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 procedual aspects independently allows those higher-order objects. They are allowed for only practical purposes, and often beyond the foundamental 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 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[1] 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 langauge, 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

[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

higher-order objects in L_{HS}, making L_{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 langauge of this paper, L_{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 and, which makes a logical conjunction of two atoms, may be dafined in Prolog-like style as :

```
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 ψ-terms. Since the variables in LIFE can refer to any ψ-terms, including atoms, the and clause[2] above is in LIFE's theoretical foundations.

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

```
map(P, [], []) :- .
map(P, [E1|R1], [E2|R2]) :-
    P(E1, E2), 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, map clauses above are higher-order. Since LIFE does not have higher-order variables for expressing only predicate names, the 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, apply2, and constants, λnot, λinc, ···, corresponding to the predicate not, inc, ···, as follows:

```
map(P, [], []) :- .
map(P, [E1|R1], [E2|R2]) :- .
  apply2(P,E1,E2), map(P,R1,R2).
apply2(λnot, X, Y) :-not(X, Y).
apply2(λinc, X, Y) :-inc(X, Y).
apply2(λ··· .
```

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 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 λnot and the predicate not. Thus, a clause that is semantically incorrect; *e.g.*,

```
apply2(λinc, X, Y) :- 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 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 and

[2]In this paper, an atom that has a predicate xxx is called a "xxx atom." A clause whose head is a yyy atom is called a "yyy clause."

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_{HS} [11], used in this paper. A directed acyclic graph, $(\mathbf{C}, \rightarrow)$, and a set, $\mathbf{V}$, are given. $\mathbf{C} \cap \mathbf{V} = \phi$ must hold. $\mathbf{C}$ and $\rightarrow$ 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 $\langle a,b \rangle$ exists, then the edge is expressed as $a \rightarrow b$.

3.1 Atomic Objects

An edge $a \rightarrow b$ 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 $a \overset{*}{\rightarrow} b$ iff one of the following conditions holds:

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

If $a \overset{*}{\rightarrow} b$ holds,then there must exist a finite sequence $a = x_1 \rightarrow \cdots \rightarrow x_n = b$, 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 $s \succeq t$, iff the following condition holds:

$$n \geq m, \text{ and } s_i \overset{*}{\rightarrow} t_i, \text{for } i = 1, \cdots, m. \quad \square$$

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

Definition 4 (Atomic Object): An *atomic object* (or simply an object) is a triple $\langle X, c, s \rangle$. $\mathcal{A}$ stands for the set of all atomic objects, Each part of $\langle X, c, s \rangle$ 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 $\langle X, (c_1, \cdots, c_m), (s_1, \cdots, s_n) \rangle$ 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 $\mathbf{C} = \{5, \mathtt{num}, \mathtt{nil}, \mathtt{cons}, \mathtt{append}\}$ and $\mathbf{V} = \{\mathtt{X}\}$ are given, are:

- `[(5)]`
- `[(list num)]`
- `[(cons num) [(5)] [(nil num)]]`
- `[(append) [(nil)] X:[()] X]`

As shown in these example, classes express not only types (*i.e.*, `()`, `(list num)`) but also data

(*i.e.*, (5), (cons num), (nil), (nil num)) 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

$$H \leftarrow B_1, \cdots, B_n.$$

where $H, B_1, \cdots, B_n \in \mathcal{A}\ (n \geq 0)$. H is called a *head* (or a head object). Each B_i is called a *body object*, and the sequence $B_1, \cdots, B_n$ 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 fomulae(atoms). Relations {list→cons, list →nil, pred →append, pred →and, pred →map } are given. First, consider the first-order predicate append, which forms an atom with three lists. The proglam for append is

```
[(append) [(nil)] X:[()] X] ←.
[(append) [(cons) A:[()] X:[()]]
    Y:[()]
    [(cons) A Z:[()]]] ←
[(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 and, which makes a logical conjunction of two atoms. The program for and in Prolog-like style is :

```
and(P, Q) :- P, Q.
```

The corresponding program in L_{HS} is :

```
[(and) P:[(pred)] Q:[(pred)]]←
  P, Q.
```

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

Next, consider the higher-order predicate map, described in Section 1. The program for map in Prolog-like style is :

```
map(P, [], []) :-
map(P, [E1|R1], [E2|R2]) :-
    P(E1, E2), map(P, R1, R2).
```

The corresponding program in L_{HS} is :

```
[(map) [(pred)] L:[(nil)] L]←.
[(map) P:[(pred)]
  [(cons) E1:[()] R1:[(list)]]
  [(cons) E2:[()] R2:[(list)]]]←
P:[(pred) E1 E2], [(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:[(pred)] and P:[(pred) E1 E2] are unified in P′:[(pred) E1 E2]. However, since the former P:[(pred)] must express only the name of a predicate as the argument of map, it must not have the substructure. Thus, the two objects contradict each other, and we can not 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 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:[(pred)]` and `P:[(pred) E1 E2]` 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:[(pred)]`. 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 "the same classes." It is called a class object.

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

Greek letters are used for the variables of class objects[3]. 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:

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

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

```
[(map) α:(pred)]
  [(cons) E1:[()]R1:[(list)]]
  [(cons) E2:[()]R2:[(list)]]]←
[α E1 E2], [(map) α R1 R2].
```

There is no class violation with these clauses.

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

```
[(and) [α:(pred)] [β:(pred)]] ←
    [α], [β].
```

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 `[(append) X Y Z]`, then `α:(pred)` and `(append)` are unified in `α′:(append)`, and then the name of the predicate `(append)` is passed to the body object via the variable `α`. Other information of the `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

```
[(and)
  [α:(pred) X:[()]Y:[()]Z:[()]]
  [β:(pred) P:[()]Q:[()]]] ←
[α X Y Z], [β P Q].
```

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 `[(and) [(pred) X1 Y1] ⋯]` is given, `[α:(pred) X Y Z]` and `[(pred) X1 Y1]` are unified in `[α′:(pred) X2 Y2 Z]`, and then the information that `X1` and `Y1` originally have is passed to the body. In this case, `Z` is useless. If a query whose first argument is `[(pred) W X Y 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

[3]There are no essential differences between variables for class objetcs and those for atomic 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_{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_{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_{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:

```
map(τ)([E1|R1], [E2|R2]) :-
    τ(E1, E2), map(τ)(R1, R2).
```

Since the variable τ 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.

λProlog [16,17,18] provides higher-order expressions by amalgamating Horn clauses with λ- calculus. It expresses the second clause of map as :

```
map P (X::L) (Y::K) :-
    P X Y, map P L K.
```

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

LIFE[1] (or its ancestor LOGIN[2]) is based on ψ-terms, which express all objects, including atoms. Variables refer to any ψ-terms, and the and clause in LIFE is :

```
and(1=>P, 2=>Q) :- P, Q.
```

However, since there is no theoretical mechanism that takes only predicates (sorts in LIFE) from atoms (ψ-terms in LIFE)[4], LIFE can not express map[5].

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

```
AND =
  [A:#P ATOM, B:#Q 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_{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_{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 root_sort () is provided.

[5]In fact, 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

[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,pp.185-215(1986).

[3] Akama K, Kawaguchi Y. and Miyamoto E.: "Equivalent transformation for member constraints on the term domain", Journal of Jsai (1998). to be appeared.

[4] Akama K, Kawaguchi Y. and Miyamoto E.: "Solving logical problems by equivalent transformation(2) — limitations of SLD resolution", Journal of Jsai (1997). now posting.

[5] W. Chen, M. Kifer and D. S. Warren: "HiLog: A first-order semantics for higher- order logic programming constructs", Proceedings of the North American Conference on Logic Programming (Eds.by E. Lusk and R.Overbeek), Cleveland, Ohio, PP.1090-1114(1989).

[6] M. C. Emele: "TFS the typed feature structure representation formalism", Proceedings of the International Workshop on Sharable Natural Language Resources (1994). http://www2.ims.uni-stuttgart.de/ ~emele/ TFS.html.

[7] Hagiya M.: "Higher-order unification and generalization of proofs", Journal of Jsai., 6,3,pp.78-86(1991).

[8] M. Hanus: "Logic programming with type specifications", Types in Logic Programming (Ed.by F. Pfenning), Logic Programming Serises, The MIT Press, pp.91-140 (1992).

[9] Harao M. and Iwanuma K.: "Complexity of higher-order unification algorithm", Computer Software, 8 ,1,pp.41-53(1989).

[10] Kawaguchi Y., Akama K., and Miyamoto E.: "A construction of semantics for a declarative language with hierarchies and substructures", Technical Report jerr 970604.01, Dept. of Joho, Tomakomai National College of Technology, yuuichi @jo.tomakomai-ct.ac.jp (1997).

[11] Kawaguchi Y., Akama K., and Miyamoto E.: "Representation and calculation of objects with classes and substructures — a simple computational framework based on logic —", ournal of Jsai.,12,1,pp.48-57(1997).

[12] Kawaguchi Y., Akama K., and Miyamoto E.: "Applying program transformation to type inference on a logic language", Technical Report jerr 970604.02, Dept. of Joho, Tomakomai National College of Technology, yuuichi@jo.tomakomai-ct.ac.jp (1997).

[13] Kogure K.: "Feature structures(1)", Journal of Jsai.,8,2(1993).

[14] Kogure K.: "Feature structures(2)", Journal of Jsai.,8,3(1993).

[15] J. W. Lloyd: "Foundations of Logic Programming", Springer-Verlag, second edition (1987).

[16] G. Nadathur and D. Miller: "An overview of λProlog ", Proceedings of the Logic Programming: Fifth International Conference on Logic Programming (Eds. by R. Kowalski and K. Bowen), The MIT Press, pp.810-827(1988).

[17] G. Nadathur and F. Pfenning: "The type system of a higher-order logic programming language", Types in Logic Programming (Ed. by F. Pfenning), Logic Programming Series, The MIT Press, chapter 9, pp.245-283(1992).

[18] G. Nadathur and D. Miller: "Higher-order logic programming", Technical report, Computer Science Department, Duke University(1995).
ftp:// ftp.cis.upenn.edu/ pub/ papers/ miller/ Handbook_Logic_AI_LP.dvi.Z.

[19] E. Yardeni, T. Frühwirth and E. Shapiro: "Polymorphically typed logic programs", Types in Logic Programming (Ed.by F. Pfenning), Logic Programming Series, The MIT Press, chapter 2,pp.63-90(1992).