Unification Algorithm for Objects with Classes and Substructure

Yuuichi KAWAGUCHI*  Kiyoshi AKAMA**  Eiichi MIYAMOTO***
(Received 29 November 1999)

Abstract

Unification is a basic and important operation for executing logic programs. Traditional theories of logic programming languages provide unification algorithms for terms consisting of constants, variables, and functions. There are many situations in which more complex expressions than such terms are needed to represent complicated knowledge or other objects. Although some unification algorithms have been provided for such objects, their theoretical basis lack of clarity. This paper presents an unification algorithm for objects that have class hierarchies and substructures. We provide a formal definition of the target language. The definition consists of two parts: (1) the structure of objects and (2) basic methods to operate them. Based on these theoretical basis, we present an algorithm and prove that our algorithm can unify two objects and that it terminates in a finite number of steps.

Key words : Class Hierarchy, Substructure, Class Object, Atomic Object, Unification.

1 Introduction

In many logic programming languages, resolutions are generally used for executing programs. Resolutions use unifications. Unification is a basic and important method for executing programs. Lloyd [5] provided a unification algorithm for atoms based on terms. He proved that the algorithm unifies two atoms and terminates in a finite number of steps. Logic programming languages use not only terms but also more complex objects for expressing complicated knowledge or data efficiently. For objects that have a more complex structure than terms do, unification algorithms are complicated. Previous reports [1, 2] have described such objects and provided an unification algorithm, but they maybe not interested in proofs of it.

This paper presents a unification algorithm for objects that have class hierarchies and substructure. Such objects are more complex than terms are. They can express some knowledge or data more efficiently than terms can. We have constructed a framework for expressing and executing programs based on such objects. Using the framework, we present an algorithm and prove that it is a unification algorithm and that it terminates in a finite number of steps.

2 The Language

This section provides formal definitions of the target language, LHS. Intuitive meanings of the lan-
guage are described in [3].

2.1 Program

A set $\mathbf{C}$ is given. Each element of $\mathbf{C}$ is called a constant. The set $\mathbf{C}$ has a relation over it, denoted as $\rightarrow$, and the following two conditions are assumed to hold.

Condition 1: For $\forall a \in \mathbf{C}$, any sequence $x_1, \cdots, x_n \in \mathbf{C}$ ($0 \leq n$) must not satisfy $a \rightarrow x_1 \rightarrow \cdots \rightarrow x_n \rightarrow a$. In the case of $n = 0$, this condition forbids $a \rightarrow a$.

Condition 2: For $\forall a \in \mathbf{C}$ and $\forall x, y \in \mathbf{C}$, if both $x \rightarrow a$ and $y \rightarrow a$ hold, then $x = y$ holds.

Definition 1 (Descendant Relation for $\mathbf{C}$)

For $a, b \in \mathbf{C}$ and the relation $\rightarrow$ over $\mathbf{C}$, $a$ is the descendant of $b$ and is denoted as $b \overset{\bullet}{\rightarrow} a$, iff one of the following conditions holds:

\[
\begin{align*}
(1) & \quad a = b, \text{ or} \\
(2) & \quad \text{there exists some constants} x_1, \cdots, x_n \in \mathbf{C} \text{ s.t.} \\
& \quad b = x_1 \rightarrow \cdots \rightarrow x_n = a \ (1 \leq n < \infty)
\end{align*}
\]

The above two conditions are mutually exclusive. By Condition 1, if the condition (2) holds, then $a \neq b$.

Proposition 1 For $\forall a, b \in \mathbf{C}$, $a \overset{\bullet}{\rightarrow} b$ and $b \overset{\bullet}{\rightarrow} a$ $\Rightarrow$ $a = b$.

Proof. Suppose $a \neq b$. By Definition 1, $a \overset{\bullet}{\rightarrow} b$ implies that there exists some constants $x_1, \cdots, x_m \in \mathbf{C}$ ($0 \leq m$) such that $a = x_1 \rightarrow \cdots \rightarrow x_m = b$. Similarly, $b \overset{\bullet}{\rightarrow} a$ implies that $b = y_1 \rightarrow \cdots \rightarrow y_n = a$ ($0 \leq n$). We then have a sequence $a = x_1 \rightarrow \cdots \rightarrow x_m = b = y_1 \rightarrow \cdots \rightarrow y_n = a$. This contradicts Condition 1. Therefore, $a = b$ must hold.

Definition 2 (Class)

A class is a sequence of zero or more constants. The length of a class is the number of constants constructing the class. $\mathbf{C}^\ast$ stands for a set of all classes.

Suppose $\mathbf{C} = \{\text{list}, \text{nil}, \text{num}\}$, then the following are examples of elements in $\mathbf{C}^\ast$:

$(\text{list}, 6), (\text{list}, \text{num}), (\text{nil})$.

Definition 3 (Descendant Relation for $\mathbf{C}^\ast$)

Similar to the case of $\mathbf{C}$, for two classes $c = (c_1, \cdots, c_m)$ and $d = (d_1, \cdots, d_n)$ ($0 \leq m, n$), $c$ is the descendant of $d$ and is denoted as $d \overset{\bullet}{\rightarrow} c$, iff the following conditions hold:

\[
\begin{align*}
& \quad n \leq m, \text{ and} \\
& \quad d_i \overset{\bullet}{\rightarrow} c_i \ (i = 1, \cdots, n).
\end{align*}
\]

Another set $\mathbf{V}$ is given. Each element of $\mathbf{V}$ is called a variable. The set $\mathbf{V}$ is a disjoint of $\mathbf{C}$, i.e., $\mathbf{C} \cap \mathbf{V} = \phi$.

Definition 4 (Class Object)

A class object is a pair $(\alpha, c)$, where $\alpha \in \mathbf{V}$ and $c \in \mathbf{C}^\ast$. $\mathcal{C}$ stands for a set of all class objects.

Definition 5 (Atomic Object)

Let $\mathcal{A}$ be a set such that:

\[
\begin{align*}
\mathcal{A} &= \{(X, x, (a_1, \cdots, a_n)) | \\
& \quad X \in \mathbf{V}, x \in \mathbf{C}, \\
& \quad a_1, \cdots, a_n \in \mathcal{A} \cup \mathcal{C}, 0 \leq n < \infty\}
\end{align*}
\]

Each element in $\mathcal{A}$ is called an atomic object (or simply an object). The class of the class object $x$ is also called the class of the atomic object. Each $a_i (i = 1, \cdots, n)$ is called the subobject of the object, and the sequence $(a_1, \cdots, a_n)$ is called the substructure of the object. Note that atomic objects always have a substructure. Even in the case of $n = 0$, it has a null sequence of subobjects, i.e., $(\cdot)$, as its substructure.

The variable of a class object is sometimes called a class variable, and the variable of an object is sometimes called an object variable, when variables need to be distinguished. Let $\mathbf{V}_c$ be a set of all class variables, and $\mathbf{V}_o$ be a set of all object variables. $\mathbf{V}_c, \mathbf{V}_o \subset \mathbf{V}$ holds, and it is assumed here that $\mathbf{V}_c \cap \mathbf{V}_o = \phi$.

For an object $a \in \mathcal{A} \cup \mathcal{C}$, the set of all variables appearing in $a$ is denoted by $V(a)$.

Fig. 1 shows examples of atomic objects, where $\mathbf{C} = \{\text{append}, \text{list}, \text{nil}, \text{num}, \text{5, inc, apply}\}$, $\mathbf{V}_c = \{\alpha, \beta, \gamma, \cdots\}$, and $\mathbf{V}_o = \{A, B, C, \cdots\}$.

- $(X, (\alpha, (\text{nil})), (\cdot))$
- $(X, (\beta, (\text{list}, \text{num})),$
  $(\langle A, (\gamma_1, (\text{5})), (\cdot)\rangle, \langle R, (\gamma_2, (\text{list})), (\cdot)\rangle))$
• \((A, (\alpha, (\text{append}))),
\langle X, (\beta, (\text{nil})), ()\rangle, \langle Y, (\gamma, (\text{list})), ()\rangle\rangle
• \((A, (\alpha, (\text{apply})),
\langle \pi, (\text{inc})\rangle, \langle X, (\beta, (\text{list})), ()\rangle, \langle Y, (\gamma, (\text{list})), ()\rangle\rangle\)

Figure 1: Examples of Atomic Objects

Definition 6 (Clause and Program)
A definite clause (or simply a clause) is an expression \(H \leftarrow B_1, \cdots, B_n\), with some \(H, B_1, \cdots, B_n \in \mathcal{A}(0 \leq n)\). \(H\) is called the head (or head object) of the clause. Each \(B_i (i = 1, \cdots, n)\) is called the body object of the clause, and the sequence \(B_1, \cdots, B_n\) is called the body of the clause.

A declarative program (or simply program) is a set of some definite clauses.

2.2 Operations

This section describes operators to operate objects. Operations are called specializations. Some specializations have a direction.

2.2.1 MGCD

Definition 7 (MGCD) For two classes \(s, t \in \mathcal{C}\), a class \(u \in \mathcal{C}\) that satisfies the following conditions is called the most general common descendant (MGCD) of \(s\) and \(t\), and such \(u\) is denoted as \(\Upsilon(s, t)\):

\[
\{ s \geq u, t \geq u, \text{ and for } \forall u' \in \mathcal{C}, s \geq u' \geq t \geq u' \Rightarrow u \geq u' \}.
\]

The MGCD of two classes does not always exist. If it does not exist, it is denoted as \(\Upsilon(s, t) = \perp\). For any class \(s\), \(\Upsilon(s, \perp)\) and \(\Upsilon(\perp, s)\) are not defined.

Proposition 2 If the MGCD of two classes is not \(\perp\), it is unique.

Proof. For \(s, t \in \mathcal{C}\), assume that \(u = \Upsilon(s, t)\) and \(v = \Upsilon(s, t)\) \((u, v \neq \perp)\). Both \(u \geq v\) and \(v \geq u\) hold by Definition 7. The length of \(u\) and the length of \(v\) are equal by Definition 3. Suppose that the length of \(u\) (and \(v\)) is \(n\), then we have \(u = (u_1, \cdots, u_n), v = (v_1, \cdots, v_n)\) and both \(u_i \rightarrow v_i\) and \(v_i \rightarrow u_i\) for each \(i = 1, \cdots, n\). By Proposition 1, \(u_i = v_i\) for each \(i = 1, \cdots, n\). Thus, \(u = v\).

For two classes \(s = (s_1, \cdots, s_m)\) and \(t = (t_1, \cdots, t_n)\) \((0 \leq n \leq m)\), let us consider the following algorithm.

Algorithm 1

1. For each \(i = 1, \cdots, n\),
   - if \(s_i \rightarrow t_i\), then assign \(u_i := t_i\).
   - if \(t_i \rightarrow s_i\), then assign \(u_i := s_i\).
   - otherwise assign \(u = \perp\), and exit this algorithm.

2. Assign \(u_i := s_i\) for each \(i = n + 1, \cdots, m\).
3. Assign \(u := (u_1, \cdots, u_m)\).

In the following propositions, we use the same notations as those used in Algorithm 1.

Proposition 3 If \(u \neq \perp\), then \(u = \Upsilon(s, t)\).

Proof. [Most general] Suppose that for some \(v = (v_1, \cdots, v_k) \in \mathcal{C}\), both \(s \geq v\) and \(t \geq v\) hold. \(s \geq v\) implies \(m \leq k\). By the definition of \(\geq\), \(s_i \rightarrow v_i\) and \(t_i \rightarrow v_i\) for each \(i = 1, \cdots, m\). If \(s_i \rightarrow t_i\) then \(u_i = t_i \geq v_i\). Alternatively, if \(t_i \geq s_i\) then \(u_i = s_i \geq v_i\). Therefore, in both cases, \(u \geq v\) holds.

[Common] The length of the class \(u\) is \(m\). The length of \(s\) is equal to \(m\), and the length of \(t\), i.e., \(n\), is less than or equal to \(m\). For each \(i = 1, \cdots, n\), if \(s_i \rightarrow t_i\) then \(u_i = t_i\), and this implies that \(s_i \rightarrow u_i\) and \(t_i \rightarrow u_i\). Alternatively if \(t_i \rightarrow s_i\) then \(u_i = s_i\), and this implies that \(t_i \rightarrow u_i\) and \(s_i \rightarrow u_i\). Thus, in any case, \(t \geq u\). For each \(i = n + 1, \cdots, m\), we have \(s_i \rightarrow u_i\), since \(u_i = s_i\). Thus, \(s \geq u\).

Proposition 4 If the MGCD of \(s\) and \(t\) is not \(\perp\), then Algorithm 1 always derives it.

Proof. Let \(v = (v_1, \cdots, v_k)\) be \(\Upsilon(s, t) \neq \perp\) \((m, n \leq k)\).

1. Assume Algorithm 1 succeeds and derives \(u \neq \perp\). We have \(u = v = \Upsilon(s, t)\) by Proposition 2 and 3.

2. Assume Algorithm 1 derives \(\perp\). This implies that neither \(s_i \rightarrow t_i\) nor \(t_i \rightarrow s_i\) for some values of \(i \in \Upsilon(s, t)\) implies that \(s_i \rightarrow v_i\) and \(t_i \rightarrow v_i\) for all \(i = 1, \cdots, n\). There are four possible cases according to Definition 1:
1. \( s_i = v_i \) and \( t_i = v_i \); this implies that \( s_i \xrightarrow{*} t_i \) and \( t_i \xrightarrow{*} s_i \).

2. \( s_i = s_i^1 \rightarrow \cdots \rightarrow s_i^a = v_i \) (1 \( \leq a \)) and \( t_i = v_i \); this implies \( s_i \xrightarrow{*} t_i \).

3. \( s_i = v_i \) and \( t_i = t_i^1 \rightarrow \cdots \rightarrow t_i^b = v_i \) (1 \( \leq b \)); this implies \( t_i \xrightarrow{*} s_i \).

4. \( s_i = s_i^1 \rightarrow \cdots \rightarrow s_i^a = v_i \) and \( t_i = t_i^1 \rightarrow \cdots \rightarrow t_i^b = v_i \) (1 \( \leq a, b \)); we have \( s_i^1 = t_i^1 \), and by Condition 2, \( s_i^m = t_i^m \), \( \cdots , s_i^1 = t_i^b-a \) if \( a \leq b \). This implies \( t_i \xrightarrow{*} s_i \). Otherwise, if \( b \leq a \), then we have \( s_i^{a-b} = t_i \). This implies \( s_i \xrightarrow{*} t_i \).

In all four cases, either \( s_i \xrightarrow{*} t_i \) or \( t_i \xrightarrow{*} s_i \) holds. This contradicts the assumption. Therefore, Algorithm 1 always succeeds, if \( \Upsilon (s, t) \neq \bot \).

Propositions 3 and 4 ensure that the MGCD of two classes is not \( \bot \), if Algorithm 1 derives it.

**Proposition 5** For \( s, t \in C \), suppose \( u = \Upsilon (s, t) \neq \bot \), then \( \Upsilon (s, u) = \Upsilon (t, u) = u \).

**Proof.** By Proposition 4, \( u \) is derived by the algorithm. For each \( i = 1, \cdots, n \), either \( s_i \xrightarrow{*} t_i \) or \( t_i \xrightarrow{*} s_i \) holds according to the assumption of this proposition.

The length of \( \Upsilon (s, u) \) is \( m \), since the length of \( s \) is \( n \), the length of \( u \) is \( m \) and \( n \leq m \). For \( i = 1, \cdots, n \), if \( s_i \xrightarrow{*} t_i \), then \( u_i = t_i \) and we have \( s_i \xrightarrow{*} u_i \). Otherwise, if \( t_i \xrightarrow{*} s_i \), then \( u_i = s_i \) and we have \( s_i \xrightarrow{*} u_i \). Thus, \( s_i \xrightarrow{*} u_i \) always holds for \( i = 1, \cdots, n \). We have \( u = \Upsilon (s, u) \) by Algorithm 1.

The length of \( \Upsilon (t, u) \) is \( m \), since the length of \( t \) is \( m \) and the length of \( u \) is \( m \). For \( i = 1, \cdots, m \), if \( s_i \xrightarrow{*} t_i \), then \( u_i = t_i \) and we have \( t_i \xrightarrow{*} u_i \). Otherwise, if \( t_i \xrightarrow{*} s_i \), then \( u_i = s_i \) and we have \( t_i \xrightarrow{*} u_i \). Thus, \( t_i \xrightarrow{*} u_i \) always holds for \( i = 1, \cdots, m \). We have \( u = \Upsilon (t, u) \) by Algorithm 1.

2.2.2 Specialization

To operate atomic objects or class objects, there are four basic operators. Suppose a target object is \( z \in A \cup C \) and a basic operator is \( \Theta \), the application of \( \Theta \) to \( z \) is denoted as \( z[\Theta] \).

**Definition 8 (Basic Operator)** (1) For \( X, Y \in V_\alpha \), the operator \( X \xrightarrow{\alpha} Y \) exchanges all \( X \) appearing in the object with \( Y \).

(2) For \( \alpha, \beta \in V_c \), the operator \( \alpha \cdot_c \beta \) exchanges all \( \alpha \) appearing in the object with \( \beta \).

(3) For \( \alpha \in V_c \) and \( t \in C^* \), the operator \( \alpha \cdot t \) exchanges the class of all class objects specified by the variable \( \alpha \) appearing in the object with \( t \).

This operator has a required condition: there must be a descendant relation between the source class and the destination class. For example, when \( \alpha \cdot t \) is performed, \( \Upsilon (s, t) \neq \bot \) must hold.

(4) For \( X \in V_\alpha \) and \( b \in A \cup C \), the operator \( X \cdot b \) adds \( b \) in the tail part of substructure of the target atomic object whose variable is \( X \).

When the operator is applied to atomic objects, the required condition is effective for subobjects recursively. If a basic operator holds the condition, then it is said to be applicable to the object. If the class variable of an operator and the class variable of a target class object are different, then the operator is also said to be applicable to the object, but it does not have any effect.

As shown in Definition 8, for a basic operator \( \Theta \), if \( z \in A \) then \( z[\Theta] \in A \), otherwise, if \( z \in C \) then
z[θ] ∈ C.

**Definition 9 (Specialization Operator)** The specialization operator (or simply specialization) is a sequence of zero or more basic operators. S stands for a set of all specialization operators. A null sequence [] is called the unit specialization operator (or simply unit specialization).

[Concatenation] A concatenation operation of two specializations θ = [θ1, ··· , θm] and σ = [σ1, ··· , σn] (0 ≤ m, n) is denoted as θ·σ and the result is defined as

\[
θ·σ = [θ1, ··· , θm, σ1, ··· , σn].
\]

[Applicability] When a specialization θ is applied to an object a, all basic operators in θ must be applicable to a. In this case, θ is said to be applicable to a.

[Application] The application of an applicable specialization θ to an object a is denoted as aθ and the result is defined as

\[
\begin{aligned}
& a[\_] = a, \\
& a[θ1, θ2, ··· , θm] = (a[θ1])[θ2, ··· , θm].
\end{aligned}
\]

Specialization operators are extended into clauses. Suppose that C = (H → B1, ··· , Bn) is a clause and θ is a specialization operator that is applicable to all objects H, B1, ··· , Bn, then the application of θ to C is denoted as Cθ and is defined as

\[
Cθ = (H[θ] → B1[θ], ··· , Bn[θ]).
\]

## 3 Unification Algorithm

This section gives a definition of the unification algorithm for atomic objects. Section 3.1 shows the algorithm, and Section 3.2 proves that it is a unification algorithm.

### 3.1 Unification

**Definition 10 (Unification Operator)**

For a, b ∈ A ∪ C, a unification operator is a pair of two specialization operators (θ, σ) ∈ S × S, which satisfies aθ = bσ.

Obviously, we can not unify any atomic object and any class object.

**Algorithm 2** For two class objects, (α, s) and (β, t), the algorithm for unifying them is defined as follows. If the unification of them is successful, a unification operator is obtained.

1. Compute u = Y(s, t). If u = ⊥, then the whole unification fails. If u ≠ ⊥, then assign τ := [α → u, β → u].
2. Assign τ := τ · [α ↠ ω, β ↠ ω], where ω is a new class variable.
3. We have (τ, τ) as a unification operator.

We omit the proof that Algorithm 2 produces the unification operator for class objects.

An atomic object that includes its object variable in its subobject, such as

\[
(X, x, ((X, x', (\cdots)), \cdots)),
\]

can cause a problem. Variables work as labels of objects in applying specializations. Objects that have the same variable should be the same. As to the above example, the outside object — its object variable is X — includes a subobject whose object variable is also X. The subobject should include a subobject whose object variable is X. The nest of objects infinitely succeeds. In applying a specialization operator, for example [X ↠ Y], to such an object, the operation recursively continues and will not end. In this paper, we do not deal with this kind of object.

The restriction is defined in Definition 11.

**Definition 11 (Regular Object)** An atomic object (X1, (α, s), (a1, ··· , an)) (0 ≤ n) is called a regular object, if the following conditions hold:

1. The object variable X does not appear in each subobject a1, ··· , an, and
2. ai is recursively a regular object for i = 1, ··· , n.

Here, all atomic objects are assumed to be regular objects. If there is no ambiguity, we call them simply objects.

Here, we show a unification algorithm for regu-
lar atomic objects \(a\) and \(b\):
\[
\begin{align*}
\{ a &= \{ x, (\alpha, s), (a_1, \ldots, a_m) \}, \\
\{ b &= \{ y, (\beta, t), (b_1, \ldots, b_n) \}, \text{ where } 0 \leq n \leq m. \}
\end{align*}
\]
As to names of variables, with appropriate specializations \(\rho_a\) and \(\rho_b\), let \(V(\alpha) \cap V(\beta, t)\) be \(\phi\). Before performing Algorithm 3, the re-assignings of \(\alpha \cup \beta \cup a \cup b \) to \(a\) and \(b \cup \beta \cup \alpha \) to \(b\) must be performed.

As to the number of subobjects, the algorithm allows only two cases: \((1) n = m\) or \((2) one or both of \(n\) and \(m\) are zero. In any other case, the algorithm immediately terminates in failure.

The algorithm is defined recursively. If the unification is successful, a pair of operators \((\tau, \tau)\) is obtained. In Section 3.2, we prove that it is a unification operator.

**Algorithm 3**

1. When \(m = 0\), this is the case of \(n = 0\); go to Step 6. When \(m \neq 0\) and \(n = 0\), go to Step 3. In both cases, let \(\tau' = \{ \} \).

2. This is the case of \(n = m \neq 0\). Renamings for subobjects are not needed.

   Suppose that \(a_1\) and \(b_1\) are unified successfully and a unification operator \((\tau_1, \tau_1)\) is obtained. The specializations can have effects on the class, and all subobjects of \(a\) and \(b\). \(\Theta_1\) must be applicable to \((\alpha, s)\), and \(\sigma_1\) must be applicable to \((\beta, t)\). In the next step, \(a_2 \tau_1\) and \(b_2 \tau_1\) are unified.

   For each \(i = 1, \ldots, n = m\), if subobjects \(a_i \tau_1 \cdot \tau_{i-1} \cdot b_i \tau_1 \cdot \tau_{i-1}\) are unified successfully, then we have \((\tau_i, \tau_i)\) as a unification operator. Let \(\tau_i = a_i \tau_1 \cdot \tau_{i-1} \cdot b_i \tau_1 \cdot \tau_{i-1} \cdot \tau_i (i = 1, \ldots, n = m).\) Let \(\tau' = \tau_1 \cdots \tau_m\).

   For some value of \(i\), if a unification fails, then the whole unification ends in failure.

3. If \(n = m\), then let \(\tau' := \{ \} \). Otherwise, if \(n \neq m\), this is the case of \(n = 0\) and \(m = 0\), and Step 2 is not performed, then let \(\tau' = [ Y \Rightarrow a_1, \ldots, Y \Rightarrow a_m ]\).

4. Unify \(\langle \alpha, s \rangle \tau' \cdot \tau'\) and \(\langle \beta, t \rangle \tau' \cdot \tau'\). If the unification fails, then the whole unification ends in failure. Otherwise, the unification operator \(\langle \tau', \tau' \rangle\) is obtained.

**Suppose that** \((\alpha, s) \tau' \cdot \tau' = (\alpha', s')\), and \((\beta, t) \tau' \cdot \tau' = (\beta', t')\), then \(\tau' = \{ \alpha' \Rightarrow u, \beta' \Rightarrow u, \alpha' \Rightarrow u, \beta' \Rightarrow u, u = Y(s', t'), \text{ and } \omega \in V_c\}\) is a new variable. Proposition 5 ensures that \((\alpha', s') \tau' \cdot u = (\alpha', Y(s', u)) = (\alpha', u)\) and \((\beta', t') \tau' \cdot u = (\beta', u).\)

If the basic operator \(\tau\) is not applicable to some of the subobjects, then the whole unification ends in failure.

5. Equalize each object variable. Since \(a\) and \(b\) are regular, \(X\) and \(Y\) do not appear in their substructures, and they are not influenced by \(\tau' \cdot \tau' \cdot \tau'.\) Assign \(\tau' := [ X = Z, Y = Z ]\), where \(Z\) is a new object variable. \(\tau'\) does not have any effect on the subobjects.

6. The whole unification for \(a\) and \(b\) is successful, and we have an unification operator \((\tau, \tau)\), where \(\tau = \tau' \cdot \tau' \cdot \tau'.\)

\[\square\]

As to original objects \(a\) and \(b\) that have not been renamed and not re-assigned, \((\tau_a \rho_a, \tau_b \rho_b)\) is the unification operator.

### 3.2 Theorems and Proofs

In the following theorems, we use the same notations as those used in Algorithm 3.

**Theorem 1** For two regular objects \(a\) and \(b\), Algorithm 3 terminates in a finite number of steps.

**Proof.** In the cases of \(n = 0\) and \(m \neq 0\), or \(n = m = 0\), Step 2 is not performed and Algorithm 3 terminates immediately by Step 1. In the case of \(n \neq 0\), Step 2 is recursively performed. No other steps are performed repeatedly. We focus on only the case of \(n = m \neq 0\), and on Step 2.

In general, the **height** of an atomic object \(g = (G, r, (g_1, \ldots, g_k))\) is denoted as \(|g|\), and is defined as
\[
\left\{
\begin{align*}
|g| &= 0, \text{ if } k = 0, \\
|g| &= \max(|g_i|) + 1, \text{ if } 1 \leq k.
\end{align*}
\right.
\]
By the definition, \(1 \leq |g| - |g_i|\) for any \(i\), and \(0 \leq |g|\) is satisfied. Since \(g\) is regular, \(|g| < \infty\) is satisfied.
\( n = m \) implies \( \tau' = [ ] \), and \( n \) and \( m \) therefore remain unchanged. \( 1 \leq |g| - |g'| \) implies that an execution of Step 2 decreases the height of the object by one or more. The minimum value of height is zero. In finite number of execution, the height of the object drops to zero. For an object whose height is zero, such as \( (X, (\alpha, s), (\)) \), Algorithm 3 terminates immediately in Step 1.

**Theorem 2** For any two regular objects \( a \) and \( b \), assume that \( V(a) \cap V(b) = \emptyset \). If a pair of specialization operators \((\theta, \sigma)\) is obtained by Algorithm 3, then \( a\theta = b\sigma \) holds.

**Proof.**

(1) In the case of \( n = m = 0 \): \( \tau' = [ ] \) and \( \tau'' = [ ] \).

\[
\begin{align*}
\alpha\tau' &= \langle X, (\alpha, s), (\))\rangle\tau' \cdot \tau'' \\
&= \langle Z, (\omega, u), (\)\rangle
\end{align*}
\]

\[
\begin{align*}
\beta\tau' &= \langle Y, (\beta, t), (\))\rangle\tau' \cdot \tau'' \\
&= \langle Z, (\omega, u), (\)\rangle
\end{align*}
\]

Thus, we have \( \alpha\tau = \beta\tau \).

(2) In the case of \( n = 0 \) and \( m \neq 0 \): \( \tau' = [ ] \) and \( \tau'' = [Y \mapsto a_1, \cdots, Y \mapsto a_m] \).

\[
\begin{align*}
\alpha\tau' &= \langle X, (\alpha, s), (a_1, \cdots, a_m)\rangle\tau' \cdot \tau'' \\
&= \langle X, (\alpha, s), (a_1, \cdots, a_m)\rangle\tau' \cdot \tau'' \\
&= \langle Z, (\omega, u), (a_1 \tau' \cdot \tau'', \cdots, a_m \tau' \cdot \tau'')\rangle \\
&= \langle Z, (\omega, u), (a_1 \tau', \cdots, a_m \tau')\rangle
\end{align*}
\]

Since the object \( a \) is regular, the variable \( X \) does not appear in the substructure of \( a \). Since \( V(a) \cap V(b) = \emptyset \), the variable \( Y \) does not appear in the substructure of \( a \). \( \tau' \) does not have any effect on \( a \), and \( \tau'' = [X \mapsto Z, Y \mapsto Z] \) does not have any effect on subobjects of \( a \).

\[
\begin{align*}
\beta\tau' &= \langle Y, (\beta, t), (\))\rangle\tau' \cdot \tau'' \\
&= \langle Z, (\beta, t), (a_1, \cdots, a_m)\rangle\tau' \cdot \tau'' \\
&= \langle Z, (\omega, u), (a_1 \tau', \cdots, a_m \tau')\rangle
\end{align*}
\]

Thus, we have \( a\theta = b\sigma \).

(3) In the case of \( n = m \neq 0 \): In this case, we use \( n \) for the number of subobjects. \( \tau' = \tau_1 \cdots \tau_n \), and \( \tau'' = [ ] \).

\[
\begin{align*}
\alpha\tau' &= \langle X, (\alpha, s), (a_1, \cdots, a_n)\rangle\tau_1 \cdot \tau_2 \cdots \tau_n \cdot \tau'' \\
&= \langle X, (\alpha, s)\tau_1, (z_1, a_2 \tau_1, \cdots, a_n \tau_1)\rangle \\
&= \langle X, (\alpha, s)\tau_1 \cdot \tau_2, (z_1 \tau_2, z_2, \cdots, a_n \tau_1)\rangle \\
&= \langle X, (\alpha, s)\tau_1 \cdot \tau_2 \cdots \tau_n \cdot \tau'' \\
&= \cdots \\
&= \langle X, (\alpha, s)\tau', (z_1 \tau_1, z_2 \tau_2 \cdots \tau_n, \cdots, z_n)\rangle\tau' \cdot \tau'' \\
&= \langle Z, (\omega, u), (z_1 \tau_1, z_2 \tau_2 \cdots \tau_n, \cdots, z_n)\rangle
\end{align*}
\]

Thus, we have \( \alpha\tau = \beta\tau \). In all cases (1) \sim (3), \( \alpha\theta = \beta\tau \) holds.

**4 Conclusions**

We have proven that Algorithm 3 is a unification algorithm and that it terminates in a finite number of steps. As shown in this paper, unification is not the most basic operation. In particular, for objects that have a more complex structure than simple terms do, unification is a very large operation. We successfully proved, using small and simple operators, i.e., specializations, that our algorithm is based on.

**Acknowledgments**

Authors thank to Youichiro Kojima in TNCT. He often helped one of authors with many jobs in Somei-Ryou.

**References**


