Atomic objects and Class objects : Two Objects Based on a
Theory of Equivalent Transformations

Yuuichi KAWAGUCHI*, Kiyoshi AKAMA** and Eiichi MIYAMOTO***
(Received 28 November 1997)

Abstract
There are two kinds of higher-order objects. One is predicate name, and the other is a whole atom. Many programming language which deal with declarative aspects of programs and procedural aspects independently allows those higher-order objects. They are allowed for only practical purposes and often beyond the fundamental theories of the language. We have developed a declarative language named L_HS, based on a theory of equivalent transformations. The language originally has an ability for dealing with one kind of higher-order objects. They are called atomic objects. This paper aims to extend the ability to deal with other kind of higher-order objects. They are called class objects.

1 Introduction
We are interested in two aspects of programming languages, declarative semantics and procedural semantics. Our computational framework based on equivalent transformations (ET) can deal with these two flexibly and efficiently. Two programs are said to be equivalent when their declarative semantics are equal. A program transformation algorithm is an equivalent transformation when a given program and a program obtained by applying the algorithm are equivalent. In the ET framework, any method for executing programs is allowed, as long as it is an equivalent transformation. Comparing the ET framework to other computational frameworks, (e.g., the framework of logic programming (LP)), the declarative semantics of programs are independent of procedural

1 The word "object" in this paper is different from the one in object-oriented languages.

* Associate Professor : Department of Computer Engineering
** Associate Professor : Department of System Information Engineering, Faculty of Engineering, Hokkaido University
*** Professor : Department of System Information Engineering, Faculty of Engineering, Hokkaido University

semantics of them, but the method for executing programs is restricted to logical inference. It has been shown that programs can be executed more flexibly and efficiently in the ET framework than in the LP framework [3].

In the ET framework, the method for expressing objects is also flexible. In contrast with the LP framework in which objects are expressed only by terms, the ET framework allows many representations of objects. A method how to obtain the declarative semantics from representations of objects (or programs) is generally defined [10]. The method is independent of the representation. In solving a given problem within the ET framework, there are two alternative ways to express objects: (1) using expressions that are specialized for each given problem, or (2) using general expressions that can express many different objects in a single and general form.

In order to establish a general method for expressing many different kinds of objects, we have developed a declarative language, named L_HS [11]. In L_HS, all objects are expressed by "atomic objects." Each atomic object has a class and a substructure. The class determines the relationship between one object and another, and the substructure expresses the structure of the object. For example, an object that represents a formula has a class (e.g., number, term, etc.) and a substructure, (e.g., 5, (+ 18 2), etc.). This paper provides a theoretical foundation for
higher-order objects in \( L_{\text{HS}} \), making \( L_{\text{HS}} \) more expressive. This paper focuses on the expression of higher-order objects and shows some problems and a solution. In the next section (Section 2), there is a general discussion on some examples. In Section 3, the definition of the target language of this paper, \( L_{\text{HS}} \), is given. Section 4 shows those higher-order expressions in the language. Section 5 provides an extension to the language. In Section 6, our method is compared with other methods.

2 Two Higher-Order Expressions

The predicate \texttt{and}, which makes a logical conjunction of two atoms, may be defined in Prolog-like style as:

\[
\text{and}(P, Q) :- P, Q.
\]

The theoretical foundation of Prolog is the first-order predicate logic [15]. It does not allow variables to appear at places where atoms are located. Variables referring to atoms, such as \( P \) and \( Q \) on the right-hand side, are not allowed in the first-order predicate logic, since atoms should be placed there. Therefore, they are beyond pure Prolog. Prolog uses such expressions for practical purposes.

Variables are higher-order when they appear at places where variables are not allowed to appear by the first-order predicate logic. Clauses, atoms, and predicates are higher-order when they have higher-order variables.

Many logic programming languages for such higher-order expressions have been developed with their theoretical foundations, i.e., higher-order logic. For example, the language LIFE [1] expresses all objects with \( \Psi \)-terms. Since the variables in LIFE can refer to any \( \Psi \)-terms, including atoms, the \texttt{and} clause\(^2\) above is in LIFE's theoretical foundations.

As another example of a higher-order expression, let us consider \texttt{map} clauses. They are expressed in Prolog-like style as

\[
\text{map}(P, [], []) :- .
\]
\[
\text{map}(P, [E1|R1], [E2|R2]) :- P(E1, E2), \text{map}(P, R1, R2).
\]

The variable \( P \) in the atom \( P(E1, E2) \) on the right-hand side of the second clause refers to a predicate name. Such occurrences of variables are also not allowed in first-order predicate logic. Therefore, \texttt{map} clauses above are higher-order. Since LIFE does not have higher-order variables for expressing only predicate names, the \texttt{map} clauses are beyond the theoretical foundations of LIFE.

M. Hanus tried to explain the above higher-order expressions only with first-order expressions [8]. He used an additional predicate, \texttt{apply2}, and constants \( \lambda \texttt{not}, \lambda \texttt{inc}, \ldots \), corresponding to the predicate \texttt{not}, \texttt{inc}, \ldots, as follows:

\[
\text{map}(P, [], []) :- .
\]
\[
\text{map}(P, [E1|R1], [E2|R2]) :- .
\]
\[
\text{apply2}(P, E1, E2), \text{map}(P, R1, R2).
\]
\[
\text{apply2}(\lambda \texttt{not}, X, Y) :- \texttt{not}(X, Y).
\]
\[
\text{apply2}(\lambda \texttt{inc}, X, Y) :- \texttt{inc}(X, Y).
\]
\[
\text{apply2}(\lambda \ldots)
\]

The expressions here are all first-order. No variable appears at a place where first-order predicate logic does not allow it to appear. Are the \texttt{map} clauses, which are originally higher-order, expressed as first-order expressions?

However, as M. Hanus pointed out, a new problem arises. There is no mechanism for making any relationship between the constant \( \lambda \texttt{not} \) and the predicate \texttt{not}. Thus, a clause that is semantically incorrect, e.g.,

\[
\text{apply2}(\lambda \texttt{inc}, X, Y) :- \texttt{not}(X, Y).
\]

is syntactically correct in his method. The original program, which uses higher-order clauses, did not involve such a clause. This shows that \texttt{map} clauses can not be expressed by only first-order expressions.

In conclusion, we found that (1) there are two kinds of higher-order expressions, such as \texttt{and}
and map, and that (2) higher-order expressions can not be expressed exactly by only first-order expressions.

3 The Target Language
This section defines the target language, $L_{\text{HS}}$ [11], used in this paper. A directed acyclic graph, $(\mathbf{C}, \to)$, and a set, $\mathbf{V}$, are given. $\mathbf{C} \cap \mathbf{V} = \emptyset$ must hold. $\mathbf{C}$ and $\to$ express a set of nodes and a set of edges, respectively. Each element of $\mathbf{C}$ is called a constant and each element of $\mathbf{V}$ is called a variable. For two constants $a, b \in \mathbf{C}$, if an edge $(a, b)$ exists, then the edge is expressed as $a \to b$.

3.1 Atomic Objects
An edge $a \to b$ can be regarded as a parent-child relationship. The constant $a$ is the parent of $b$, and $b$ is the child of $a$. The parent-child relationship makes a partial order on $\mathbf{C}$, defined below.

**Definition 1 (Descendant Relations on $\mathbf{C}$):** For $a, b \in \mathbf{C}$, $b$ is a descendant of $a$ and is expressed as $a \to b$ iff one of the following conditions holds:

- $a = b$, or
- $\exists a' \in \mathbf{C}$ s.t. $a \to a'$, $a' \to b$.

If $a \to b$ holds, then there must exist a finite sequence $a = x_1 \to \cdots \to x_n = b$, where $x_1, \cdots, x_n \in \mathbf{C}$ ($n \geq 1$).

**Definition 2 (Class):** A class is a sequence of zero or more constants. $\mathbf{C}'$ stands for a set of all classes. The length of a class is the number of constants included in the class.

Classes are also ordered by descendant relations.

**Definition 3 (Descendant Relations on $\mathbf{C}'$):**
For $s = (s_1, \cdots, s_m), t = (t_1, \cdots, t_n) \in \mathbf{C}'$ ($m, n \geq 0$), $t$ is a descendant of $s$, expressed as $s \supset t$, iff the following condition holds:

$n \geq m$, and $s_i \to t_i$ for $i = 1, \cdots, m$.

The class whose length is zero, i.e., $()$, always holds that $() \supset c$ for any class $c \in \mathbf{C}'$. Classes in $L_{\text{HS}}$ are similar to the so-called types in other languages. In $L_{\text{HS}}$, the word “class” is used.

**Definition 4 (Atomic Object):** An atomic object (or simply an object) is a triple $(X, c, s)$. $\mathcal{A}$ stands for the set of all atomic objects. Each part of $(X, c, s)$ is defined recursively by:

- $X \in \mathbf{V}$,
- $c \in \mathbf{C}'$,
- $s = (s_1, \cdots, s_n)$, $s_i \in \mathcal{A}$, $i = 1, \cdots, n$ ($n \geq 0$).

The class $c$ is called the class of the object. The sequence $s$ is called a substructure of that object, and each element of the substructure is called a subobject.

In this paper, upper case letters, such as $X, Y, \cdots$, are used for variables of atomic objects.

Variables of atomic objects are used as references of them. In one atomic object, a variable should always refer to the same atomic object. Different atomic objects should not be referred by the same variable. This rule is called the regularity of atomic objects. A regular atomic object (or simply a regular object) is an object that satisfies this rule. In this paper, all atomic objects are regular objects. According to this rule, we introduce the following notation and abbreviations:

- Using a programming-language-like style, an atomic object $(X, (c_1, \cdots, c_m), (s_1, \cdots, s_n))$ is written as $X : [(c_1, \cdots, c_m), (s_1, \cdots, s_n)]$.

- In an object, if there is no need to indicate that some objects are the same, then the variables of such objects are omitted.

- The second time (and later times) a subobject appears in an object, it is simply noted by its variable.

Some examples of atomic objects, assuming that $\mathbf{C} = \{\text{num}, \text{nil}, \text{cons}, \text{append}\}$ and $\mathbf{V} = \{X\}$ are given, are:

$$[	ext{((5))}]$$
$$[	ext{((list num))}]$$
$$[	ext{((cons num) ((5)) ((nil num))}]$$
$$[	ext{((append) ((nil)) X : []]) X}$$

As shown in these example, classes express not only types (i.e., $()$, $(\text{list num})$) but also data.
or predicates (i.e., (append)).

3.2 Programs

Definition 5 (Definite Clause): A definite clause (or simply a clause) is a formula of the form

\[ H \leftarrow B_1, \ldots, B_n. \]

where \( H, B_1, \ldots, B_n \in \mathcal{A} (n \geq 0) \). \( H \) is called a head (or a head object). Each \( B_i \) is called a body object, and the sequence \( B_1, \ldots, B_n \) is called a body.

The regularity of objects and abbreviated notations are now applied in the scope of one clause.

Definition 6 (Programs): A program is a set of clauses.

For simplicity, assume in \( L_{HS} \) are executed by SLD-resolution, which is usually used in LP languages such as Prolog. It is known that SLD-resolution is an equivalent transformation [4]. The definition of unification algorithm used in SLD-resolution is omitted in this paper.

4 Higher-Order Expressions

Here, let us consider expressing some atomic formulae (atoms). Relations \{ \text{list} \rightarrow \text{cons}, \text{list} \rightarrow \text{nil}, \text{pred} \rightarrow \text{append}, \text{pred} \rightarrow \text{and}, \text{pred} \rightarrow \text{map} \} are given. First, consider the first-order predicate \( \text{append} \), which forms an atom with three lists. The program for \( \text{append} \) is

\[
([\text{append}] \ [\text{nil}] \ X:[()]) \ X \leftarrow.
\]
\[
([\text{append}] \ [\text{cons}] \ A:[()]) \ X:[()]
\]
\[
\quad Y:[]
\]
\[
\quad ([\text{cons}] \ A \ Z:[]) \leftarrow
\]
\[
([\text{append}] \ X \ Y \ Z).
\]

Thus, \( L_{HS} \) can express first-order clauses with atomic objects. Atomic objects give theoretical foundations to clauses.

Next, consider the higher-order predicate \( \text{and} \), which makes a logical conjunction of two atoms. The program for \( \text{and} \) in Prolog-like style is:

\[
\text{and}(P, Q) \leftarrow P, Q.
\]

The corresponding program in \( L_{HS} \) is:

\[
[(\text{and}) \ P:([\text{pred}]) \ Q:([\text{pred}])] \leftarrow P, Q.
\]

Thus, \( L_{HS} \) also seems to be able to express higher-order predicates such as \( \text{and} \).

Next, consider the higher-order predicate \( \text{map} \), described in Section 1. The program for \( \text{map} \) in Prolog-like style is:

\[
\text{map}(P, [], []) \leftarrow
\]
\[
\text{map}(P, [E1|R1], [E2|R2]) \leftarrow P(E1, E2), \text{map}(P, R1, R2).
\]

The corresponding program in \( L_{HS} \) is:

\[
[(\text{map}) \ [\text{pred}] \ L:([\text{nil}]) \ L] \leftarrow.
\]
\[
[(\text{map}) \ P:([\text{pred}])]
\]
\[
[(\text{cons}) \ E1:[]] \ R1:([\text{list}])
\]
\[
[(\text{cons}) \ E2:[]] \ R2:([\text{list}])]
\]
\[
\quad P:([\text{pred}] \ E1 \ E2), \ [\text{map}] P R1 R2.
\]

This results in a class violation, since the first occurrence of an object referred to by the variable \( P \) includes no substructure, and the second object referred to by the variable \( P \) includes two subobjects, \( E1 \) and \( E2 \), as a substructure. Of course, \( P:([\text{pred}]) \) and \( P:([\text{pred}] \ E1 \ E2) \) are unified in \( P':([\text{pred}] \ E1 \ E2) \). However, since the former \( P:([\text{pred}]) \) must express only the name of a predicate as the argument of \( \text{map} \), it must not have the substructure. Thus, the two objects contradict each other, and we cannot express the clauses that satisfy demands from both objects referred to by \( P \). This is not a problem of abbreviated notations, but rather one of limited abilities for higher-order expressions in \( L_{HS} \).

Atomic objects defined in Definition 4 can not give theoretical foundations to expressions such as \( \text{map} \). In the next section (Section 5), as a solution for this problem, we provide an extension to \( L_{HS} \).
5 An Extension to $L_{HS}$-Class Objects

The above class violation occurred because two different objects $P : \{ (p\text{red}) \}$ and $P : \{ (p\text{red}) \} E_1 E_2$ were taken as the same objects. The first object should represent only a predicate name as the first argument of map, and the second object should represent a whole atom. These objects are essentially different.

In fact, the first subobject should not be the atomic object $P : \{ (p\text{red}) \}$; Atomic objects represent atoms; however, the first subobject must represent only the name of a predicate. Classes represent names of predicates in atoms. Therefore, we provide a new method in order to express the idea of "same classes." It is called a class object.

**Definition 7 (Class Object):** A class object is a pair $\langle \alpha, c \rangle$, where $\alpha \in V$ and $c \in C^*$. $C$ stands for the set of all class objects.

Greek letters are used for the variables of class objects. Using a programming language style, a class object $\langle \alpha, c \rangle$ is written as $\alpha : c$. and abbreviations that are similar to those of atomic objects are used.

The definition of the atomic object is changed from Definition 4 to the following definition.

**Definition 8 (Atomic Object (new)):** An atomic object is a triple $\langle X, C, S \rangle$. Each part of it is defined recursively:

- $X \in V$,
- $c \in C$,
- $s = \langle s_1, \ldots, s_n \rangle$, $s_i \in A \cup C$, $i = 1, \ldots, n$ ($n \geq 0$).

Using the class object, the map clause is expressed as:

$$\begin{align*}
\{(\text{map}) \alpha : (\text{pred})\} \\
\{(\text{cons}) E_1 : [()] R_1 : [\text{list}]\} \\
\{(\text{cons}) E_2 : [()] R_2 : [\text{list}]\} & \leftarrow \\
[\alpha, E_1 E_2], \{(\text{map}) \alpha R_1 R_2\}.
\end{align*}$$

There is no class violation with these clauses.

Here, let us consider the and clause again. Supposing we use class objects, it may be:

$$\begin{align*}
\{(\text{and}) \alpha : (\text{pred})\} \{(\beta : (\text{pred}))\} & \leftarrow \\
[\alpha], \beta.
\end{align*}$$

On the left-hand side, and receives two atoms as its arguments. In executing this program, supposing a query is given and the first argument of the query is $\{(\text{append}) X Y Z\}$, then $\alpha : (\text{pred})$ and $\text{append}$ are unified in $\alpha' : (\text{append})$, and then the name of the predicate $\text{append}$ is passed to the body object via the variable $\alpha$. Other information of the $\text{append}$ atom, which is expressed by $X, Y, Z$, however, is not passed to the body object, since the variables that refer to them are not specified. Alternatively, the arguments of atoms that are arguments of and may be explicitly specified as

$$\begin{align*}
\{(\text{and}) \alpha : (\text{pred}) X : [()] Y : [()] Z : [()]\} \\
\{(\beta : (\text{pred}) P : [()] Q : [()]\} & \leftarrow \\
[\alpha, Y, Z], \beta P Q.
\end{align*}$$

In this case, the number of arguments of the atom that is the first argument of and is fixed at three, and the number of arguments of the second atom is fixed at two. If a query $\{(\text{and}) \{(\text{pred}) X_1 Y_1 \cdots\} \}$ is given, $\alpha : (\text{pred}) X Y Z$ and $\{(\text{pred}) X_1 Y_1\}$ are unified in $\alpha' : (\text{pred}) X_2 Y_2 Z$, and then the information that $X_1$ and $Y_1$ originally have is passed to the body. In this case, $Z$ is useless. If a query whose first argument is $\{(\text{pred}) \text{X} \text{Y} \text{Z}\}$ is given, then the unification ends in success, but $Z$ is not passed to the body. Of course, these are similar to the second arguments of and. Thus, this style of and that uses class objects is incomplete. The and clause that uses variables to refer only to atomic objects and does not use class objects is correct.

Thus, two kinds of higher-order clauses exist: one is a whole atom (for and) and the other is only the name of a predicate (for map).

In this way, class objects extend the ability of atomic objects. Atomic objects and class objects
give theoretical foundations to the two kinds of higher-order expressions, i.e. and and map.

6 Related Works

As pointed out by M. Hanus, higher-order expressions, such as map, can not be expressed by replacing them with first-order objects. This means that a framework for dealing with higher-order objects is essential. Every objects in \( L_{\text{HS}} \) has a variable, a class and a substructure. The substructure is a sequence of (sub) objects. Since objects can express atoms, and variables can refer to any objects, objects in the original version of \( L_{\text{HS}} \) can deal with higher-order expressions such as and. Since atomic objects can not refer to predicates, they can not express other higher-order expressions such as map. This means that there are two kinds of higher-order expressions: one is a whole atom and the other is only a predicate.

Since class objects can express predicates, class objects can refer only to predicates from atom. As a result, \( L_{\text{HS}} \) objects have had the ability to deal with both and clauses and map clauses with theoretical foundations.

This section discusses some other higher-order logic programming languages with respect to these two points: i.e., (1) whether the variables can refer to atoms (for and), and (2) whether there is a theoretical mechanism for taking only predicates from atoms (for map).

HiLog [5,19] expresses the second clause of map as follows:

\[
\text{map}(\tau) ([E_1|R_1], [E_2|R_2]) :- \\
\tau(E_1, E_2), \text{map}(\tau)(R_1, R_2).
\]

Since the variable \( \tau \) stands for some predicate names, HiLog can take a predicate name from an atom. However, since HiLog's variables do not refer to whole atoms, HiLog may express the and clause in an inflexible style.

\[\text{map} \ P \ (X::L) \ (Y::K) :- \\
P \ X \ Y, \ \text{map} \ P \ L \ K.\]

The variable \( P \) refer to a predicate name or a \( \lambda \)-term that expresses a function (or an anonymous function). There are no variables that refer to atoms. \( \text{map} \) may also express the and clause in an inflexible style.

\[\text{and}(1=>P, 2=>Q) :- P, Q.\]

\( \lambda \text{Prolog} \ [16,17,18] \) provides higher-order expressions by amalgamating Horn clauses with \( \lambda \)-calculus. It expresses the second clause of map as:

\[\text{map} \ P \ (X::L) \ (Y::K) :- \\
P \ X \ Y, \ \text{map} \ P \ L \ K.\]

The variable \( P \) refer to a predicate name or a \( \lambda \)-term that expresses a function (or an anonymous function). There are no variables that refer to atoms. \( \lambda \text{Prolog} \) may also express the and clause in an inflexible style.

\[\text{and}(1=>P, 2=>Q) :- P, Q.\]

However, since there is no theoretical mechanism that takes only predicates (sorts in LIFE) from atoms (\( \psi \)-terms in LIFE)\(^\text{4}\), LIFE can not express map\(^\text{5}\).

Just as in LIFE, variables in Typed Feature Structure (TFS) \([6,13,14]\) can refer to atoms. The and clause in TFS is

\[\text{AND} = \\
[A: #P \ \text{ATOM}, \ B: #Q \ \text{ATOM}] :- #P, #Q.\]

However, also as in the case of LIFE, since there are no variables for only predicate names (types in TFS), TFS can not express map.

7 Conclusions

As described above, there are two different kinds of higher-order clauses. One is a whole atom and the other is only the name of predicate. Variables in the original version of \( L_{\text{HS}} \) refer to whole atoms but can not refer to only predicates. As a solution, this paper provided class objects as an extension to the class system in \( L_{\text{HS}} \). Class objects allow atomic objects to refer to predicates. Atomic objects and class objects give theoretical foundations to these two higher-order expressions.

\(^4\)For practical purposes, a built-in function \textbf{root.sort} () is provided.

\(^5\)In fact, \textbf{map} is implemented as a built-in function in LIFE.
Acknowledgment

We wish to thank Mark W. McCann for his proofreadings. He delicately read and corrected our paper in English.

References


