# A Coq Formalization of Unification Modulo Exclusive-Or

by

Yichi Xu

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2023

APPROVED:

Professor Dan Dougherty, Thesis Advisor

Professor Craig A Shue, Head of Department

# Contents

**Abstract**

*Equational Unification* is a critical problem in many areas such as automated theorem proving and security protocol analysis. In this paper, we focus on XOR-Unification, that is, unification modulo the theory of exclusive-or. This theory contains a function with the properties Associativity, Commutativity, Nilpotency, and the presence of an identity. In the proof assistant Coq, we implement an algorithm inspired by Liu and Lynch's inference rules and prove it sound, complete, and terminating. Using Coq's code extraction capability one obtains an implementation in the programming language Ocaml.

# 1 Introduction

Proof assistants are computer programs that enable users to do mathematical reasoning on a computer. Unlike many computer programs which focus on computing numerical or symbolical aspects, proof assistant focuses more on *Proving* and *Defining*. So in proof assistants, users can define properties and do logical reasoning about any function. Proof assistants are often used by people who formalize mathematical theories and prove theorems. Proof assistants are especially useful for some proofs people are unsure about: if the proof can be adopted in a proof assistant, then it is correct [10].

Unification is the process of solving equations between symbolic expressions, i.e. finding appropriate substitutions for variables such that the equation is satisfied. E-Unification is unification with some identities E, e.g. Commutativity or Associativity. Exclusive-or (XOR) is a well-known algebraic operation that arises in many applications; the axioms are given later in Section 3.1.

The symbolic model is an important topic in security protocol analysis. A common way to analyze protocols is to perform syntactic unification with the protocol rules to explore the reachable states. If an attack state is reachable from the initial state then an attack exists and the protocol is flawed. However, the limit of using syntactic unification to analyze protocol is that it only captures the case when terms, representing messages, are exactly the same, which in many protocols it is not enough. For example, Vernam cipher and cipher-block chaining mode for block ciphers rely on XOR [15]. Some protocols can be proved to be secure when we consider the XOR as a free operator but are flawed otherwise. For example, the original version of Bull's recursive authentication protocol was formally proved correct in the Dolev-Yao model, but the protocol used XOR for encryption and was thus vulnerable to an attack that exploited the self-cancellation property [20]. Therefore, XOR unification is important because it will help analysis calculate a more precise reachability, which moreover will provide a more accurate analysis of the protocols using xor properties.

In this thesis, we adopt a modified version of the algorithm developed by Lynch and Liu [12], and implement it and prove the algorithm correct in Coq. Then we can use Coq to automatically generate the certified Ocaml code for XOR unification algorithms. The algorithm of Liu and Lynch works over a signature with variables, constants, and uninterpreted function symbols. Here we work with the sub-signature with variables and constants, omitting uninterpreted function symbols of arity greater than 0. Incorporating unlimited uninterpreted function symbols is a good topic for future work.

## 1.1 Contributions

There are relatively few formalizations of unification algorithms beyond syntactic unification. We make the following contributions:

- We develop a fully verified implementation of the XOR unification algorithm in Coq.

- Using Coq's extraction mechanism we can extract code for the conventional programming language Ocaml, which is guaranteed to be correct and terminating.

- We develop a new data structure representing terms (if Associativity and Commutativity are in the identity E), and prove it correct.

- We develop a rewrite system so that all equivalence terms are syntactically equal in their reduced form, and prove it correct.

## 1.2 Related Work

There are a variety of proof assistants, a sample includes Coq [5], Isabelle [16], Lean [8] and PVS [17]. The *Archive of Formal Proofs* is "a collection of proof libraries, examples, and larger scientific developments, mechanically checked in the theorem prover Isabelle." [6]. The Archive currently only presents one first-order unification formalization; there are not any treatments of equational unification at the time of this writing.

*Syntactic* unification is unification modulo the empty equational theory. There are many algorithms for syntactic unification, but there are only a few which have been verified and formalized. The earliest formalization is the algorithm from Manna and Waldinger [13] and it is proved by Paulson [18] using LCF. This formalization is used as a basis for later researchers Coen, Slind, and Krauss[21] of the same in Isabelle. Urban, Pitts, and Gabbay [23] also formalized first-order unification in Isabelle. A relatively recent formalization for syntactic unification is from Avelar, Galdino, deMoura and Ayala-Rincon [1] using PVS .

*E*-unification is unification modulo an equational theory. Dougherty [9] has verified two algorithms for boolean unification algorithm. Ayala-Rincón *et.al.* [2] have verified an AC-Unification algorithm using PVS. For XOR unification, there are only a few algorithms but no formalization. Tuengethal, Kusters and Turuani [22] mentioned a relatively easy and intuitive way to design such an algorithm by combining theories such that their overall output satisfies the XOR properties. Guo, Narendran, and Wolfram [11] mentioned using Gaussian elimination over a boolean ring to compute unifiers for XOR unification. Liu and Lynch [12] give several terminating inference rules so the unification problems can reach a solved form if they are solvable. However, the above papers only give algorithms but not a formalization. We decided to do a formalization for a subtheory of the theory that Liu and Lynch's algorithm treats; they consider homomorphism functions and uninterpreted functions and we did not.

# 2 Preliminaries

In this chapter, we cover the basic background necessary to understand the concepts of XOR-Unification. And the following discussion is mainly based on the book: Term rewriting and all that[4].

## 2.1 Terms

Here we introduce the basic definition will be used in the later sections. The following are standard notations and definitions.

We use $\mathbb{V}$ to denote the set of variables. A *signature* $\Sigma$ is a set of function symbols where each $f \in \Sigma$ is associated with a non-negative integer $n$ representing the arity of the function $f$, i.e. the number of parameters of the function $f$. Function $f \in \Sigma$ is allowed to have arity of 0 where in this case function $f$ represents a constant. A *Term* will be built from function symbols $\Sigma$ and variables $\mathbb{V}$. A variable itself is a term, a 0-arity function symbol is a term, and a combination of functions and variables is a term. We use $\mathbb{T}(\Sigma, \mathbb{V})$ to denote term.

**Example 1** *If $x$,$y$ are variables, and $f$ is a binary function, $g$ is a 0-arity function then:*

- *$x$ and $y$ are two terms.*
- *$g()$ is a term*
- *$f(x, f(y, g()))$ is a term.*

In any term, we use $Vars(t)$ to denote the set of variables occurring in t.

**Example 2** *Let term $t = f(x, y, g(z, z))$ then $Vars(t) = \{x, y, z\}$.*

In any term, we use $\Sigma(t)$ to denote the set of function symbols occurring in t.

**Example 3** *Let term $t = f(x, y, g(z, z))$ then $\Sigma(t) = \{f, g\}$.*

## 2.2 Substitution

A *substitution* $\sigma$ is a homomorphism function $\sigma : \mathbb{V} \to \mathbb{T}(\Sigma, \mathbb{V})$ such that $\sigma(x) \neq x$ for only finite many $x$.

**Example 4** *Let substitution* $\sigma = \{x \mapsto y, z \mapsto f(x)\}$ *then:*

- $\sigma(x) = y$.
- $\sigma(y) = y$.
- $\sigma(z) = f(x)$.
- $\sigma(g(x, z)) = g(\sigma(x), \sigma(z) = g(y, f(x))$.

The *domain* of a substitution is $\sigma : Dom(\sigma) := \{x \in \mathbb{V} | \sigma(x) \neq x\}$. The *range* of a substitution is $\sigma : Ran(\sigma) := \{\sigma(x) | x \in Dom(\sigma)\}$. The *variable range* of a substitution is $\sigma : VRan(\sigma) := \bigcup_{x \in Dom(\sigma)} Var(\sigma(x))$.

**Example 5** *Let substitution* $\sigma = \{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_i \mapsto t_i\}$ *then:*

- $Dom(\sigma) := \{x_1, x_2, \ldots, x_i\}$.
- $Ran(\sigma) := \{t_1, t_2, \ldots, t_i\}$.
- $VRan(\sigma) := Var(t_1) \bigcup Var(t_2) \bigcup, \ldots, \bigcup Var(t_i)$.

A substitution $\sigma$ is *more general* than a substitution $\sigma'$ if there is a substitution $\delta$ such that $\sigma' \approx \delta\sigma$ for all variables in $\mathbb{V}$. In this case, we write $\sigma \lesssim \sigma'$. This definition will be important when we discuss unification.

**Example 6** *Let substitution* $\sigma = \{x \mapsto f(y)\}$*,* $\sigma' = \{x \mapsto f(a), y \mapsto a\}$*. Then* $\sigma \lesssim \sigma'$ *because there exists a substitution* $\delta = \{y \mapsto a\}$ *such that* $\sigma \approx \delta\sigma'$*. We can check that* $\delta\sigma \approx \sigma'$ *by:*

- $\delta\sigma(x) \approx \delta(f(y)) \approx f(a)$, $\sigma'(x) \approx f(a)$.
- $\delta\sigma(y) \approx \delta(y) = a$, $\sigma'(y) = a$.
- *No change on any other variables.*

## 2.3 Equational Theories

Equational theories are used to describe the properties of algebraic structures and to define the rules for manipulating expressions built from these structures. The equations in an equational theory specify how expressions can be transformed into other expressions that are equivalent according to the given theory. Here we first give the definition for $\Sigma$-identities:

Let $\Sigma$ be a signature and $\mathbb{V}$ a countably infinite set of variables disjoint from $\Sigma$. A $\Sigma$-Identity (or simply identity) is a pair $(s, t) \in T(\Sigma, \mathbb{V}) \times T(\Sigma, \mathbb{V})$.

Let E be a set of $\Sigma$-identities. The relation $\approx_E$ is the smallest equivalence relation on $T(\Sigma, \mathbb{V})$ that contains E and is closed under substitutions.

**Example 7** *Let identities:* $E = \emptyset$, $C = \{f(x, y) = f(y, x)\}$*:*

- $f(x) \approx_E f(x)$.
- $f(y, x) \not\approx_E f(x, y)$.
- $f(y, x) \approx_C f(x, y)$.

A substitution $\sigma$ is more general modulo $\approx_E$ than a substitution $\sigma'$ if there is a substitution $\delta$ such that $\sigma' \approx_E \delta\sigma$ for all variables in $\mathbb{V}$. In this case, we write $\sigma \lesssim_E \sigma'$.

**Example 8** *Let substitution* $\sigma = \{x \mapsto f(y_1, y_2)\}$, $\sigma' = \{x \mapsto f(y_2, y_1), z \mapsto a\}$. *Let* $C = \{f(x, y) \approx f(y, x)\}$. *Then* $\sigma \lesssim \sigma'$ *because there exists a substitution* $\delta = \{z \mapsto a\}$ *such that* $\sigma \approx_C \delta\sigma'$. *We can check that* $\delta\sigma \approx_C \sigma'$ *by:*

- $\delta\sigma(x) \approx_C \delta(f(y_1, y_2)) \approx_C f(y_1, y_2)$, $\sigma'(x) \approx_C f(y_2, y_1)$. *And* $f(y_1, y_2) \approx_C f(y_2, y_1)$.
- $\delta\sigma(z) \approx_C \delta(z) \approx_C a$, $\sigma'(z) \approx a$.
- *No change on any other variables.*

## 2.4 Unification

A Unification problem is a finite set of equations:

$$S = \{s_1 \approx_E^? t_1, s_2 \approx_E^? t_2, \ldots, s_n \approx_E^? t_n, \} \tag{1}$$

Unification[19] is the process of solving the satisfiability problem: Given identity $E$, s and t, find a substitution $\sigma$ such that $\sigma(s) \approx_E \sigma(t)$, which we trying to achieve:

$$S = \{\sigma(s_1) \approx_E \sigma(t_1), \sigma(s_2) \approx_E \sigma(t_2), \ldots, \sigma(s_n) \approx_E \sigma(t_n), \} \tag{2}$$

A unifier or solution of S is a substitution $\sigma$ such that $\sigma(s_i) \approx_E \sigma(t_i)$. $\mathbb{U}(S)$ denotes the set of all unifiers of S. S is unifiable if $\mathbb{U}(S) \neq \emptyset$. A substitution $\sigma$ is a most general unifier (mgu) of S if $\sigma \in \mathbb{U}(S) \wedge \forall \sigma' \in \mathbb{U}(S), \sigma \lesssim \sigma'$. In the theory we are dealing with, there is only one unique unifier.

### 2.4.1 Syntactic Unification

If $E = \emptyset$ then the problem becomes syntactic unification where we are trying to find a substitution $\sigma$ such that $\sigma(s) \approx \sigma(t)$, because $s \approx_E t \leftrightarrow s \approx t$. Hence the Unification problem becomes:

$$S = \{s_1 \approx^? t_1, s_2 \approx^? t_2, \ldots, s_n \approx^? t_n, \} \tag{3}$$

Here is an example of a syntactic unification problem

**Example 9** *Given identities:* $E = \emptyset$*, unification problem:*$S = \{x \approx_E z, y \approx_E f(x,z)\}$ *or* $S = \{x \approx z, y \approx f(x,z)\}$*:*

- $\sigma = \{x \mapsto a, z \mapsto a, y \mapsto f(a,a)\}$ *is a unifier of S*
- $\sigma' = \{x \mapsto z, y \mapsto f(z,z)\}$ *is a unifier of S.*
- $\sigma' \lesssim_E \sigma$*.*
- *In fact* $\sigma'$ *is the mgu of S.*

### 2.4.2 Equational Unification

If $E \neq \emptyset$ then the problem is referred to as equational unification, where in addition to syntactic unification, we have to consider the identities it carries.

Here is an example of an equational unification problem:

**Example 10** *Let the unification problem* $S' = \{f(x,y) =^?_C f(a,b)\}$*, where* $C = \{f(x,y) \approx f(y,x)\}$ *i.e. commutative:*

- $\delta = \{x \mapsto a, y \mapsto b\}$ *is a unifier of S'.*
- $\delta' = \{x \mapsto b, y \mapsto a\}$ *is a unifier of S.*
- $\delta' \not\lesssim_C \delta$*.*
- $\delta \not\lesssim_C \delta'$*.*

Noticed the $\delta$ and $\delta'$ is not comparable, but they are both unifiers of the original problems. It can be shown that both $\delta$ and $\delta'$ are minimal, therefore there are no most general unifier in this problem. Here we introduce the complete set for E-unification problems.

A *complete set* of E-unifiers of S is a set of substitutions $C$ Satisfies:

- each $\sigma \in C$ is an E-unifiers of S
- $\forall \theta \in U(S)$ there exists $\sigma \in C$ such that $\sigma \approx_E \theta$

A *minimal complete set* of E-unifiers is a complete set of E-unifiers C that satisfies the additional condition: $\forall \sigma, \sigma' \in C$,$\sigma \lesssim_E \sigma'$ implies $\sigma = \sigma'$.

**Example 11** *Let the unification problem $S' = \{f(x, y) =^?_C f(a, b)\}$, where $C = \{f(x, y) = f(y, x)\}$ i.e. commutative:*

- $\delta = \{x \mapsto a, y \mapsto b\}$ *is a unifier of S'.*
- $\delta' = \{x \mapsto b, y \mapsto a\}$ *is a unifier of S.*
- $\{\delta, \delta'\}$ *is a complete set of problems S.*
- *It can be shown that* $\{\delta, \delta'\}$ *is actually the minimal complete set of problems S.*

### 2.4.3 Undecidability of Unification

While many unification problems can be solved efficiently, some are proven to be undecidable. This means that there is no algorithm that can always determine whether a solution exists for these problems or not. There are many undecidable unification problems. Examples include unification modulo theories such as a group [3] or a ring of polynomials over a field [7], as well as various forms of higher-order unification [14]. These undecidable problems demonstrate the inherent limitations of automated reasoning and highlight the importance of identifying and restricting the classes of problems that can be solved effectively by computer algorithms[4].

# 3 The theory of Exclusive Or

The following chapter details the algorithm discussed in the introduction.

## 3.1 Axioms for XOR

Here are the formal axioms for XOR where the signature is $\Sigma = \{\oplus, 0\}$:

- Associativity: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
- Commutativity: $x \oplus y = y \oplus x$
- Unity: $x \oplus 0 = x$
- Nilpotency: $x \oplus x = 0$

For this equational theory, there exists a confluent and terminating rewrite system in which every term has unique normal forms, confluent means that if two terms are equivalence then their normal form are equal. We will explain and prove the rewrite system in detail in later sections.

## 3.2 XOR-unification algorithm

The notations we used in this chapter are a simpler version of Liu and Lynch's paper because they are dealing with a richer theory.

In this development, we use $\Gamma \| \Lambda$ to indicate our system, $\Gamma$ denotes the unification problem consisting of a set of equations $\{S \approx^?_E 0\}$, because of the Nilpotency, we can move the term from the right-hand side to the left-hand side without losing equivalency. $\Lambda$ denotes a set of equations in solved form. Initially, the unification is stored in $\Gamma$, while $\Lambda$ remains empty. If a system is in the normal form regarding these inference rules, then $\Lambda$ is in solved form if the original problem is solvable.

A solved form means that if all left-hand sides ($s_i$) are pairwise distinct variables and none of which occurs in any of the right-hand sides ($t_i$).

$$S = \{s_1 \approx^? t_1, s_2 \approx^? t_2, \ldots, s_n \approx^? t_n, \} \tag{4}$$

And we define the substitution extraction from problem S as below, note that it has to be in the solved form:

$$\overrightarrow{S} = \{s_1 \mapsto t_1, s_2 \mapsto t_2, \ldots, s_n \mapsto t_n, \} \tag{5}$$

Our inference rules are listed below.

**Trivial**

$$\frac{\Gamma \cup \{0 \approx^?_E 0\} \| \Lambda}{\Gamma \| \Lambda} \tag{6}$$

This inference rule seeks a problem that is already balanced and deletes it.

**Variable Substitution**

$$\frac{\Gamma \cup \{x \oplus S \approx^?_E 0\} \| \Lambda}{\sigma\Gamma \| \sigma\Lambda \cup \{x \approx^?_E S\}} \tag{7}$$

where $\sigma = x \mapsto S$.

This inference rule seeks a problem that contains a variable, moves everything else to the right-hand side of the problem, and applies its corresponding substitution to the whole system.

Therefore, for this thesis, we need to prove this set of inference rules correct. Correct here means it will return an idempotent of mgu (most general unifier) of the original problem if it is solvable. An idempotent substitution is a substitution that gives the same result whether it is applied once or multiple times.

### 3.2.1 Termination

Repeated application of these two inference rules guarantees termination, as every time either rule is applied, the length of $\Gamma$ is reduced by one. Eventually, $\Gamma$ becomes empty or irreducible with respect to the two inference rules, leading to termination.

### 3.2.2 Unifiers preserves through Inference Rules

Here we will show that after applying either rule, the set of unifiers will not change.

**Theorem 12** *Let* $\Gamma \| \Lambda$ *and* $\Gamma' \| \Lambda'$ *be two systems satisfying* $\Gamma \| \Lambda \Rightarrow^*_{Inf} \Gamma' \| \Lambda'$*. Then* $\mathbb{U}(\Gamma \cup \Lambda) = \mathbb{U}(\Gamma' \cup \Lambda')$

Recall that $\mathbb{U}$ stands for the set of all unifiers. Notation $\Rightarrow^*_{Inf}$ means that apply either inference rule any number of times.

In the following section, we will develop some lemma that helps to prove this theorem.

**Lemma 13** *Let* $\Gamma \| \Lambda$ *and* $\Gamma' \| \Lambda'$ *be two systems satisfying* $\Gamma \| \Lambda \Rightarrow_{Trivial} \Gamma' \| \Lambda'$*. Then* $\mathbb{U}(\Gamma \cup \Lambda) = \mathbb{U}(\Gamma' \cup \Lambda')$

*Proof.* Because for all substitution $\sigma$, $\sigma\{0 =^? 0\} = \{0 =^? 0\}$, therefore $\{0 =^? 0\}$ does not affect the solutions, i.e., $\mathbb{U}(\Gamma \cup \Lambda) = \mathbb{U}(\Gamma' \cup \Lambda')$

**Corollary 14** *Let* $\Gamma \| \Lambda$ *and* $\Gamma' \| \Lambda'$ *be two systems satisfying* $\Gamma \| \Lambda \Rightarrow^*_{Trivial} \Gamma' \| \Lambda'$*. Then* $\mathbb{U}(\Gamma \cup \Lambda) = \mathbb{U}(\Gamma' \cup \Lambda')$

*Proof.* Immediate

**Lemma 15** *Let* $\Gamma \| \Lambda$ *and* $\Gamma' \| \Lambda'$ *be two systems satisfying* $\Gamma \| \Lambda \Rightarrow_{VariableSubstitution} \Gamma' \| \Lambda'$*. Then* $\mathbb{U}(\Gamma \cup \Lambda) = \mathbb{U}(\Gamma' \cup \Lambda')$

*Proof.* Recall the inference rule for Variable substitution:

$$\frac{\Gamma \cup \{x \oplus S \approx_E^? 0\} \| \Lambda}{\sigma\Gamma \| \sigma\Lambda \cup \{x \approx_E^? S\}} \tag{8}$$

i.e.

$$\frac{\Gamma \cup \{x \oplus S \approx_E^? 0\} \| \Lambda}{\sigma\Gamma \| \sigma\Lambda \cup \{x \oplus S \approx_E^? 0\}} \tag{9}$$

We want to show that $\mathbb{U}(\Gamma \cup \{x \oplus S \approx_E^? 0\} \cup \Lambda) = \mathbb{U}(\sigma\Gamma \cup \sigma\Lambda \cup \{x \approx_E^? S\})$. We start with $(\Gamma \cup \{x \oplus S \approx_E^? 0\} \cup \Lambda)$ and consider arbitrary substitution $\theta \in \mathbb{U}(\Gamma \cup \{x \oplus S =^? 0\} \cup \Lambda)$, and $\sigma = \{x \mapsto S\}$:

$$\begin{aligned}
&\theta \in \mathbb{U}(\Gamma \cup \{x \oplus S \approx_E^? 0\} \cup \Lambda) \\
\leftrightarrow\ &\theta \in \mathbb{U}(\Gamma \cup \Lambda) \cup \theta\{x \oplus S \approx_E 0\} \\
\leftrightarrow\ &\theta \in \mathbb{U}(\Gamma \cup \Lambda) \\
\leftrightarrow\ &\theta\sigma \in \mathbb{U}(\Gamma) \cup \theta\sigma \in \mathbb{U}(\Lambda) \\
\leftrightarrow\ &\theta \in \mathbb{U}(\sigma\Gamma) \cup \theta \in \mathbb{U}(\sigma\Lambda) \\
\leftrightarrow\ &\theta \in \mathbb{U}(\sigma\Gamma) \cup \theta \in \mathbb{U}(\sigma\Lambda \cup \{x \oplus S \approx_E 0\}) \\
\leftrightarrow\ &\theta \in \mathbb{U}(\sigma\Gamma \cup \sigma\Lambda \cup \{x \oplus S \approx_E^? 0\})
\end{aligned} \tag{10}$$

Hence proved.

**Corollary 16** *Let* $\Gamma \| \Lambda$ *and* $\Gamma' \| \Lambda'$ *be two systems satisfying* $\Gamma \| \Lambda \Rightarrow^*_{VariableSubstitution} \Gamma' \| \Lambda'$. *Then* $\mathbb{U}(\Gamma \cup \Lambda) = \mathbb{U}(\Gamma' \cup \Lambda')$

*Proof.* Immediate.

**Theorem 17** *Let* $\Gamma \| \Lambda$ *and* $\Gamma' \| \Lambda'$ *be two systems satisfying* $\Gamma \| \Lambda \Rightarrow^*_{Inf} \Gamma' \| \Lambda'$. *Then* $\mathbb{U}(\Gamma \cup \Lambda) = \mathbb{U}(\Gamma' \cup \Lambda')$

*Proof.* Immediate by corollary 13 and corollary 15.

#### 3.2.3 Reduced Form to Solved Form

**Lemma 18** *Let $\Gamma \| \Lambda$ and $\Gamma' \| \Lambda'$ be two systems satisfying $\Gamma \| \Lambda \Rightarrow^*_{Inf} \Gamma' \| \Lambda'$, if the variables in the left-hand side of $\Lambda$ are disjoint from all the variables in $\Gamma$, then the variables in the left-hand side of $\Lambda'$ are also disjoint from all the variables in $\Gamma'$.*

**Proof.** Consider the inference rules for our unification algorithm. Since the Trivial rule does not change anything, we can omit it from this part of the proof. Now let's consider the Variable Substitution rule.

For proving purposes, We introduce a new notation: $lhs()$ and $LHS()$. The $lhs()$ function takes an equation as input and returns its left-hand side term as output. The $LHS()$ function takes a list of equations or a unification problem as input and returns a list of terms that are the $lhs()$ of all the equations.

The only way a new variable gets added to $LHS(\Lambda')$ is through:

1. The substitution of a variable in $\Gamma'$, or
2. The substitution of a variable in $LHS(\Lambda)$, or
3. The application of the rule Variable Substitution itself.

In case (1), the variable in $\Gamma'$ would already be disjoint from all variables in $\Gamma$ since $\Gamma | \Lambda$ implies that the variables in $\Gamma$ and $LHS(\Lambda)$ are disjoint.

In case (2), we know that the variables in $LHS(\Lambda)$ are disjoint from all variables in $\Gamma$ by assumption. Therefore, the variable substitution would not affect the disjointness of variables in $\Gamma$ and $LHS(\Lambda')$.

In case (3), the variable being substituted is either not in $\Gamma$ or it is in $LHS(\Lambda)$. If the variable is not in $\Gamma$, then it would not affect the disjointness of variables in $\Gamma$ and $LHS(\Lambda')$. If the variable is in $LHS(\Lambda)$, then the same argument as in case (2) applies.

Therefore, we have shown that the variables in the left-hand side of $\Lambda'$ are disjoint from all variables in $\Gamma'$ if they are disjoint from all variables in $\Gamma$ and $LHS(\Lambda)$.

**Theorem 19** *Let $\Gamma \| \Lambda$ and $\Gamma' \| \Lambda'$ be two system satisfying $\Gamma \| \Lambda \Rightarrow^*_{Inf} \Gamma' \| \Lambda'$. if $\Lambda$ is in solved form then $\Lambda'$ is in solved form.*

**Proof.** To prove this result, we first observe that inference rule Trivial does not affect $\Lambda'$ and can therefore be omitted from this part of the proof. Regarding inference rule Variable Substitution, we use induction on the number of steps to show that $\Lambda'$ is in solved form, given the assumption that $\Lambda$ is in solved form.

For the base case, we note that $\Lambda'$ does not change, and since $\Lambda$ is in solved form by assumption, the base case is trivially proven. For the inductive case, assuming that $\Lambda'$ is in solved form, we want to show that $\sigma\Lambda' \cup x \approx^?_E S$, where $\sigma = x \mapsto S$, is also in solved form.

To prove that the variables in the left-hand side are pairwise distinct, we use Lemma 17, which shows that the variables in $\Gamma$ are disjoint from those in $\Lambda$, and hence every newly added variable through Variable Substitution is also disjoint from the variables in $LHS(\Lambda')$.

To prove that the left-hand side does not occur on the right-hand side, we again use Lemma 17, which shows that the variables in $\Gamma$ are disjoint from those in $\Lambda$, and hence every newly added S and $LHS(\Lambda')$ are also pairwise disjoint.

Combining these arguments, we conclude that $\Lambda'$ is always in solved form, given that $\Lambda$ is in solved form.

#### 3.2.4 Solved Form to Substitution

We aim to demonstrate that if a solved form S is given, then $\overrightarrow{S}$ is always an idempotent substitution and a most general unifier (mgu) of S.

To prove that $\overrightarrow{S}$ is idempotent, we need to show that it satisfies the definition of solved form, which implies that the left-hand side does not occur on the right-hand side, and therefore this substitution is idempotent.

To prove that $\overrightarrow{S}$ is an mgu of S, we consider two cases. First, assume that $\sigma$ is a unifier of S, and we want to show that $\sigma$ and $\sigma\overrightarrow{S}$ behave the same on all the variables in $\mathbb{V}$, which means that $\sigma x = \sigma\overrightarrow{S}x$ for all $x \in \mathbb{V}$.

Consider an arbitrary variable x. If $x \in LHS(S)$, then $\sigma x = \sigma t$, where t is the right-hand side of $x$ in $S$ because $\sigma$ is a unifier of S. Furthermore, since $\overrightarrow{S}$ replaces $x$ with its corresponding right-hand side, $\sigma t = \overrightarrow{S}x$, and hence $\sigma x = \overrightarrow{S}x$. If $x \notin LHS(S)$, then $\sigma x = x$ and $\overrightarrow{S}x = x$, which again implies that $\sigma x = \overrightarrow{S}x$.

Therefore, we have shown that $\overrightarrow{S}$ is an mgu of S and an idempotent substitution.

#### 3.2.5 Solvablility

**Theorem 20** *Given the Unification problems S. if the system* $S\|\{\}$ *have an irreducible form* $\Gamma\|\Lambda$ *where* $\Gamma$ *is not empty. Then S is not solvable.*

**Proof.** Consider the first problem in $\Gamma$. If it is irreducible then there exist no unifiers that unify some problem in $\Gamma$, meanwhile, according to theorem 11, unifiers are the same during transformation. If there exists no unifiers for $\Gamma\|\Lambda$, then there exists no unifiers for $S\|\{\}$ or $S$. i.e. S is not solvable. Hence Proved.

**Theorem 21** *Given the Unification problems S. if the system* $S\|\{\}$ *have an irreducible form* $\Gamma\|\Lambda$ *where* $\Gamma$ *is empty. Then S is not solvable.*

An empty set $\{\}$ is in solved form, from theorem 18, we know that $\Lambda$ is in solved form. And from 3.2.4. We know that $\overrightarrow{\Lambda}$ solves $\Lambda$ or $\Gamma\|\Lambda$ because $\Gamma$ is empty. By Theorem 11, we know that $\overrightarrow{\Lambda}$ solves $S\|\{\}$ or $S$. Hence S is solvable.

### 3.2.6 Overall Proof for correctness

The given algorithm solves Unification problems by exhaustively applying inference rules on $S\|\{\}$, which results in an irreducible form $\Gamma'\|\Lambda'$. If $\Gamma'$ is non-empty, the problem $S$ is shown to be unsolvable by Theorem 20. However, if $\Gamma'$ is empty, it implies that $\Lambda$ is in solved form as per Theorem 18. The proof in 3.2.4 shows that the substitution $\overrightarrow{\Lambda}$ is an idempotent mgu of $\Lambda$. Theorem 11 states that $S \cup \{\}$ has the same set of unifiers as $\Gamma' \cup \Lambda'$, which is equal to $S$ has the same set of unifiers as $\Lambda'$ since $\Gamma'$ is empty. Therefore, both systems have the same mgu. Thus, we have established that the algorithm correctly determines whether a problem is solvable and returns an idempotent mgu of the original problem.

# 4 Coq Implementation

This chapter illustrates the Coq implementation of our work, including the definition of different data structures, the algorithm, and the theorems in Coq. Please note that we only provide the statement of the theorems in this chapter, as the full proofs have 11,000 lines of code. For complete development, please refer to our Coq code.

Here is a outline:

- ***Coq: Bool and Prop***: Introduce the basic definition of Bool type and Prop type. Illustrate the difficulties of a comparison bool function.
- ***Basic Data structure***: Introduce how we use Coq code to represent the basic data structure and equivalence relationships.
- ***XOR-Rewrite System***: Illustrate the overall rewrite system, and explain how we use the rewrite system to define the bool function to determine the equivalency between two terms .
- ***Substitution***: Introduce how we use Coq code to represent the data structure for substitution.
- ***XOR-Unification Algorithm***: Illustrate the algorithms and the approaches we use to prove it correct.

## 4.1 Coq: Bool and Prop

Two important types in Coq are Prop and Bool, which have distinct purposes and uses.

Prop is a type in Coq that represents mathematical propositions, i.e., statements that can be either True or False. Propositions in Coq can be expressed using logical connectives such as and ($\vee$), or ($\wedge$), not ($\sim$), and implies ($\rightarrow$), as well as quantifiers such as forall

($\forall$) and exists ($\exists$). Propositions in Coq can be used to express and prove the correctness of programs, algorithms, and mathematical theorems.

**Example 22** *Here are some examples of* Prop*:*

- $x = y$ *is a* Prop
- $P \to Q$ *is a* Prop *where* $P$ *and* $Q$ *are* Prop *as well.*
- $\forall n, n < 1$ *is a* Prop

On the other hand, Bool is a type in Coq that represents Boolean values, i.e., true or false. Bool in Coq are **computational** types that can be used in algorithms and functions. For example, a function that checks whether a given number is even or odd would return a bool value. Bool in Coq can be combined using logical operators such as and, or, and not, but they cannot be used to express mathematical propositions directly.

The differences between Prop and Bool in Coq are fundamental and reflect the distinction between computational and logical reasoning. While Bool is a type that represents computational values and can be used in algorithms and functions, Prop is a type that represents mathematical propositions and can be used to reason about the correctness and properties of programs and systems.

The main difficulty that causes this to happen is computation. An intuitive example would be, to design a Prop for detecting prime numbers would be easy:

$$(n : nat) := \forall (m1, m2 : nat), m1 \times m2 = n \to False \tag{11}$$

However, to actually compute a number is prime or not would be difficult.

Note that Prop is much easier to design because it only needs to capture the specific logical reasoning while Bool have to compute the exact value. And if we have a bool function then the Prop function is trivial, e.g. we have bool function $f()$, then the Prop function can be designed as $f() = true$.

In this development, we successfully defined a prop function directly, $t1, t2 := t1 == t2$ directly which will be explained in detail in later sections, to determine if the two terms are equivalent. But we failed to come up with a bool function $f(t1, t2)$ returns true if $t1 == t2$ directly. Here is what the road map looks like to achieve a bool function for XOR equivalence:

1. Define a bool function $f(\_, \_)$ to determine if two terms are exactly the same.

2. Define a rewrite function $r(\_)$ to rewrite any term into another form.

3. Prove that this rewrite does not lose equivalency, i.e. $\forall t : term, t == r(t)$.

4. Define a prop function $P_R(\_)$ to capture if a term is in certain form.

5. Prove the after rewrite it always satisfied some prop $P_R$, i.e. $\forall t : term, P_R(r(t))$

6. Prove that if two terms are in $P_R$, then function $f(\_, \_)$ can determine their xor equivalency, i.e. given $P_R(t1) \wedge P_R(t2)$, then $t1 == t2 \leftrightarrow f(t1, t2) = true$.

Now we can build the bool function for xor equivalence and prove it correct, $f(r(t1), r(t2)) = true \leftrightarrow t1 == t2$. These higher-level ideas give a clearer picture of the Bool and Prop functions. Bool function returns a **Computational Results**, whereas the Prop function is a **Mathematical Propositions** that can be used to prove some properties. We need bool function to compute a result to carry the algorithm further and prop function to verify the bool function is actually working as we want it to work.

## 4.2 Basic Data Structure

Given that this work only concerns constants, variables, and the $\oplus$ function, which exhibits XOR properties, the associated data structure can be described in the following way in Coq.

```
Definition var := string.
Inductive term: Type :=
| C : nat –> term
| V : var –> term
| Oplus : term –> term –> term.
Definition T0:term:= C 0.
Notation "x +' y" := (Oplus x y) (at level 50, left associativity).
```

The constructor C takes a natural number, which is a built in data structure from coq, as its input and outputs a constant term, while the constructor V takes a string, which requires export from Coq library, as input and outputs a variable term. The function $\oplus$ takes two terms as inputs and outputs an oplus term. Note that we define constant 0 as the unit.

**Example 23** *Some terms and their representations in Coq.*

- $1 \oplus v$ *in Coq is:* *C 1 +'V"v"*
- $a \oplus b \oplus c$ *in Coq is:* *(V "a"+'V "b")+' V"c"*

After introducing the fundamental term representations in Coq, it is necessary to define the equivalence relation modulo XOR. In addition to the four axioms of associativity, commutativity, unity, and nilpotency, this relation must also satisfy the properties of reflexivity, symmetry, and transitivity, as it is an equivalence relation. Since this is a congruence relation, we also must define oplus_compat which also plays a crucial role in this equivalence relation.

```
Reserved Notation "x == y" (at level 70).
Inductive eqv : term –> term –> Prop :=
| eqvA: forall x y z, (x +' y) +' z == x +' (y +' z)
```

```
| eqvC: forall x y, x +' y == y +' x
| eqvU: forall x, T0 +' x == x
| eqvN: forall x, x +' x == T0

| eqv_ref: forall x, x == x
| eqv_sym: forall x y, x == y -> y == x
| eqv_trans: forall x y z, x == y -> y == z -> x == z

| Oplus_compat : forall x x' , x == x' -> forall y y' ,
      y == y' -> x +' y == x' +' y'
where "x == y" := (eqv x y).
```

An easy intuition checks:

```
Lemma cancel_R: forall x y z, x +' z == y +' z -> x == y.
Lemma cancel_L: forall x y z, z +' x == z +' y -> x == y.
Lemma eqv_eqv0 (s t: term): s == t ↔ (s +' t) == T0.
```

Also note that there is some special case where this equivalence relationship implies equal. This is not easy to prove in Coq.

```
Theorem const_eqv_to_eq: forall n m, C n == C m -> n = m.
Theorem var_eqv_to_eq: forall v m, V v == V m -> v = m.
```

Having defined the propositional version of the equivalence relationship, our next step is to develop an algorithm that can determine if two terms are equivalent. However, this task presents a challenge since the terms can be viewed as a tree structure with the $\oplus$ operator at the top and two sub-terms on the left and right-hand sides. Due to the properties of associativity and commutativity, the tree can be rotated freely. Additionally, the task becomes even more complex if we add in nilpotency.

**Example 24** *Here are some examples of equivalence terms.*

- $a \oplus (0 \oplus b)$, $a \oplus b$, $c \oplus a \oplus b \oplus c$
- $a \oplus b \oplus a$, $b$, $b \oplus b \oplus b \oplus 0$

The examples above give an intuition of why developing a bool function returns true for two terms if they are equivalent is hard.

Despite the difficulties involved, it is essential to develop a Boolean version of the equivalence relation function, because at the end when we want to check whether an equation is unified or not, we have to have an algorithm return true or false for the left-hand side and the right-hand side. We will start by constructing a simple syntactic equivalence checker that returns true if two terms are exactly the same, i.e. modulo empty theory. Note this function does not have relationship with unification yet.

```
Fixpoint term_beq_syn (t1 t2:term):bool:=
  match t1, t2 with
  |C 0, t2 ⇒ t2
  |C n1, C n2 ⇒ beq_nat n1 n2
  |V v, V w ⇒ beq_var v w
  |Oplus t11 t12, Oplus t21 t22 ⇒
                  andb (term_beq_syn t11 t21) (term_beq_syn t12 t22)
  |_, _ ⇒ false
  end.
```

The correctness of syntactic checker is easy to prove:

```
Theorem term_beq_syn_true_iff: forall (t u: term),
term_beq_syn t u = true ↔ t = u.
Theorem term_beq_syn_false_iff: forall (t u: term),
term_beq_syn t u = false ↔ t <> u.
```

Now we have the syntactic checker. Since every term has a unique normal form, so we will build a rewrite system to reduce all terms to their normal form and then we can use the syntactic checker to decide whether two terms are equal or not.

## 4.3 XOR-Rewrite System

The following section details the design of the rewrite system and its correctness proof.

### 4.3.1 Road map and Important Correctness Theorem

Our ultimate goal is to transform the lTerm representation into a unique normal form: lTerm , more details will be introduced in the later section.

The first step is to design two functions $f_{tlt}(\_)$ $f_{ltt}(\_)$ to transfer the term into lTerm and back, and a predicate $\approx\approx$ for lTerm. Then we need to prove these two predicates capture the same equivalency for two data structures. i.e. the following lemmas have to be true.

$$\forall (t1\, t2 : term), t1 \approx_{XOR} t2 \leftrightarrow f_{tlt}(t1) \approx\approx f_{tlt}(t2) \tag{12}$$

$$\forall (tl1\, tl2 : lTerm), tl1 \approx\approx tl2 \leftrightarrow f_{ltt}(tl1) \approx_{XOR} f_{ltt}(tl2) \tag{13}$$

The second step is to design the rewrite system $f_R(\_)$ so that two equivalent lTerm are equal after rewriting. i.e. the following lemma has to be true:

$$\forall (tl1\, tl2 : lTerm), tl1 \approx\approx tl2 \leftrightarrow f_R(tl1) = f_R(tl2) \tag{14}$$

These three Lemma overall build the most important Theorem:

$$\forall (t1\, t2 : Term), t1 \approx_{XOR} t2 \leftrightarrow f_{tlt}(t1) \approx\approx f_{tlt}(t2) \leftrightarrow f_R(f_{tlt}(t1)) = f_R(f_{tlt}(t2)) \tag{15}$$

That is:

$$\forall (t1\, t2 : Term), t1 \approx\approx t2 \leftrightarrow f_R(f_{tlt}(t1)) = f_R(f_{tlt}(t2)) \tag{16}$$

Once we have these functions and proof in place, we can use the syntactic checker to compute the results for xor equivalency. In the later section, we will introduce how each function and proof is adopted in Coq in detail.

### 4.3.2 Alternative data representations:lTerm

Here we introduce the datatype, lTerm. The data structure for lTerm in Coq is simply a list of terms. The intuition here is to break down the term tree into individual singleton terms and put every individual singleton term into a list.

**Example 25** *Some examples of term to lTerm:*

- $a \oplus (0 \oplus b)$ *in lTerm is [V"a";C 0 +' V"b"]*
- $a \oplus b \oplus a$ *in lTerm is [V"a";V"b";V"a"]*

First, we need two functions $f_{tlt}(\_)$ and $f_{ltt}(\_)$ mentioned in the outline to transform between an ordinary term and a lTerm.

```
Fixpoint term_to_lTerm (t:term):lTerm:=
  match t with
  |t1 +' t2 ⇒ (term_to_lTerm t1) ++ (term_to_lTerm t2)
  |_ ⇒ [t]
  end.
Fixpoint lTerm_to_term (tl:lTerm):term:=
  match tl with
```

```
  | [] ⇒ T0
  | t::tl' ⇒ t +' (lTerm_to_term tl')
  end.
```

Given our choice to work with lTerm, we must to establish an equivalence relation between two lTerm representations that captures the same equivalence relationship defined for term, which is ≈≈ mentioned in the outline. This is necessary to ensure that our work with lTerm remains consistent with our work on term.

The intuition behind designing an equivalence relation for lTerm is straightforward, as the goal is for these two relations to capture the same equivalence relations. We can simply make modifications to the term data structures. The four axioms for XOR are represented by lr_perm (Associativity and Commutativity), lr_N(Nilepotency), and lr_T0 (Unity). We also require reflexivity, symmetry, and transitivity to indicate the equivalence relations. The connection between term and lTerm is established in lr_eqv_add1, which can capture all transformations between term and lTerm since l1 and l2 represent any arbitrary lTerm, in the case of singleton term:empty list. Finally, lr_oplus captures the fundamental concept of transforming term into lTerm by breaking the ⊕ and placing them in a list. Having defined the equivalence relations and transformation functions, we can now prove their correctness.

```
Inductive lTerm_eqv: lTerm -> lTerm -> Prop:=
|lr_T0 ll lr: lTerm_eqv (ll++lr) (ll++T0::lr)
|lr_eqv_add1 x y l1 l2: x == y -> lTerm_eqv (l1++x::l2) (l1++y::l2)
|lr_N l: lTerm_eqv (l++l) [T0]
|lr_perm x y: Permutation x y -> lTerm_eqv x y
|lr_compat l1 l2 l3 l4: lTerm_eqv l1 l2->lTerm_eqv l3 l4
                                        -> lTerm_eqv (l1++l3) (l2++l4)
```

```
|lr_trans l1 l2 l3: lTerm_eqv l1 l2 -> lTerm_eqv l2 l3->lTerm_eqv l1 l3
|lr_sym l1 l2: lTerm_eqv l1 l2 -> lTerm_eqv l2 l1
|lr_relf l1: lTerm_eqv l1 l1
|lr_oplus t1 t2: lTerm_eqv [t1 +' t2] [t1;t2].
```

And we can state the proof from equations (12) and (13) in Coq and prove it.

```
Theorem term_eqv_ok: forall (t1 t2:term),
  (term_to_lTerm t1) =l= (term_to_lTerm t2) ↔ t1 == t2.


Theorem lTerm_eqv_ok: forall (l1 l2:list term),
  l1 =l= l2 ↔ lTerm_to_term l1 == lTerm_to_term l2.
```

The full proof can be viewed in the full development. And now we move on to designing the function $f_R$ mentioned in the outline.

### 4.3.3 Rewrite System: Associativity Reducing

Now that we have established transformation functions and equivalence relations for lTerm, we can proceed with designing the rewrite system. Our first goal is to eliminate Associativity or any $\oplus$ in lTerm. We developed the lTerm data structure precisely to avoid dealing with multiple layers of terms, so eliminating these layers is the initial step in the rewriting process.

**Example 26** *Some examples of for A-Reduced:*

- *[V"a";C 0 +' V"b"] is not A-Reduced, because it can be further down broken into [V"a";C 0;V"b"]*
- *[V"a";V"b";V"a"] is A-Reduced*

Then we can define the Prop for A-Reduced:

```
Inductive AReduced: lTerm->Prop:=
|AReduced_nil: AReduced []
|AReduced_cons_const: forall (tl:list term)(n:nat),
    AReduced tl -> AReduced ((C n) :: tl)
|AReduced_cons_var: forall (tl:list term)(v:var),
    AReduced tl -> AReduced ((V v) :: tl).
```

which simply means that in lTerm, we can only have constants and variables. Then we can develop the A-Reducing algorithm to rewrite any lTerm into their A-Reduced form:

```
Fixpoint AReducing_lr(tl:lTerm):lTerm:=
  match tl with
  |[] ⇒ []
  |t::tl' ⇒ if (Oplus_term t)
      then app (term_to_lTerm t) (AReducing_lr tl')
      else t::(AReducing_lr tl')
  end.
```

Then the correctness:

```
Theorem AReducing_lr_Correct_Reduced:forall (tl:lTerm),
  AReduced (AReducing_lr tl).
Theorem AReducing_lr_Correct_eqv:forall (tl:lTerm),
  tl =l= (AReducing_lr tl).
Theorem AReduced_AReducing_idpt:forall (tl:lTerm),
  AReduced tl -> AReducing_lr tl = tl.
```

The first theorem states that every lTerm after A-Reducing is A-Reduced. The second theorem states that every lTerm is equivalent to the same lTerm after being A-Reducing. The third theorem states that this algorithm is idempotent, i.e. terminating.

#### 4.3.4 Rewrite System: Nilpotency Reducing

We now proceed with the second part of our rewrite system, which involves handling Nilpotency. The underlying principle for Nilpotency reduction is based on the following lemma:

```
Lemma NReducing_Base:forall(lt1 lt2 lt3:lTerm)(t:term),
lt1 ++ [t] ++ lt2 ++ [t] ++ lt3 =l= lt1 ++ lt2 ++ lt3.
```

This lemma states that given any lTerm, if it contains two identical terms, the lTerm is equivalent to the one obtained by removing these two terms due to Nilpotency. Thus, we first define our criterion for a lTerm being N-Reduced.

```
Inductive NReduced:lTerm->Prop:=
|NReduced_nil : NReduced []
|NReduced_cons_const : forall (n : nat) (tl : lTerm),
        ~In (C n) tl -> NReduced tl -> NReduced ((C n) :: tl)
|NReduced_cons_var : forall (v : var) (tl : lTerm),
        ~In (V v) tl -> NReduced tl -> NReduced ((V v) :: tl).
```

The addition of a term to a lTerm is equivalent to removing that term from the lTerm if the term is already present in the lTerm as a constant or a variable. So we can first design an algorithm about Nilpotency add.

```
Fixpoint n_add (a:term) (x:lTerm) : lTerm :=
    match x with
    | nil ⇒ a :: nil
    | a1 :: x1 ⇒
        if term_beq_syn a1 a
            then x1
            else a1 :: n_add a x1
```

```
    end.

Fixpoint NReducing'(tl:lTerm):lTerm:=
  match tl with
  |[] ⇒ []
  |t::tl ⇒ (n_add t (NReducing' tl))
  end.

Definition NReducing(tl:lTerm):lTerm:= rev(NReducing' tl).
```

Note that the final N-Reducing algorithm takes a reverse, that's because the recursive call will flip the order. Here we use reverse to rotate back so we can achieve idempotent. Then the correctness:

```
Theorem NReducing_Correct_alllist: forall (tl : lTerm),
  AReduced tl -> NReduced (NReducing tl).
Theorem NReducing_eqv: forall (tl:lTerm),
  (NReducing tl) =l= tl.
Theorem NReduced_NReducing: forall (tl :lTerm),
  NReduced (tl) -> NReducing tl = tl.
```

These three theorems prove similar properties compared to A-Reducing. However, note that when checking for N-Reduced, an additional assumption is needed stating that the lTerm is already A-Reduced. This is because the N-Reduced properties and N-Reducing function assume that the lTerm is already A-Reduced. As long as the A-Reducing rewrite always happens before the N-Reducing rewrite, we could use the assumption that the lTerm we are dealing with is already A-Reduced.

#### 4.3.5 Rewrite System: Unity Reducing

This is the easiest rewrite system, we just need to go through the lTerm and remove all T0. Meanwhile, note that in lTerm data structure:

```
Lemma T0_lTerm_eqv_nil:
  [T0] =l= nil.
```

The algorithm and the Prop for U-Reduced is straightforward. As long as the term is not equal to T0, it can stay in the lTerm. The code is listed below:

```
Inductive UReduced: list term–> Prop:=
|UReduced_nil : UReduced []
|NReduced_cons_const : forall (n : nat) (tl : list term),
     n<>0 –> UReduced tl –> UReduced ((C n) :: tl)
|NReduced_cons_var : forall (v : var) (tl : list term),
     UReduced tl –> UReduced ((V v) :: tl).


Fixpoint UReducing (tl: list term) : list term:=
  match tl with
  |[] ⇒ []
  |t::tl' ⇒ if term_beq_syn t T0 then UReducing tl' else t::UReducing tl'
  end.
```

Then the correctness, i.e. equivalence relations preserve and it is indeed U-Reduced after U-Reducing:

```
Lemma UReducing_eqv: forall(tl: list term),
  (UReducing tl) =l= tl.
Lemma UReduced_UReducing_alllist: forall(tl:list term),
  AReduced tl –> UReduced (UReducing tl).
```

#### 4.3.6 Rewrite System: Commutativity Reducing

To establish a normal form with respect to Commutativity, we need to impose an ordering on the constants and variables in the lTerm. For the constants, we can use the natural number ordering directly. However, for the strings, it is not that straightforward. We find a comparison function provided in the Coq library and decided to use it. This ordering will allow us to ensure that equivalent lTerms have the same ordering of constants and variables, making it easier to compare them for equality syntactically.

```
Fixpoint compare (s1 s2 : string) : comparison :=
  match s1, s2 with
  | EmptyString, EmptyString ⇒ Eq
  | EmptyString, String _ _ ⇒ Lt
  | String _ _ , EmptyString ⇒ Gt
  | String c1 s1', String c2 s2' ⇒
    match Ascii_compare c1 c2 with
    | Eq ⇒ compare s1' s2'
    | ne ⇒ ne
    end
  end.
```

To ensure that the lTerm is ordered in the desired way, we can define a proposition to capture this property. The design is straightforward: every constant should be less than every variable, and within the constants and variables, we should use the respective ordering relations mentioned above. Note that this is a term ordering.

```
Inductive Rvc: term –> term –> Prop:=
|rvv: forall (v1 v2:var), order_string v1 v2 –> Rvc (V v1) (V v2)
|rvc: forall (v:var) (n:nat), Rvc (C n) (V v)
```

```
|rcc: forall (n1 n2:nat), n1 <= n2 -> Rvc (C n1) (C n2).

Inductive ltSorted : list term -> Prop :=
| ltSorted_nil : ltSorted []
| ltSorted_cons1 a : ltSorted [a]
| ltSorted_consn a b l :
    ltSorted (b :: l) -> Rvc a b -> ltSorted (a :: b :: l).
```

The binary relation between terms, Rvc, and ltSorted, the name for C-Reduced, are used to ensure that every term in an lTerm is in order by Rvc. If this property holds, then the lTerm is considered ltSorted or C-Reduced. The algorithm to achieve this is:

```
Definition sort_term(l:list term): list term:=
   (sort_constterm l) ++ (sort_varterm l).
```

where sort_constterm is the function filtering out all non-constant and sorting every constant in the lTerm and sort_constterm is the function filtering out all non-variable and sorting every variable in the lTerm.

Then to the correctness, this time we need to first have the correctness proof for the function sort_term, which is the input and out put is Permutation of each other and the lTerm is sorted.

```
Theorem sort_ltSorted: forall (l:list term),
    ltSorted(sort_term l).

Theorem sort_ltSorted_Permutation: forall(l:list term),
    AReduced l -> Permutation l (sort_term l).
```

### 4.3.7 The whole rewrite system

To create a complete rewrite system, we need to combine all the pieces we have designed so far. Since many rewrite systems require the input lTerm to be A-Reduced, we first apply the A-Reducing algorithm to the lTerm. After that, the order of the other rewriting algorithms does not matter.

```
Definition Reducing_lr (tl:list term):list term:=
  UReducing (NReducing (AReducing_lr tl)).
```

Note that here we haven't add in the C-Reducing part yet. But to prove this function works correctly, as before we need the following Theorem. Note that Reduced:= AReduced/\NReduced/\UReduced

```
Theorem Reducing_lr_Correct_Reduced: forall(tl:list term),
  Reduced(Reducing_lr tl).
Theorem Reducing_lr_Correct_eqv: forall(tl:list term),
  tl =l= (Reducing_lr tl).
```

Then we add in the ltSorted or C-Reduced, the function is simple, we just need to add sort_term in front of the reducing system described above. Recall that this is the function $f_R(\_)$ we mentioned in the outline section.

```
Definition Reducing_lr_Ord (l:list term):list term:=
  sort_term (Reducing_lr l).
```

And then we need a similar correctness proof:

```
Theorem Reducing_lr_Ord_Correct_Reduced_Ord: forall(lt:lTerm),
  Reduced_Ord (Reducing_lr_Ord lt).
Theorem Reducing_lr_Ord_Correct_eqv: forall(tl:list term),
  tl =l= (Reducing_lr_Ord tl).
```

Now we have our complete rewrite system. The benefit of this rewrite system is that every equivalence class has a unique normal form, or every two equivalence terms are exactly the same in their normal form. Note that every Lemma stated above is the most important connection between rewrite system correctness proof, for more details, please refer to our full development.

Now since we have all the small connection pieces together, we could prove:

$$\forall (tl1\, tl2 : lTerm), tl1 \approx\approx tl2 \leftrightarrow f_R(tl1) \approx\approx f_R(tl2) \tag{17}$$

by transitivity of the predicates $\approx\approx$ with an ease. Our original lemma (14) in the outline:

$$\forall (tl1\, tl2 : lTerm), tl1 \approx\approx tl2 \leftrightarrow f_R(tl1) = f_R(tl2) \tag{18}$$

took a bit more work to prove, please refer to our full development for more details. Once we have those, we can easily state and prove the most important theorem (16) in Coq as below.

```
Theorem lTerm_eqv_eq_correct: forall (l1 l2:list term),
  l1 =l= l2 ↔ Reducing_lr_Ord l1 = Reducing_lr_Ord l2.
```

## 4.4 Substitutions

Now that we have our basic definitions and equivalence relation in place, we can start building substitutions. There are many ways to implement substitutions, but here we will define substitution as a function that takes a variable as input and returns a term as output.

```
Definition sub : Type := var -> term.
```

Then we define our lft function just like any ordinary lft:

```
Equations lft (sb:sub): (term -> term) :=
  lft sb (C n) := C n;
  lft sb (V v) := sb v;
  lft sb (t1 +' t2) := lft sb t1 +' lft sb t2.
```

Note that we use the Equations module instead of the Fixpoint in our implementation of lft. This is because we initially used Equations to handle termination. Although our final algorithm does not rely on the Equations module, we decided to keep the previous implementations.

Since we are working with lTerm, the ordinary lft function is not the right fit for lTerm. However, with a little twist, we can make it work. We define our lft function for lTerm as lftl, which is as follows:

```
Definition lftl (sb:sub): lTerm -> lTerm:=
  (fun lt ⇒ map (lft sb) lt).
```

This is fairly straightforward: we just replace every term in that lTerm that is in the domain of the substitution by the range of the substitution. Note here the lTerm after lftl is not neccessary in reduced form.

We also need to prove the correctness of our new lftl function to ensure that it has the same functionality as the original lft function:

```
Lemma lft_lftl_correct: forall (sb:sub)(t:term),
  term_to_lTerm (lft sb t) =l= lftl sb (term_to_lTerm t).
Lemma lftl_lft_correct:forall(sb:sub)(tl:lTerm),
  lTerm_to_term (lftl sb tl) == lft sb (lTerm_to_term tl).
```

To prove the first lemma above, we proceed as follows: For any term t and substitution sb, if we apply lft to t with sb and then transform the resulting term into a lTerm, it is equivalent to applying lftl directly to the lTerm form of t with sb.

To prove the second lemma above, we proceed as follows: For any lTerm lt and substitution sb, if we apply lftl to lt with sb and then transform the resulting lTerm into a term, it is equivalent to applying lft directly to the term form of lt with sb.

Another important function that is worth mentioning in this development is the update substitution function. The function is listed below:

```
Equations update_sub : sub –> var –> term –> sub :=
  update_sub tau x t :=
    fun v ⇒ if eq_dec_var x v then
                 t else
                 tau v.
```

The function takes a substitution sb, a variable v, and a term t, return a new substitution that adds the bind $v \mapsto t$ into the substitution. The other utility functions used in substitutions, such as domain, range, vrange, and idempotent, are defined according to their standard definitions. we have omitted the Coq definitions of these functions in our presentation here.

## 4.5 XOR-Unification Algorithm

This chapter details how the xor-unification is laid out and how did we prove it.

### 4.5.1 Raw Problems and Reduced Problems

Here, "problems" refer to the unification problems that we want to solve, which have the form of $\{t_1 \approx^?_E s_1, \ldots, t_n \approx^?_E s_n\}$. We represent each problem as a pair of lTerms, or (lTerm,lTerm), since we have proven that terms can be transformed into lTerms without losing equivalence. We use "problem" to refer to a single unification problem and "problems" to refer to a list of problems that capture all unification problems.

Raw problems in our development means the lTerm on both hand side can be in any form, they don't have to be reduced.

Reduced problems in our development means the right-hand side of the problems are empty and the left-hand side of the problems are all reduced.

**Example 27** *Here are some examples of a raw problem, raw problems, a reduced problem.*

- *(nil,[C 0;C 0]) is a raw problem*
- *[(nil,nil);(C 1 +' C 1,nil)] is raw problems*
- *([C 1;C 0],nil) is not a reduced problem because the lhs is not Reduced*
- *([C 1],[C 1]) is not a reduced problem because the rhs is not empty*
- *([C 1], nil) is reduced*

And then we define functions transform raw problems to reduced problems:

```
Definition rawP_to_reducedP(p:problem):problem:=
  ((Reducing_lr_Ord ((lhs p) ++ (rhs p)) ),[]).
Definition rawPs_to_reducedPs(ps:problems):problems:=
  map rawP_to_reducedP ps.
```

This transformation does not change the equality and does not lose any unifiers while transforming:

```
Lemma lTerm_eqv_same_side_nil:forall(lt1 lt2:lTerm),
  lt1 =l= lt2 ↔ lt1 ++ lt2 =l= [].
Lemma reducedPs_to_rawPs_sub_preserve:forall(ps:problems)(sb:sub),
  solves_problems sb (rawPs_to_reducedPs ps) –>
```

```
  solves_problems sb ps.
Lemma rawPs_to_reducedPs_sub_preserve:forall(ps:problems)(sb:sub),
  solves_problems sb ps ->
  solves_problems sb (rawPs_to_reducedPs ps).
```

### 4.5.2 Solved Form

Here define problems in solved Form:

```
Definition solved_form (ps:problems):Prop:=
  NoDup (map fst ps) /\
  Forall single_variable_lTerm_Prop (map fst ps) /\
  Forall Reduced_Ord (map snd ps) /\
  disjoint_In (app_list_lTerm (map fst ps)) (app_list_lTerm (map snd ps)).
```

The left-hand side consists of pairwise distinct variables, which is represented by NoDup (map fst ps) and Forall single_variable_lTerm_Prop (map fst ps). The variables on the left-hand side do not appear on the right-hand side, which is represented by disjoint_In (app_list_lTerm (map fst ps))(app_list_lTerm (map snd ps)). Lastly, we add one more property that to simplify the proof, which is Forall Reduced_Ord (map snd ps). This property indicates that the lTerms on the right-hand side are reduced to their normal form.

Then we need to define a function that extracts a substitution from problems in solved form:

```
Fixpoint solved_form_to_sub (ps:problems):sub:=
  match ps with
  |[] ⇒ id_sub
  |p::ps' ⇒ match single_variable_lTerm_var (fst p) with
```

```
                |Some v ⇒ compose_sub (singleton_sub v (lTerm_to_term (snd p)
                    )) (solved_form_to_sub ps')
                |None ⇒ solved_form_to_sub ps'
  end end.
```

Here are some very important proof results from the substitution extraction function. It directly connects the algorithm to the final property of correctness.

```
Theorem solved_form_sub_solves:forall(ps:problems),
  solved_form ps –> solves_problems (solved_form_to_sub ps) ps.
Theorem solved_form_sub_mgu:forall(ps:problems),
  solved_form ps –> mgu_xor (solved_form_to_sub ps) ps.
Theorem solved_form_sub_idpt:forall(ps:problems),
  solved_form ps –> idempotent (solved_form_to_sub ps).
```

The first Theorem indicates that if a problems is in solved form, then the substitution generates from it solves the problems.

The Second Theorem indicates that if a problems is in solved form, then the substitution generates from it is the most general unifier of the problems.

The Third Theorem indicates that if a problems is in solved form, then the substitution generates from it is idempotent.

### 4.5.3 Terminating Function

Coq only accepts functions that it is certain will terminate. For relatively large algorithms, we need to prove to Coq that our algorithm will terminate. One approach to do this is to use the concept of executing a function a certain number of times. This is easier to prove will terminate since the natural numbers are well-founded and will eventually reach zero.

In this development, we adopt the approach mentioned above - supplying a "fuel" to the function, meaning we give the function the number of executions it needs to run. This approach requires us to consider two aspects: first, what each execution does, and second, how many executions we need to perform to guarantee termination.

### 4.5.4 Problems Set and Solved Problems

The goal here is to design a function that transforms reduced problems into problems in solved form. To do so, we use the two inference rules introduced in Chapter 3.2: (1) eliminate the function that is already solved or balanced, as Trivial states, and (2) if a problem is solvable, transform it into solved form and apply the resulting substitution to the rest of the problems.

To transform these two rules into Coq code, we define another data structure: Problem Set.

```
Definition problems_set:=prod problems problems.
```

The intuition behind this data structure is similar to the mathematical symbol used in Chapter 3.2. The problems on the left-hand side are problems that remain to be processed, while the problems on the right-hand side are all the problems that have already been processed and are in solved form. So the input problem set would be something like `(_,nil)`.

The first question we need to answer is what happens during one execution of the function. We start by checking the first problem. If it is already balanced or solved, i.e., the left-hand side is equivalent to the right-hand side, then we can just remove it because it will not affect the remaining problems. This is essentially what the Trivial function does, as introduced in Section 3.2. If the problem is not already balanced, we need to check whether it is solvable or not. If it is solvable, then we can transform it into its solved form and apply the resulting substitution to both the remaining unprocessed problems and the already processed problems, just like the Variable Substitution function introduced in

Section 3.2.

Now, here is the code:

```
Definition problem_to_sub(p:problem):option (var * sub).
  remember (rawP_to_reducedP p) as rp.
  destruct (a_var_in_lTerm (fst rp)) eqn:H.
  – exact (Some (v, (singleton_sub v (lTerm_to_term (remove (V v) (fst rp)))))).
  – exact None.
Defined.


Definition step(pss:problems_set):problems_set:=
  match pss with
  |([], _) ⇒ pss
  |(p::pss',sp) ⇒ match (lTerm_eqv_bool (fst p) []) with
                   | true ⇒ (pss',sp)
                   | false ⇒ match problem_to_sub p with
                              |Some (v,sb) ⇒
                                (
                                Rproblems (apply_sub_problems sb pss')
                                ,
                                ([V v], (remove (V v) (fst p))) ::
                                Rproblems (apply_sub_problems sb sp)
                                )
                              |None ⇒ pss
  end end end.
```

The problem-to-sub function generates a substitution that solves a given problem, provided that the problem is solvable. The function follows the same approach as described above: it first checks if the problem is already balanced or solved, and if so, it removes the problem. If the problem is not already balanced, it checks if it is solvable. If the problem is solvable, it applies the generated substitution to both the processed and unprocessed problems. If the problem is not solvable, the function returns that the entire problem is not solvable.

After defining what needs to be done during one execution, we must determine exactly how many times the algorithm executes, i.e., the termination argument. From the algorithm step above, it is obvious that every execution deals with only one problem. So the measure is straightforward - it is just the number of problems in the input problems set.

```
Definition measure(pss:problems_set):nat:=(length_nat (fst pss))
```

Now we can formally define our steps algorithm, which consists of executing the function step a number of times equal to the measure:

```
Fixpoint steps(measure:nat)(sys:problems_set):problems_set:=
  match measure with
  |0 ⇒ sys
  |S measure' ⇒ (steps measure' (step sys))
  end.
```

Another design consideration is how to determine when the algorithm has halted, either because it has reached a solved form or because the problem is not solvable. To address this, we adopt an approach of returning the original problem set unchanged if we detect a problem that is not solvable. This allows the algorithm to continue unchanged until the maximum number of executions have been performed, ensuring that if the problem is solvable, it will eventually be transformed into the solved form and left with nothing in the left-hand side of the problem set. To support this approach, we need to prove a lemma that relates the measure and problems.

```
Definition fixed_pss(pss:problems_set):Prop:=
  step pss = pss.
Lemma steps_fixed_sys:forall(sys:problems_set),
  fixed_pss(steps (measure sys) sys).
```

The lemma states that after "measure" number of executions, further executions do not change the problem set anymore, either because the first problem on the left-hand side is not solvable or the left-hand side is empty.

With this lemma in place, we can now prove that the steps function does the right thing, which is to transform the problems into a solved form:

```
Theorem rawPs_to_solvedPs_solved_form:forall(ps:problems) (ps1 ps2:
    problems),
  steps (measure (problems_set_ready ps)) (problems_set_ready ps) = (ps1,
      ps2) ->
  solved_form ps2.
Theorem steps_sub_preserve_ps2_rev:forall(ps:problems)(ps1 ps2:problems)(
    sb:sub),
  steps (measure (problems_set_ready ps)) (problems_set_ready ps) = (ps1,
      ps2) ->
  ps1 = [] -> solves_problems sb ps2 -> solves_problems sb ps.
```

Note that during the transformation, the substitution that solves the original problem always solves the processed problems because the processed problems are subproblems of the original problem that is in solved form. Regarding the backwards direction, an extra assumption is needed that the set of processed problems, ps1, is empty. This indicates that the function steps did not encounter any unsolvable problem and all the problems have been processed into solved form.

Moreover, we also need to ensure that this process does not lose any unifiers:

```
Theorem steps_sub_preserve_ps2:forall(ps:problems)(ps1 ps2:problems)(sb:
    sub),
  steps (measure (problems_set_ready ps)) (problems_set_ready ps) = (ps1,
      ps2)
  -> solves_problems sb ps -> solves_problems sb ps2.
```

### 4.5.5 Assemble

Now we can wrap every piece together.

First, lets look at our raw problems to solved problems:

```
Definition rawPs_to_solvedPs(ps:problems):option problems:=
  match steps (measure((rawPs_to_reducedPs ps),[])) ((rawPs_to_reducedPs
      ps),[]) with
  |(nil,ps) ⇒ Some ps
  |(_,_) ⇒ None
  end.
```

The algorithm first converts the input raw problems to reduced problems, and then performs a fixed number of executions on the reduced problems. If the left-hand side of the problem set is empty after these executions, then the problem is solvable, and the function returns the right-hand side, which is the solved form of the original problem and then transformed it into a substitution. If the left-hand side is not empty, it means the problem is not solvable and the function returns None. Note that during the execution, the function maintains a set of processed problems and applies the generated substitution to both the processed and unprocessed problems. This ensures that the transformed solved form does not lose any unifiers.

```
Definition XORUnification(ps:problems) : option sub:=
  match (rawPs_to_solvedPs ps) with
  |Some ps' ⇒ Some (solved_form_to_sub ps')
  |None ⇒ None
  end.
```

This means that if a problem has a complete solved form, then we return the substitution generated from that solved form. If not, then it is not solvable.

### 4.5.6 Final Correctness

```
Theorem XORUnification_not_solvable:forall(ps:problems),
  XORUnification ps = None -> not_solvable_problems ps.
```

The algorithm returns None means the original unification problem is not solvable.

```
Theorem XORUnification_solves:forall(ps:problems)(sb:sub),
  XORUnification ps = Some sb -> solves_problems sb ps.
```

The algorithm returns some substitution means this substitution solves the original unification problem.

```
Theorem XORUnification_mgu:forall(ps:problems)(sb:sub),
  XORUnification ps = Some sb -> mgu_xor sb ps.
```

The algorithm returns some substitution means this substitution is the mgu(most general unifier) of the problems.

```
Theorem XORUnification_idpt:forall(ps:problems)(sb:sub),
  XORUnification ps = Some sb -> idempotent sb.
```

The algorithm returns some substitution means this substitution is idempotent.

We also need the chain of reasoning backward:

```
Definition problems_unifiable(ps:problems):Prop:=
  exists sb:sub, solves_problems sb ps.
Theorem unifiable_return_sub:forall(ps:problems),
  problems_unifiable ps -> (exists sb:sub, XORUnification ps = Some sb).
Theorem not_unifiable_return_None:forall(ps:problems),
  ~(problems_unifiable ps) -> XORUnification ps = None.
```

To sum up, in this development, we proved that: If the original unification problem is solvable, then the algorithm will return a substitution that is a most general unifier and it is idempotent. If the original unification problem is not solvable then the algorithm will return None.

# 5 Future Work

An immediate area for future work would be to incorporate uninterpreted functions and homomorphism functions into the algorithm. While the overall proving steps would not change significantly, the unification algorithms would need to be modified to handle these new functions.

```
Definition var := string.
Definition fname := string.
Inductive term: Type :=
| C : nat -> term
| V : var -> term
| Oplus : term -> term -> term
| f1 : fname -> term -> term
| f2 : fname -> term -> term -> term
| h : fname -> term -> term
```

Modifying the data structure to include uninterpreted functions and homomorphism functions is relatively straightforward. We can introduce two new sets, f1 and f2, to store the uninterpreted functions with arities 1 and 2, respectively, and a new function h to represent the homomorphism function. The function name (fname) for each symbol is used to differentiate between them.

For equivalence relations, we need to add in the appropriate rules for functions, where the function names must match and their corresponding terms must be identical.

To incorporate the rewrite system, we first apply an H-Reducing step to reduce all h-functions to their base terms. Next, we can A-Reduce, N-Reduce, and U-Reduce f1, f2, and the beq function as before. For C-Reducing, we need to determine a specific order for obtaining the normal form, such as constants before variables, followed by f1 terms, f2 terms, and finally h-terms.

The most significant change would be in the unification algorithm, which becomes non-deterministic with the addition of uninterpreted function symbols.

**Example 28** *Consider two xor unification problems with some uninterpreted function f().* $S_1 = \{x_1 \oplus x_2 \approx y_1 \oplus y_2\}$, $S = \{f(x_1) \oplus f(x_2) \approx f(y_1) \oplus f(y_2)\}$.

*The unifiers for $S_1$ could be:*

- $\sigma_1 := \{x_1 \mapsto x_2 \oplus y_1 \oplus y_2\}$
- $\sigma_2 := \{x_2 \mapsto x_1 \oplus y_1 \oplus y_2\}$
- $\sigma_1 = \sigma_1\sigma_2$ *i.e.* $\sigma_1 \lesssim_E \sigma_2$
- $\sigma_2 = \sigma_2\sigma_1$ *i.e.* $\sigma_2 \lesssim_E \sigma_1$

*The unifiers for $S_2$ could be:*

- $\sigma_1 := \{x_1 \mapsto x_2, y_1 \mapsto y_2\}$
- $\sigma_2 := \{x_1 \mapsto y_1, x_2 \mapsto y_2\}$
- $\sigma_3 := \{x_1 \mapsto y_2, x_2 \mapsto y_1\}$
- $\sigma_1 \not\lesssim_E \sigma_2$
- $\sigma_2 \not\lesssim_E \sigma_1$
- $\sigma_2 \not\lesssim_E \sigma_3$
- $\sigma_3 \not\lesssim_E \sigma_2$
- $\sigma_1 \not\lesssim_E \sigma_3$
- $\sigma_3 \not\lesssim_E \sigma_1$

The future goal is to tackle non-deterministic unification problems, which is challenging due to the need to capture the minimal set of solutions. To address this, the N-Decomposition inference rule will be incorporated into the algorithm to eliminate uninterpreted functions. Moreover, to capture the complete set of solutions for the system $\Gamma \| \Lambda$, $\Delta$ will be introduced, and the system will be modified to $\Gamma \| \Delta \| \Lambda$ to capture disequations. In the proof section, we will need to account for the fact that N-Decompositions will split the problem into two parts and may not preserve the unifiers exactly.

More precisely: (Note N-D stands for N-Decompositions)

**Theorem 29** *Let* $\Gamma \| \Delta \| \Lambda$, $\Gamma' \| \Delta' \| \Lambda'$ *and* $\Gamma'' \| \Delta'' \| \Lambda''$ *be three systems satisfying that* $\Gamma \| \Delta \| \Lambda \Rightarrow_{N-D} \Gamma' \| \Delta' \| \Lambda' \bigvee \Gamma'' \| \Delta'' \| \Lambda''$. *Then unifiers* $\sigma$ *unifies* $\Gamma' \| \Delta' \| \Lambda'$ *or* $\Gamma'' \| \Delta'' \| \Lambda''$ *unifies* $\Gamma \| \Delta \| \Lambda$.

**Theorem 30** *Let* $\Gamma \| \Delta \| \Lambda$, $\Gamma' \| \Delta' \| \Lambda'$ *and* $\Gamma'' \| \Delta'' \| \Lambda''$ *be three systems satisfying that* $\Gamma \| \Delta \| \Lambda \Rightarrow_{N-D} \Gamma' \| \Delta' \| \Lambda' \bigvee \Gamma'' \| \Delta'' \| \Lambda''$. *Then if unifiers* $\sigma$ *unifies* $\Gamma \| \Delta \| \Lambda$, *there exists some substitution* $\delta$ *and* $\delta'$ *such that* $\delta\sigma$ *unifies* $\Gamma' \| \Delta' \| \Lambda'$ *and* $\delta'\sigma$ *unifies* $\Gamma'' \| \Delta'' \| \Lambda''$.

# References


[1] 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.

[2] Mauricio Ayala-Rincón, Maribel Fernández, Daniele Nantes-Sobrinho, and Gabriel Ferreira Silva. A certified algorithm for AC-unification. In *Proceedings of FSCD 2022*. LIPIcs, 2022.

[3] Franz Baader. Unification in a theory of primitive recursive functions with applications to semigroups and groups. *Information and Computation*, 95(1):1–36, 1991.

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

[5] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[6] Jacques Fleuriot Carlin MacKenzie, James Vaughan. Archive of formal proofs. Accessed: 2023-01-11.

[7] James H Davenport and John C Reynolds. The undecidability of equivalence and disjointness problems for polynomial ideals over the integers. *Journal of Symbolic Computation*, 1(1):23–35, 1985.

[8] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[9] Daniel J Dougherty. A Coq formalization of Boolean unification. In *33rd International Workshop on Unification Dortmund, June 24, 2019*, 2019.

[10] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[11] Qing Guo, Paliath Narendran, and David A Wolfram. Unification and matching modulo nilpotence. In *International Conference on Automated Deduction*, pages 261–274. Springer, 1996.

[12] Zhiqiang Liu and Christopher Lynch. Efficient general unification for XOR with homomorphism. In *International Conference on Automated Deduction*, pages 407–421. Springer, 2011.

[13] Zohar Manna and Richard Waldinger. Deductive synthesis of the unification algorithm. In *Computer Program Synthesis Methodologies*, pages 251–307. Springer, 1983.

[14] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[15] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[16] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[17] Sam Owre, John M Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[18] Lawrence C Paulson. Verifying the unification algorithm in LCF. *Science of computer programming*, 5:143–169, 1985.

[19] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[20] P. Y. A. Ryan and S. A. Schneider. An attack on a recursive authentication protocol. a cautionary tale. *Inf. Process. Lett.*, 65(1):7–10, jan 1998.

[21] Christian Sternagel and René Thiemann. First-order terms. *Archive of Formal Proofs*, February 2018. `https://isa-afp.org/entries/First_Order_Terms.html`, Formal proof development.

[22] Max Tuengerthal, Ralf Küsters, and Mathieu Turuani. Implementing a unification algorithm for protocol analysis with XOR. *arXiv preprint cs/0610014*, 2006.

[23] Christian Urban, Andrew M Pitts, and Murdoch J Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.