Developing an efficient and reliable software system requires the adoption of some strategy that helps software developers to communicate without ambiguity. A solution is the use of formal methods for specification and design, together with a formal semantic definition of paradigm’s main concepts. The main contribution of this paper is to define mathematical models of the most important concepts in object-oriented systems: the object, the inheritance and the concurrency. This is achieved by distinguishing three conceptual levels at the system construction stage: i) the object specification level, ii) the implementation level and iii) the application programs level. Separating object’s semantics from client program’s semantics allows the formal observation of three different types of concurrency: intra internal concurrency (inside a method), internal concurrency (among methods which are invoked by the same object) and external concurrency (among methods which are invoked by different objects); and it also permits to formally observe the inheritance mechanism as a static mechanism for building type hierarchies. Also we analyze the relation between the algebraic models and the operational models of an object oriented specification.

1 Introduction

We think formal methods for specification and design help on the development of reliable software, and give a precise description of concepts independently from any implementation, offering a basis for testing correctness. Several attempts have been made to give rigorous mathematical foundations for object oriented systems using algebraic theories developed for abstract data types [Bre91, Bre93, EGS93, ES89, GTW78, GM86, Lu93, MC94]. Particularly, algebraic models for inheritance were studied [CO90, GM92] based on the order-sorted algebra notions. Another works have used others formalisms; in [BM92, CHC90, WZ88] is defined a formal semantics of inheritance, which is applicable to the analysis of several object oriented programming languages. In [Bru91, CP89, Ame90] the denotational semantic formalism is used for explaining the objects’ meaning. Furthermore, some researches have attempted to define formal semantics for concurrent object oriented languages [MW91, Mes93].

The main contribution of this paper is to define mathematical models of the most important concepts in object-oriented systems: the object, the inheritance and the concurrency. This is achieved by distinguishing three conceptual levels at the system construction stage: i) the object specification level, ii) the implementation level and iii) the application programs level.

The first level consists in the specification of objects types from the universe of discourse using some formal method. At the second level we deal with the selection of appropriate data structures for every object type defined in the previous level. We also write sound feature and behavior algorithms, i.e. satisfying the specifications for object types.

The remaining level corresponds to the construction of the applications programs, using objects of types previously defined. In an application we can find newborn objects, dropped-off objects and object displaying their behavior. The design of applications is independent from object implementations. Abstract specifications designed in the first level completely define the objects being modeled. Thus, application programs can be built without taking care of implementations.

This separation in levels is worth valuable because it is widely accepted that separating specifications from implementation is a good design mechanism [GTW78] [Por92], as it provides understanding, reusing and verification. What is more, separating objects semantics from client program semantics allows the formal observation of three different types of concurrency: intra internal concurrency (inside a method), internal concurrency (among methods which are invoked by the same object) and external concurrency (among methods which are invoked by different objects); and it also permits to formally observe the inheritance mechanism as a static mechanism for building type hierarchies. From the behavioral point of view, is_a relationships exist at the object type specification level but can be forgotten at the application programs level. Based on this model, we will formally describe the meaning of a type specification giving a clear interpretation of the meaning of inheritance and the meaning of methods.

The rest of the paper is organized as follows: in section 2 we survey the main aspects of our reference formal object
model.; in Section 3 we formally describe the three conceptual steps at a object based system. Finally, our conclusions are exposed in section 4.

2 A Formal Object Model
We assume the reader is familiar not only with object-oriented concepts but also with notions belonging to the denotational semantics formalism [Gor79] [Sch86].

In this approach, the syntactic entities are interpreted within certain mathematical structures called semantic domains. In denotational semantics, domains are usually understood to be complete partially ordered sets (cpos.)

The semantic domain where objects become significant is the set of records in a similar way as Cardelli proposes in [Car84] and [CW85]. More precisely, we model an object as a pair <v,B> where i) v is the object’s value, that is, a private internal state (which cannot be observed from outside); and ii) B is the set of all methods the object can perform (own and inherited). We consider the set of objects’ methods is partitioned in three subgroups: creators, observers and mutators. This discrimination of methods is deeply studied in [LW93]. An observer returns a result without altering internal states of objects. A mutator changes one or more objects states. Thus, object’s behavior is determined by observer and mutator operations (the domain of B is the object’s interface, that is, the set of messages the objects can perform). Creator operations do not belong to concrete objects, they are treated as type (class) methods. Each object type has got only one creator; the object_oriented languages usually denote it with the parameterized statement new(type_name). Example is:

Stack = < Stack_State, Stack_Behavior >
Domain(Stack_Behavior) = { top, pop, push } where: top is observer, pop and push are mutators, and new(Stack) is the creator.

Figure 1 shows the definition of semantic domains for objects.

- MESSAGE = String, primitive domain for procedure names.
- ID, primitive domain for object identifiers.
- VALUE, primitive domain for values.
- OBSERVER = [STORE → VALUE], domain for observable qualities of objects.
- MUTATOR = [STORE → STORE], domain for objects’ behaviors.
- METHOD = OBSERVER + MUTATOR, domain for methods.
- BEHAVIOR = [MESSAGE → METHOD], domain for objects’ behavior.
- OBJECT = (VALUE x BEHAVIOR), domain for objects.
- STORE = [ID → (OBJECT + {unused})], domain for object community.

Fig. 1 - Semantic domains for objects

The definition of a formal object model presented above covers the main aspects of any object model: internal state, information hiding and object community. The reader should notice the concept of inheritance is missing on purpose; inheritance is a mechanism for building class hierarchies. An object is a sound model (i.e. an instance) for an abstract definition (i.e. a type). Thus, objects are independent from their specification’s construction processes.

Store is the domain for object community. Intuitively, a store holds persistent objects.

Mutator is the domain for methods which can alter internal state of objects; when the evaluation of a mutator is performed in a specific object, the mutator may modify this object and it may modify another objects too by raising the evaluation of other method (through a message call).

Observer is the domain for methods which return a result without altering internal state of objects, that means, they do not modify the store.

3 Semantics for the Design Steps
3.1 The Object Specification Level
It is not easy to formally determine the meaning of a type specification, specially when the concept of inheritance is involved. The semantics of type specifications is subtle. In particular, the combination of inheritance, subtyping overwriting and dynamic binding often leads to unexpected results. The principal difficulty follows from the use of self with the overwriting of inherited methods and dynamic binding. When self occurs in a method body which has been invoked on an object, self refers to that object. When using dynamic binding, if a method is inherited in some subtype, all method names referred to in that method body are resolved in the context of the subtype (not in the type where the method was originally defined.) This mechanism may carry out undesirable effects.

Another dangerous feature in object-orientation is the fact that objects can display their behavior parallely. This concurrency may occur:
- among methods which are invoked by different objects (external concurrency).
- among methods which are invoked by the same object (internal concurrency).
- inside an individual method (intra_internal concurrency).

To illustrate:
We analyze the problems that the concurrency may yield:
- in the external concurrency there is not problem because we are working over different objects.
- in the internal concurrency may occur same interference because there are two methods trying to accede an object at the same time. Observers may accede together, but mutators must have exclusive access.

We will show how this explosive combination of object-oriented programming features can be carefully handled if we give semantics to type specifications. In the following sections, we present two specification alternatives: a more abstract one (algebraic) and a more concrete one (operational).

### 3.1.1 Algebraic Specification

**Syntax:**
We adopt the classical approach for the algebraic specification of abstract types [EM85]. We illustrate its use with the next example:

<table>
<thead>
<tr>
<th>Type</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>creators:</td>
</tr>
<tr>
<td></td>
<td>newSquare: \rightarrow Square</td>
</tr>
<tr>
<td></td>
<td>mutators:</td>
</tr>
<tr>
<td></td>
<td>enlarge: Square, Integer \rightarrow Square</td>
</tr>
<tr>
<td></td>
<td>observers:</td>
</tr>
<tr>
<td></td>
<td>length: Square \rightarrow Integer</td>
</tr>
<tr>
<td></td>
<td>area: Square \rightarrow Integer</td>
</tr>
<tr>
<td>Axioms</td>
<td>A_1: length(newSquare) = 0</td>
</tr>
<tr>
<td></td>
<td>A_2: length(enlarge(S,i)) = length(S) + i</td>
</tr>
<tr>
<td></td>
<td>A_3: area(S) = length(S) * length(S)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>DoubleSquare inherits Square</th>
</tr>
</thead>
</table>

**Semantics:**
Several attempts have been made to give a rigorous mathematical foundation for object-based systems using algebraic theories developed for abstract data types.

In the loose algebraic approach [Wir86] a syntactic object (\Sigma,E) with \Sigma signature and E equations is associated to a semantic object C, which is the set of (classes of equivalencies of) \Sigma-algebra satisfying E. Formally speaking,

\[ \text{Models } ((\Sigma,E)) = \{ |A|/A \Sigma \text{-algebra, } E \models A \} \]

An algebra is a domain for abstract objects. The following algebra is a model for the Square specification:

\[ A = \langle \text{Integer}, 0, \_+\_, \text{id}_\_, (\_)^2 \rangle, \]

where:
- Square\text{A} = \text{Integer}
- Integer\text{A} = \text{Integer}
- newSquare\text{A} = 0 : \rightarrow \text{Integer}, \text{constant zero}
- enlarge\text{A} = \_+\_: Integer, Integer \rightarrow \text{Integer}, \text{sum}
- length\text{A} = \text{id}_\_: Integer \rightarrow \text{Integer}, \text{identity}
- area\text{A} = (\_)^2: \text{Integer} \rightarrow \text{Integer}, \text{square}

The following algebra is a model for the DoubleSquare specification:

\[ B = \langle \text{Integer}, 0, \_+\_, \_\times\_2, (\_)^2*4 \rangle, \]

where:
- Square\text{B} = \text{Integer}
- Integer\text{B} = \text{Integer}
- newSquare\text{B} = 0 : \rightarrow \text{Integer}, \text{constant zero}
- enlarge\text{B} = \_+\_: Integer, Integer \rightarrow \text{Integer}, \text{sum}
- length\text{B} = \_\times\_: Integer, Integer \rightarrow \text{Integer}, \text{square}
- area\text{B} = (\_)^2*4: \text{Integer} \rightarrow \text{Integer}, \text{square} * 4

In the presence of overwriting of inherited methods the relation between the models of the type and its supertype is subtle. It does not always happen that:
- models(K) \subseteq Models(parent(K)), or
- \forall A: (A \in Models(K) \rightarrow \exists B (B \in Models(parent(K) and A is sub-algebra of B)).

This conditions holds only if the inheritance is conservative, that is, when i) parent’s specifications are inherited without being modified; and ii) only new operations and/or consistent axioms are introduced. The reader may find more details in [WZ88].

An algebraic specification specifies functionality (what
A type specification is a generic (parameterized) definition of objects with similar characteristics. Such specification describes a pattern, i.e. a function building a new object. A type specification is a recursive definition, bodies of methods can contain self occurrences. We remark that the recursion is present in the methods, not in the values.

The new(T) statement denotes the creation of the youngest additional object of type T. The meaning of such object is found by taking a fixed point of T [CP89] [Br91] [BM92]. Thus, object’s methods are interpreted in an environment in which self denotes the entire object. The value of a new(T) object is the constant term corresponding to the constructor of some algebra which is model for T, that is: value(new(T)) = c, where c is a constructor constant in A and A ∈ Models(T).

We now describe a function conforming the denotational semantic domain; using a variant of MacCarlthy’s formalism, a functional language based on the Lambda notation. The function is called Sem, and associates a type with the record it denotes, that is, the object it generates.

\[
\text{Sem} :: \text{TYPE} \rightarrow \text{OBJECT}
\]

where:

\[
\text{value and behavior} \text{ are the first and the second projection of the OBJECT <v, B> pair, respectively.}
\]

Behavior is a semantic entity, while methods are syntactic entities. A behavior is a function from message keys to functions or to ⊥ (the undefined element), while type methods are functions from key messages to expressions or to ⊥. A behavior also differs from type methods in the fact that the former contains a function for any message the type deals with; while the later associate an expression only to local messages of the type.

A type pattern is the functional associated to its recursive definition. A pattern is a function mapping behavior into behavior. The behavior for a particular instance of that type is obtained as the minimum fix point of its pattern.

Notation: function redefinition

\[
\{ g(0), \text{if } f(0) = \perp
\}
\]

In the context of a type hierarchy supporting single inheritance we define:

- parent(T): returns the immediate ancestor of type T.
- methods(T): returns the expressions of local methods belonging to type T.
- root(T): returns tt if it happens type T to be the root type of the type hierarchy.
The equation above states: if \( T \) is the root type, its pattern is completely determined by its own (proper) methods. If it happens \( T \) to inherit from another type, its pattern construction becomes more complicated.

We call the `parent_object` to the object obtained by the application of the `parent(T)` pattern to `self`. The function `methods(T)` only provides the (proper) methods of type \( T \) (that is, the behavioral increase added in \( T \)'s specification); these ones may contain occurrences of the special variables `self` and `super`.

The parent-object will play the `super` role in such context. This way we obtain a `son_object` which has the behavior as defined in its type, but having resolved its super occurrences.

Finally, the `son_object`, attached to the `parent_object` through function redefinition, determines an object bringing together proper and inherited behavior.

**Semantics for the expressions **`exp`**>**

We define the semantic interpretation function \( E \), which maps expressions to functions:

\[
E :: \text{EXP} \to \text{STORE} \to (\text{VALUE} + \{\text{error}\})
\]

The denotational semantics for expressions can be understood easily:

\[
E[\text{nat}] = \lambda \sigma.\text{nat}
\]
\[
E[\text{bool}] = \lambda \sigma.\text{bool}
\]
\[
E[\text{string}] = \lambda \sigma.\text{string}
\]
\[
E[\text{X}] = \lambda \sigma.\sigma(X)
\]
\[
E[[e_1, e_2, \ldots, e_n]] = \lambda \sigma. f(E[e_1]_{\sigma}, \ldots, E[e_n]_{\sigma})
\]
\[
E[\text{self}] = \lambda \sigma. \sigma(\text{self})
\]
\[
E[\text{super}] = \lambda \sigma. \sigma(\text{super})
\]

For the evaluation to be correct, `self` and `super` must be bound to the corresponding objects in store \( \sigma \).

\[E[\text{state}] = \lambda \sigma.\text{value}(\sigma(\text{self}))\]

Accesses the internal state of the object bound to `self` in store \( \sigma \); this shows that only the object can access to its own internal structure.

\[E[\text{new}(T)] = \lambda \sigma. \text{Sem}(T)\]

A type specification (syntax) has a well-determined meaning (semantics): is an object which value is the constant term of the algebra and which behavior is the fixpoint of its pattern. A type is a pattern for generating object. The meaning of `new(T)` is obtained by taking a new copy of such pattern.

\[E[e_i,o] = \lambda \sigma. \text{cases } E[e_i]_{\sigma} \text{ of}\]
\[\text{error: error}\]
\[v: \text{error}\]
\[<v,B>: (B(o))_{\sigma}\]

If the evaluation of \( e_i \) reports an error or a value, the evaluation of \( e_i,o \) reports an error. Otherwise, if the result of evaluating \( e_i \) is an object (of the form \( <v,B> \)) with \( v \) its value and \( B \) its behavior, we obtain the method attached to the message \( o \), that is \( B(o) \). This method is a function from a store into values; it is applied on store \( \sigma \) returning a value or an error.

**Semantics for the commands **`com`**>**

We define the semantic interpretation function \( C \), which maps commands into functions:

\[
C :: \text{COM} \to \text{STORE} \to (\text{STORE} + \{\text{error}\})
\]

\[C[e_i := e_j] = \lambda \sigma. \text{Cases } E[e_i]_{\sigma} \text{ of}\]
\[\text{error: error}\]
\[X: \text{Cases } E[e_j]_{\sigma} \text{ of}\]
\[\text{error: error}\]
\[\text{value: } \sigma[X/value]\]

\[C[e_i,m] = \lambda \sigma. \text{cases } E[e_i]_{\sigma} \text{ of}\]
\[\text{error: error}\]
\[v: \text{error}\]
\[<v,B>: (B(m))_{\sigma}\]

If the evaluation of \( e_i \) reports an error or a value, the evaluation of \( e_i,m \) reports an error. Otherwise, if the result of evaluating \( e_i \) is an object (of the form \( <v,B> \)) with \( v \) its value and \( B \) its behavior, we obtain the method attached to the message \( m \), that is \( B(m) \). This method is a function from stores into stores; it is applied on store \( \sigma \) returning a new store which reflects the execution of method \( m \) using the old store. In other words, the meaning of message sending is to "carry out" the method (function) bound to label \( m \) inside object (record) \( o \). The method execution may modify the store (the object itself or other objects in the store).

\[C[\text{input}(X)] = \lambda \sigma. \text{Cases } \sigma(\text{IN}) \text{ of}\]
The input command takes the info in the special position IN of the store. If the position is empty, input denotes an error.

\[ C[\text{output}(X)] = \lambda \sigma. \text{Cases } \sigma(X) \text{ of } \]
\[ \text{unbound} : \text{error} \]
\[ \text{data: } \sigma[\text{OUT}/\sigma(\text{OUT}) + \text{data}] \]

The output command adds the content of the X variable in the special position OUT of the store. If X is unbound, then output denotes an error.

\[ C[c_{1};c_{2}] = \lambda \sigma. \text{Cases } C[c_{1}\sigma] \text{ of } \]
\[ \text{error} : \text{error} \]
\[ \sigma' : E[c_{2}\sigma] \]

For evaluating a sequence of commands, we first evaluate the former one (say \(c_{1}\)); then the latter (say \(c_{2}\)) using the updated (if so) store \(\sigma'\).

\[ C[c_{1};c_{2}] = \lambda \sigma. \text{Cases } C[c_{1}\sigma,c_{2}\sigma] \text{ of } \]
\[ \text{error, } \sigma' : \text{error} \]
\[ < \sigma', \text{error} > : \text{error} \]
\[ < \text{error,error}, \sigma ' > : \text{error} \]
\[ < \sigma_{1}, \sigma_{2} > : \text{compose}(\sigma, \sigma_{1}, \sigma_{2}) \]

where:
\[ \text{compose} :: \]
\[ \text{STORExSTORExSTORE} \rightarrow \text{STORE} \]

Both commands \(c_{1}\) and \(c_{2}\) are evaluated in parallel. By convention, they work on different and disjoin store sections. Therefore, they do not interfere on each other. Function compose returns the new updated store.

3.1.3 Relationship Between Both Semantics
Given a type specification T, let A be an algebra, \(A \in \text{Models}(T)\): A= \(\langle A, m_{1}, \ldots, m_{k} \rangle\).

Let \(A^{0} \subseteq \text{OBJECT}\). It exists a function, say \text{Obj} (one to one and onto), with Val its inverse function. Obj associates values in A with their corresponding object values.

Definition:

\[ \text{Obj} :: A \rightarrow A^{0} \]
\[ \text{Obj}(v) = < v, < m_{k}^{\text{obj}}, m_{1}^{\text{obj}}, \ldots, m_{k}^{\text{obj}} > > \]
\[ \text{Val}(< v, B >) = v \]

where:

functions \(m_{n}^{\text{obj}} : \text{OBJECT} \rightarrow \text{OBJECT}, \)
satisfy: \(m_{n}^{\text{obj}}(o) = \text{Obj}(m_{n}(\text{Val}(o)))\)

The algebra is the domain for abstract objects. Function \text{Obj} maps values from that algebra into concrete objects. \text{Obj} is compatible with the functional behavior of objects.

3.2 The Application Programs Level
In this section we define the semantics of object-based programs, within a parallel framework, as store transformations. Classically, a program is a function mapping inputs into outputs. A program handling objects has the same meaning, although internally its behavior differs from the one belonging to a classical program.

To be simpler, let us consider a reduced form for object programs that can be easily extended: a object-based program like a partially ordered set of object creations and message sendings. This means some parts of the program may be executed simultaneously, while others are sequential.

The programs define the concurrency among messages, that means both the external and internal concurrency need to be expressed in the client programs, the intra Internal concurrency remains transparent.

Syntax:

\(<\text{Prog}> :: \text{Program com}\)

where \text{com} is the syntactic category defined in section 3.1.2, provided an exception: the special variables \text{self}, \text{super} and \text{state} do not occur in program expressions.

Semantics:

\(P : \text{Prog} \rightarrow \text{value}^* \rightarrow (\text{value}^* + \{\text{error}\})\)

\(P[\text{Program } c] =
\begin{align*}
\lambda \text{inputs}. \text{cases}(C[c]_{\sigma[0]}[\text{IN/inputs}]) & \\
\text{error: error} & \\
\sigma & : \sigma(\text{OUT})
\end{align*}\)

The meaning of a program is a function that first receives an input, it then executes the c command over an initial store \(\sigma[0][\text{IN/inputs}]\), (\(\sigma[0]\) denotes the empty store and \(\sigma[0][\text{IN/inputs}]\) denotes the store just containing the input); and finally returns some information in the OUT store position.

If it happens to the command evaluation to denote an error, the program denotes an error.
3.3 The Implementation Level

Software reuse is an old idea. Nowadays, with the rapidly increasing demand for complex software systems, reuse is a central point in software development. Reuse depends crucially on i) the construction of abstract software modules (using parameterization and modularization), and ii) the identification of reusable software. Both points require the use of formal specification.

If at the specification level objects have been defined formally, then we can build highly reusable correct implementations at the implementation level. We want to remark that also a single specification may have multiple implementations, conforming a tree [WHS89] where each node is an implementation of its parent. Nodes in the tree represent different levels of abstraction beginning with the root (which is the most abstract level) and ending in the leaves, which are the concrete implementations. The fact of dealing with several implementation alternatives for a type, allows the selection of the best choice for a particular situation, increasing in this sense efficiency and reusability.

4 Conclusions

We present a formal model which explains the meaning of an object type specification (which may be built using the inheritance mechanism).

Separating object’s semantics from client program semantics allows the formal observation of three different types of concurrency: intra_internal concurrency (inside a single method), internal concurrency (among methods of the same object) and external concurrency (among methods of different objects). In an object-oriented environment both the external and internal concurrency need not to be reflected in the specification of object’s functionality, but in the client programs specification. An object specification may reflect intra_internal concurrency.

Algebraic specifications, although allow parallel models do not permit the explicit expression of concurrency. We believe this is a good feature as a specification describe what is done rather the way things are done. If even necessary, formalisms for expressing concurrency are always applicable.

The relation between the algebraic models and the operational models of an object oriented specification is another important topic, where the constructor methods play an eminent roll.

References


[GTW78] J.Goguen, J.Thatcher and E.Wagner. An Initial Algebra approach to the specification,


