Temur Kutsia[5]

[1] Universidade de Brasília, Exact Sciences Institute, Brasília D.F., Brazil
[2] Universidade Federal de Goiás, Institute of Mathematics and Statistics, Goiânia, Brazil
[3] Universidade de Brasília, Graduate Program in Informatics, Brasília D.F., Brazil
[4] Analytical Mechanics Associates Inc., Hampton, VA, U.S.A.
[5] Research Institute for Symbolic Computation, Johannes Kepler Universität, Linz, Austria

Abstract. Anti-unification is the problem of computing the commonalities between syntactic terms. Among its main applications, the detection of code regularities stands out since it is widely used in industrial settings as the foundation for technologies for the recognition of code-cloning, improvement of parallel compilation, and software error detection and correction. This paper discusses the verification of an anti-unification algorithm in the Prototype Verification System (PVS). The algorithm is based on inference rules that constructively compute the common structure between two terms, providing a substitution that expresses the *least general generalizer* between them as output. To the best of the authors' knowledge, this is the first formalization of an anti-unification procedure containing proofs of relevant properties, such as termination and soundness.

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

1 Introduction

Motivation and Contextualization. Generalization problems, in one form or another, arise in various areas of mathematics, computer science, and artificial intelligence. One such form involves finding a common generalization of given logical expressions, usually terms, which should preserve their similarities as much as possible while uniformly abstracting over their differences. In logic, the process of finding such common generalizations is called *anti-unification*. Research in this area was initiated by Plotkin [25] and Reynolds [27], who independently proposed the first algorithms for first-order syntactic anti-unification. Since then, the problem has been studied in various theories, motivated by different applications. In recent years, there has been a renewed interest in this technique due to its relevance in many modern applications, such as, automatic program repair [10,16,30], software clone detection and refactoring [11,12,29], programming by examples [26], preventing bugs and misconfiguration in services [19], linguistic structure learning for chatbots [17], and reasoning in natural language [31], to name just a few. A recent survey [15] provides more details about the state-of-the-art in theory and applications of anti-unification.

First-order syntactic anti-unification, the focus of this paper, remains central to all subsequent advanced anti-unification algorithms and applications. Therefore, formalizing and verifying the syntactic anti-unification algorithm can have significant implications for the reliability of many algorithms and software tools that depend on it.

Main Contribution. This paper presents a novel formalization of an anti-unification algorithm in the proof assistant PVS [23]. Along with the specification of the algorithm itself, proofs of relevant properties, such as termination and soundness, are provided. The presented work builds upon some definitions of the algebra of terms and substitutions that were adapted from the PVS libraries `TRS` and `nominal`, part of NASALib –a collection of mathematical developments formalized in PVS and maintained by NASA[6]. Such libraries were contributed by some of the authors of this paper in previous works focused on the formalization of the theory of syntactic unification [3], nominal unification modulo theories [6,7], and AC-unification [8]. The development presented in this paper also aims at building the basis for extensions of anti-unification in the nominal syntax and modulo theories, as done in the `nominal` library for unification algorithms. Additionally, the formalization presented in this paper is expected to serve as a foundational stone for further validation of software based on generalization algorithms.

The algorithm is presented as a general functional procedure to apply anti-unification inference rules, which is a well-known logical technique to *decompose* terms according to their commonalities, *merging* different occurrences of common elements equally distributed on the terms, and *solving* subterms without commonalities that appear in common positions of the terms. The formalization profits from relevant features of the PVS language, such as predicate subtyping and the support for dependent types. These features allow discharging properties such as termination and well-behavior ("niceness") of the computed generalization in an automatic way. The referred automation is possible by attributing elaborated types to the inference rules carrying information about the size decrement of the problem at hand and the construction of well-behaved substitutions. As part of the type checking of these definitions, PVS generates proof obligations called *Type Correctness Conditions* (TCC). Due to the expressiveness of the PVS language, not all the TCCs can be proven automatically. Nevertheless, even the ones having to be manually proven state particular properties on each inference rule, which makes the proof much more focused and then accessible than proving the corresponding property on the whole algorithm. Having termination and the well-behavior of the computed generalizations, the proof of

[6] NASALib can be accessed at https://github.com/nasa/pvslib.

soundness consists of an inductive proof verifying that the output substitutions effectively generalize the input terms.

The PVS formalization presented in this paper is publicly available as part of NASALib[7]. To the best of the authors' knowledge, this is the first formalization of an anti-unification procedure including proofs of its termination and soundness.

Organization. Section 2 presents the required background following a notation close to the one used in the formalization. Section 3 presents analytical proofs, whose formalization is explained in detail in Section 4. Finally, Section 5 concludes and gives perspectives for future work.

2 Background

Definition 1 (Terms). *Let Σ be a signature with uninterpreted function symbols and consider $\mathbb{X}$ a countable infinite set of variables. The set $T(\Sigma, \mathbb{X})$ of Σ-terms is generated by the grammar:*

$$s, t ::= c \mid X \mid \langle\rangle \mid (s, t) \mid f\, s$$

Above, c denotes a constant, X is a variable, $\langle\rangle$ is the unit symbol, (s, t) is a pair, and $f\, s$ is a function application.

The size *of a term t is defined as: $size(t) = 1$, where t is a variable, a constant, or the unit symbol, $size((s, t)) = 1 + size(s) + size(t)$, and $size(f\, s) = 1 + size(s)$.*

The set of variables appearing in a term t is denoted as $vars(t)$. This notation is extended to sets of terms: $vars(M) = \cup_{t \in M} vars(t)$.

In the following, constants are denoted by lowercase Latin letters of the first part of the alphabet, and variables by capital letters from the end of the alphabet. Also, Σ-terms are simply referred as "terms.". The grammar is adapted from the notation for the nominal syntax used in the PVS library `nominal` [8]. The inclusion of unity and pairs in the nominal syntax is standard (e.g., [28]) and of great utility in allowing functions of arbitrary arity. The use of $\langle\rangle$ and pairs ease the implementation of operations for *flattening* terms, which in its turn, allows to deal efficiently with associativity. The same technique has been applied in formalizations of unification modulo associativity (e.g., [5, 9]) and will be useful in extensions to anti-unification modulo equational theories.

Definition 2 (Substitution). *A* substitution *σ is a mapping from $\mathbb{X}$ to the set of terms $T(\Sigma, \mathbb{X})$ that moves only a finite set of variables in $\mathbb{X}$. It is represented by $\sigma = (X_1 \mapsto t_1) \cdots (X_n \mapsto t_n)$, where each $(X_i \mapsto t_i)$, $1 \le i \le n$, is called a* basic substitution*. The empty substitution is called the* identity *and is denoted as id. The set $sup\_dom = \{X_1, \ldots, X_n\}$ is called the* super-set domain *of σ. In particular, $sup\_dom(id) = \emptyset$.*

[7] The version referred in this text can be found in the `TRS` library, commit: 0835a23.

Substitutions, except for the identity, are denoted by lowercase Greek letters.

Definition 3 (Action of a substitution). *The* action *of a basic substitution* $(X \mapsto s)$ *over a term* t *is defined inductively as:*

i) $(X \mapsto s)(X) = s$;
ii) $(X \mapsto s)(Y) = Y$, *for* $Y \neq X$;
iii) $(X \mapsto s)(u) = u$, *for either* $u = \langle\rangle$ *or* $u = c$, *a constant;*
iv) $(X \mapsto s)((u, v)) = ((X \mapsto s)(u), (X \mapsto s)(v))$; *and,*
v) $(X \mapsto s)(fu) = f(X \mapsto s)(u)$.

The action of basic substitutions is extended inductively to define the action of a substitution σ *on terms as:*

i) $\sigma(t) = t$, *if* $\sigma = id$;
ii) $(\sigma(X \mapsto s))(t) = \sigma((X \mapsto s)(t))$

The domain *and* range *of a substitution are respectively defined as the sets*

$$dom(\sigma) = \{X \mid \sigma(X) \neq X\} \quad and \quad ran(\sigma) = \{\sigma(X) \mid X \in dom(\sigma)\}.$$

The set of variables in $ran(\sigma)$, $vars(ran(\sigma))$, *is denoted as* $\mathcal{V}ran(\sigma)$. *From the definition of the action of a substitution, the* composition *of two substitutions* δ *and* σ *is naturally defined as the concatenation* $\delta\sigma$. *The substitution* σ restricted *to* $\mathbb{V}$, *a subset of* $dom(\sigma)$, *is defined as a* mapping from variables into terms $\sigma|_{\mathbb{V}}$ *with* $dom(\sigma|_{\mathbb{V}}) = \mathbb{V}$, *and such that for all* $X \in \mathbb{V}, \sigma|_{\mathbb{V}}(X) = \sigma(X)$.

Remark 1. Prefix notation agrees with the definition of substitution composition, i.e., $(\delta\sigma)(t) = \delta(\sigma(t))$. Also, substitution composition is associative.

Definition 4 (Renaming and Variant). *A* renaming *is an injective substitution* ρ *into a set of variables, i.e., for all* $X, Y \in dom(\rho)$, *if* $\rho(X) = \rho(Y)$ *then* $X = Y$.

A pair of terms t_1 *and* t_2 *are called* variants *if there exist renamings* ρ_1 *and* ρ_2 *such that* $dom(\rho_i) = vars(t_i)$ *for* $i = 1, 2$, *and* $\rho_1(t_1) = \rho_2(t_2)$.

Example 1. The terms $s = f(h(X, c), h(Y, X))$, $t = f(h(U, c), h(X, U))$, $u = f(h(U, c), h(V, U))$, and $v = f(h(Y, c), h(X, Y))$, are variants. For instance, let $\rho_1 = (Y \mapsto U)(X \mapsto V)$, and $\rho_2 = (X \mapsto U)(Y \mapsto V)$. Then, $\rho_1(s) = \rho_2(v)$. Also, notice that for this definition of substitution, there exists no renaming ρ such that $\rho(s) = v$, but $\rho_2^{-1}\rho_1(s) = v$ and $\rho_1^{-1}\rho_2(v) = s$.

Remark 2. Variants are terms known as *equivalent up to variable renaming*, i.e., terms that can be obtained one from the other by the action of the composition of renaming.

Definition 5 (More General Term). *Given two terms* s *and* t, s *is* more general *than* t *if there exists a substitution* σ *such that* $\sigma(s) = t$. *In this case,* t *is said to be* less general *or* more specific *than* s. *It is denoted by* $s \leq t$.

Example 2. Consider $t = f(f(a,b),a)$, $s_1 = f(X,Y)$ and $s_2 = f(f(Z,b),Z)$. If $\sigma_1 = (X \mapsto f(a,b))(Y \mapsto a)$ and $\sigma_2 = (Z \mapsto a)$ then $\sigma_1(s_1) = t$ and $\sigma_2(s_2) = t$. Thus, $s_1 \leq t$ and $s_2 \leq t$. Also, notice that $s_1 \leq s_2$.

Definition 6 (Term Generalizer). *Given a finite set of terms M, a term s is a* generalizer *of M if $s \leq t$ for all $t \in M$:*

$$generalizers(M) = \{s \in T(\Sigma, \mathbb{X}) \mid \forall t \in M,\ s \leq t\}.$$

Remark 3. For any finite set of terms M, a term generalizer always exists since a variable X is more general than any $t' \in M$. Also, $generalizers(\emptyset) = T(\Sigma, \mathbb{X})$.

Definition 7 (Least General Generalizer). *Given a finite set of terms M, a term $s \in generalizers(M)$ is called a* least general generalizer *(for short,* `lgg`*) of M if $t \leq s$ for all $t \in generalizers(M)$.*

Example 3. Consider $M = \{f(f(a,b),a), f(f(c,b),c)\}$. Thus, up to variable renaming,

$$\begin{aligned} generalizers(M) = \{&X, f(X,Y), f(f(X,Y),Z), f(f(X,Y),X),\\ &f(f(X,b),Z), f(f(X,b),X)\}. \end{aligned}$$

Notice that $f(f(X,b),X)$ is an `lgg` of M. Indeed, it is unique up to variable renaming, and then one can say that it is *the* `lgg` of M.

Remark 4. Notice that $\emptyset$ does not have an `lgg`. For finite non-empty sets, the syntactic case of the anti-unification problem is of type *unitary*, i.e., for any problem, there exists a unique `lgg` up to renamings [15, 25, 27]. Nevertheless, this is not necessarily the case for other theories in which the problem can be of type *finitary* [2], *infinitary* [13], or even of *type zero* [14].

Definition 8 (Anti-unification Triple). *An* anti-unification triple *(AUT) has the form $s \underset{X}{\triangleq} t$, where s and t are terms, and X is a variable such that $X \notin vars(\{s,t\})$. It is denoted as eq_X, its* label *is X, and its left- and right-hand sides, s and t, are denoted as $\mathtt{lhs}(eq_X)$ and $\mathtt{rhs}(eq_X)$, respectively.*

Given a set of AUTs, $A = \{eq_{X_1}, \ldots, eq_{X_n}\}$, the expression $\mathtt{lbls}(A)$ denotes the set of labels of A, $\{X_i \mid 1 \leq i \leq n\}$. Also, the set of variables in A, $vars(\{\mathtt{lhs}(eq_{X_i}) \mid 1 \leq i \leq n\} \cup \{\mathtt{rhs}(eq_{X_i}) \mid 1 \leq i \leq n\})$, is denoted by $vars(A)$.

The size *of an AUT eq_X is defined as $size(\mathtt{lhs}(eq_X)) + size(\mathtt{rhs}(eq_X))$, and the size of a set of $AUTs$ A as above is defined as $\sum_{1 \leq i \leq n} size(eq_{X_i})$. Additionally, A is called a* valid *set of $AUTs$ if all its labels are pairwise different ($X_i \neq X_j$, for $i \neq j, 1 \leq i,j \leq n$) and $\mathtt{lbls}(A) \cap vars(A) = \emptyset$.*

In contrast to unification, in which unifiers are built by binding variables occurring on the terms of the problem, in anti-unification, the terms (in the AUTs) are rigid, and solutions are built by binding labels. The necessity of labels is made precise in Section 4.

Definition 9 (AUT Classification). *An AUT $s \underset{X}{\triangleq} t$ is classified as:*

i) Decomposable*, if s and t are pairs or function applications beginning with the same function symbol.*
ii) Solved*, if it is not* decomposable *and s and t are different terms.*
iii) Trivial*, if both s and t are the unit symbol, the same constant or the same variable.*

Two different AUTs eq_X and eq_Y with $\mathtt{lhs}(eq_X) = \mathtt{lhs}(eq_Y)$ and $\mathtt{rhs}(eq_X) = \mathtt{rhs}(eq_Y)$ are called repeated*. An AUT eq_X is* repeated *in a set of AUTs A, if there exists an AUT $eq_Y \in A$ such that eq_X and eq_Y are* repeated*.*

Definition 10 (Anti-unification Problem). *Consider the valid set of AUTs $A = \{eq_{X_1}, \dots, eq_{X_n}\}$. An Anti-Unification Problem consists in finding a set of terms $\{t_1, \dots, t_n\}$ and two substitutions τ_l and τ_r satisfying, for all $1 \leq i \leq n$:*

i) t_i *is an* $\mathtt{lgg}$ *of* $\{\mathtt{lhs}(eq_{X_i}), \mathtt{rhs}(eq_{X_i})\}$*;*
ii) $\tau_l(t_i) = \mathtt{lhs}(eq_{X_i})$ *and* $\tau_r(t_i) = \mathtt{rhs}(eq_{X_i})$*.*

Note that the unique solution for an empty set of AUTs is an empty set of terms.

Example 4. Consider $A = \{f(f(a,a),b) \underset{X_1}{\triangleq} f(f(a,b),b), f(a,b) \underset{X_2}{\triangleq} f(c,d)\}$. The set of terms $\{t_1 = f(f(a,X),b), t_2 = f(Y,Z)\}$ and the substitutions $\tau_l = (Z \mapsto b)(Y \mapsto a)(X \mapsto a)$ and $\tau_r = (Z \mapsto d)(Y \mapsto c)(X \mapsto b)$ satisfy the Definition 10.

Remark 5. Let A, $\{t_1, \dots, t_n\}$, τ_l and τ_r be in terms of Definition 10. Suppose $(\bigcup_{1 \leq i \leq n} vars(t_i)) \cap \mathtt{lbls}(A) = \emptyset$. Consider $\gamma = (X_n \mapsto t_n) \cdots (X_1 \mapsto t_1)$. Then, $(\tau_l\gamma)(X_i) = \mathtt{lhs}(eq_{X_i})$ and $(\tau_r\gamma)(X_i) = \mathtt{rhs}(eq_{X_i})$, for $1 \leq i \leq n$. Thus, the classical and standard version of the Anti-Unification Problem, stated in Def. 10, can be paraphrased in terms of finding a substitution γ with $dom(\gamma) = \mathtt{lbls}(A)$, instead of a set of terms $\{t_1, \dots, t_n\}$. In the following, an alternative version of the Anti-Unification Problem is stated.

Definition 11 (More General Substitution). *Let γ and σ be substitutions and $\mathbb{V}$ a finite set of variables. γ is said to be more general than σ on $\mathbb{V}$ if there exists a substitution δ such that $(\delta\,\gamma)|_{\mathbb{V}} = \sigma|_{\mathbb{V}}$. It is denoted by $(\gamma \preceq \sigma)\big|_{\mathbb{V}}$. Alternatively, σ is said to be more specific than γ on $\mathbb{V}$.*

Example 5. Consider the substitutions $\gamma = (X \mapsto f(f(Z,Y),W))(Y \mapsto f(W,X))$ and $\sigma = (X \mapsto f(f(Z,b),Z))$. If $\delta = (Y \mapsto b)(W \mapsto Z)$, then $(\delta\,\gamma)(X) = \sigma(X)$; thus, $(\gamma \preceq \sigma)\big|_{\{X\}}$. Also, verify that $(\sigma \preceq \gamma)\big|_{\{Y\}}$ but neither $(\sigma \preceq \gamma)\big|_{\{X\}}$ nor $(\gamma \preceq \sigma)\big|_{\{Y\}}$. Consider now the substitutions $\gamma' = (X \mapsto f(f(Z,b),Z))(Y \mapsto f(W,X))$ and $\sigma' = (X \mapsto f(f(Z,Y),W))$. Then, $(\sigma' \preceq \gamma')\big|_{\{X,Y\}}$.

Definition 12 (Substitution Generalization). *Let $A = \{eq_{X_1}, \dots, eq_{X_n}\}$ be a valid set of AUTs. A substitution γ, with $dom(\gamma) = \mathtt{lbls}(A)$, is a substitution generalization of A if there exist substitutions τ_l and τ_r such that $(\tau_l\gamma)(X_i) = \mathtt{lhs}(eq_{X_i})$ and $(\tau_r\gamma)(X_i) = \mathtt{rhs}(eq_{X_i})$, for all $1 \leq i \leq n$.*

$$\text{(Decompose-Function)}\ \frac{\langle \{f\,s \underset{X}{\triangleq} f\,t\} \cup U' \mid S \mid \sigma\rangle}{\langle \{s \underset{Y}{\triangleq} t\} \cup U' \mid S \mid (X \mapsto f\,Y)\,\sigma\rangle}$$

$$\text{(Decompose-Pair)}\ \frac{\langle \{(s,u) \underset{X}{\triangleq} (t,v)\} \cup U' \mid S \mid \sigma\rangle}{\langle \{s \underset{Y}{\triangleq} t, u \underset{Z}{\triangleq} v\} \cup U' \mid S \mid (X \mapsto (Y,Z))\,\sigma\rangle}$$

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

$$\text{(Solve-Non-Repeated)}\ \frac{\langle \{s \underset{X}{\triangleq} t\} \cup U' \mid S \mid \sigma\rangle}{\langle U' \mid \{s \underset{X}{\triangleq} t\} \cup S \mid \sigma\rangle}\ \text{if } s \underset{X}{\triangleq} t \text{ is solved and non-repeated in } S$$

$$\text{(Syntactic)}\ \frac{\langle \{s \underset{X}{\triangleq} s\} \cup U' \mid S \mid \sigma\rangle}{\langle U' \mid S \mid (X \mapsto s)\,\sigma\rangle}\ \text{if } s \underset{X}{\triangleq} s \text{ is trivial}$$

Fig. 1. Anti-unification Rules.

Note that the unique substitution generalization of the empty set of AUTs is *id*.

Definition 13 (Anti-unification Problem: Alternative Version). *Let A be a valid set of AUTs. The Anti-Unification Problem consists in finding a substitution generalization of A, σ, such that if γ is also a substitution generalization of A, then $(\gamma \preceq \sigma)|_{\mathtt{lbls}(A)}$.*

In the following, Anti-Unification Problem refers to the one stated in Definition 13. The formalization follows the approach proposed by [1]. It is based on a terminating procedure developed from a set of inference rules over configurations, notions that will be presented next.

Definition 14 (Configuration). *A* configuration *is a triple*

$$\mathcal{C} := \langle \mathcal{C}_{\mathtt{Uns}} \mid \mathcal{C}_{\mathtt{Sol}} \mid \mathcal{C}_{\mathtt{Sub}}\rangle$$

where $\mathcal{C}_{\mathtt{Uns}}$ and $\mathcal{C}_{\mathtt{Sol}}$ are sets of AUTs called the unsolved *and* solved *parts of $\mathcal{C}$, respectively, and $\mathcal{C}_{\mathtt{Sub}}$ is a substitution called the* substitution *of $\mathcal{C}$.*

In addition, a valid configuration *satisfies the following constraints:*

i) $\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}$ is a set of valid AUTs: the labels are unique and disjoint from $vars(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}})$;
ii) $dom(\mathcal{C}_{\mathtt{Sub}})$ and $vars(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}) \cup \mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}})$ are disjoint.
iii) $\mathcal{C}_{\mathtt{Sol}}$ consists of non-repeated solved AUTs.

In Figure 1, the system of inference rules to build substitution generalizations for valid configurations is shown. The inference rules modify valid configurations according to an *active* AUT in the unsolved part.

The *decompose* rules, (Decompose-Function) and (Decompose-Pair), decompose *active* AUTs. They add *fresh* labels (one for function-decomposable AUTs and two for pair-decomposable AUTs) and append to the substitution a new basic substitution recording the common structure of the terms in the *active* AUT. A *fresh* variable is a variable that does not appear in the configuration. Fresh labels are highlighted in the Figure. The *solve* rules, (Solve-Repeated) and (Solve-Non-Repeated), consider differences given by a solved *active* AUT. If the *active* AUT is repeated in the solved part, the substitution is modified adding a basic substitution from the *active* label to the label of the repeated AUT and the *active* AUT is deleted; otherwise, the active AUT is moved to the solved part of the configuration. Finally, the (Syntactic) rule deletes a trivial *active* AUT and adds to the substitution a basic substitution from its label to its term. The anti-unification algorithm, denoted as `Antiunify`, consists of the exhaustive application of the inference rules.

Abbreviations (DecF), (DecP), (SolR), (SolNR), and (Synt) of the rule names are used in this document when convenient. The symbols $\Longrightarrow$ and $\Longrightarrow^*$ denote one-step rule *reduction* and *derivation*, i.e., the reflexive transitive closure of $\Longrightarrow$; thus, $\mathcal{C} \Longrightarrow^* \mathcal{C}'$ means that the configuration $\mathcal{C}'$ is obtained from configuration $\mathcal{C}$ after an arbitrary number of applications of the inference rules. Superscripts will highlight the inference rule applied in reductions; for instance, $\mathcal{C} \overset{\text{DecP}}{\Longrightarrow} \mathcal{C}'$ means that $\mathcal{C}'$ is obtained from $\mathcal{C}$ applying the rule (DecP).

Example 6. The Figure 2 exemplifies a derivation using the inference rules. Also, for brevity, configuration substitutions are restricted to the input active label X: for example, after the first application of the rule (DecP), the substitution $(X \mapsto f(Z_1, Z_2)) = ((Y \mapsto (Z_1, Z_2))(X \mapsto fY))\big|_{\{X\}}$. The substitution of the final configuration, $(X \mapsto f(f(Z, b), Z)) = ((Z_2 \mapsto Z)(Z_4 \mapsto b)(Z_3 \mapsto (Z, Z_4))(Z_1 \mapsto fZ_3)(Y \mapsto (Z_1, Z_2))(X \mapsto fY))|_{\{X\}}$ is a substitution generalization of the unsolved part of the initial configuration, $\{f(f(c, b), c) \underset{X}{\triangleq} f(f(d, b), d)\}$. Also, the solved part of the final configuration, $\{c \underset{Z}{\triangleq} d\}$ gives the needed information to build τ_l and τ_r (see Definition 12): $\tau_l = (Z \mapsto c)$ and $\tau_r = (Z \mapsto d)$.

3 Analytical Proofs

The anti-unification algorithm `Antiunify` is proved to *terminate* by induction, using as measure the size of the configuration *size*($\mathcal{C}$) defined as the size of its unsolved part (*size*($\mathcal{C}_{\texttt{Uns}}$)). It is easy to verify that this measure strictly decreases after each rule application. Indeed, the *size* of the configurations obtained after applying the rules (DecF) and (DecP) decreases by two; and the rules (SolR), (SolNR) and (Synt) eliminate an AUT from $\mathcal{C}_{\texttt{Uns}}$. Therefore, final configurations are of the form $\langle \emptyset \mid S \mid \sigma \rangle$.

Additionally, some *preservation* properties are required. In particular, the algorithm transforms valid configurations into valid configurations. This is also

$$\begin{array}{lc}
 & \langle \{ f(f(c,b),c) \underset{X}{\triangleq} f(f(d,b),d) \} \mid \emptyset \mid id \rangle \\ \hline
\text{(DecF)} & \langle \{ (f(c,b),c) \underset{Y}{\triangleq} (f(d,b),d) \} \mid \emptyset \mid (X \mapsto f\, Y) \rangle \\ \hline
\text{(DecP)} & \langle \{ f(c,b) \underset{Z_1}{\triangleq} f(d,b), c \underset{Z_2}{\triangleq} d \} \mid \emptyset \mid (X \mapsto f\,(Z_1, Z_2)) \rangle \\ \hline
\text{(DecF)} & \langle \{ (c,b) \underset{Z_3}{\triangleq} (d,b), c \underset{Z_2}{\triangleq} d \} \mid \emptyset \mid (X \mapsto f\,(f\, Z_3, Z_2)) \rangle \\ \hline
\text{(DecP)} & \langle \{ c \underset{Z}{\triangleq} d, b \underset{Z_4}{\triangleq} b, c \underset{Z_2}{\triangleq} d \} \mid \emptyset \mid (X \mapsto f\,(f\,(Z, Z_4), Z_2)) \rangle \\ \hline
\text{(SolNR)} & \langle \{ b \underset{Z_4}{\triangleq} b, c \underset{Z_2}{\triangleq} d \} \mid \{ c \underset{Z}{\triangleq} d \} \mid (X \mapsto f\,(f\,(Z, Z_4), Z_2)) \rangle \\ \hline
\text{(Synt)} & \langle \{ c \underset{Z_2}{\triangleq} d \} \mid \{ c \underset{Z}{\triangleq} d \} \mid (X \mapsto f\,(f\,(Z, b), Z_2)) \rangle \\ \hline
\text{(SolR)} & \langle \emptyset \mid \{ c \underset{Z}{\triangleq} d \} \mid (X \mapsto f\,(f\,(Z, b), Z)) \rangle
\end{array}$$

Fig. 2. Derivation Example.

proved inductively by analysis of cases. For instance, suppose $\mathcal{C} \overset{\text{DecF}}{\Longrightarrow} \mathcal{C}'$. Then, the three constraints in the definition 14 of valid configuration are preserved. Indeed, suppose the active AUT in the valid configuration $\mathcal{C}$ is $f\, s \underset{X}{\triangleq} f\, t$.

1. Then, $\mathcal{C}'_{\mathtt{Uns}}$ includes the AUT $s \underset{Y}{\triangleq} t$, and $\mathcal{C}'_{\mathtt{Sub}} = (X \mapsto f\, Y)\mathcal{C}_{\mathtt{Sub}}$, where Y is fresh. Thus, the labels in $\mathcal{C}'_{\mathtt{Uns}}$ and $\mathcal{C}'_{\mathtt{Sol}}$ are unique and disjoint from $vars(\mathcal{C}'_{\mathtt{Uns}} \cup \mathcal{C}'_{\mathtt{Sol}})$. Then, $\mathcal{C}'_{\mathtt{Uns}} \cup \mathcal{C}'_{\mathtt{Sol}}$ is a set of valid AUTs.
2. $dom(\mathcal{C}'_{\mathtt{Sub}}) = \{X\} \cup dom(\mathcal{C}_{\mathtt{Sub}})$, $vars(\mathcal{C}'_{\mathtt{Uns}} \cup \mathcal{C}'_{\mathtt{Sol}}) = vars(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}})$, and $\mathtt{lbls}(\mathcal{C}'_{\mathtt{Uns}} \cup \mathcal{C}'_{\mathtt{Sol}}) = \{Y\} \cup \mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}) \setminus \{X\}$, which guarantees the second constraint.
3. $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}}$, which consists of non-repeated solved AUTs.

To prove that the algorithm *correctly* computes generalizations, some restrictions on the configurations in a derivation are required. By termination, a valid configuration leads to a final valid configuration as $\langle A \mid \emptyset \mid id \rangle \Longrightarrow^* \langle \emptyset \mid S \mid \sigma \rangle$. So, σ should generalize the input problem, that is for all $s \underset{X}{\triangleq} t \in A$, $\sigma(X) \preceq s$ and $\sigma(X) \preceq t$. In addition, the final solved part S provides information about the construction of τ_l and τ_r: $\tau_l = (Z_m \mapsto \mathtt{lhs}(eq_{Z_m})) \cdots (Z_1 \mapsto \mathtt{lhs}(eq_{Z_1}))$ and $\tau_r = (Z_m \mapsto \mathtt{rhs}(eq_{Z_m})) \cdots (Z_1 \mapsto \mathtt{rhs}(eq_{Z_1}))$, where eq_{Z_i}, for $1 \leq i \leq m$ are the AUTs in S.

The proof proceeds by induction on the size of configurations. The *base case* holds since when $size(\mathcal{C}) = 0$, the unsolved part is empty. For the *inductive step*, a derivation of the form $\mathcal{C} \Longrightarrow \mathcal{C}' \Longrightarrow^* \langle \emptyset \mid S \mid \sigma \rangle$ is considered. One assumes that the result holds for $\mathcal{C}' \Longrightarrow^* \langle \emptyset \mid S \mid \sigma \rangle$. Then, the proof is finished by case analysis on the rule applied in the first reduction; for instance, if (decF) applies, then following derivation holds:

$$\langle \{ f\, s \underset{X}{\triangleq} f\, t \} \cup U' \mid S_0 \mid \sigma_0 \rangle \overset{\text{DecF}}{\Longrightarrow} \langle \{ s \underset{Y}{\triangleq} t \} \cup U' \mid S_0 \mid (X \mapsto f\, Y)\sigma_0 \rangle.$$

By hypothesis, for all $s_i \underset{X_i}{\triangleq} t_i \in U'$, $1 \leq i \leq n$, one has that $\tau_l \sigma X_i = s_i$ and $\tau_r \sigma X_i = t_i$. To conclude, since $\tau_l \sigma Y = s$ and $\tau_r \sigma Y = t$, and the basic substitution $(X \mapsto f\, Y)$ is part of the final substitution σ, one concludes that $\tau_l \sigma X = f\, s$ and $\tau_r \sigma X = f\, t$. More precisely, it is required to prove (inductively) that for some θ, $\sigma = \theta(X \mapsto f\, Y)\sigma_0$. Also, by the item 2 of validity (Definition 14) of $\mathcal{C}$ and $\mathcal{C}'$, $\sigma_0 X = X$ and $(X \mapsto f\, Y)\sigma_0 Y = Y$. So, $\tau_l \sigma X = \tau_l \theta f\, Y = f\, \tau_l \theta Y = f\, \tau_l \sigma Y = f\, s$. Similarly, one proves that $\tau_r \sigma X = f\, t$.

The proof of completeness also requires an inductive analysis. Suppose that there exists a substitution generalization γ with associated ρ_l and ρ_r as in Definition 12 for the set of AUTs $\mathcal{C}_{\texttt{Uns}}$, where $\mathcal{C}$ is a valid configuration. It must be proved that γ is more general than the computed substitution generalization, say σ: $(\gamma \preceq \sigma)|_{\texttt{lbls}(\mathcal{C}_{\texttt{Uns}})}$. For instance, the *inductive step* is analyzed for the case in which $\mathcal{C} \overset{\text{DecF}}{\Longrightarrow} \mathcal{C}' \Longrightarrow^* \langle \emptyset \mid S \mid \sigma \rangle$, as above. Two subcases need to be analyzed: either $\gamma(X) = U$, or $\gamma(X) = f\, s'$. The difficult subcase is the former. For this, consider $\gamma' = (Y \mapsto W)(U \mapsto f\, W)\gamma$, where W is a fresh variable regarding all variables in the final configuration, which include all variables in $\mathcal{C}$. Thus, the substitutions $\rho_l' = (W \mapsto s)\rho_l$ and $\rho_r' = (W \mapsto t)\rho_r$ are such that $\rho_l'\gamma'(Z) = \texttt{lhs}(eq_Z)$ and $\rho_r'\gamma'(Z) = \texttt{rhs}(eq_Z)$, for all $eq_Z \in \mathcal{C}'_{\texttt{Uns}}$. In particular, $\rho_l'\gamma'(Y) = \rho_l'(W) = (W \mapsto s)(W) = s$, and $\rho_r'\gamma'(Y) = \rho_r'(W) = (W \mapsto t)(W) = t$. Therefore, $(\gamma')|_{\texttt{lbls}(\mathcal{C}'_{\texttt{Uns}})}$ is proved to be a substitution generalization (associated with ρ_l' and ρ_r'), and by the induction hypothesis, it is more general than σ. Thus, there exists δ' such that $(\delta'\gamma' \preceq \sigma)|_{\texttt{lbls}(\mathcal{C}'_{\texttt{Uns}})}$. The next step consists in building a δ for γ such that $(\delta\gamma \preceq \sigma)|_{\texttt{lbls}(\mathcal{C}_{\texttt{Uns}})}$. Notice that $\delta'\gamma' = \delta'(Y \mapsto W)(U \mapsto f\, W)\gamma$. Choose $\delta = \delta'(Y \mapsto W)(U \mapsto f\, W)$. Thus, for any $Z \in \texttt{lbls}(\mathcal{C}_{\texttt{Uns}} \setminus \{X\})$, $\delta\gamma(Z) = \sigma(Z)$. For $Z = X$, $\delta'(Y \mapsto W)(U \mapsto f\, W)\gamma(X) = \delta'(Y \mapsto W)(U \mapsto f\, W)(U) = \delta'(f\, W) = f(\delta'(W)) = f(\delta'\gamma'(Y)) = f(\sigma(Y)) = \sigma(f\, Y) = \sigma(X)$. Therefore, $(\delta\gamma \preceq \sigma)|_{\texttt{lbls}(\mathcal{C}_{\texttt{Uns}})}$.

4 Formalization

This section described how the main concepts and proofs explained in Sections 2 and 3 were formalized in PVS, focusing on how the application of important features of the language allowed to improve the readability of the specification and to reach a high degree of automation in its mechanical verification. In PVS, declarations and proofs are stated in different files. The former must be included in files with extension `.pvs` and the latter are stored by the system in files with extension `.prf`. PVS features an interactive proof environment in which the user applies proof commands guiding the proof engine towards tree derivations in a Gentzen style [24]. It also provides resources to simplify recovering, editing, developing, debugging, maintaining, and presenting proofs. In the following, the symbol ↗ indicates a link to specific parts of the specification.

4.1 Basic Notions: Terms and Substitutions

A deep embedding of syntactic terms is defined as an abstract datatype called `first_order_term` ↗, parametric on arbitrary types representing constant and function symbols (`constant` and `f_symbol`) and a nonempty type for variable symbols (`variable`). From this declaration, the type-checker generates implicit declarations such as axioms for extensionality for each constructor, an axiom for the well-foundedness regarding the proper subterm relation $\ll$, and an inductive scheme.

For the actual definition of the anti-unification algorithm, the variable and constant symbols are represented by natural numbers, as can be seen in the theory `first_order_terms_properties` ↗. Since the terms are finitely sized and hence contain a finite number of variables, considering variables as natural numbers allows easy access to fresh symbols during the application of the anti-unification rules, simply by applying the `epsilon`[8] function on the set of natural numbers not already appearing as variable or label symbols in the equations being processed by the algorithm (see the `freshLabel` ↗ definition).

A basic substitution is denoted by the type `basic_sub` ↗ built from the ordered pair of types `[variable, first_order_term]`. The type `sub` ↗ is defined as a list of basic substitutions, which allows considering the composition between substitutions `delta` and `sigma` just as `append(delta,sigma)`. The action of a basic substitution `sigma_basic` and a substitution `sigma` on a term t are specified as the recursive functions `subs(sigma_basic, t)` ↗ and `subs(sigma)(t)` ↗. Since in PVS a decreasing measure must be explicitly provided along with the specification of a recursive function, the (polymorphic) relation $\ll$ built for the datatypes `first_order_term` and `list` was used as such in the declaration of both functions. All these specifications are in the PVS theory `first_order_substitution` ↗. There, the predicate `nice?` ↗ over substitutions is also specified, along with relevant properties about it. For instance, `nice_disjoint_dom_img` ↗ states that if `nice?(sigma)` holds, then the domain and variables in the range of `sigma` are disjoint . This property fits exactly with the ones of a $\mathcal{C}_{\texttt{Sub}}$ in a given configuration $\mathcal{C}$. Other practical properties of `nice?` substitutions, such as idempotency, are also proven ↗.

These theories also contain several results constituting a solid framework to deal with algebraic properties of these objects in the formalization of the algorithm `Antiunify`. The PVS type-checker generates proof obligations, called type correctness conditions (TCCs), which must be discharged to guarantee type correctness. This implies manually performing an assisted verification of the specified types whenever the system cannot discharge the TCCs automatically. The theories `first_order_terms_properties` and `first_order_substitution` have 119 (including 37 TCCs) and 115 (18 TCCs) proved formulas, respectively.

[8] The `epsilon` function is a generic function defined in the PVS prelude that takes an arbitrary element from a non-empty set.

4.2 Termination

The anti-unification triples from Definition 8 are formalized as objects of the record type `AUT` ↗, with fields of type `variable` and `first_order_term`. Also, the classification of an AUT `eq` as decomposable, trivial, or solved from Definition 9 is represented by the predicates `match_DecF?(eq)` ↗, `match_DecP?(eq)` ↗, `match_Synt?(eq)` ↗, and `match_Sol?(eq)` ↗.

The type `Configuration` ↗ and the predicate `validConfiguration?` ↗ state the notions expressed in the Definition 14. The collections of solved and unsolved equations of a configuration are represented using lists instead of sets to simplify the specification of the inference rules and the analysis regarding termination, correctness, and completeness of the anti-unification algorithm `Antiunify`. This allows the deterministic classification of the derivability of a configuration `c` based on the classification of its first unsolved AUT, expressed by `car(c'unsolved)`. An example of the use of subtyping can be seen in the definition of the Configuration ↗ type, where the substitutions of a configuration are required to be *nice* by fulfill the `nice?` predicate.

The rules (DecF), (DecP) and (Synt) were specified as the function declarations `DecF(c)` ↗, `DecP(c)` ↗, and `Synt(c)` ↗, respectively. The solve rules (SolR) and (SolNR) were integrated into a unique rule `Solve(c)` ↗. To facilitate the automation of the proofs of termination, configuration validity, and preservation of *niceness* of the `Antiunify` algorithm, these properties were encoded in the types of the functions representing the rules. This was possible thanks to the support for dependent types provided by PVS. For instance, consider the type of the function `DecF(c)` ↗. Its input type is defined as `(match_DecF_conf?)`. In PVS, this is syntactic sugar to represent the type of the elements for which the predicate `match_DecF_conf?` holds. Such predicate determines, given a valid configuration `c`, if the first of the AUT in its list of unsolved AUTs, `car(c'unsolved)`, is function-decomposable. On the other hand, the output type denotes those valid configurations `cp`, with the following properties:

1. `cp` has a smaller size than the input configuration `c`,
2. the unsolved AUTs of `c` and `cp` only differ in the first element, and
3. the action of `cp'substitution` over the label in `car(c'unsolved)` results in a function application of the function symbol from the left-hand side of `car(c'unsolved)` over the label in `car(cp'unsolved)`, as expressed in the represented rule.

While these properties between input and output types of the function needed to be manually proven (as TCCs generated by the type checker), these proofs are much more focused and organized than having to prove the same properties directly at the `Antiunify` algorithm level.

The `Antiunify` ↗ algorithm is defined in PVS as a recursive function of type `[(validConfiguration?) -> (validConfiguration?)]`. It takes as input a valid configuration `c` and makes a recursive call on the result of the application of an inference rule to `c` according to its classification. As explained in Section3, this classification is determined directly by the classification of the list of unsolved equations of `c`, i.e., `c‘unsolved`, which in its turn is given by the classification of the first element of such list `car(c‘unsolved)`. The type polymorphism supported by PVS allowed the use of the same name for predicates expressing these properties on single equations and lists of them, namely, `match_DecF?`, `match_DecP?`, `match_Synt?`, and `match_Sol?`. This improved the readibility of the formalization.

A great benefit of restricting the types for the functions representing the inference rules as mentioned above, is that PVS automatically proves `Antiunify`'s termination, since the output type of each inference rule forces the resulting configuration to be smaller than the input. Consequently, termination of `Antiunify` required no additional effort. Similarly, PVS automatically proves that every output of the `Antiunify` algorithm fulfills the `validConfiguration?` predicate.

4.3 Soundness

The soundness of the `Antiunify` algorithm is proven by showing that the substitution component of its output is in fact a generalization of the `unsolved` field of the input configuration. This proof required a series of auxiliary lemmas stating specific preservation properties on the algorithm. Three of these lemmas are highlighted below.

1. `antiunify_sub_preserves_terms` states that if a term `t` is in the range of the substitution of a configuration `c` and its variables are not labels of the AUTs in the unsolved part of `c`, then the action of the final computed substitution `Antiunify(c)'substitution` and the current substitution on `t` is the same: `(Antiunify(c)'substitution)(t) = (c'substitution)(t)`. The proof of this lemma consists of 375 lines and also depends on another classification lemma whose formalization requires 208 lines.
2. `antiunify_dom_sub_preserves_vars_unsolved` states that the domain of the final substitution `Antiunify(c)'substitution` and the variables in `c'unsolved` are disjoint. Its proof consists of 418 lines, and its formalization depends on other preservation lemmas, the previous one, applied twice, and another two, whose formalization required 71 and 208 lines, applied three times and twice, respectively.
3. `antiunify_solved_labels_preserve_vars_unsolved` states that the labels of `Antiunify(c)'solved` are disjoint of the variables in `c'unsolved`. Its proof consists of 276 lines and depends on application of other preservation lemmas, whose proofs required 71 and 276 lines.

Lemma (1) is applied only once to the analysis of the solved rule, more specifically, in the subcase (Solve-Repeated). In this case, the first AUT of `c'unsolved` has the form $\mathtt{s} \underset{\mathtt{X}}{\triangleq} \mathtt{t}$ and is repeated in `c'solved` with another label, say $\mathtt{s} \underset{\mathtt{X'}}{\triangleq} \mathtt{t}$. In the derived configuration, `solve(c)`, the label X' belongs to the range of the substitution but not to the labels of the unsolved part. Therefore, the substitution of the final configuration, `Antiunify(c)`, maps X' into X', according to this lemma. This is the key to infer that the final substitution maps X into X'.

Lemmas (2) and (3) are applied once and twice, respectively, to the case analysis of the inference rule (Syntactic). In this case, the first AUT of `c'unsolved` has the form $\mathtt{s} \underset{\mathtt{X}}{\triangleq} \mathtt{s}$. Lemma (2) guarantees that the final substitution maps s into s as it maps X into s, since variables in terms of `c'unsolved` are not modified by the final substitution. Therefore, the final substitution maps X into s. Lastly, Lemma (3) is used to guarantee that substitutions playing the role as τ_l and τ_r (Definition 12) applied to the image of X by the final substitution result in the term s. It holds since $dom(\tau_l)$ and $dom(\tau_r)$ are both subsets of the labels of the solved part of the final configuration.

No preservation lemma was required to prove the cases related to the rules (DecF) and (DecP), and the case of rule (SolveNR) depends on three very simple preservation lemmas other than those highlighted.

The soundness theorem, `antiunif_is_sound`, was mechanically proven by induction on the size of the input configuration and applying a by-case analysis closely following the discussion in Section 3 on the analytical proof. The PVS proof has 848 lines of length, 64 for the analysis of rule (DecF), 140 for the analysis of rule (DecP), 269 for the (Syntactic) rule, 245 for the (SolveR) and 63 for (SolveNR).

The fact that the analysis of (SolveR) uses once the first highlighted auxiliary lemma (which proof consists of 375 lines) can be seen as an indication of the higher complexity of the analysis of (SolveR) with respect to the cases for (decF), (decP), and (SolveNR). Also, this demonstrates that deleting a repeated AUT and adding to the configuration substitution a basic substitution between variables implies much more considerations than those required in informal proofs in which several assumptions are implicitly applied according to intuition and convenience. Even more surprising is the case of the (Syntactic) rule. It uses once the second and twice the third highlighted auxiliary lemmas above. But also, indirectly the first lemma, used in the proof of the second one. Indeed, the proof of the second lemma depends on another three preservation lemmas. Therefore, what apparently would be the simpler case, i.e., eliminating an AUT with an equation between the same constant, variable or unit, adding a basic substitution from the AUT label to the constant, variable, or unity to the configuration substitution, required much more work than all other cases.

Table 1 compiles quantitative data. It depicts the total number of proved formulas in each one of the three theories, discriminating how many formulas are TCCs. Also, it provides quantitative information on the proof of the theorem `antiunif_is_sound`. As a rough estimation of the effort involved in the formal analysis of each inference rule, the last two columns include the number of lines required for each of them, directly in the formalization of the theorem, and the total number of lines required in the proofs of the applied auxiliary lemmas (related to preservation properties). Using a MacBook Pro Laptop (2.6 GHz 6-Core Intel Core i7 with memory of 16 GB 2667 MHz DDR4), the type checking of the theories developed for this work plus all their dependencies and the rerunning of all the proofs in Table 1 took about 170 seconds. Only 30 out of the 96 TCCs

Table 1. Formalization in numbers.

PVS theory	**Formulas**	**TCCs**	**Inference Rule**	**Proof size (# lines)**	**Dependencies (# lines)**
Terms	119	37	-	-	-
Substitution	115	18			
Anti-unification	116	41	(DecF)	64	-
			(DecP)	140	-
			(Synt)	269	1624
			(SolveR)	245	663
			(SolveNR)	63	111

in the whole development needed human guidance to be proved, the rest were closed by automatic proof strategies assigned by default by PVS.

5 Conclusion

Although anti-unification algorithms are a vital element of widely used systems and frameworks, no report was found on the formal study of anti-unification in interactive proof assistants. Notably, a specification in the Coq proof assistant of the Plotkin's algorithm for computing the least common generalization was used to automatically extract Haskell code from it in the past, but no proofs on the correctness of the algorithm were reported [18, 20].

The lack of formal studies on anti-unification contrasts with the availability of formal developments related to unification algorithms, as those mentioned in the introduction. Although anti-unification and unification may be informally considered dual problems, the verification in this paper clarifies the different amount of effort necessary because of the higher level of complexity introduced in the formal analysis of the standard method of addressing anti-unification by using fresh variables while guaranteeing that the substitution generalization being built incrementally does not modify the elements of the input problem.

The formalized algorithm follows a straight-forward application of the inference rules presented in Figure 1. In that sense, practical aspects, such as temporal and space efficiency, were minimally considered when formally describing the anti-unification procedure. Naturally, this leaves space for improvement on several fronts. In particular, simple modifications could be easily implemented; for instance, the process to decide which rule is applicable can be implemented as just one matching function. Even though, having a simple version of the anti-unification algorithm formally verified will undoubtedly be useful in further verification efforts, focused on more sophisticated and efficient generalization algorithms. For instance, PVS provides code extraction features, which allows to generate executable Lisp code from ground PVS expressions. Additionally, the PVSIO framework [21], adds input-output capabilities. Leveraging these features, it would be possible to apply differential testing to validate advanced implementations of anti-unification procedures using the formalization presented in this paper as an oracle, following the MINERVA approach [22].

The work presented in this paper makes good use of advanced features provided by the PVS language, such as predicate sub-typing and dependent types. These features allowed decoupling otherwise big proofs in localized smaller efforts along the specification. This formalization can be seen as an example for the application of such an approach, which can reduce the workload in the development of complex specifications.

Future work includes formally verifying other important properties of the presented algorithm, notably its completeness, which is informally sketched in Section 3. Additionally, further extensions of the anti-unification rules from Section 2 will be explored. For instance, the changes needed to support nominal syntax modulo combinations of equational theories as commutative, associative, idempotent, absorptive, and others (e.g., [2,4,13]). Also, the use of this formalization to validate the correctness of existing implementations of anti-unification algorithms will be explored. Of course, from the authors' experience in formalizing different unification algorithms, it is not expected that such extensions will be straightforwardly obtained as effortless exercises for several reasons. For the sake of illustration, the reader is invited to note that minor additions to the theory, such as giving equational properties to the function symbols, change the complexity and the anti-unification kind of the problem [15]. Additionally, even minor algorithmic improvements have an impact on existent proofs.

Acknowledgments. This work was partially supported by the NASA's System Wide Safety Project, under the RSES contract 80LARC23DA003; the Austrian Science Fund (FWF) project P 35530; the Goiás Research Foundation FAPEG, process 202310267000223; the Brazilian Research Council CNPq, Grant Universal 409003/21-2, and Research Grant 313290/21-0; and the Brazilian Higher Education Council (CAPES) through the PrInt Program, finance code 001.

References

1. María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014.
2. María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. Order-sorted equational generalization algorithm revisited. *Ann. Math. Artif. Intell.*, 90(5):499–522, 2022.
3. Andréia Borges Avelar, André Luiz Galdino, Flávio Leonardo Cavalcanti de Moura, and Mauricio Ayala-Rincón. First-order unification in the PVS proof assistant. *Log. J. IGPL*, 22(5):758–789, 2014.
4. Mauricio Ayala-Rincón, David M. Cerna, Andres Felipe Gonzalez Barragan, and Temur Kutsia. Equational anti-unification over absorption theories. In *Automated Reasoning - 12th Int. Joint Conference, IJCAR, Proc., Part II*, volume 14740 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2024.
5. Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Daniele Nantes-Sobrinho, and Ana Cristina Rocha Oliveira. A formalisation of nominal α-equivalence with A, C, and AC function symbols. *Theor. Comput. Sci.*, 781:3–23, 2019.

6. Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. Formalising nominal C-unification generalised with protected variables. *Math. Struct. Comput. Sci.*, 31(3):286–311, 2021.
7. Mauricio Ayala-Rincón, Maribel Fernández, and Ana Cristina Rocha Oliveira. Completeness in PVS of a nominal unification algorithm. In *Proc. of the Tenth Workshop on Logical and Semantic Frameworks, with Appl., LSFA*, volume 323 of *Electronic Notes in Theor. Comput. Sci.*, pages 57–74. Elsevier, 2015.
8. Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, Temur Kutsia, and Daniele Nantes-Sobrinho. Certified First-Order AC-Unification and Applications. *J. Autom. Reason.*, 68(25):1–48, 2024.
9. Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, Temur Kutsia, and Daniele Nantes-Sobrinho. Certified first-order ac-unification and applications. *J. Autom. Reason.*, 68(4):25, 2024.
10. Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019.
11. Adam D. Barwell, Christopher Brown, and Kevin Hammond. Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions via anti-unification. *Future Gener. Comput. Syst.*, 79:669–686, 2018.
12. 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 Int. Andrei Ershov Memorial Conference, PSI. Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009.
13. David M. Cerna and Temur Kutsia. Idempotent anti-unification. *ACM Trans. Comput. Log.*, 21(2):10:1–10:32, 2020.
14. David M. Cerna and Temur Kutsia. Unital anti-unification: Type and algorithms. In *5th Int. Conference on Formal Structures for Computation and Deduction FSCD*, volume 167 of *LIPIcs*, pages 26:1–26:20, 2020.
15. David M. Cerna and Temur Kutsia. Anti-unification and generalization: A survey. In *Proc. of the 32nd Int. Joint Conference on Artificial Intelligence, IJCAI*, pages 6563–6573. ijcai.org, 2023.
16. 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, SBES*, pages 74–83. ACM, 2021.
17. Boris Galitsky. *Developing Enterprise Chatbots - Learning Linguistic Structures*. Springer, 2019.
18. Adelaine Gelain, Cristiano D. Vasconcellos, Carlos Camarão, and Rodrigo Geraldo Ribeiro. Type Inference for GADTs and Anti-unification. In *Programming Languages - 19th Brazilian Symposium SBLP*, volume 9325 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2015.
19. Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Shekhar 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, NSDI 2020*, pages 435–448. USENIX Association, 2020.
20. Gabriela Moreira, Cristiano D. Vasconcellos, and Rodrigo Geraldo Ribeiro. Type inference for GADTs, outsidein and anti-unification. In *Proc. of the XXII Brazilian Symposium on Programming Languages, SBLP*, pages 51–58. ACM, 2018.
21. César A. Muñoz. Rapid Prototyping in PVS. NASA/CR-2003-212418, NIA Report 2003-03, NASA, 2003.

22. Anthony Narkawicz, César Muñoz, and Aaron Dutle. The MINERVA software development process. In *Automated Formal Methods*, volume 5 of *Kalpa Publications in Computing*, pages 93–108. EasyChair, 2018.
23. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proc. of the 11th Int. Conference on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.
24. Sam Owre and Natarajan Shankar. The Formal Semantics of PVS. Technical Report CSL-97-2R, SRI International Computer Science Laboratory, 1999.
25. Gordon D. Plotkin. A note on inductive generalization. *Machine Intell.*, 5(1):153–163, 1970.
26. Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Programming by example using least general generalizations. In *Proc. of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 283–290. AAAI Press, 2014.
27. John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intell.*, 5(1):135–151, 1970.
28. Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.
29. Wim Vanhoof and Gonzague Yernaux. Generalization-driven semantic clone detection in CLP. In *Logic-Based Program Synthesis and Transformation - 29th Int. Symposium, LOPSTR, Revised Selected Papers*, volume 12042 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2019.
30. 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 *Proc. of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, pages 1578–1588. ACM, 2022.
31. Kaiyu Yang and Jia Deng. Learning symbolic rules for reasoning in quasi-natural language. *Trans. Mach. Learn. Res.*, 2023.