STUDIA UNIV BABEŞ-BOLYAI, MATHEMATICA, XXXVIII, 3, 1993

FORMAL SPECIFICATION FOR SMALLTALK THROUGH LAMBDA-CALCULUS. A COMPARATIVE STUDY

Simona MOTOGNA*

Dedicated to Professor Emil Muntean on his 60th anniversary

Received January 31, 1994
MS subject classification 68N05, 68Q55, 68Q60

REZUMAT. - Specificarea formală prin lambda-calcul a limbajului Smalltalk. Studiu comparativ. În această lucrare sunt discutate două modele de specificații prin lambda-calcul ale limbajului Smalltalk. Prin considerarea unei ierarhii în mediul Smalltalk au fost comparate cele două modele din punctul de vedere al criteriilor pe care o specificație trebuie să le respecte

Introduction. Denotational semantics based on lambda-calculus has been a very used specification method in some models for formalization of the object oriented languages Cardelli [1] stated that the only notion critically associated with object oriented programming is inheritance This paper tends to present a comparative study of some denotational specification models for inheritance. All the models presented are based on the object oriented language Smalltalk so the study will be somehow easily

Inheritance is the possibility to define a new class (named subclass) using the definition of one or more existing classes (named superclass) A subclass can inherit instance variables or methods from the parent class The meaning of this property can be understand using a "look-up" method Suppose a message, containing the call of a method, is sent to an object Then the look-up method search the class containing the method.

* *"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, 3400 Cluj-Napoca, Romania*

```
        procedure lookup (name, class)
    if name = local_name then do local_action
      else if (inherited_module= NIL) then undefined_name
          else lookup (name, inherited_module)
```

In Smalltalk there are two special variables which can appear in a message These two variables are ***self*** and ***super*** When the message contains the variable **self** the search begins in the instance class

lookup (name, instance class)

and if the message contains the variable **super** then the search begins in the superclass of the instance class (which contains the method)

lookup (name, superclass of the instance class)

The mechanism of **self** and **super** supports the access of the methods which have the same names either from the superclass and the subclass, although they have a different action If a subclass redefines a method which was defined in superclass then this mechanism became very useful

Kamin's specification model

In [5], Samuel Kamin proposes a denotational definition for Smalltalk The major characteristic of this definition is the simple way in which inheritance is handled and the paper contains an version of this semantics in Standard ML which can be executed

The Smalltalk defined by Kamin has some modifications.

- only a few primitives are defined,
- the only literals which are permitted in the language are the integers and the arrays,
- the *pools* variables are omitted, excepting class variables,

- contexts are not objects,

- methods are not objects, so it isn't possible to create methods dynamically,

- there is a special way in which the array constants are handled any time when an array constant is evaluated a new array is created;

The definition is based on some semantic maps which assign meaning to syntactic entities These maps models the hierarchy, the inheritance and message passing mechanisms

Let's consider now the following example in Smalltalk. The hierarchy H contains two classes Point and Point1, where Point1 is a subclass of class Point In Point are defined two methods, the first being redefined in Point1 and the second method invokes the first one

```
      class Point
instanceVariablesNames
  ' x y'
method DistFrom Orig
 sqrt(self x^2 + self y^2)
method CloserToOrig(p)=
 (self DistFromOrig < p.DistFromOrig)
      Point superclass  Point1
      method DistFromOrig
 (self x + self y)
```

Let H be the hierarchy containing Point and Point1 For an easy reading, we will denote

m1 = method DistFromOrig

m2 = method CloserToOrig and

R = C[H]

In this example D[H] = YR = sup{ ⊥, R⊥, R(R⊥), .}

For a complete understanding of the example we shall recall the notations used in the specification model Kamin has defined some semantic maps to specify the behavior of the object oriented mechanisms Inheritance is modeled by the two semantic maps C and D

D Hier --> Env
C Hier --> Env --> Env

D[H] = **Y(C[H])**
C[H]ρ
= λ<c,m>
let H(c) = C S w x F
in if F(m) = no-def **then** ρ<S,m> **else M[**F(m)**]**ρ

where **C[H]** defines an application from the environment (meaning of the hierarchy H) to the environment (noted Env) which executes an "inheritance step" For example, if H is a hierarchy containing the class Point1 and it's superclass Point, m2 is an attribute not defined in Point1 and ρ(<Point,m>) is defined, then (**Point1[H]**ρ)(<Point1,m>) will be defined equivalent with ρ(<Point,m>) So, Point1 has "inherited" the definition of m2 from Point **Point1[H]** executes only an inheritance step· if D is a subclass of Point1, which doesn't define the attribute m2, then (**Point1[H]**ρ)(<D,m>) is not defined, but (**Point1[H](Point1[H]**ρ))(<D,m>) is All the inheritances are resolved here

We use ⊥ to denote the primitive routines (e g machine arithmetic).

We will construct some of these environments to understand the inheritance mechanism

R⊥ = { <Point, m1> -> ⊥,
<Point, m2> -> ⊥,
<Point1, m1> -> ⊥,
<Point1, m2> -> ⊥,
<Smallinteger, +> -> . , }

R(R⊥) = { <Point, m1> -> euclidian distance,
<Point, m2> -> if the arguments are from the class
Point then compare the euclidian
distance, else ⊥,
<Point1, m1> -> distance,
<Point1, m2> -> R⊥(<Point,m2>) = ⊥, .}

At first, all the methods are undefined After one step (see R⊥) are defined only those methods which send no messages (like * or +) or invoke primitive methods After two steps (see R(R⊥)), in addition to R⊥, are defined the two versions of method DistFromOrig and the method CloserToOrig only for the class Point After three steps CloserToOrig is defined because it can see the definition of the method DistFromOrig from class Point1 (at this step the method can be applied only for arguments from the Point class - the method is inherited) After four steps Point1 has inherited the complete definition of CloserToOrig (it can be applied for arguments from Point or Point1)

We will transcribe the denotational definition given above in Standard ML

```
val no_methods  Methods = fn m => no_def,

val H0   Hierarchy =
      fn P => (P, "Object", [], [], no_methods),

val psi0 = (fn obj => (simple (intval 0),"Object"),
        fn P => null env),

val Point_methods   Methods =
 fn "m1"=>normal("m1",[],[],literal(intconst 10,10))=>no_def,
 fn "m2"=>normal("m2",[],[], call(self,"m1",inconst 15,15))

   val Point_class   ClassDef =
   ("Point", "Object", [], [], Point_methods)

   val Point1_methods   Methods =
 fn "m1"=>normal("m1",[],[],literal(intconst 20 20))=>no_def,

   val Point1_class   ClassDef =
   ("Point1", "Point", [], [], Point1_methods),

   val H   Hierarchy = H0 mod ("Point" --> Point_class)
                  mod ("Point1" --> Point1_class),

   val prog   Prg = (call(new "Point", "m2", []), H),
```

pp prog psi0,

val prog . Prg = (call(new "Point1", "m2", []), H),

pp prog psi0,

This example illustrates how inheritance works In ML syntax **fn x** represents a lambda abstraction If this program is executed and prog1 is evaluated then it returns (10,10) because m2 representing the method CloserToOrig compare the points(10,10) and (15,15) by the euclidian distance from origin The evaluation of prog2 returns (15,15) because 20+20 > 15+15 (the two points are compare by the distance defined in Point1)

Cook's specification model

Cook's definition [2] is based on three essential aspects related to the inheritance mechanism

- the addition of new methods or replacement of the inherited methods,
- the **self** reference must be redirectionated to access the modified methods,
- the **super** reference must be redirectionated to access the original methods

We will describe this definition using the same example The modifications are expressed as a record, **Point** $\oplus$ **Point1** The new methods from class Point1 are combined with the original methods from the parent class Point, such that the method defined in Point1, in this case **DistFromOrig**, substitutes the corresponding method in class Point

The variable **self** is used to refer to the Point1 version of DistFromOrig and **super** can be used to refer to the Point version of the same method So, the modifications can be expressed as a two arguments function, self and super, and returning the record described

above These functions are called wrappers.

Also the self-reference must be changed in the inherited methods These methods are contained in a function named generator The result is a new class definition, namely a new generator This mechanism is provided by [illegible]

The generator associated with Point is .

$$\begin{aligned}&\text{GenPoint}(x,y) = \lambda\,\text{self}\\&\quad\{\ \text{DistFromOrig} \mapsto\\&\quad\ \text{sqrt}(\text{self}\,x^2 + \text{self}\,y^2),\\&\ \text{CloserToOrig} \mapsto\\&\quad\ \lambda p\ (\text{self DistFromOrig} < p\ \text{DistFromOrig})\}\end{aligned}$$

The wrapper associated with Point1 is

$$\begin{aligned}&\text{Point1Wrapper} = \lambda x,y\ \lambda\text{self}\\&\{\ \text{DistFromOrig} \mapsto\\&\quad(\text{self}\,x + \text{self}\,y)\}\end{aligned}$$

The wrapper application will be

$$\begin{aligned}&\text{Point1Wrapper} \triangleright \text{GenPoint}(x,y) =\\&\lambda x,y\ \lambda\text{self}\\&\{\ \text{DistFromOrig} \mapsto\\&\quad(\text{self}\,x + \text{self}\,y)\\&\ \text{CloserToOrig} \mapsto\\&\quad\lambda p\ (\text{self DistFromOrig} < p\ \text{DistFromOrig})\}\end{aligned}$$

After presenting these two models of specification we shall make some comments The greatest advantage of the Kamin's model is the simple treatment of inheritance The related papers appeared before seems to have some disadvantages Kamin resolved them using fixed points to model inheritance He had defined the semantic maps we have talked a little earlier Indeed, for our example it is a nice specification way But what happens when we have a larger hierarchy? The specification will be sometimes not too easy to be followed. On the other hand we haven't used yet the definition of the E map, which is far more complicated.

The model has it's advantage the specification is concentrated on inheritance and it's mechanism is treated very simple and so it's easy to understand Also, Kamin has described all the mechanisms appeared in an object oriented program the meaning of the hierarchy, the inheritance process, the message passing, the methods evaluations, the evaluation of the primitive methods which provide access to low-level operations

What about Cook's model? This model seems easier to understand maybe because it is provided with an intuitive explanation of inheritance as a mechanism for incremental programming The whole specification is based on this motivation Also, Cook proved the correctness of his model demonstrating that it is equivalent with an operational semantics of inheritance based upon a method-lookup algorithm This way of specifying the inheritance shows that this is not only an object oriented features but a general mechanism that can be applied to any form of recursive definition Although Kamin's model is closely related, he described inheritance as a global operation on programs, which blurs scope issues and inheritance Here is the most important difference between the two models

Kamin's model versus Cook's model

Every specification has to respect some well-known criteria We will discuss how these specifications respect them

Formalization verifies if the specification behaves conforming with the implementation

Kamin's model can be transcribed in an executable version in Standard ML so this criterium is easy to verify We must also notice that the language had suffered some modifications and omissions But Kamin's goal was to specify the mechanism of inheritance

and the missing details are not essential related with this concept

Cook proved that his model is equivalent with an operational semantics and it's obvious that it respects this criterium

Constructability A specification must be easy to construct even if the notation used is formal

The omitted details make Kamin's specification easier to build, but even so if the hierarchy is thick then the construction of the C and D maps seems to be hard to follow

Comprehensibility The specification must be easy to understand The specification given by Kamin seems difficult to understand when we have to deal with the maps E and *E*

Minimality All the non-essential details had been omitted (we have already present the omissions and the modifications of the language, because they are not direct related with the inheritance process)

Applicability From the applicability point of view Cook's model seems to be more interesting since his definition of inheritance, although it was developed first for object oriented languages, shows that, in fact, inheritance is a general mechanism that can be applied to any form of recursive definition

The major problem of object oriented languages is that they lack a solid formal fundamentation There have been some attemps in specifying object oriented features in operational, axiomatic, denotational and algebraic semantics We have focus our attention on denotational semantics because it provides a good mathematical instrument for specification based on lambda-calculus and, on the other hand, an instrument which is not such complicated and hard to understand as the algebraic theory used in algebraic specification techniques This comparative presentation of these two model tries to be a study for choosing

the most suitable formal specification

REFERENCES

1 L Cardelli, P Wegner, On Understanding Types, Data Abstraction and Polymorphism, Computing Surveys, vol 17, no 4, Dec 1985, pg.471-522

2 W Cook, J Palsberg, A Denotational Semantics of Inheritance and its Correctness, OOPSLA'89 Proceedings, 1989, pg 433-444

3 W Cook, W L.Hill, P S Canning, Inheritance is not Subtyping, Proceedings of POPL'90, ACM Press 1990

4 A Goldberg, D Robson, Smalltalk-80 The Language and Its Implementation, Addison-Welsey, Reading, MA, 1983

5 S Kamin, Inheritance in Smalltalk-80 A Denotational Definition, Proceedings of the 15th Annual ACM SIGACT - SIGPLAN Symposium on Principles of Programming Languages, San Diego, Jan 1988, pg 80-87

6 Soo Dong Kim, Formal Specification in Object-Oriented Software Development, Ph D Thesis, The University of Iowa, 1991