

DOI: 10.15514/ISPRAS-2026-38(6)-2



Certified Unification for Rational Terms

*E.A. Domoratskiy, ORCID: 0009-0005-9020-3403 <eridan200@mail.ru>
D.Yu. Boulytchev, ORCID: 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>
Saint-Petersburg State University,
7/9, Universitetskaya emb., Saint-Petersburg, 199034, Russia.*

Abstract. We present a unification algorithm for first-order rational terms certified with the Rocq proof assistant. The algorithm is based on the results of Martelli and Rossi but reformulated in order to derive a practically useful implementation. We introduce the basic notions of rational terms and equation systems as an intermediate state of the algorithm and the algorithm itself, and prove their main properties. All statements are formally verified in Rocq without any additional axioms. We use an induction-based approach to define a bisimilarity for coinductive types in Rocq that allows one to avoid productivity checker issues. The Rocq specification is automatically extracted into a Haskell implementation. To our knowledge this is the first work that provides a mechanized specification of unification algorithm for rational terms.

Keywords: rational terms; unification; certified programming; coinduction; bisimilarity.

For citation: Domoratskiy E.A., Boulytchev D.Yu. Certified Unification for Rational Terms. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 6, part 1, 2026, pp. 23-40. DOI: 10.15514/ISPRAS-2026-38(6)-2.

Формальная верификация алгоритма унификации рациональных термов

*Э.А. Доморацкий, ORCID: 0009-0005-9020-3403 <eridan200@mail.ru>
Д.Ю. Булычев, 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>
Санкт-Петербургский Государственный Университет,
Россия, 199034, г. Санкт-Петербург, Университетская наб., д. 7/9.*

Аннотация. Мы представляем алгоритм унификации рациональных термов первого порядка, формально верифицированный в системе автоматической проверки доказательств Rocq. Алгоритм основан на подходе Мартелли и Росси, но переформулирован для удобства практической реализации. В работе описаны основные понятия, связанные с рациональными термами, системами уравнений как промежуточным состоянием работы алгоритма, и сам алгоритм, а также доказательства их основных свойств. Все утверждения из текста формально верифицированы в Rocq без использования дополнительных аксиом. Для определения отношения наибольшей бисимуляции на коиндуктивных типах в Rocq используется подход, основанный на индукции, позволяющий избежать проблем с проверкой продуктивности. Из реализации алгоритма в Rocq автоматически получена программа на Haskell. Это первая работа, предлагающая формально верифицированную спецификацию для алгоритма унификации рациональных термов.

Ключевые слова: рациональные термы; унификация; коиндукция; бисимуляция.

Для цитирования: Доморацкий Э.А., Булычев Д.Ю. Формальная верификация алгоритма унификации рациональных термов. Труды ИСП РАН, том 38, вып. 6, часть 1, 2026 г., стр. 23–40 (на английском языке). DOI: 10.15514/ISPRAS-2026-38(6)0-2.

1. Introduction

First-order unification is a fundamental operation in many areas of computer science, including automated theorem proving, logic programming (e.g., in Prolog) and type inference. Typically, unification is studied for finite (Herbrand) terms built from variables and constructors. However, in some cases (such as recursive data types, lazy data structures, infinite streams, cyclic term graphs, etc.) the unification of infinite terms is desirable. Of particular interest are rational terms, which are infinite terms that have only finitely many distinct subterms [1].

Our research is motivated by the application of relational programming to the problem of type systems implementation. Relational programming [2-3] is a branch of logic programming studying the implementation techniques and applications of pure declarative relational specifications. Such specifications enjoy a nice property: they provide sound results regardless the “direction” they are queried in. For example, a single type system description can be used for type checking, type inhabitation test, and type inference [4]. However, being implemented in a conventional relational language equipped with first-order unification of Herbrand terms a type system specification would be incapable of dealing with recursive types, which we consider a fatal setback. Extending unification to support rational terms would fix this deficiency once and for all since recursive types are rational terms.

Although unification of rational terms is a well-studied area, the reuse of existing approaches in relational settings requires reiteration. In relational programming a distinctive search strategy – interleaving search [5] – is employed (in contrast with Prolog, where depth-first search is conventionalized). This search strategy utilizes quite a different persistent/immutable interior data structures in order to make it possible to switch between different search tree branches with no performance penalty. At the same time the only used in practice [6-7] Huet’s algorithm [8] for rational terms unification, while showing brilliant results in many applications, doesn’t work well with a truly persistent intermediate state representation. Additionally, the correctness properties

(soundness and completeness) of existing algorithms were never certified using proof automation tools.

In this work we study an algorithm by Martelli and Rossi [9] which explores an interesting approach to avoid infinite looping by reducing the size of equation system. We reformulate the algorithm in convenient terms to be used in practice and make a certified implementation with the aid of the Rocq proof assistant in purely intuitionistic logic. From this implementation a correct-by-construction Haskell or OCaml program can be automatically extracted by Rocq.

The paper is organized as follows. In Section 2 we introduce rational terms, variable substitutions, and prove their main properties. In Section 3 we introduce equation systems as an intermediate state of our algorithm, the algorithm itself, and prove its correctness and termination properties. In Section 4 we briefly describe the differences between the presented text and the Rocq implementation. In Section 5 we discuss related work; Section 6 concludes.

2. Rational Terms

In this section we introduce the notions of rational terms and substitutions, and discuss their properties.

First, we define the set of variables $x, y, z \in \mathbb{V}$ and the set of function symbols (named “constructors”) $f, g, h \in \mathbb{C}$ as countably infinite sets. Also, we assume there is a function $arity : \mathbb{C} \rightarrow \mathbb{N}$ which determines the arities of constructors and denote constructors as f^n where $f \in \mathbb{C}$ is a constructor and $n = arity(f)$ is its arity. In the examples we omit the explicit arity specification since it can be easily reconstructed.

2.1 Syntax

In order to define terms and infinite terms we first define an endofunctor $F : \text{Set} \rightarrow \text{Set}$:

$$F(X) = \mathbb{V} \cup \{f^n(t_1, \dots, t_n) \mid f^n \in \mathbb{C}, t_i \in X\}.$$

We define the set of terms T as the least fixed point of F : $T = \mu F$. It corresponds to the following syntactic definition:

$$T ::= \mathbb{V} \mid \mathbb{C}^n(T_1, \dots, T_n).$$

On the other hand, we define the set of (possibly) infinite terms T^∞ as the greatest [10] fixed point of F : $T^\infty = \nu F$. As a result, T consists of all finite Herbrand trees and T^∞ consists of all possibly infinite Herbrand trees, thus $T \subset T^\infty$.

We define the set of subterms of a term, $subterms : T^\infty \rightarrow 2^{T^\infty}$, in a conventional way inductively:

- $t \in subterms(t)$;
- $t \in subterms(t_i) \Rightarrow t \in subterms(f^n(\dots, t_i, \dots))$.

We say that possibly infinite term $t \in T^\infty$ is rational iff the set of its subterms is finite¹ [1]. Now we can define the set of rational terms: $T^R = \{t \in T^\infty \mid subterms(t) \text{ is finite}\}$. For example, $f(f(f(\dots)))$ is a rational term since it has a single subterm (itself) but $f_1(f_2(f_3(\dots)))$, where the ellipsis stands for the sequence of different constructor applications, is not. Obviously, any finite term is rational, thus $T \subset T^R$.

2.2 Substitution

We define variable substitution in a conventional way and denote the substitution of all occurrences of a variable x in a term $t \in T$ with a term $t' \in T$ as $t[x \mapsto t']$. The substitution naturally extends to infinite terms.

¹ In the text, we assume equality of infinite terms to be equivalent to their bisimilarity.

More precisely, we define a substitution $\sigma \in \Sigma$ as a mapping $\mathbb{V} \rightarrow T^\infty$ with finite substitution domain $dom : \Sigma \rightarrow 2^{\mathbb{V}}$ which is defined as a set of variables that the substitution maps to another term:

$$dom(\sigma) = \{x \in \mathbb{V} \mid \sigma(x) \neq x\}.$$

The most trivial examples of substitutions are empty substitution $[]$ and singleton substitution $[x \mapsto t]$ with domains $dom([]) = \emptyset$ and $dom([x \mapsto t]) = \{x\}$ (if $t \neq x$) respectively.

The substitution application $T^\infty \times \Sigma \rightarrow T^\infty$, denoted in a postfix manner as $t\sigma$, is a homomorphic extension of the application as a function:

$$\begin{aligned} x\sigma &= \sigma(x), \\ f^n(t_1, \dots, t_n)\sigma &= f^n(t_1\sigma, \dots, t_n\sigma). \end{aligned}$$

We say that substitution σ is rational iff its image is rational for every variable: $\forall x \in \mathbb{V}. \sigma(x) \in T^R$. Obviously, the application of a rational substitution to a rational term gives a rational term.

Consider now a substitution σ that unifies the variable x with the term t , i.e., $\sigma(x) = t\sigma$. In a finite case we clearly know that $x \notin subterms(t)$ or $t = x$. But in an infinite case we have a more general fact. For example, let $\sigma = [x \mapsto f(f(\dots))]$. We may see that

$$\sigma(x) = f(f(\dots)) = \dots = f(f(x))\sigma = f(x)\sigma.$$

Lemma 2.1 (Recursive unifier). Let us have a substitution σ , a variable x and a possibly infinite term $t \neq x$. If $\sigma(x) = t\sigma$, then $\sigma(x) = t'\sigma$, where $t' = t[x \mapsto t']$.

Proof. If $x \notin subterms(t)$, the claim is obvious since $t' = t[x \mapsto t'] = t$.

Otherwise, we may see that $t\sigma$ forms a term where at each variable x occurrence we encounter the term $t\sigma$ again. Descending recursively, we will encounter $t\sigma$ over and over again infinitely. Therefore, actually $\sigma(x)$ is not only $t\sigma$ but also $t[x \mapsto t]\sigma$ and moreover $t[x \mapsto t] \dots [x \mapsto t]\sigma$.

Since after the first substitution of x we have eliminated all its occurrences except those in t , $t[x \mapsto t][x \mapsto t] = t[x \mapsto t[x \mapsto t]]$. Finally:

$$\begin{aligned} \sigma(x) &= t[x \mapsto t] \dots [x \mapsto t]\sigma \\ &= t[x \mapsto t[x \mapsto t]][x \mapsto t] \dots [x \mapsto t]\sigma \\ &\dots = t[x \mapsto t[x \mapsto t[x \mapsto \dots]]]\sigma \\ &= t'\sigma, \\ t' &= t[x \mapsto t']. \end{aligned}$$

QED.

We define substitution composition $\sigma_1 \circ \sigma_2$ as the substitution with the following universal property:

$$t\sigma_1\sigma_2 = t(\sigma_2 \circ \sigma_1).$$

In other words, we consider substitution composition as function composition w.r.t. the substitution application. Similarly to functions, substitutions form a monoidal structure under composition.

2.3 Unification Properties

We say that substitution σ is a unifier for the terms $t_1, t_2 \in T^\infty$ iff its application to both terms gives equal results: $t_1\sigma = t_2\sigma$. Also, for some substitution σ we may consider a “set of unified terms”:

$$\{(t_1, t_2) \in T^\infty \times T^\infty \mid t_1\sigma = t_2\sigma\}.$$

We define a preorder relation “to be more general” $\sigma_1 \leq \sigma_2$ between substitutions in a conventional [11] manner: $\exists \sigma \in \Sigma. \sigma_2 = \sigma \circ \sigma_1$. Obviously, the relation is reflexive, transitive and not symmetric, and also not antisymmetric. For example, consider $\sigma_1 = [x \mapsto y]$ and $\sigma_2 = [y \mapsto x]$. We have $\sigma_2 = [y \mapsto x] \circ \sigma_1$, so $\sigma_1 \leq \sigma_2$, and $\sigma_1 = [x \mapsto y] \circ \sigma_2$, so $\sigma_2 \leq \sigma_1$, but $\sigma_1 \neq \sigma_2$.

We may notice an important connection between the generality of substitutions and their unifying properties. If σ_1 unifies the terms $t_1, t_2 \in T^\infty$ then any σ_2 such that $\sigma_1 \leq \sigma_2$ unifies them, too:

$$t_1\sigma_2 = t_1(\sigma \circ \sigma_1) = t_1\sigma_1\sigma = t_2\sigma_1\sigma = t_2(\sigma \circ \sigma_1) = t_2\sigma_2.$$

Taking this into account, we may see that $\leq$ is a partial order w.r.t. the sets of unified terms. Indeed, if some $\sigma_1 \leq \sigma_2$ and $\sigma_2 \leq \sigma_1$, we may see that they unify the same pairs of terms so their sets of

unified terms are equal. This also explains the usage of “less or equal” notation in contrast with “more general” pronunciation.

We say that for given terms $t_1, t_2 \in T^\infty$ a substitution σ' is a “unifying extension” of a substitution σ iff it is a unifier for t_1, t_2 and $\sigma \leq \sigma'$.

Since we have the preorder structure on substitutions, we may consider a “minimal” substitution satisfying some property. In particular, instantiating this notion with the unifier property, we get the most general unifier (MGU) property:

$$t_1\sigma = t_2\sigma \wedge \forall \sigma' \in \Sigma. t_1\sigma' = t_2\sigma' \Rightarrow \sigma \leq \sigma'.$$

Additionally, we may consider a “minimal extension” of a substitution σ and its minimal unifying extension σ_1 for terms $t_1, t_2 \in T^\infty$:

$$\sigma \leq \sigma_1 \wedge t_1\sigma_1 = t_2\sigma_1 \wedge \forall \sigma_2 \in \Sigma. \sigma \leq \sigma_2 \wedge t_1\sigma_2 = t_2\sigma_2 \Rightarrow \sigma_1 \leq \sigma_2.$$

Obviously:

- a minimal unifying extension is a unifying extension, too;
- if some σ unifies $t_1, t_2 \in T^\infty$ it is already a minimal unifying extension of itself;
- a minimal unifying extension of the empty substitution is MGU.

As a corollary of Lemma 2.1, we may form a minimal unifying extension for any unbound (that is not in the domain) variable in a generic way. The proof is omitted due to its triviality.

Lemma 2.2. Let us have a substitution σ , a variable x and a term $t \in T^\infty$ such that $x \notin \text{dom}(\sigma)$ and $t\sigma \neq x$. Then $\sigma' = [x \mapsto t\sigma] \circ \sigma$ is a minimal unifying extension of σ for x and t .

3. Rational Unification

In this section we introduce equation systems and the unification algorithm, and prove its correctness and termination properties.

For convenience we will use the notation $A^? = \{\perp\} \cup A$ for some set A and call it “optional A ”. Also, we use the notation π_i for the i -th projection of a tuple; for example, $\pi_1 : A \times B \rightarrow A$.

3.1 Equation System

We define an equation $e \in \mathcal{E}$ as a pair:

- a pointed finite set $X \subset \mathbb{V}$ named “left-hand side” with some variable named “representing variable” chosen as a base point;
- an optional non-variable term $t \in T^? \setminus \mathbb{V}$ named “the right-hand side”.

For convenience we will use the notations $lhs : \mathcal{E} \rightarrow 2^{\mathbb{V}}$, $repr : \mathcal{E} \rightarrow \mathbb{V}$ and $rhs : \mathcal{E} \rightarrow T^?$ for corresponding equation components. Also, we will use the notation $\underline{x}, y, z \equiv t$, where $lhs(\underline{x}, y, z \equiv t) = \{x, y, z\}$, $repr(\underline{x}, y, z \equiv t) = x$ and $rhs(\underline{x}, y, z \equiv t) = t$.

We define an equation system $\xi \in \Xi$ as a finite set of equations $\Xi \subset 2^{\mathcal{E}}$ with pairwise disjunctive left-hand sides. For convenience we will use the notation ξ, e for the equation system extension $\{e\} \cup \{e' \in \xi \mid lhs(e) \cap lhs(e') = \emptyset\}$.

We consider our equation definition as a representation of the special case of equation system in conventional sense. For example:

Conventional equation system	Our equation
$\begin{cases} x = f(z) \\ y = x \\ z = x \end{cases}$	$\underline{x}, y, z \equiv f(z)$

In this sense, we may see that trivial equations of the form $\underline{x} \equiv \perp$ do not equate anything so we consider them illegal, i.e., $\forall x \in \mathbb{V}, \xi \in \Xi. \underline{x} \equiv \perp \notin \xi$. However, we will still allow them in $\mathcal{E}$ for the simplicity of the following definitions.

The main operation defined on equation systems is $walk : \Xi \times \mathbb{V} \rightarrow \mathcal{E} \times T$. This operation looks up for a variable in an equation system returning a corresponding pair of an equation and a term. More formally:

$$walk(\xi, x) = \begin{cases} (\underline{x} \equiv \perp, x) & \text{if there is no } e \in \xi \text{ such that } x \in lhs(e), \\ (e, repr(e)) & \text{if } e \in \xi, x \in lhs(e) \text{ and } rhs(e) = \perp, \\ (e, rhs(e)) & \text{otherwise.} \end{cases}$$

Basically, we may consider our definition of equation system as an “enhanced” triangular substitution [11] w.r.t. $walk$. For example:

Triangular substitution	Equation system
$\begin{cases} x \mapsto f(z) \\ y \mapsto x \\ z \mapsto w \end{cases}$	$\begin{cases} \underline{x}, y \equiv f(z) \\ z, w \equiv \perp \end{cases}$

Indeed, we may consider equation systems as substitutions (denoted as $\llbracket \cdot \rrbracket : \Xi \rightarrow \Sigma$) which are being applied recursively to a term. More formally, we may consider a homomorphic extension of $walk$ w.r.t. the second projection of its result and apply it infinitely:

$$\begin{aligned} x \llbracket \xi \rrbracket &= \pi_2(walk(\xi, x)) \llbracket \xi \rrbracket \\ f^n(t_1, \dots, t_n) \llbracket \xi \rrbracket &= f^n(t_1 \llbracket \xi \rrbracket, \dots, t_n \llbracket \xi \rrbracket) \end{aligned}$$

Consider an application of an equation system ξ to a variable x . There are two main cases:

- if $\pi_2(walk(\xi, x)) = y \in \mathbb{V}$, we have $x \llbracket \xi \rrbracket = y \llbracket \xi \rrbracket = \dots = y \llbracket \xi \rrbracket = \dots$ that leads to an infinite recursion so we must let $x \llbracket \xi \rrbracket = y$;
- otherwise, we simply descend into the right-hand side term.

Therefore, equation system application may be defined as a corecursive function:

```

function ApplyEqSys( $\xi \in \Xi, t \in T^\infty$ )
  if  $x \leftarrow t \in \mathbb{V}$  then
     $(\_, t) \leftarrow walk(\xi, x)$ 
  if  $x \leftarrow t \in \mathbb{V}$  then
    return  $x$ 
  if  $f^n(t_1, \dots, t_n) \leftarrow t$  then
    return  $f^n(ApplyEqSys(\xi, t_1), \dots, ApplyEqSys(\xi, t_n))$ 

```

Under the equation system application, we may consider equation systems as a special case of substitutions.

Theorem 3.1. Given an equation system $\xi \in \Xi$, $\llbracket \xi \rrbracket$ is a rational substitution.

Proof. Let $subterms(\perp) = \emptyset$; for any equation $e \in \mathcal{E}$, $subterms(e) = \{repr(e)\} \cup subterms(rhs(e))$ and for any equation system $\xi \in \Xi$, $subterms(\xi) = \bigcup_{e \in \xi} subterms(e)$. Since ξ is finite and $subterms(t)$ is finite for any $t \in T$, too, $subterms(\xi)$ is finite.

Given a variable x , let $T_x = \{x\} \cup subterms(\xi)$ which is obviously finite. It is easy to show that $subterms(\llbracket \xi \rrbracket(x)) \subseteq \{t \llbracket \xi \rrbracket \mid t \in T_x\}$ so it must be finite. Therefore, $\llbracket \xi \rrbracket$ is a rational substitution.

QED.

The second key operation of equation systems is $union : \Xi \times \mathbb{V} \times \mathbb{V} \rightarrow \Xi \times (T \times T)^?$. Given two variables x, y and an equation system ξ , this operation builds a new equation system from ξ such that x, y belong to the same equation.

More formally:

```

1. function UnOionEqSys( $\xi \in \Xi, x, y \in \mathbb{V}$ )
2.   ( $e_x, \_$ )  $\leftarrow$  walk( $\xi, x$ )
3.   ( $e_y, \_$ )  $\leftarrow$  walk( $\xi, y$ )
4.   if repr( $e_x$ ) = repr( $e_y$ ) then
5.     return ( $\xi, \perp$ )
6.   if rhs( $e_x$ ) =  $\perp$  and rhs( $e_y$ ) =  $\perp$  then
7.     return ( $[\xi, \text{lhs}(e_x) \cup \text{lhs}(e_y) \equiv \perp], \perp$ )
8.   if rhs( $e_x$ ) =  $\perp$  then
9.     return ( $[\xi, \text{lhs}(e_x) \cup \text{lhs}(e_y) \equiv \text{rhs}(e_y)], \perp$ )
10.  if rhs( $e_y$ ) =  $\perp$  then
11.    return ( $[\xi, \text{lhs}(e_x) \cup \text{lhs}(e_y) \equiv \text{rhs}(e_x)], \perp$ )
12.  return ( $[\xi, \text{lhs}(e_x) \cup \text{lhs}(e_y) \equiv \text{rhs}(e_x)], (\text{rhs}(e_x), \text{rhs}(e_y))$ )

```

When we add a new equation at lines 6-12, we may choose either *repr*(e_x) or *repr*(e_y) as the representing variable. The same is true about the right-hand side of the new equation at line 12: it could be either *rhs*(e_x) or *rhs*(e_y). As we can see, the second element of *union* result is intended to inform a caller about a collision in right-hand sides during the union.

We consider the following properties of *union* and leave them without pen-and-paper proofs:

- if *union*(ξ, x, y) = ($\xi', \perp$), then $\llbracket \xi' \rrbracket$ is a minimal unifying extension of $\llbracket \xi \rrbracket$ for x, y (mostly done by Lemma 2.2);
- if *union*(ξ, x, y) = ($\xi_1, (t_1, t_2)$) and $\llbracket \xi_2 \rrbracket$ is a minimal unifying extension of $\llbracket \xi_1 \rrbracket$ for t_1, t_2 , then $\llbracket \xi_2 \rrbracket$ is a minimal unifying extension of $\llbracket \xi \rrbracket$ for x, y ;
- if *union*(ξ, x, y) = ($\xi_1, (t_1, t_2)$) and $\llbracket \xi_2 \rrbracket$ is a unifying extension of $\llbracket \xi \rrbracket$ for x, y , then $\llbracket \xi_2 \rrbracket$ is a unifying extension of $\llbracket \xi_1 \rrbracket$ for t_1, t_2 .

These properties are required for the soundness and completeness of the unification algorithm. Some other properties required for the termination are discussed below.

3.2 Common Part

We define a “common part” [9] of two terms *common_part* : $T \times T \rightarrow T^?$ inductively:

$$\begin{aligned}
\text{common_part}(x, _) &= x \\
\text{common_part}(_, y) &= y \\
\text{common_part}(f^n(\dots, l_i, \dots), f^n(\dots, r_i, \dots)) &= f^n(\dots, \text{common_part}(l_i, r_i), \dots)
\end{aligned}$$

The definition is intentionally left non-total since two terms may not have a common part (in such cases the result is $\perp$) and slightly non-deterministic (in the case of two variables) to allow optimizations. Hereafter we require the implementation of *common_part* to be a total deterministic function.

There is a connection between common part existence and unifiability:

- common_part*(t_1, t_2) = t iff *common_part*(t_2, t_1) = t (as a predicate);
- if some substitution σ is a unifier of $t_1, t_2 \in T$ and *common_part*(t_1, t_2) = t , then $t \neq \perp$ and σ is a unifier of t_1, t (and of t_2, t by the previous point).

3.3 The Unification Algorithm

We define the unification algorithm as a recursive function *unify* : $\Xi \times T \times T \rightarrow \Xi^?$:

```

1. function RationalUnify( $\xi \in \Xi, t_1, t_2 \in T$ )
2.   if  $x \leftarrow t_1 \in \mathbb{V}$  and  $y \leftarrow t_2 \in \mathbb{V}$  then
3.     ( $\xi, ts$ )  $\leftarrow$  union( $\xi, x, y$ )
4.     if ( $t_1, t_2$ )  $\leftarrow ts$  then
5.       return RationalUnify( $\xi, t_1, t_2$ )
6.     return  $\xi$ 
7.   if  $x \leftarrow t_1 \in \mathbb{V}$  then
8.     ( $e, \_$ )  $\leftarrow$  walk( $\xi, x$ )
9.     if rhs( $e$ ) =  $\perp$  then
10.      return  $\xi, \text{lhs}(e) \equiv t_2$ 
11.      $t \leftarrow \text{common\_part}(\text{rhs}(e), t_2)$ 
12.     if  $t = \perp$  then
13.       failure
14.      $\xi \leftarrow \xi, \text{lhs}(e) \equiv t$ 
15.     return RationalUnify( $\xi, \text{rhs}(e), t_2$ )
16.   if  $y \leftarrow t_2 \in \mathbb{V}$  then
17.     return RationalUnify( $\xi, y, t_1$ )
18.   if  $f^n(l_1, \dots, l_n) \leftarrow t_1$  and  $g^m(r_1, \dots, r_m) \leftarrow t_2$  then
19.     if  $f^n \neq g^m$  then
20.       failure
21.     for  $i = 1, \dots, n$  do
22.        $\xi \leftarrow \text{RationalUnify}(\xi, l_i, r_i)$ 
23.     return  $\xi$ 

```

Theorem 3.2 (Soundness). Let us have $\xi \in \Xi$ and $t_1, t_2 \in T$. If *unify*(ξ, t_1, t_2) = $\xi' \neq \perp$, then $\llbracket \xi' \rrbracket$ is a minimal unifying extension of $\llbracket \xi \rrbracket$ for t_1, t_2 .

Proof. Mostly done by induction on the inference tree using the properties of *union*.

The only interesting case is when $t_1 \in \mathbb{V}$ (let us name it x) and $t_2 \notin \mathbb{V}$:

- if *walk*(ξ, x) = ($e, _$) and *rhs*(e) = $\perp$, the proof follows from Lemma 2.2;
- otherwise, we have *common_part*(*rhs*(e), t_2) = $t \neq \perp$ and *unify*($[\xi, \text{lhs}(e) \equiv t], \text{rhs}(e), t_2$) = $\xi' \neq \perp$ such that $\llbracket \xi' \rrbracket$ is a minimal unifying extension of $\llbracket \xi, \text{lhs}(e) \equiv t \rrbracket$ for *rhs*(e), t_2 (by the induction hypothesis), we may prove that $\llbracket \xi' \rrbracket$ is a minimal unifying extension of $\llbracket \xi \rrbracket$ for x, t_2 similarly to *union*.

QED.

The immediate corollary of this theorem is that *unify*($\emptyset, t_1, t_2$) is an MGU of t_1, t_2 (unless it is not $\perp$).

Theorem 3.3 (Completeness). Let us have $\xi \in \Xi$ and $t_1, t_2 \in T$. If *unify*(ξ, t_1, t_2) = $\perp$, then there is no unifying extension of $\llbracket \xi \rrbracket$ for t_1, t_2 .

Proof. The proof is mostly similar to the proof of the previous theorem.

The only interesting case is when $t_1 \in \mathbb{V}$ ($x = t_1$), $t_2 \notin \mathbb{V}$, *walk*(ξ, x) = ($e, _$) and *rhs*(e) $\neq \perp$:

- if *common_part*(*rhs*(e), t_2) = $t = \perp$, there is no unifier of *rhs*(e), t_2 and therefore no unifying extension of $\llbracket \xi \rrbracket$ for x, t_2 , too;
- otherwise, if there is a unifying extension of $\llbracket \xi, \text{lhs}(e) \equiv t \rrbracket$ for *rhs*(e), t_2 , then we may prove that it must be a unifying extension of $\llbracket \xi \rrbracket$ for x, t_2 , too (similarly to *union*).

QED.

We may² also consider a “direct” unification of rational terms $t_1, t_2 \in T^R$. Although it doesn’t straightforwardly come from the unification of finite terms, it could be obtained using an equation system representation for them. The fact [12] is that any rational term may be represented as a pair of an equation system and a finite term, so we may “include” the equation systems of t_1, t_2 into the current state and apply the algorithm as usual. In practice, it is not much helpful to unify rational terms directly since we may always represent them as pairs of an equation system and a finite term.

3.4 Termination of the Algorithm

To prove the termination of the algorithm we need to introduce some new notions.

3.4.1 Roots

We say that a variable x is a “root” of an equation system ξ iff it is either:

- a representing variable for some equation;
- or it does not occur in the left-hand sides of ξ .

More formally, we say that x is a root of ξ iff $\text{repr}(\pi_1(\text{walk}(\xi, x))) = x$.

We define the roots of an equation system modulo a finite variable set X , $\text{roots}_X : \Xi \rightarrow 2^{\mathbb{V}}$ as:

$$\text{roots}_X(\xi) = \{x \in X \mid \text{repr}(\pi_1(\text{walk}(\xi, x))) = x\}.$$

The motivation for the introduction of roots as a separate concept (instead of the representing variables) is the unification for unbound variables. For example, when we unify x with $f(y)$ and there are no equations with x in the left-hand side, we are required to add a new equation $x \equiv f(y)$. It increases the number of representing variables but doesn’t change the number of roots modulo X if $x \in X$.

Indeed, we may consider an equation system as if there was an equation for every $x \in \mathbb{V}$ by just populating it by trivial equations in the form $x \equiv \perp$. We may notice that the behavior of walk would remain unaffected in this case. So, a root of ξ is just a representing variable of a populated in such a manner version of ξ . The issue of this approach is that after the population of a system with non-essential equations their number (and the number of representing variables) will become infinite and therefore cannot be reduced by one during unification. So, we additionally use a finite set X to focus our attention only at “interesting” variables.

3.4.2 Non-variable size

We define non-variable size of a term $t \in T$ as a number of its constructors. More formally, we consider the following recursively-defined function $\text{size} : T \rightarrow \mathbb{N}$:

$$\begin{aligned} \text{size}(x) &= 0 \\ \text{size}(f^n(t_1, \dots, t_n)) &= 1 + \sum_{i=1}^n \text{size}(t_i) \end{aligned}$$

We also consider a non-variable size of an equation system ξ as the sum of non-variable sizes of its right-hand sides, $\text{size} : \Xi \rightarrow \mathbb{N}$, and a size of an equation system ξ modulo a finite variable set X as a pair of the number of roots modulo X and the non-variable size of ξ , $\text{size}_X : \Xi \rightarrow \mathbb{N} \times \mathbb{N}$:

$$\begin{aligned} \text{size}(\xi) &= \sum_{\substack{e \in \xi \\ \text{rhs}(e) \neq \perp}} \text{size}(\text{rhs}(e)), \\ \text{size}_X(\xi) &= (|\text{roots}_X(\xi)|, \text{size}(\xi)). \end{aligned}$$

² This claim is not proven in Rocq since it requires the axiom of choice.

3.4.3 Cost

We define the “cost” of two terms, $\text{cost} : T \times T \rightarrow \mathbb{N}$, as a recursively-defined function:

$$\begin{aligned} \text{cost}(x, t_2) &= \text{size}(t_2) \\ \text{cost}(t_1, y) &= \text{size}(t_1) \\ \text{cost}(f^n(\dots, l_i, \dots), g^m(\dots, r_j, \dots)) &= 1 + \sum_{i=1}^{\min(n, m)} \text{cost}(l_i, r_i) \end{aligned}$$

In fact, independently of the actual definition, we only need that:

- $\text{cost}(x, t) = \text{cost}(t, x) = \text{size}(t)$ for any $x \in \mathbb{V}, t \in T$;
- if $\text{common_part}(t_1, t_2) = t \neq \perp$, then $\text{cost}(t_1, t_2) + \text{size}(t) = \text{size}(t_1) + \text{size}(t_2)$.

3.4.4 Unification task

We define unification task $\tau \in \mathcal{T}$ as a triple of an equation system and two terms: $\tau = \Xi \times T \times T$. Now we can define a “recursive call” relation between unification tasks $\tau_1, \tau_2 \in \mathcal{T}$ modulo a finite set $X \subset \mathbb{V}$ denoted as $\tau_1 \xrightarrow{X} \tau_2$ as presented in **Ошибка! Источник ссылки не найден.** The definition is aligned with all 4 possible recursive call kinds in the definition of *unify*. More precisely (the item numbers are the listing line numbers):

5. UnionCall;
16. CommonPartCall;
18. SwapVarTermCall;
23. SubtermCall.

We define a unification task size modulo a finite set $X \subset \mathbb{V}$, $\text{size}_X : \mathcal{T} \rightarrow \mathbb{N} \times \mathbb{N}$ as:

$$\text{size}_X(\xi, t_1, t_2) = (|\text{roots}_X(\xi)|, \text{size}(\xi) + \text{cost}(t_1, t_2)).$$

Further we consider the lexicographical order on pairs. Under these definitions we may see that every considered recursive call strictly reduces a unification task size in a constant number of steps.

Moreover, we may prove that $\xrightarrow{X}$ is a well-founded relation for any $X \subset \mathbb{V}$.

Lemma 3.1 ($\xrightarrow{X}$ is well-founded). Given a finite set $X \subset \mathbb{V}$, $\xrightarrow{X}$ is a well-founded relation.

Proof. Given $(\xi_1, t_1, t_2) \in \mathcal{T}$, we need to show that there is no an infinite recursive call chain:

$$(\xi_1, t_1, t_2) \xrightarrow{X} \tau_2 \xrightarrow{X} \dots \xrightarrow{X} \tau_n \xrightarrow{X} \dots$$

$\frac{x, y \in \mathbb{V} \quad \text{roots}_X(\xi_2) < \text{roots}_X(\xi_1) }{(\xi_1, x, y) \xrightarrow{X} (\xi_2, t_1, t_2)}$	(UnionCall)
$\frac{x \in \mathbb{V} \quad t_x, t_y \notin \mathbb{V} \quad \text{common_part}(t_x, t_y) = t \neq \perp \quad \text{roots}_X(\xi_1) = \text{roots}_X(\xi_2) \quad \text{size}(\xi_1) + \text{size}(t) = \text{size}(\xi_2) + \text{size}(t_x)}{(\xi_1, x, t_y) \xrightarrow{X} (\xi_2, t_x, t_y)}$	(CommonPartCall)
$\frac{y \in \mathbb{V} \quad t \notin \mathbb{V}}{(\xi, t, y) \xrightarrow{X} (\xi, y, t)}$	(SwapVarTermCall)
$\frac{1 \leq i \leq n \quad \text{size}_X(\xi_2) \leq (\text{roots}_X(\xi_1) , \text{size}(\xi_1) + \sum_{j=1}^{i-1} \text{cost}(l_j, r_j))}{(\xi_1, f^n(l_1, \dots, l_n), g^n(r_1, \dots, r_n)) \xrightarrow{X} (\xi_2, l_i, r_i)}$	(SubtermCall)

Fig. 1. The definition of $\tau_1 \xrightarrow{X} \tau_2$.

Well-founded induction on the unification task size modulo X . In other words, we show that any recursive call chain started from (ξ_1, t_1, t_2) is finite if it holds for any $\tau' \in \mathcal{T}$ such that $size_X(\tau') < size_X(\xi_1, t_1, t_2)$, we call it “induction hypothesis” or IH. Case analysis on t_1 :

- If $t_1 \in \mathbb{V}$ (let us call it x), the only allowed recursive call kinds are (UnionCall) and (CommonPartCall):
 - ✓ We have $t_2 = y \in \mathbb{V}$, $(\xi_1, x, y) \xrightarrow{X} (\xi_2, t'_1, t'_2)$ and $|roots_X(\xi_2)| < |roots_X(\xi_1)|$. Obviously, we may use IH since the latter implies $size_X(\xi_2, t'_1, t'_2) < size_X(\xi_1, x, y)$ by lexicographic ordering.
 - ✓ We have $(\xi_1, x, t_2) \xrightarrow{X} (\xi_2, t_x, t_2)$, $t_x, t_2 \notin \mathbb{V}$, $common_part(t_x, t_2) = t \neq \perp$, $|roots_X(\xi_1)| = |roots_X(\xi_2)|$ and $size(\xi_1) + size(t) = size(\xi_2) + size(t_x)$.
 1. Let us show that $size_X(\xi_1, x, t_2) = size_X(\xi_2, t_x, t_2)$:
 - Since $|roots_X(\xi_1)| = |roots_X(\xi_2)|$, we only need to show that $size(\xi_1) + cost(x, t_2) = size(\xi_2) + cost(t_x, t_2)$.
 - By the definition, $cost(x, t_2) = size(t_2)$.
 - By the property of common part and $cost$, $cost(t_x, t_2) + size(t) = size(t_x) + size(t_2)$.
 - Finally:

$$\begin{aligned} size(\xi_1) + \underline{cost(x, t_2)} + size(t) &= size(\xi_1) + \underline{size(t_2)} + size(t) \\ &= size(\xi_1) + \underline{size(t)} + size(t_2) \\ &= size(\xi_2) + \underline{size(t_x)} + size(t_2) \\ &= size(\xi_2) + \underline{cost(t_x, t_2)} + size(t). \end{aligned}$$
 - 2. Let us consider the possible recursive call kinds from (ξ_2, t_x, t_2) . Since $t_x, t_2 \notin \mathbb{V}$, it is only (SubtermCall). We may see that any (SubtermCall) recursive call strictly reduces $cost$ of the terms because of the top-level constructors, so we may use IH.
- If $t_1 \notin \mathbb{V}$, the only allowed recursive call kinds are (SwapVarTermCall) and (SubtermCall). The first one just swaps terms of the task and immediately turns into the previous case. The latter always strictly reduces the terms' $cost$ because of the top-level constructors so we may use IH.

QED.

3.4.5 The Proof of Termination

Finally, we define the variables of an equation system $vars : \Xi \rightarrow 2^{\mathbb{V}}$ as the set of variables which occur anywhere in the equation system.

We can prove the following properties of $union$:

- if $union(\xi, x, y) = (\xi', \perp)$, $vars(\xi) \subseteq X$ and $x, y \in X$, then $|roots_X(\xi')| \leq |roots_X(\xi)|$;
- if $union(\xi, x, y) = (\xi', \perp)$, then $size(\xi') \leq size(\xi)$;
- if $union(\xi, x, y) = (\xi', \perp)$, then $vars(\xi') \subseteq \{x, y\} \cup vars(\xi)$;
- if $union(\xi, x, y) = (\xi', (t_1, t_2))$ and $vars(\xi) \subseteq X$, then $|roots_X(\xi')| = |roots_X(\xi)| + 1$;
- if $union(\xi, x, y) = (\xi', (t_1, t_2))$, then $vars(\xi') \subseteq vars(\xi)$ and there are $e_1, e_2 \in \xi$ such that $rhs(e_1) = t_1$ and $rhs(e_2) = t_2$.

Using these properties, we first prove that if $unify(\xi, t_1, t_2) = \xi' \neq \perp$, then $vars(\xi') \subseteq vars(t_1) \cup vars(t_2) \cup vars(\xi)$ where $vars : T \rightarrow 2^{\mathbb{V}}$ is defined similarly to the $vars$ of an equation system.

Theorem 3.4 (Termination). Given $\xi \in \Xi$ and $t_1, t_2 \in T$, the computation of $unify(\xi, t_1, t_2)$ terminates.

Proof. We need to prove a more general fact here: for any finite set $X \subset \mathbb{V}$ such that $vars(\xi) \cup vars(t_1) \cup vars(t_2) \subseteq X$, the algorithm terminates and if its result $\xi' \neq \perp$, $size_X(\xi') \leq size_X(\xi, t_1, t_2)$. It means that we can increase the equation system size up to $cost(t_1, t_2)$.

This fact straightforwardly comes from the properties of $union$ and the previous lemma by well-founded induction on $\xrightarrow{X}$. Finally, we instantiate the generalized fact with $X = vars(\xi) \cup vars(t_1) \cup vars(t_2)$.

QED.

4. Specification in Rocq

In this section we describe some simplifications and tricks used in the formally verified proof in the Rocq proof assistant. Every statement formulated before has been formally proven in Rocq [13] not using any axioms including any kind of extensionality or the law of excluded middle. However, in many cases we require the decidability of finite sets to get along with intuitionistic logic. The algorithm is successfully extracted from the Rocq specification as a Haskell program.

For technical simplicity, we assume that all constructors have arity equal to either 0 or 2. This restriction doesn't affect the generality of the proof since we may encode any complex term as a list with a nullary constructor as a tag [14]. More precisely, we deal with the following definitions of term and possibly infinite term:

```
Definition name := nat.
Inductive term : Set :=
| Var : name -> term
| Cst : name -> term
| Con : name -> term -> term -> term.
CoInductive inf_term : Set :=
| InfVar : name -> inf_term
| InfCst : name -> inf_term
| InfCon : name -> inf_term -> inf_term -> inf_term.
```

Although the conventional [15] way to internalize bisimilarity of such coinductive structures is the usage of coinductively-defined predicate, we faced a number of serious limitations of this approach especially in the usage of corecursively-proven transitivity due to productivity check. Instead, we have used an alternative induction-based predicate and proved its equivalence to the conventional one:

```
CoInductive inf_term_coinductive_eq : inf_term -> inf_term -> Prop :=
| InfVarEq x : inf_term_coinductive_eq (InfVar x) (InfVar x)
| InfCstEq c : inf_term_coinductive_eq (InfCst c) (InfCst c)
| InfConEq f l1 l2 r1 r2 : inf_term_coinductive_eq l1 l2
-> inf_term_coinductive_eq r1 r2
-> inf_term_coinductive_eq (InfCon f l1 r1) (InfCon f l2 r2).

Inductive path : Set :=
| Here : path
| Left : path -> path
| Right : path -> path.
Inductive inf_path_to : inf_term -> path -> inf_term -> Prop :=
| InfHere t : inf_path_to t Here t
| InfLeft f l r p t : inf_path_to l p t -> inf_path_to (InfCon f l r) (Left p) t
| InfRight f l r p t : inf_path_to r p t -> inf_path_to (InfCon f l r) (Right p) t.
Definition inf_term_eq_node (l r : inf_term) : Prop :=
match l, r with
| InfVar l, InfVar r => l = r
| InfCst l, InfCst r => l = r
| InfCon l _ _, InfCon r _ _ => l = r
| _, _ => False
```

```

end.
Definition inf_term_eq (l r : inf_term) : Prop :=
  forall p l', inf_path_to l p l' -> exists r', inf_path_to r p r' /\ inf_term_eq_node l' r'.
Lemma inf_term_eq_correct t1 t2 (H : inf_term_eq t1 t2) : inf_term_coinductive_eq t1 t2.
Lemma inf_term_eq_complete t1 t2 (H : inf_term_coinductive_eq t1 t2) : inf_term_eq t1 t2.

```

While this definition allowed us to avoid issues with the productivity checker which led to the rejection of initially more compact proofs, the proofs themselves became more technically challenging.

In Rocq, instead of defining equation system as a set of equations, we encode them as a finite partial mapping (similarly to substitutions) and use representing variables to distinguish between equations. This approach is mostly inspired by triangular substitutions [11] and Union-Find [16] data structure (disjoint set union), *walk* and *union* follow the corresponding operations of the conventional Union-Find. The main advantage of this encoding is a familiarity with practical implementations of unification algorithms so one may rewrite our definitions line-by-line in another programming language and get almost ready-to-use implementation.

In Rocq, we define *common_part* as an inductive predicate and prove its decidability to avoid difficulties with non-determinism and partiality:

```

Inductive is_common_part : term -> term -> term -> Prop :=
| VarLCP x t : is_common_part (Var x) t (Var x)
| VarRCP x t : is_common_part t (Var x) (Var x)
| CstTCP c : is_common_part (Cst c) (Cst c) (Cst c)
| ConTCP f l1 l2 l r1 r2 r : is_common_part l1 l2 l -> is_common_part r1 r2 r
  -> is_common_part (Con f l1 r1) (Con f l2 r2) (Con f l r).

```

Finally, the unification algorithm itself is defined as an inductive predicate, too, in order to separate the termination proof from the correctness proofs. The termination proof is encoded directly in Rocq as was presented in the text.

Table 1. The correspondence between mentioned objects and statements in the Rocq implementation.

term, inf_term, inf_term_eq	Term, infinite term and bisimilarity of infinite terms
inf_subterm	$subterms : T^\infty \rightarrow 2^{T^\infty}$ as a predicate
is_rational_term	T^R as a predicate
term_to_inf_rational	$T \subset T^R$
inf_subst, inf_subst_empty, inf_subst_singleton, inf_subst_dom, inf_image	Substitution: type, empty and singleton instances, $dom : \Sigma \rightarrow 2^V$ and the functional representation (image)
inf_subst_apply	Substitution application
is_rational_subst	Rational substitution
inf_subst_apply_rational	Claim “application of a rational substitution to a rational term gives a rational term”
inf_subst_recursive_unifier	Lemma 2.1 “Recursive unifier”
inf_subst_eq	Equivalence of substitutions
inf_subst_compose, inf_subst_compose_spec, inf_subst_compose_empty_l, inf_subst_compose_empty_r, inf_subst_compose_assoc	Substitution composition: definition, universal property and monoidal properties
inf_subst_more_general, inf_subst_more_general_refl, inf_subst_more_general_trans, inf_subst_more_general_not_antisym	“To be more general” relation and preorder structure
inf_unifier, inf_unifier_more_general	“Unifier” property and its connection with the previous ones
inf_unifying_extension, inf_mgu, inf_min_unifying_extension, inf_min_unifying_extension_same, inf_min_unifying_extension_empty	“Unifying extension”, “minimal unifying extension”, “MGU” and relations between them

inf_min_unifying_extension_unbound	Lemma 2.2
eqsys, eqsys_lookup	Low-level representation of equation systems as a finite mapping discussed above
is_eqsys_root, eqsys_roots, eqsys_roots_num	$roots_X : \Xi \rightarrow 2^V$
term_nonvar_size, eqsys_rhs_nonvar_size, eqsys_nonvar_size_hlp, eqsys_nonvar_size	$size : T \rightarrow \mathbb{N}$ $size : \Xi \rightarrow \mathbb{N}$
eqsys_walk_result, eqsys_walkable, eqsys_well_formed	Well-formedness property of the low-level representation: there are no infinite (cyclic) chains of variables
wf_eqsys	Equation system: low-level representation with a well-formedness proof
eqsys_empty, eqsys_empty_well_formed, wf_eqsys_empty	Empty equation system
eqsys_walk_hlp, eqsys_walk_aux, wf_eqsys_walk	$walk : \Xi \times V \rightarrow \mathcal{E} \times T$
wf_eqsys_image_hlp, wf_eqsys_image, wf_eqsys_apply, wf_eqsys_to_subst	$\llbracket \cdot \rrbracket : \Xi \rightarrow \Sigma$
eqsys_subterms, wf_eqsys_inf_subterms, wf_eqsys_to_subst_rational	Theorem 3.1
is_common_part, common_part_aux, common_part	$common_part : T \times T \rightarrow T^?$ as a predicate and a function
is_common_part_sym, is_common_part_inf_unifiable, is_common_part_inf_unifiable_inv	Main properties of <i>common_part</i>
eqsys_union, eqsys_union_well_formed, wf_eqsys_union_aux, wf_eqsys_union	$union : \Xi \times V \times V \rightarrow \Xi \times (T \times T)^?$
wf_eqsys_union_spec, wf_eqsys_union_prop	All mentioned properties of <i>union</i>
rational_unify_vt_spec, rational_unify_vt_impl, rational_unify_vt_impl_well_formed, rational_unify_vt_aux, rational_unify_vt, rational_unify_vt_prop	Lines 8-14 of the definition of $unify : \Xi \times T \times T \rightarrow \Xi^?$ and their relevant properties; “vt” stands for “variable-term”
rational_unification	Definition of $unify : \Xi \times T \times T \rightarrow \Xi^?$ as a predicate
rational_unification_correct	Theorem 3.2 “Soundness”
rational_unification_complete	Theorem 3.3 “Completeness”
rational_unification_result_vars	Claim “if $unify(\xi, t_1, t_2) = \xi' \neq \perp$, then $vars(\xi') \subseteq vars(t_1) \cup vars(t_2) \cup vars(\xi)$ ”
rational_unification_system_size	$size_X : \Xi \rightarrow \mathbb{N} \times \mathbb{N}$
rational_unification_cost	$cost : T \times T \rightarrow \mathbb{N}$
rational_unification_cost_sym, rational_unification_cost_common_part	Properties of <i>cost</i>
rational_unification_task	$\mathcal{T} = \Xi \times T \times T$
rational_unification_size	$size_X : \mathcal{T} \rightarrow \mathbb{N} \times \mathbb{N}$
rational_unification_recursive_call	$\overset{X}{\rightsquigarrow}$
rational_unification_recursive_call_well_founded	Lemma 3.1 “ $\overset{X}{\rightsquigarrow}$ is well-founded”
rational_unification_task_xs, rational_unification_exists_hlp, rational_unification_exists_aux, rational_unification_exists	Theorem 3.4 “Termination”

5. Related Work

Rational terms have been studied extensively from the formulation of the notion [1] to the development of unification algorithms. Although there are many works [8-9, 17-18] presenting algorithms for first-order unification of rational terms, they all follow one of two possible approaches: either as in Huet's algorithm [8] or as in Martelli and Rossi's algorithm [9]. The first approach is to accumulate in the intermediate state a set of equivalences between input terms and the second is to accumulate equivalences between variables only, simplifying more complex terms if needed.

While the first approach works well when we may assign a unique index to every term (or use a mutable variable inside it), it does many “dereference” operations and only grows the intermediate state which poorly affects any “truly” persistent (e.g., immutable) representation of the intermediate state. In contrast, the second approach requires only “dereferencing” of variables that is needed a priori to perform a unification efficiently (e.g., using triangular substitutions) and reduces the intermediate state when it is needed to prevent infinite looping. These differences allow us to deal with a truly persistent representation of the intermediate state faster than in the first approach.

Although the correctness and termination of both approaches have been proven before, there is no mechanized specification in an automated proof checking system for any of them (at least available publicly). The most related work from this point of view is Spadotti's PhD thesis [19] providing a mechanized specification of rational trees in a coinductive and a syntactic manner, and proofs of their equivalence. However, this work is focused on equivalence theories and mechanization itself and doesn't refer to the unification problem.

Also, there are many works which address more than first-order unification: second-order [6], higher-order [20], etc. These works generally either directly rely on the first-order unification of rational terms [6] or at least deal with similar notions such as equivalence classes and representing variables [20]. The core principles (discussed above) established by the pioneers are mostly staying unchanged.

The work of Rozplochas et al. [14] is not directly connected to this work but is supported by a mechanized proof in Rocq and is related to relational programming which is known to require a truly persistent representation of the intermediate state due to utilization of the interleaving search procedure. In their work, which was a starting point for this research, the notions related to the unification were borrowed from the classical works such as [21].

6. Conclusion and Future Work

In this paper we presented a certified unification algorithm for rational terms based on the work of Martelli and Rossi [9]. The algorithm operates on equation systems as an intermediate state, and its correctness and termination have been formally verified in the Rocq proof assistant. The implementation is done in a practical-oriented style to simplify porting to other programming languages with a confidence of correctness. The full source code of the verification is provided.

The future work may include an experimental performance evaluation of our algorithm versus Huet's algorithm [8] in the context of persistent intermediate state representations. Additionally, the current Rocq implementation does not expose which variable bindings were added during the unification; this limitation prevents its direct using for formalizing of disunification and will be addressed in future.

References

- [1]. Colmerauer A. Prolog and infinite trees. *Logic programming*, vol. 16, issue 2, 1982, pp. 231-251.
- [2]. Friedman D.P., Byrd W.E., Kiselyov O., Hemann J. *The Reasoned Schemer*. 2nd ed. Cambridge, Massachusetts, The MIT Press, 2018, 189 p. DOI: 10.7551/mitpress/5801.001.0001.

- [3]. Hemann J., Friedman D.P. μ Kanren: A Minimal Functional Core for Relational Programming. In *Proc. of the 2013 Annual Workshop on Scheme and Functional Programming*, 2013, pp. 1-10.
- [4]. Byrd W.E. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis. Indiana University, 2017. 296 p.
- [5]. Kiselyov O., Shan C., Friedman D.P., Sabry A. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proc. of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*, 2005, pp. 192-203. DOI: 10.1145/1086365.1086390.
- [6]. Gauthier N., Pottier F. Numbering matters: First-order canonical forms for second-order recursive types. In *Proc. of ACM SIGPLAN Notices*, 2004, vol. 39, issue 9, pp. 150-161.
- [7]. `swipl/src/pl-prim.s.c` at 45e83178379f0fd6f918bd294bb361f3db7bf1d7 - SWI-Prolog/swipl - GitHub. Available at: <https://github.com/SWI-Prolog/swipl/blob/45e83178379f0fd6f918bd294bb361f3db7bf1d7/src/pl-prim.s.c#L71>, accessed 13.04.2026.
- [8]. Huet G. *Résolution d'Équations dans des Langages d'Ordre 1, 2, ..., ω* . PhD thesis. Université de Paris VII, 1976. 281 p.
- [9]. Martelli A., Rossi G. Efficient Unification with Infinite Terms in Logic Programming. In *Proc. of Fifth Generation Computer Systems*, 1984, pp. 202-209.
- [10]. Simon L., Mallya A., Bansal A., Gupta G. Coinductive logic programming. In *Proc. of International Conference on Logic Programming*, 2006, vol. 4079, pp. 330-345. DOI: 10.1007/11799573_25.
- [11]. Baader F., Snyder W. Unification theory. *Handbook of automated reasoning*, 2001, vol. 1, pp. 445-532. DOI: 10.1016/B978-044450813-3/50010-2.
- [12]. Courcelle B., Kahn G., Vuillemin J. Algorithms for equivalence and reduction to minimal form for a class of simple recursive equations. From *Semantics to Computer Science Essays in Honour of Gilles Kahn*, 1974, pp. 169-184. DOI: 10.1017/CBO9780511770524.009.
- [13]. `miniKanren-coq`. Available at: <https://github.com/dboulytchev/miniKanren-coq/tree/master/src/Rational>, accessed 18.08.2026.
- [14]. Rozplochas D., Vyatkin A., Boulytchev D. Certified Semantics for Relational Programming. In *Proc. of Asian Symposium on Programming Languages and Systems*, 2020, vol. 12470, pp. 167-185. DOI: 10.1007/978-3-030-64437-6_9.
- [15]. Coinductive types and corecursive functions — The Rocq Prover 9.3+alpha documentation. Available at: <https://rocq-prover.org/doc/master/refman/language/core/coinductive.html>, accessed 13.04.2026.
- [16]. Aho A.V., Hopcroft J.E., Ullman J.D. *The Design and Analysis of Computer Algorithms*. Michigan, Addison-Wesley Publishing Company, 1974, 470 p.
- [17]. Mukai K. A unification algorithm for infinite trees. In *Proc. of International Joint Conference on Artificial Intelligence*, 1983, vol. 1, pp. 547-549.
- [18]. Jaffar J. Efficient unification over infinite terms. *New Generation Computing*, 1984, vol. 2, pp. 207-219. DOI: 10.1007/BF03037057.
- [19]. Spadotti R. *A mechanized theory of regular trees in dependent type theory*. PhD thesis. Paul Sabatier University, 2016, 213 p.
- [20]. Chen Zh., Pfenning F. A Saturation-Based Unification Algorithm for Higher-Order Rational Patterns. *ACM Transactions on Computational Logic*, 2025, vol. 26, issue 1, pp. 1-33. DOI: 10.1145/3704265.
- [21]. McBride C. First-order unification by structural recursion. *Journal of functional programming*, 2003, vol. 13, issue 6, pp. 1061-1075. DOI: 10.1017/S0956796803004957.

Информация об авторах / Information about authors

Эридан Алексеевич ДОМОРАЦКИЙ – аспирант второго года обучения кафедры системного программирования математико-механического факультета Санкт-Петербургского государственного университета. Область научных интересов: языки программирования и автоматический вывод типов, функциональное программирование, реляционное программирование.

Eridan Alekseevich DOMORATSKIY – second-year postgraduate student at the Software Engineering Chair, faculty of Mathematics and Mechanics, Saint-Petersburg State University. Research interests: programming language and type inference, functional programming, relational programming.

Дмитрий Юрьевич БУЛЫЧЕВ – кандидат физико-математических наук, доцент кафедры системного программирования математико-механического факультета Санкт-Петербургского государственного университета. Область научных интересов: языки и инструменты программирования, функциональное, логическое и реляционное программирование, декларативное программирование, синтез программ.

Dmitry Yurievich BOULYTCHEV – Cand. Sci. (Phys.-Math.), Assoc. Prof. at the Software Engineering Chair, faculty of Mathematics and Mechanics, Saint-Petersburg State University. Research interests: programming languages, compilers and tools, functional, logic and relational programming, declarative programming, program synthesis.