Declarative Semantics for A Programming Language with Class Hierarchies and Substructure

Yuuichi KAWAGUCHI* Kiyoshi AKAMA** Eiichi MIYAMOTO***

(Received 29 November 1999)

Abstract

This paper constructs declarative semantics for a language that can deal with class hierarchies and substructure. Lloyd has successfully constructed declarative semantics for a programming language with terms. In some cases, objects that are more complex than terms are needed for efficient or direct expressing. For each of those objects, its original extention has added to Lloyd's theories. They lack a global viewpoint. Our theoretical basis for constructing semantics is a theory of declarative computation model (DP). DP gives us a method of constructing semantics independently of particular objects. DP deals with programs on a four-tuple $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$, called a specialization system. We construct the specialization system for our language in this paper.

Key words : *Declarative Semantics, Declarative Programming Language, Class Hierarchy, Substructure, Specialization System.*

1 Introduction

There are two kind of semantics of programs; one is declarative semantics and another is procedural semantics. We are interested in declarative semantics. This paper constructs declarative semantics for our target language LHS [7] that can deals with classes and substructures.

Lloyd [8] has constructed declarative semantics for Prolog. Prologonly uses terms that consist of constants, variables and functions. Terms can express many kinds of knowledges or data, but there are some cases where more efficient or direct expressions are needed. Expressing classes and substructures are the instances of such cases. Some programming languages for those objects are presented, such as LIFE [2] and λ Prolog [9], and declarative semantics of them are constructed.

In many cases, they have constructed theories by adding its original extension to Lloyd's declarative semantics. Each theories is independent of other theories. It is hard for us to see the relationship between their theories. For example, we do not see the relationship between LIFE's Ψ-terms and programs in λ Prolog. They lack the global theories that combine them.

This paper construct declarative semantics for LHS. Our theoretical basis is a theory of declarative computation model. It is also called DP [3]. DP gives a method of constructing semantics of programs. The method is independent of particular programming languages or objects. It is very flexible. We get a global viewpoint with DP.

* 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

2 General Framework

This section introduces our theoretical basis for constructing semantics. It is called *a theory of declarative computation model* or simply DP [3].

2.1 Specialization System

DP can construct theories independently of particular target objects. If a set of all target objects and a set of operators for them are given, DP can judge the correctness of computations. To do this, DP needs a four-tople $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$ that is defined by Definition 1.

Definition 1 (Specialization System)
$\mathcal{A}, \mathcal{G}, \mathcal{S}$ are sets. $\mu : \mathcal{S} \to$ partial_map$(\mathcal{A})$[1] is a function. The four-pair $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$ holding the following conditions is called a specialization system.

(1) $\forall s_1, s_2, \exists s \in \mathcal{S} : \mu(s) = \mu(s_1) \circ \mu(s_2)$[2]
(2) $\exists s \in \mathcal{S}, \forall a \in \mathcal{A} : \mu(s)(a) = a$
(3) $\mathcal{G} \subset \mathcal{A}$

Each element in $\mathcal{A}$ is called an atomic object *or simply object. Each element in $\mathcal{G}$ is called a* ground object. *Each element in $\mathcal{S}$ is called a* specialization. *The specialization shown in (2) is called a* unit specialization. □

Users of DP must construct the specialization system for target objects. They then can apply theories of DP to their computations.

2.2 Program and Declarative Semantics

Definition 2 (Declarative Program)
Let Γ be a specialization system $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$. A declarative program on Γ is defined below.

(1) A definite clause on Γ is a formula that has a form as $H \leftarrow B_1, \cdots, B_n$ $(0 \leq n)$, where $H, B_1, \cdots, B_n \in \mathcal{A}$. H is called the head of the definite clause. The sequence $B_1, \cdots, B_n$ is called the body of the definite clause.

(2) A declarative program on Γ is a set of some definite clauses on Γ. □

In the rest of this paper, we call a definite clause simply a clause and call a declarative program simply a program. $\mathcal{P}(\Gamma)$ stands for a set of all programs on a specialization system Γ.

Definition 3 (Some Notations) *Given a specialization system $\Gamma = \langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$.*

(1) For a clause C on Γ, head(C) stands for the head of C and body(C) stands for a set of all objects appearing in the body of C.

(2) For $P \in \mathcal{P}(\Gamma)$, Gclause$(P)$ stands for a set of all ground instances of clauses in P. Here, a ground instance of a clause $H \leftarrow B_1, \cdots, B_n$ is a clause $H' \leftarrow B'_1, \cdots, B'_n$, where $\exists \theta \in \mathcal{S}$, $H' = H\theta$, $B'_1 = B_1\theta$, $\cdots$, $B'_n = B_n\theta$. Note that we call an operation that makes a ground instance from a clause groundizing *of the clause.* □

Definition 4 (Declarative Semantics)
Given a specialization system $\Gamma = \langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$. For $P \in \mathcal{P}(\Gamma)$, the declarative semantics of P is denoted as $\mathcal{M}(P)$ and defined as

$$\mathcal{M}(P) = \bigcup_{n=0}^{\infty} T_P{}^n(\phi),$$

where T_P is a function defined as

$$T_P : 2^{\mathcal{G}} \to 2^{\mathcal{G}},$$
$$T_P(x) = \{head(C) \mid body(C) \subset x, C \in Gclause(P)\},$$
$$T_P{}^n(x) = \overbrace{T_P(T_P(\cdots(T_P(x))))}^{n \text{ times}}.$$

□

This is our declarative semantics. It has some resemblance to the semantics described by Lloyd [8]. However, our semantics is different from it at least in two points. One: ours does not depend on particular target objects, but Lloyd's depends on terms. Two: to compute the declarative semantics of a given program, Lloyd's definition of the declarative semantics must use logical consequences $\models$. This means that it fixes the method of computations to resolutions based on unifications. Our $\mathcal{M}(P)$ does not fix the method of computations. All computations constructed with elements in $\mathcal{S}$ are allowed.

[1] partial_map(X) is a set of all partial maps on X.
[2] $\mu(s_1) \circ \mu(s_2)$ is a combination function of $\mu(s_1)$ and $\mu(s_1)$.

3 The Language

This section provides formal definitions of our target language. Intuitive meanings of the language are described in [7].

3.1 Objects – $\mathcal{A}$, $\mathcal{G}$

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 $a \rightarrow a$. □

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. □

Definition 5 (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 $b \xrightarrow{*} a$, iff one of following conditions holds:*

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

□

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

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

Definition 6 (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.* □

Definition 7 (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:*

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

□

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 8 (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.* □

Definition 9 (Atomic Object) *Let $\mathcal{A}$ be a set such that:*

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

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.* □

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 $V(a)$.

Definition 10 (Ground Object) *An atomic object $a \in \mathcal{A}$ holding $V(a) = \phi$ is called a* ground object. *$\mathcal{G}$ stands for a set of all ground objects.* □

3.2 Operations – S, μ

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

3.2.1 MGCD

Definition 11 (MGCD) *For two classes $s, t \in \mathbb{C}^*$, a class $u \in \mathbb{C}^*$ that satisfies the following conditions is called the* most general common descendant *(MGCD) of s and t, and such u is denoted as $\Upsilon(s, t)$:*

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

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

Proposition 2 *If the MGCD of two classes is not* $\bot$, *it is unique.* □

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* $s_i \xrightarrow{*} t_i$, *then assign* $u_i := t_i$.
   - *if* $t_i \xrightarrow{*} s_i$, *then assign* $u_i := s_i$.
   - *otherwise assign* $u = \bot$, *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* $u \neq \bot$, *then* $u = \Upsilon(s, t)$. □

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

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

3.2.2 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 12 (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* $\alpha \triangleright t$ *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 $(\alpha, s)[\alpha \triangleright t]$ *is performed,* $\Upsilon(s, t) \neq \bot$ *must hold.*

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

(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 12, 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 13 (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 $\theta = [\![\theta_1, \cdots, \theta_m]\!]$ *and* $\sigma = [\![\sigma_1, \cdots, \sigma_n]\!]$ *$(0 \leq m,n)$ 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* $\mu(\theta)(a)$ *or simply* $a\theta$ *and the result is defined as*

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

$\mathcal{S}$ stands for a set of all specializations. □

For specializing clausess, specialization operators are extended. 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 $\mu(\theta)(C)$ or $C\theta$ and is defined as

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

4 Discussion

4.1 Specialization System

We describe that the specialization system constructed in Section 3 holds the required condition defined by Definition 1.

As to the condition (1), for $\forall\theta_1, \theta_2 \in \mathcal{S}$ and $a \in \mathcal{A}$

$$\begin{aligned} (\mu(\theta_1) \circ \mu(\theta_2))(a) &= \mu(\theta_1)(\mu(\theta_2)(a)) \\ &= \mu(\theta_1)(a\theta_1) \\ &= (a\theta_1)\theta_2 \\ &= a(\theta_1{\cdot}\theta_2) \\ &= \mu(\theta_1{\cdot}\theta_2)(a). \end{aligned}$$

Thus, the condition (1) holds with $s = \theta_1{\cdot}\theta_2 \in \mathcal{S}$.

As to the condition (2), the unit specialization $[\![\,]\!]$ is the desired operator.

As to the condition (3), Definition 10 implies that $\mathcal{G} \subset \mathcal{A}$.

Therefore, we succeeded to constructed the specialization system for our target language. This implies that we succeeded to construct declarative semantics for the language.

4.2 Related Work

Lloyd [8] has constructed declarative semantics for logic programming languages with terms. Terms can express many kinds of knowledges or data. However, as to some complex objects, such as classes or substructures, expressing them by terms is not the best way. For more efficient or direct method to express class hierarchies, LOGIN [2] (or its descendant LIFE [1]) has introduced ϕ–terms. For substructures, λ Prolog [9] has combined the first order logic and the λ calculus. LOGIN and λ Prolog have succeeded to construct their theories. However, we can not see the relationship between the two theories.

As described in Section 2, DP provides a general way to construct semantics for many programming languages. We succeeded to construct declarative semantics for a language with classes and substructures in this paper. If we succeed to construct theories for other kind of programming languages, then we can see them with a unified viewpoint, *i.e..*, DP. This is the main value of this paper. Indeed, we have succeeded for programming languages with terms, multi-set, and constraints [4, 6, 5].

5 Conclusion

We succeeded to construct declarative semantics on DP for our target language that deals with classes and substructures, by constructing the specialization system.

Acknowledgement

Authors thank to Youichiro Kojima in TNCT. He often carred one of authors to the office by his car.

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] K. Akama. Declarative semantics of logic programs on parameterized representation systems. *Advances in Software Science and Technology*, 5:45 – 63, 1993.

[4] K. Akama, Y. Kawaguchi, and E. Miyamoto. Equivalent transformation for member constraints on the term domain. *Journal of Jsai*, 13(2):274 – 282, May 1998. in Japanese.

[5] K. Akama, Y. Kawaguchi, and E. Miyamoto. Equivalent transformations for equational constraints and a specialization system on the multiset domain. IEICE Trans. on Inf. & Syst. SS97-55, IEICE, January 1998. in Japanese.

[6] K. Akama, Y. Kawaguchi, and E. Miyamoto. Equivalent transformations for equational constraints on the multiset domain. *Journal of Jsai.*, 13(3):395 – 403, May 1998. in Japanese.

[7] 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.

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

[9] G. Nadathur and D. Miller. An overview of λ Prolog. In R. Kowalski and K. Bowen, editors, *Proceedings of the Logic Programming: Fifth International Conference on Logic Programming*, pages 810 – 827. The MIT Press, 1988.

Appendix

Proof of Proposition 1.
Suppose $a \neq b$. By Definition 5, $a \xrightarrow{*} b$ 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, $b \xrightarrow{*} a$ 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. □

Proof of Proposition 2.
For $s, t \in \mathbb{C}^*$, assume that $u = \Upsilon(s, t)$ and $v = \Upsilon(s, t)$ $(u, v \neq \bot)$. Both $u \succeq v$ and $v \succeq u$ hold by Definition 11. The length of u and the length of v are equal by Definition 7. 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 $u_i \xrightarrow{*} v_i$ and $v_i \xrightarrow{*} u_i$ for each $i = 1, \cdots, n$. By Proposition 1, $u_i = v_i$ for each $i = 1, \cdots, n$. Thus, $u = v$. □

Proof of Proposition 3.
[Most general]
Suppose that for some $v = (v_1, \cdots, v_k) \in \mathbb{C}^*$, both $s \succeq v$ and $t \succeq v$ hold. $s \succeq v$ implies $m \leq k$. By the definition of $\succeq$, $s_i \xrightarrow{*} v_i$ and $t_i \xrightarrow{*} v_i$ for each $i = 1, \cdots, m$. If $s_i \xrightarrow{*} t_i$ 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 $s_i \xrightarrow{*} t_i$ then $u_i = t_i$, and this implies that $s_i \xrightarrow{*} u_i$ and $t_i \xrightarrow{*} u_i$. Alternatively if $t_i \xrightarrow{*} s_i$, then $u_i = s_i$, and this implies that $t_i \xrightarrow{*} u_i$ and $s_i \xrightarrow{*} u_i$. Thus, in any case, $t \succeq u$. For each $i = n+1, \cdots, m$, we have $s_i \xrightarrow{*} u_i$, since $u_i = s_i$. Thus, $s \succeq u$. □

Proof of Proposition 4.
Let $v = (v_i, \cdots, v_k)$ be $\Upsilon(s, t) \neq \bot$ $(m, n \leq k)$.

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

(2) Assume Algorithm 1 derives $\bot$. This implies that neither $s_i \xrightarrow{*} t_i$ nor $t_i \xrightarrow{*} s_i$ for some values of i. $v = \Upsilon(s, t)$ implies that $s_i \xrightarrow{*} v_i$ and $t_i \xrightarrow{*} v_i$ for all $i = 1, \cdots, n$. There are four possible cases according to Definition 5:

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. $s_i = s_i^1 \to \cdots \to s_i^a = v_i$ 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 $\Upsilon(s, t) \neq \perp$. □