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-

* Department of Computer Engineering, Tomakomai National College of Technology, Aza-Nish-ikioka 443, Tomakomai, Hokkaido, 059-1275, Japan
yuuichi@jo.tomakomai-ct.ac.jp

** Center for Information and Multimedia Studies, Hokkaido University, Kita 10, Nishi 5, Kita-ku, Sapporo, Hokkaido, 060-0810, Japan

*** Division of System and Information Engineering, Hokkaido University, Kita 13, Nishi 8, Kita-ku, Sapporo, Hokkaido, 060-8628, Japan

guage are described in [3].

2.1 Program

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

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

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

Definition 1 (Descendant Relation for $\mathbb{C}$)
For $a, b \in \mathbb{C}$ and the relation $\rightarrow$ over $\mathbb{C}$, a is the descendant *of b and is denoted as $b \overset{*}{\rightarrow} a$, iff one of following conditions holds:*

$$\begin{cases} (1) & a = b, \text{ or} \\ (2) & \text{there exists some constants} \\ & x_1, \cdots, x_n \in \mathbb{C} \text{ s.t.} \\ & b = x_1 \rightarrow \cdots \rightarrow x_n = a \ (1 \leq n < \infty) \end{cases}$$

$\square$

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

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

Proof. Suppose $a \neq b$. By Definition 1, $a \overset{*}{\rightarrow} b$ implies that there exists some constants $x_1, \cdots, x_m \in \mathbb{C}$ $(0 \leq m)$ such that $a = x_1 \rightarrow \cdots \rightarrow x_m = b$. Similarly, $b \overset{*}{\rightarrow} a$ 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. $\square$

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. $\mathbb{C}^*$ stands for a set of all classes.* $\square$

Suppose $\mathbb{C} = \{list, nil, num\}$, then the followings are examples of elements in $\mathbb{C}^*$:

$$(),\ (list),\ (list, num),\ (nil), \cdots$$

Definition 3 (Descendant Relation for $\mathbb{C}^*$)
Similar to the case of $\mathbb{C}$, 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 $d \succeq c$, iff the following conditions holds:*

$$\begin{cases} n \leq m, \text{ and} \\ d_i \overset{*}{\rightarrow} c_i \ (i = 1, \cdots, n). \end{cases}$$

$\square$

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

Definition 4 (Class Object) *A* class object *is a pair (α, c), where $\alpha \in \mathbb{V}$ and $c \in \mathbb{C}^*$. C stands for a set of all class objects.* $\square$

Definition 5 (Atomic Object) *Let $\mathcal{A}$ be a set such that:*

$$\begin{aligned} \mathcal{A} = \{ & \langle X, x, (a_1, \cdots, a_n) \rangle \mid \\ & X \in \mathbb{V}, x \in C, \\ & a_1, \cdots, a_n \in \mathcal{A} \cup C, 0 \leq n < \infty \} \end{aligned}$$

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., $()$, as its substructure.* $\square$

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 $\mathbb{V}_c$ be a set of all class variables, and $\mathbb{V}_o$ be a set of all object variables. $\mathbb{V}_c, \mathbb{V}_o \subset \mathbb{V}$ holds, and it is assumed here that $\mathbb{V}_c \cap \mathbb{V}_o = \phi$.

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

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

- $\langle X, (\alpha, (nil)), () \rangle$
- $\langle Y, (\beta, (list, num)),$
  $(\langle A, (\gamma_1, (5)), () \rangle, \langle R, (\gamma_2, (list)), () \rangle) \rangle$

- $\langle A, (\alpha, (append)),$
  $(\langle X, (\beta, (nil)), ()\rangle,$
  $\langle Y, (\gamma, (list)), ()\rangle, \langle Y, (\gamma, (list)), ()\rangle)\rangle$
- $\langle A, (\alpha, (apply)),$
  $((\pi, (inc)),$
  $\langle X, (\beta, (list)), ()\rangle, \langle Y, (\gamma, (list)), ()\rangle)\rangle$

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 $H, B_1, \cdots, B_n \in \mathcal{A}(0 \le n)$. 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 $s, t \in \mathbf{C}^*$, a class $u \in \mathbf{C}^*$ that satisfies the following conditions is called the* most general common descendant *(MGCD) of s and t, and such u is denoted as $\Upsilon(s, t)$:*

$$\begin{cases} s \succeq u, t \succeq u, \text{ and} \\ \text{for } \forall u' \in \mathbf{C}^*, s \succeq u', t \succeq u' \Rightarrow u \succeq u'. \end{cases}$$

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 $s, t \in \mathbf{C}^*$, assume that $u = \Upsilon(s, t)$ and $v = \Upsilon(s, t)$ $(u, v \neq \perp)$. Both $u \succeq v$ and $v \succeq u$ 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 $u_i \overset{*}{\rightarrow} v_i$ and $v_i \overset{*}{\rightarrow} u_i$ 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 $t = (t_1, \cdots, t_n)$ $(0 \le n \le m)$, let us consider the following algorithm.

Algorithm 1

```
1. For each $i = 1, \cdots, n$,
    • if $s_i \overset{*}{\rightarrow} t_i$, then assign $u_i := t_i$.
    • if $t_i \overset{*}{\rightarrow} s_i$, 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 $v = (v_1, \cdots, v_k) \in \mathbf{C}^*$, both $s \succeq v$ and $t \succeq v$ hold. $s \succeq v$ implies $m \le k$. By the definition of $\succeq$, $s_i \overset{*}{\rightarrow} v_i$ and $t_i \overset{*}{\rightarrow} v_i$ for each $i = 1, \cdots, m$. If $s_i \overset{*}{\rightarrow} t_i$ then $u_i = t_i \succeq v_i$. Alternatively, if $t_i \succeq s_i$ then $u_i = s_i \succeq v_i$. Therefore, in both cases, $u \succeq v$ 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 $s_i \overset{*}{\rightarrow} t_i$ then $u_i = t_i$, and this implies that $s_i \overset{*}{\rightarrow} u_i$ and $t_i \overset{*}{\rightarrow} u_i$. Alternatively if $t_i \overset{*}{\rightarrow} s_i$, then $u_i = s_i$, and this implies that $t_i \overset{*}{\rightarrow} u_i$ and $s_i \overset{*}{\rightarrow} u_i$. Thus, in any case, $t \succeq u$. For each $i = n + 1, \cdots, m$, we have $s_i \overset{*}{\rightarrow} u_i$, since $u_i = s_i$. Thus, $s \succeq u$. □

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

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

(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 $s_i \overset{*}{\rightarrow} t_i$ nor $t_i \overset{*}{\rightarrow} s_i$ for some values of i. $v = \Upsilon(s, t)$ implies that $s_i \overset{*}{\rightarrow} v_i$ and $t_i \overset{*}{\rightarrow} v_i$ 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. $s_i = s_i^1 \to \cdots \to s_i^a = v_i$ $(1 \leq a)$ and $t_i = v_i$; this implies $s_i \xrightarrow{*} t_i$.
3. $s_i = v_i$ and $t_i = t_i^1 \to \cdots \to t_i^b = v_i$ $(1 \leq b)$; this implies $t_i \xrightarrow{*} s_i$.
4. $s_i = s_i^1 \to \cdots \to s_i^a = v_i$ and $t_i = t_i^1 \to \cdots \to t_i^b = v_i$ $(1 \leq a, b)$; we have $s_i^a = t_i^b$, and by Condition 2, $s_i^{a-1} = t_i^{b-1}$, $\cdots$, $s_i = t_i^{b-a}$ 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 $\Upsilon(s, t) \neq \perp$. □

Propositions 3 and 4 ensure that the MGCD of two classes is not $\perp$, iff Algorithm 1 derives it.

Proposition 5 *For $s, t \in \mathbb{C}^*$, suppose $u = \Upsilon(s, t) \neq \perp$, 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 $z \in \mathcal{A} \cup C$ and a basic operator is θ, the application of θ to z is denoted as $z[\theta]$.

Definition 8 (Basic Operator) *(1) For $X, Y \in \mathbb{V}_o$, the operator $X \xrightarrow{o} Y$ exchanges all X appearing in the object with Y.*

$$\begin{cases} a[X \xrightarrow{o} Y] = a, \text{ if } a \in C, \\ \langle Z, x, (a, \cdots)\rangle[X \xrightarrow{o} Y] \\ \qquad = \langle Z, x, (a[X \xrightarrow{o} Y], \cdots)\rangle, \\ \langle X, x, (a, \cdots)\rangle[X \xrightarrow{o} Y] \\ \qquad = \langle Y, x, (a[X \xrightarrow{o} Y], \cdots)\rangle. \end{cases}$$

(2) For $\alpha, \beta \in \mathbb{V}_c$, the operator $\alpha \xrightarrow{c} \beta$ exchanges all α appearing in the object with β.

$$\begin{cases} (\gamma, s)[\alpha \xrightarrow{c} \beta] = (\gamma, s), \\ (\alpha, s)[\alpha \xrightarrow{c} \beta] = (\beta, s), \\ \langle X, x, (a, \cdots)\rangle[\alpha \xrightarrow{c} \beta] \\ \qquad = \langle X, x[\alpha \xrightarrow{c} \beta], (a[\alpha \xrightarrow{c} \beta], \cdots)\rangle. \end{cases}$$

(3) For $\alpha \in \mathbb{V}_c$ and $t \in \mathbb{C}^*$, the operator $\alpha \rhd t$ exchanges the class of all class objects specified by the variable α 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 $(\alpha, s)[\alpha \rhd t]$ is performed, $\Upsilon(s, t) \neq \perp$ must hold.

$$\begin{cases} (\gamma, s)[\alpha \rhd t] = (\gamma, s), \\ (\alpha, s)[\alpha \rhd t] = (\alpha, \Upsilon(s, t)), \\ \langle X, x, (a, \cdots)\rangle[\alpha \rhd t] \\ \qquad = \langle X, x[\alpha \rhd t], (a[\alpha \rhd t], \cdots)\rangle. \end{cases}$$

(4) For $X \in \mathbb{V}_o$ and $b \in \mathcal{A} \cup C$, the operator $X \twoheadrightarrow b$ adds b in the tail part of substructure of the target atomic object whose variable is X.

$$\begin{cases} a[X \twoheadrightarrow b] = a, \text{ if } a \in C, \\ \langle Y, x, (a_1, \cdots, a_n)\rangle[X \twoheadrightarrow b] \\ \qquad = \langle Y, x, (a_1[X \twoheadrightarrow b], \cdots, a_n[X \twoheadrightarrow b])\rangle, \\ \langle X, x, (a_1, \cdots, a_n)\rangle[X \twoheadrightarrow b] = \\ \qquad = \langle X, x, (a_1, \cdots, a_n, b)\rangle \end{cases}$$

□

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 θ, if $z \in \mathcal{A}$ then $z[\theta] \in \mathcal{A}$, otherwise, if $z \in C$ then

$z[\theta] \in C$.

Definition 9 (Specialization Operator) *The* specialization operator *(or simply specialization) is a sequence of zero or more basic operators. $\mathcal{S}$ 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 $\theta = [\![\theta_1, \cdots, \theta_m]\!]$ and $\sigma = [\![\sigma_1, \cdots, \sigma_n]\!]$ $(0 \le m, n)$ is denoted as $\theta \cdot \sigma$ and the result is defined as

$$\theta \cdot \sigma = [\![\theta_1, \cdots, \theta_m, \sigma_1, \cdots, \sigma_n]\!].$$

[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 $a\theta$ and the result is defined as

$$\begin{cases} a[\![\,]\!] = a, \\ a[\![\theta_1, \theta_2, \cdots, \theta_m]\!] = (a[\theta_1])[\![\theta_2, \cdots, \theta_m]\!] . \end{cases}$$

□

Specialization operators are extended into clauses. Suppose that $C = (H \leftarrow B_1, \cdots, B_n)$ is a clause and θ is a specialization operator that is applicable to all objects $H, B_1, \cdots, B_n$, then the application of θ to C is denoted as $C\theta$ and is defined as

$$C\theta = (H_1\theta \leftarrow B_1\theta, \cdots, B_n\theta) .$$

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 $a, b \in \mathcal{A} \cup C$, a unification operator is a pair of two specialization operators $(\theta, \sigma) \in \mathcal{S} \times \mathcal{S}$, which satisfies $a\theta = b\sigma$. □

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 $u = \Upsilon(s, t)$. If $u = \bot$, then the whole unification fails. If $u \ne \bot$, then assign $\tau := [\![\alpha \triangleright u, \beta \triangleright u]\!]$.*
2. *Assign $\tau := \tau \cdot [\![\alpha \overset{c}{\rightarrow} \omega, \beta \overset{c}{\rightarrow} \omega]\!]$, 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

$$\langle X, x, (\langle X, x', (\cdots)\rangle, \cdots)\rangle,$$

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 $[\![X \overset{o}{\rightarrow} Y]\!]$, 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 $\langle X, (\alpha, s), (a_1, \cdots, a_n)\rangle$ $(0 \le n)$ is called a* regular object, *iff the following conditions hold:*

1. *The object variable X does not appear in each subobject $a_1, \cdots, a_n$, and*
2. *a_i is recursively a regular object for $i = 1, \cdots, n$.*

□

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

Here, we show a unification algorithm for regular atomic objects a and b:

$$\begin{cases} a = \langle X, (\alpha, s), (a_1, \cdots, a_m)\rangle, \\ b = \langle Y, (\beta, t), (b_1, \cdots, b_n)\rangle, \text{where } 0 \le n \le m. \end{cases}$$

As to names of variables, with appropriate specializations ρ_a and ρ_b, let $V(a\rho_a) \cap V(b\rho_b)$ be ϕ. Before performing Algorithm 3, the re-assignings of $a\rho_a$ to a and $b\rho_b$ to b must be performed.

As to the number of subobjects, the algorithm allows only two cases:[1] (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 (τ, τ) 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 τ' be $[\![\ ]\!]$.*

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 (τ_1, τ_1) is obtained. The specializations can have effects on the class, and all subobjects of a and b. θ_1 must be applicable to (α, s), and σ_1 must be applicable to (β, t). In the next step, $a_2\tau_1$ and $b_2\tau_1$ are unified.

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

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

3. *If $n = m$, then let $\tau^s := [\![\ ]\!]$. Otherwise, if $n \neq m$, this is the case of $n = 0$ and $m \neq 0$, and Step 2 is not performed, then let τ^s be $[\![Y \mathrel{\not\twoheadrightarrow} a_1, \cdots, Y \mathrel{\not\twoheadrightarrow} a_m]\!]$.*

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

   Suppose that $(\alpha, s)\tau' \cdot \tau^s = (\alpha', s')$, and $(\beta, t)\tau' \cdot \tau^s = (\beta', t')$, then $\tau^c = [\![\alpha' \triangleright u, \beta' \triangleright u, \alpha' \overset{c}{\rightharpoondown} \omega, \beta' \overset{c}{\rightharpoondown} \omega]\!]$, $u = \Upsilon(s', t')$, and $\omega \in \mathbb{V}_c$ is a new variable. Proposition 5 ensures that $(\alpha', s')[\alpha' \triangleright u] = (\alpha', \Upsilon(s', u)) = (\alpha', u)$ and $(\beta', t')[\beta' \triangleright u] = (\beta', u)$.

   If the basic operator τ^c 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 $\tau' \cdot \tau^s \cdot \tau^c$. Assign $\tau^v := [\![X \overset{o}{\rightharpoondown} Z, Y \overset{o}{\rightharpoondown} Z]\!]$, where Z is a new object variable. τ^v does not have any effect on the subobjects.*

6. *The whole unification for a and b is successful, and we have an unification operator (τ, τ), where $\tau = \tau' \cdot \tau^s \cdot \tau^c \cdot \tau^v$.*

□

As to original objects a and b that have not been renamed and not re-assigned, $(\tau_a \cdot \rho_a, \tau_b \cdot \rho_b)$ 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 $n = m \neq 0$, 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 $g = \langle G, r, (g_1, \cdots, g_k)\rangle$ is denoted as $|g|$, and is defined as

$$\begin{cases} |g| = 0, \text{ if } k = 0, \\ |g| = \max_{g_i \in \mathcal{A}}(|g_i|) + 1, \text{ if } 1 \le k. \end{cases}$$

By the definition, $1 \le |g| - |g_i|$ for any i, and $0 \le |g|$ is satisfied. Since g is regular, $|g| < \infty$ is satisfied.

[1]This restriction is related to the method by which programmers determine the shape of objects. In some languages, the method is implemented as type declarations. This method is not discussed in this paper (*c.f.*, [3] and [4]).

$n = m$ implies $\tau^s = [\![\,]\!]$, and n and m therefore remain unchanged. $1 \leq |g| - |g_i|$ 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 $\langle X, (\alpha, s), ()\rangle$, Algorithm 3 terminates immediately in Step 1. □

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

Proof.
(1) In the case of $n = m = 0$: $\tau^t = [\![\,]\!]$ and $\tau^s = [\![\,]\!]$.

$$\begin{aligned} a\tau &= \langle X, (\alpha, s), ()\rangle\tau^c{\cdot}\tau^v \\ &= \langle Z, (\omega, u), ()\rangle \end{aligned}$$

$$\begin{aligned} b\tau &= \langle Y, (\beta, t), ()\rangle\tau^c{\cdot}\tau^v \\ &= \langle Z, (\omega, u), ()\rangle \end{aligned}$$

Thus, we have $a\tau = b\tau$.

(2) In the case of $n = 0$ and $m \neq 0$: $\tau^t = [\![\,]\!]$ and $\tau^s = [\![Y \twoheadrightarrow a_1, \cdots, Y \twoheadrightarrow a_m]\!]$.

$$\begin{aligned} a\tau &= \langle X, (\alpha, s), (a_1, \cdots, a_m)\rangle\tau^s{\cdot}\tau^c{\cdot}\tau^v \\ &= \langle X, (\alpha, s), (a_1, \cdots, a_m)\rangle\tau^c{\cdot}\tau^v \\ &= \langle Z, (\omega, u), (a_1\tau^c{\cdot}\tau^v, \cdots, a_m\tau^c{\cdot}\tau^v)\rangle \\ &= \langle Z, (\omega, u), (a_1\tau^c, \cdots, a_m\tau^c)\rangle \end{aligned}$$

Since the object a is regular, the variable X does not appear in the substructure of a. Since $V(a) \cap V(b) = \phi$, the variable Y does not appear in the substructure of a. τ^s does not have any effect on a, and $\tau^v = [\![X \xrightarrow{o} Z, Y \xrightarrow{o} Z]\!]$ does not have any effect on subobjects of a.

$$\begin{aligned} b\tau &= \langle Y, (\beta, t), ()\rangle\tau^s{\cdot}\tau^c{\cdot}\tau^v \\ &= \langle Z, (\beta, t), (a_1, \cdots, a_m)\rangle\tau^c{\cdot}\tau^v \\ &= \langle Z, (\omega, u), (a_1\tau^c, \cdots, a_m\tau^c)\rangle \end{aligned}$$

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. $\tau^t = \tau_1 \cdots \tau_n$, and $\tau^s = [\![\,]\!]$.

$$\begin{aligned} a\tau &= \langle X, (\alpha, s), (a_1, \cdots, a_n)\rangle\tau^t{\cdot}\tau^c{\cdot}\tau^v \\ &= \langle X, (\alpha, s)\tau_1, (z_1, a_2\tau_1, \cdots, a_n\tau_1)\rangle \\ &\qquad \tau_2 \cdots \tau_n{\cdot}\tau^c{\cdot}\tau^v \\ &= \langle X, (\alpha, s)\tau_1{\cdot}\tau_2, (z_1\tau_2, z_2, \\ &\qquad a_3\tau_1{\cdot}\tau_2, \cdots, a_n\tau_1)\rangle\tau_3 \cdots \tau_n{\cdot}\tau^c{\cdot}\tau^v \\ &= \cdots \\ &= \langle X, (\alpha, s)\tau^t, \\ &\qquad (z_1\tau_2 \cdots \tau_n, z_2\tau_3 \cdots \tau_n, \cdots, \\ &\qquad z_{n-1}\tau_n, z_n)\rangle\tau^c{\cdot}\tau^v \\ &= \langle Z, (\omega, u), \\ &\qquad (z_1\tau_2 \cdots \tau_n{\cdot}\tau^c, z_2\tau_3 \cdots \tau_n{\cdot}\tau^c, \cdots, \\ &\qquad z_{n-1}\tau_n{\cdot}\tau^c, z_n\tau^c)\rangle \end{aligned}$$

$$\begin{aligned} b\tau &= \langle Z, (\omega, u), \\ &\qquad (z_1\tau_2 \cdots \tau_n{\cdot}\tau^c, z_2\tau_3 \cdots \tau_n{\cdot}\tau^c, \cdots, \\ &\qquad z_{n-1}\tau_n{\cdot}\tau^c, z_n\tau^c)\rangle \end{aligned}$$

Thus, we have $a\tau = b\tau$. In all cases (1) ~ (3), $a\tau = b\tau$ 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

[1] H. Aït-Kaci. An introduction to LIFE – programming with logic, inheritance, functions, and equations. Technical report, Digital Equipment Corporation Paris Research Laboratory, 1993. `http://www.isg.-sfu.ca/ftp/pub/hak/publish/-ilps93-life.ps.Z`.

[2] H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 3:185 – 215, 1986.

[3] Y. Kawaguchi, K. Akama, and E. Miyamoto. Representation and calculation of objects with classes and substructures – a simple computational framework based on logic –. *Journal of Jsai.*, 12(1):48 – 57, 1997.

[4] Y. Kawaguchi, K. Akama, and E. Miyamoto. Applying program transformation to type inference on a logic language. *IEICE Trans. on Inf. & Syst.*, E81–D(11):1141 – 1147, November 1998.

[5] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.