States as Specifications

Paulo Borba*
Departamento de Informática
Universidade Federal de Pernambuco

Abstract

We present a general approach for formally modelling states of object-oriented programs using OBJ [10] specifications and associated order-sorted theory presentations [9]. This formal model can then be used to define the structural operational semantics [14] of object-oriented languages. Our approach has the advantage of using the power of the theory of abstract data types for defining operations on states. As states are represented by an abstract structure, the operational semantics can be defined in a simpler way, facilitating reasoning about programs and derivation of implementations.

Resumo

Neste trabalho apresentamos uma abordagem genérica para modelar formalmente estados de programas orientados a objetos. Em particular, nós usamos especificações em OBJ [10] e teorias algébricas [9] como ferramentas básicas. O modelo formal apresentado pode ser diretamente utilizado para definir a semântica operacional estruturada [14] de várias linguagens orientadas a objetos. A grande vantagem da abordagem apresentada é usar a teoria de tipos abstratos de dados para definir estados e as operações sobre os mesmos. Como estados são representados de forma bem abstrata, a semântica operacional de uma linguagem pode ser mais facilmente definida, facilitando o raciocínio sobre programas e a derivação de implementações.

1 Introduction

A lot of effort has been done in order to define formal semantics for object oriented-languages. Most of the work in this area defines semantics in terms of mathematical models based on set theory [6], metric spaces [1], category and sheaf theory [12, 3], and hidden order sorted algebra [7]. Exceptions are [5], which defines an assertional style proof system, and [17] and [11], which give the semantics in terms of a process algebra based on operational semantics [13].

Because of the lack of a fully abstract mathematical model for interleaving, and the intrinsic details of the semantics of object identification and dynamic object creation and deletion, we think that the framework of operational semantics is quite adequate for specifying the semantics of a concurrent object-oriented language. In fact, as demonstrated in [2], the operational semantics of a language can be easily and concisely defined, being still possible to reason about it in a pragmatical way, and use it or derive language implementations.

However, in order to obtain a clear operational semantics definition it is essential to have a clear model for program states. In fact, the standard use of mappings (from variable names to values) for modelling states is not adequate for representing in a simple and natural way object-oriented concepts such as subtyping and inheritance.

Aiming to solve this problem, in this paper we present a general approach for formally modelling states of object-oriented programs using OBJ [10] specifications and associated order-sorted theory presentations [9]. We adapt and generalize some ideas from [8] and [2], where the main ideas about

*Supported in part by CNPq, Grant 301021/95-3. Address: Caixa Postal 7851, Recife, PE, Brazil, CEP 50732-970. WWW: `http://www.di.ufpe.br/~phmb`. Electronic mail: `phmb@di.ufpe.br`.

our approach have been extensively used to define the semantics of FOOPS [15, 2, 8]. In order to enforce the generality of our approach, we illustrate it with states of programs written in two rather contrasting object-oriented languages: Java [4] and FOOPS.

In particular, our approach has the advantage of using the power of the theory of abstract data types for defining operations on states. As states are represented by an abstract structure, the operational semantics can be defined in a simpler way, facilitating reasoning about programs and derivation of implementations. In fact, lots of complications can be avoided and a concise semantic definition can usually be obtained.

This paper is structured in the following way. First we give an overview of order sorted algebra [9] and OBJ [10], which are the basic tools that we will use to model states. Following that we informally present the main ideas of our approach for using OBJ specifications to model states. Those ideas are then formalized using order sorted theory presentations. Lastly, we generally analyse and compare our approach to others.

2 Order Sorted Algebra and OBJ

Order sorted algebra (OSA) is a mathematical theory supporting multiple inheritance, overloading, polymorphism, error handling, partial functions, and multiple representation in an algebraic framework [9]. OBJ [10] is a first order, purely functional language providing an algebraic style for the specification, rapid prototyping, and implementation of abstract data types. OBJ also supports the declaration of "loose" theory specifications as well as "tight" executable specifications. In particular, OSA is the theory underlying OBJ, and we will use both OBJ specifications and OSA theory presentations for modelling states of object-oriented systems.

2.1 Order Sorted Algebra

Given a "sort set" S, an S-**sorted set** A is just a family of sets A_s for each "sort" $s \in S$; we write $\{A_s \mid s \in S\}$. For a fixed S, operations on S-sorted sets are defined component-wise. For example, given S-sorted sets A and B, $A \cup B$ is defined as $(A \cup B)_s = A_s \cup B_s$, for each $s \in S$. We write $|A|$ for the distributed union of all sets in A. Also, $e \in A$ is an abbreviation for $e \in |A|$.

In order-sorted algebra, S is a **partially ordered set** (**poset**). We will often use the extension of the ordering on S to strings of equal length in S^* by $s_1 \ldots s_n \leq s'_1 \ldots s'_n$ iff $s_i \leq s'_i$ for $1 \leq i \leq n$.

A **many-sorted signature** is a pair (S, Σ), where S is called the **sort set** and Σ is an $S^* \times S$-sorted family $\{\Sigma_{w,s} \mid w \in S^* \text{ and } s \in S\}$. Elements of (the sets in) Σ are called operation symbols. An **order-sorted signature** is a triple $(S, \leq, \Sigma)$ such that (S, Σ) is a many-sorted signature and $(S, \leq)$ is a poset. When the poset (S,$\leq$) is clear, we write Σ for $(S, \leq, \Sigma)$.

Given a signature $(S, \leq, X)$, we say that X is a **ground signature** iff it is formed only by distinct constant symbols. For a signature Σ, the notation $\Sigma(X)$ abbreviates $\Sigma \cup X$, if X is a ground signature disjoint from Σ. In this case, we may call X a Σ-**variable family**.

We now turn to the models that provide actual functions to interpret the operation symbols in a signature. Let (S, Σ) be a many-sorted signature. Then an (S, Σ)-**algebra** A is a family $\{A_s \mid s \in S\}$ of sets called the **carriers** of A, together with a function $A_\sigma : A_w \to A_s$ for each σ in $\Sigma_{w,s}$ where $A_w = A_{s1} \times \ldots \times A_{sn}$ when $w = s1 \ldots sn$ and where A_w is a one point set when $w = \lambda$. Let $(S, \leq, \Sigma)$ be an order-sorted signature. An $(S, \leq, \Sigma)$-**algebra** is a many sorted (S, Σ)-algebra A such that $s \leq s'$ in S implies $A_s \subseteq A_{s'}$. When $(S, \leq)$ is clear, $(S, \leq, \Sigma)$-algebras may be called **order-sorted** Σ-**algebras**.

The algebra whose carrier sets are formed by the terms that we can construct from a given signature Σ is called the **term algebra** and it's denoted by $\mathcal{T}_\Sigma$. Similarly, $\mathcal{T}_\Sigma(X)$ is the algebra of Σ-terms with variables in the Σ-variable family X.

A term may have many different sorts. In particular, if $t \in \mathcal{T}_\Sigma$ has sort s then it also has sort s' for any $s' \geq s$. A condition on signatures called **regularity** guarantees that every term has a well defined least sort (see [9]). For a term t, this is denoted by $LS(t)$.

We also introduce the term algebra P_Σ of fully parsed terms (i.e., terms together with their type information) associated to Σ. We let P_Σ be the least S-sorted set such that $\sigma \in \Sigma_{w,s}$ and $ti \in P_{\Sigma,si}$ for, $i = 1, \ldots, n$, where $w = s1 \ldots sn$, and $s \leq s'$ imply $\sigma.ws\underline{(}t1, \ldots, tn\underline{)} \in P_{\Sigma,s'}$. The definitions introduced so far and the ones to come can be easily extended for parsed terms, but for simplicity we only consider unparsed terms.

Now, given a regular signature Σ, we can define a parsing function $\rho : \mathcal{T}_\Sigma \to \mathcal{P}_\Sigma$ which transforms an untyped term t into a fully typed term t' such that the sort of t' is the least sort of t. It's easy to extend ρ to equations, set of equations and variable families. Here we omit the details. Also, for simplicity, we let unparsed terms be used in places where parsed terms are expected, if the signature in question is regular. In those cases, we assume that an unparsed term t abbreviates $\rho(t)$. In the same way, an unparsed equation might be used when a parsed equation is expected.

For a regular order-sorted signature $(S, \leq, \Sigma)$, a Σ-**equation** is a triple $\langle X, t, t' \rangle$ where X is a Σ-variable family and t, t' are in $\mathcal{T}_{\Sigma(X)}$ with $LS(t)$ and $LS(t')$ in the same connected component of $(S, \leq)$[1]. We will use the notation $(\forall X)\ t = t'$. When the variable set X can be deduced from the context we allow it to be omitted[2]. Lastly, we say that an equation is **unquantified** if $X = \emptyset$. Order-sorted conditional equations are expressions of the form $(\forall X)\ t = t'\ \mathtt{if}\ C$, where the **condition** C is a finite set of unquantified Σ-equations involving only variables in X.

Given a set of equations Γ, there is a congruence $=_\Gamma$ relating two terms iff we can prove they are equal from the equations in Γ by equational deduction (see [9]). Furthermore, this congruence splits the term algebra into equivalence classes of terms modulo Γ. Hence, given a term t, $[t]_\Gamma$ denotes its equivalence class under Γ, and $[\![t]\!]_\Gamma$ denotes the representative of this class.

An **order-sorted presentation with a signature of non-monotonicities** is an ordered 5-tuple, $(S, \leq, \Sigma, \Omega, \Gamma)$, where $(S, \leq, \Sigma)$ is an order-sorted signature, Γ is a set of parsed Σ-equations and Ω is a signature of non-monotonicities; that is, $\Omega \subseteq \Sigma$ is used to indicate the operations which may have different interpretations when restricted to common subsorts.

For reasoning about this kind of presentation, we assume default equations relating monotonic operations having the same name and related ranks. This is necessary because parsed equations are used. The default equations are in the form: $\sigma.ws(\tilde{x}) = \sigma.w's'(\tilde{x})$, for any $\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ such that $w \leq w'$ and $\sigma \notin \Omega_{w,s}$, where X is a Σ-variable family, $x_i \in X$, for $i = 1 \,..\, k$, $\tilde{x}$ stands for $x_1, \ldots, x_k$, and $w = LS(x_1), \ldots, LS(x_k)$. We let $\Gamma^\star$ be the union of Γ with the default equations.

So, for a presentation $P = (S, \leq, \Sigma, \Omega, \Gamma)$, we let $t =_P t'$, $[t]_P$ and $[\![t]\!]_P$ respectively mean $t =_{\Gamma^\star} t'$, $[t]_{\Gamma^\star}$ and $[\![t]\!]_{\Gamma^\star}$. Lastly, given a signature $(S, \leq, \Sigma')$, we use $P \cup \Sigma'$ for the presentation $(S, \leq, \Sigma \cup \Sigma', \Omega, \Gamma)$.

2.2 OBJ Specifications

OBJ specifications are formally represented by OSA theory presentations. In fact, there is a direct correspondance between specifications and presentations. OBJ specifications (modules) can be used to define one or more abstract data types (ADTs), which are sets of data elements together with associated operations. For instance consider part of the following OBJ module[3] defining the natural numbers:

```
fmod NAT is
  sorts Zero NzNat Nat .
  subsort Zero < Nat .
  subsort NzNat < Nat .
  fn 0 : -> Zero .
  fn s : Nat -> NzNat .
  fn + : Nat Nat -> Nat .
  ⋮
```

where the keywords `sorts` and `fn` respectively introduce the name of the set of data elements, and the associated operation (function) symbols. Note how the declaration `subsort` is used to indicate that the elements of a sort are also elements of a supersort; so `Zero` is used to denote the set containing only the number 0, `NzNat` denotes the positive natural numbers and `Nat` denotes the natural numbers.

[1]Given a poset $(S, \leq)$, let $\cong$ denote the transitive and symmetric closure of $\leq$. Then $\cong$ is an equivalence relation whose equivalence classes are called the **connected components** of $(S, \leq)$.

[2]However, the reader should be aware that satisfaction of an equation depends crucially on its variable set.

[3]We are actually using a variant of the syntax of OBJ.

The meaning of the functions defined in an OBJ specification is given by axioms (equations). In a module defining code, equations are interpreted as left-to-right rewrite rules. For the example being discussed, the following equations are necessary:

```
   ⋮
   vars M N : Nat .
   ax 0 + M = M .
   ax s(N) + M = s(N + M) .
 endfmod
```

The keyword `var` introduces variables of a given sort, whereas `ax` precedes an axiom, and `endfmod` indicates the end of a functional module.

3 States as Specifications

Operationally, a system implemented in an object-oriented language consists of a database containing information about the current objects in the system. This information can be retrieved by the evaluation of attributes, and modified by the execution of methods or by the deletion and creation of objects. Modifying this information changes the *database state*.

In this section we informally present our approach for modelling (database) states of object-oriented systems. First we exemplify how those states can be represented by OBJ specifications. Later we illustrate how typical operations on states can be simulated with specifications.

We take the view that attributes correspond to general properties of objects and, therefore, can be classified as *stored* or *derived*. The value of a stored attribute is kept as part of the local state of an object. On the other hand, the value of a derived attribute is not stored by an object, but can be computed from the values of other attributes. Note that stored attributes are usually represented by instance variables, whereas derived attributes are usually represented by side-effect free operations ("functions" or methods) associated to objects.

In order to enforce the generality of our approach, we consider states of programs written in two rather contrasting object-oriented languages: FOOPS [15, 2, 8] and Java [4]. FOOPS is an extension of OBJ with concepts from object-oriented programming; so it supports constructs for defining both ADTs and classes of objects. FOOPS also inherits from OBJ its simple "declarative" style for programming using "equations"; in fact, it supports no use of variables as in imperative languages. Contrasting, Java is a radical evolution of C++ [16] being based on some fundamental C++ design decisions. In particular, it supports only some pre-defined ADTs and constructs for defining classes. Also, variables represent stored attributes and are explicitly used to store references to objects.

3.1 States

Based on ideas used to define the semantics of FOOPS (Functional Object-Oriented Programming System) [2, 8], we propose that states of object-oriented systems should be represented by OBJ specifications. In fact, states of an object-oriented program P can be represented by specifications formed by the following components:

- the definition of the data types[4] of P;
- sorts corresponding to the classes of P;
- functions corresponding to the *stored* attributes of P;
- constants denoting objects of the classes represented by the sorts of the constants; and
- equations establishing the values of *stored* attributes for objects in the database.

[4]Assuming that P was written in a language that supports both *data types* and *classes* of objects.

Moreover, the subsort relationships in these specifications reflect the subclass and subsort relationships specified by P.

In order to illustrate our approach of using specifications to model states, let us consider programs defining a class of nodes of a linked list of integer numbers. In FOOPS, such a class would be typically defined by a module such as the following:

```
omod FNODE is
  pr INT .
  class Node .
  at val : Node -> Int .
  at next : Node -> Node .
  ⋮
endomod
```

which defines a class `Node` and two associated attributes: `val` corresponding to the value associated to a node, and `next` corresponding to the node following any given node in a linked list. As illustrated above, attributes are defined as operations from an object identifier to a value that denotes a current property of the related object.

Note that the ADT of integers is defined by the module `INT`, which was written in OBJ[5] and is imported into `FNODE` to introduce the sort `Int` used to define `val`. Such a module could be defined in the same way as `NAT` was defined in Section 2.2.

In Java, a similar class of nodes could be defined as follows:

```
package JNODE;
class Node {
  int      val;
  Node    next;
  ⋮
}
```

where the attributes are represented by variables and `int` denotes Java's pre-defined ADT of integers, which are automatically imported into the package `JNODE`. We omit other details of `JNODE` and `FNODE` since, as discussed above, only data types definitions, class names and *stored* attributes are relevant for defining states of an object-oriented program.

A typical OBJ specification representing a database state of `JNODE` or `FNODE` looks like

```
fth STATE1 is
  ex INT .
  sort Node .
  fn val : Node -> Int .
  fn next : Node -> Node .
  fns n1 n2 : -> Node .
  ax val(n1) = 5 .
  ax next(n1) = n2 .
  ax val(n2) = 7 .
  ax next(n2) = n2 .
endfth
```

In particular, this specification denotes a state having two nodes (objects of class `Node`): `n2`, which stores the integer `7` and points to itself, and `n1`, which stores `5` and points to `n2`.

Note that besides introducing a sort corresponding to the class `Node` and two functions representing the stored attributes of `Node`, the specifications denoting states of `JNODE` and `FNODE` must include the OBJ specification of the data types used in those programs; in our case, this is obviously `INT` since it is explicitly imported into `FNODE` and corresponds to Java's `int` ADT.

[5] To be more precise, we should say that `INT` was written using the functional part of FOOPS. But that is just a syntactical variant of OBJ [15], so there is no problem to be less precise.

This indicates that OBJ specifications can only be used to model states of programs if we provide OBJ specifications for the data types (not classes of objects) of the language used to write the programs. Of course, this is not a problem since OBJ can actually specify any computable datatype. Also, note that it is not a particular requirement of our approach; in fact, any other formalism for modelling states would have to satisfy that.

Class inheritance can also be easily handled by our approach for modelling states. For instance, let us consider the following FOOPS module defining a subclass of `Node` corresponding to binary trees building nodes (i.e., nodes able to store two references to other nodes):

```
omod FNODE2 is
  ex FNODE .
  class Node2 .
  subclass Node2 < Node .
  at next : Node2 -> Node2 [redef] .
  at nextl : Node2 -> Node2 .
  ⋮
endomod
```

Note that `Node2` inherits from `Node` the attributes `val` and `next`, which is redefined for having a more specialized type. In addition to those attributes, `Node2` introduces the attribute `nextl`, another pointer to objects of `Node2`.

A state of `FNODE2` having objects of both `Node` and `Node2` typically looks like

```
fth STATE2 is
  ex INT .
  sorts Node Node2 .
  subsorts Node2 < Node .
  fn val : Node -> Int .
  fn next : Node -> Node .
  fn next : Node2 -> Node2 [redef] .
  fn nextl : Node2 -> Node2 .
  fns n1 : -> Node .
  fns n2 : -> Node2 .
  ax val(n1) = 5 .
  ax next(n1) = n2 .
  ax val(n2) = 7 .
  ax next(n2) = n2 .
  ax nextl(n2) = n2 .
endfth
```

where functions associated to a sort are also available to its subsorts, since elements of a sort are also elements of an associated supersort. Our approach can also represent states having undefined attributes; that is, attributes that have no associated value in the state. For instance, by simply removing the axiom `nextl(n2) = n2` from the specification above we obtain a specification which denotes a state of `FNODE2` where the attribute `nextl(n2)` is undefined.

Note that states of `FNODE2` are just natural, direct extensions of states of `FNODE`, obtained by simply adding subsorts declarations and functions respectively corresponding to the subclasses declarations and extra stored attributes introduced or redefined by `FNODE2`. Functions corresponding to redefined attributes are tagged with `[redef]` in order to indicate that they might be nonmonotonic; that is, they might have different interpretations from their associated supersort functions when restricted to common subsorts.

Contrasting to FOOPS, Java does not allow attributes to be redefined. In fact, an "equivalent" Java program to `FNODE2` could be defined in the following way:

```
package JNODE2;
class Node2
  extends Node {
```

```
    Node2    nextl, next;
      ⋮
}
```

However, that actually defines (instead of redefines!) another attribute named `next`, but having type `Node2`. So objects belonging to the class above will actually have four attributes: `val`, `nextl`, and two named `next` but of different types, where both are accessed in different ways and might be associated to completely different values.

In this way, a typical state of `JNODE2` can be represented by the following specification, where `null` is Java's "null reference" polymorphic value associated to each defined class:

```
fth STATE3 is
  ex INT .
  sorts Node Node2 .
  subsorts Node2 < Node .
  fn val : Node -> Int .
  fn next : Node -> Node .
  fn next : Node2 -> Node2 [redef] .
  fn nextl : Node2 -> Node2 .
  fns n1 : -> Node .
  fns n2 : -> Node2 .
  ax val(n1) = 5 .
  ax next(n1) = n2 .
  ax val(n2) = 7 .
  ax next.NodeNode(n2.Node2) = null.Node .
  ax next.Node2Node2(n2.Node2) = n2.Node2 .
  ax nextl(n2) = n2 .
endfth
```

Parsed equations (see Section 2.1) must be used in some cases to avoid ambiguity. This, together with the tag `[redef]`, allows both versions of `next` to be associated with different values without generating any inconsistency. On the other hand, if states were represented by specifications having unparsed equations only, it would be possible to prove that `n2.Node2` is equal to `null.Node` using the equations defining `next`. This is obviously not desirable; it would mean that our model for Java states would be equating values that are not assumed to be equal by Java.

It is interesting to note that `STATE2` is not one of the possible states of `JNODE2` since attributes in Java must be always defined and one of the versions of `next` for `n2` is not defined in `STATE2`.

3.2 Operations on States

Using our representation of states as specifications, typical operations on states can be easily defined. In fact, given a database state, *stored* attributes are evaluated simply by reducing (using term rewriting) the corresponding expression in the module representing the state. For example, considering `STATE1`, the evaluation of the FOOPS expressions

`val(n1) + val(n2)` and `next(n2)`,

respectively result in `12` and `n2`; this can be easily deduced by equational reasoning, from the equations in `STATE1`.

Similarly, the same results would be obtained by evaluating the following Java expression with `STATE1`:

`n1.val + n2.val` and `n2.next`.

In order to deduce that, we just need to translate the expressions into the notation used both by OBJ functions and by FOOPS attributes and proceed with equational reasoning. That is straightforward. For instance, the evaluation of

`n1.val + n2.val`

in STATE3 yields 12 since the values of val(n1) and val(n2) in that state are respectively 5 and 7.

When using ambiguous attribute names, a particular version of an attribute must be identified as in the following Java expression:

```
n2.next.Node
```

which can be translated to the parsed OBJ term next.NodeNode(n2.Node2). Therefore, the evaluation of this Java expression in STATE3 yields null, which the corresponding value of null.Node in Java.

Attribute evaluation just access the information stored in a state, whereas object creation and deletion actually change that information. In fact, adding a non initialized object n3 to STATE1 simply results in a state with one more constant of sort Node:

```
fth STATE4 is
  ex INT .
  sort Node .
  fn val : Node -> Int .
  fn next : Node -> Node .
  fns n1 n2 n3 : -> Node .
  ax val(n1) = 5 .
  ax next(n1) = n2 .
  ax val(n2) = 7 .
  ax next(n2) = n2 .
endfth
```

On the other hand, removing the object n2 from STATE1 yields the following state:

```
fth STATE5 is
  ex INT .
  sort Node .
  fn val : Node -> Int .
  fn next : Node -> Node .
  fns n1 : -> Node .
  ax val(n1) = 5 .
endfth
```

where the object, its related equations and all references to it were removed from the database. Therefore the attribute next(n1) is undefined in STATE5.

Method execution also changes the state of a system. For example, consider that the FOOPS method expression val n1 := 9 is "equivalent" to the Java assignment command n1.val = 9;; that is, it changes the value stored in the node n1 to 9. So the execution of either the expression or the command in STATE1 would change the database to the state represented by following specification:

```
fth STATE6 is
  ex INT .
  sort Node .
  fn val : Node -> Int .
  fn next : Node -> Node .
  fns n1 n2 : -> Node .
  ax val(n1) = 9 .
  ax next(n1) = n2 .
  ax val(n2) = 7 .
  ax next(n2) = n2 .
endfth
```

containing the same information as STATE1, except that the equation val(n1) = 9 is in place of the equation val(n1) = 5. That is, using our approach for modelling states, attribute updates are simply performed by replacing equations.

4 States as Presentations

Using OSA, in this section we formalize the ideas informally presented in the previous section. We define a formal mathematical model of states of object-oriented programs (specifications). This model can then be used to formally define the structural operational semantics [14] of object-oriented languages.

4.1 Signatures and Specifications

States are relative to programs; that is, for an object-oriented program P there is a particular class D_{P} of all database states associated to P. So, before formalizing our notion of state, we have to formalize the aspects of object-oriented programs that have a direct influence on the class of states associated to a program.

We consider that an object-oriented program (a FOOPS module or a Java package) defines a *signature* and a *specification*. A signature contains a sort (data types) and class hierarchy, and names (together with typing and redefinition information) of functions, methods, and attributes in the program. A specification is formed by a signature and some axioms (equations) that specify properties of the elements of the related signature.

As can be concluded from the examples in Section 3, in this paper we shall be interested in the following aspects of object-oriented signatures:

1. A "sort set" $U = S \cup C$, where S has sort names and C has class names. The sets S and C are disjoint because a sort and a class cannot have the same name.

2. A partial order $\leq$ on U, which establishes the sort and class hierarchy. Classes and sorts are not related: $u \leq t \Rightarrow u \in S \Leftrightarrow t \in S$, for any $t, u \in S \cup C$.

3. A $U^* \times U$-sorted family $\Sigma = F \cup A$, where F and A respectively contain names for functions and *stored* attributes. Functions are related to sorts: $F_{w,u} = \emptyset$ if $wu \notin S^+$; and stored attributes have only one parameter, which should be of a class type: $A_{w,u} = \emptyset$, if $w \notin C$.

4. A family $R \subseteq A$ formed by names of redefining (overriding) attributes.

Note that an object-oriented signature Σ having the components above can be seen as the order sorted signature $(U, \leq, \Sigma)$ (see Section 2.1). In this way, the notation and concepts related to order sorted signatures (e.g., terms, least sort, equations, etc.) are available for object-oriented signatures as well.

For our purposes, the assumptions above are general enough to model signatures of various object-oriented languages. For instance, Java interfaces[6] [4] can be seen as just standard Java classes for the purpose of representing program states. For the same purpose, interface `implements` declarations can be simply seen as a normal Java subclass declarations indicated by `extends`. Similarly, Java pre-defined arrays can be seen as classes having an arbitrary number of attributes, one for each index of the array. Finally, it is easy to see that a Java attribute `att` defined as

```
class C {
  Type   att;
}
```

corresponds to the operation `att : C -> Type` in an object-oriented signature.

For modelling states, we can just consider that an object-oriented specification P is formed by an object-oriented signature Σ and a set FE of F-equations—that is, equations formed only by the functions of Σ and variables. In fact, as illustrated in Section 3, states contain only the definitions of the data types used by a program; the definitions of methods and derived attributes are not relevant for defining the states associated to a program. Furthermore, recall that the data types used by a program should be defined by equations in OBJ before states of that program can be modelled as OBJ specifications or as OSA presentations. So it becomes trivial to map such a program to an object-oriented specification.

[6] Java's roughly equivalent to so called "abstract classes" in other languages.

4.2 States

As discussed in Section 3, states of object-oriented programs are informally represented by *specific* OBJ specifications. Thus in order to provide a formal representation for states it is enough to use the same mathematical model normally used to formally represent OBJ specifications: OSA presentations (see Section 2.2). Indeed, as first discussed in [2], states can be formally represented by OSA presentations.

However remember that, due to class inheritance, attributes with the same name and related classes might be associated with completely different values. This implies that the functions corresponding to those attributes in specifications denoting states might be nonmonotonic. Therefore states should be formally modelled by order sorted presentations with a signature of non-monotonicities; just order sorted presentations are not adequate for doing that in an elegant way and can easily lead to inconsistencies as discussed in Section 3.

Let us now make more precise what are the contents of such a presentation. But first, we have to assume that for any object-oriented specification P there is an associated C-sorted family $I(\mathsf{P})$ (just I, when not confusing) of disjoint components; that is, $I_u \cap I_{u'} = \emptyset$, if $u \not\equiv u'$. Each component is formed by symbols which can be used as object identifiers[7] (references) of a given class. This fixed connection between identifiers and classes is necessary because we are representing those concepts in the framework of OSA; so, each symbol should have a fixed, pre-defined rank. In order to ensure least parse of terms, we also assume that identifiers cannot have the same name as functional constants (formally, $|I| \cap F_{\lambda,u} = \emptyset$, for any $u \in S$).

Definition 4.1 For a specification P, a **database state** is a presentation with a signature of non-monotonicities, consisting of the following components:

1. A signature $(U, \leq, D)$, where $D = F \cup A \cup Id$, for some $Id \subseteq I$ containing the identifiers of the objects in this state[8].
2. A signature $\Omega = R \cap A$ of non-monotonicities, containing redefining (overriding) attributes.
3. A set $DE = FE \cup IdE$ of parsed D-equations, for some finite set IdE of equations establishing the values for some of the stored attributes of objects in Id. Actually, (D, DE) has to be a conservative extension of $(F \cup Id, FE)$, in the sense that the equations in DE should not relate functional expressions nor object identifiers that cannot be related by the equations in FE[9].

□

Note that I can alternatively be seen as an $U^* \times U$-sorted family, by considering that $I_{\lambda,u} = I_u$, for $u \in C$; and $I_{w,u} = \emptyset$, for $w \neq \lambda$ or $u \notin C$.

Observe that Id and IdE are the components of the database that can change from one state to another, by the execution of expressions which may update, create and remove objects. The other components remain fixed.

We can now define the family D_{P} of all database states for a given specification P; it is the family of all presentations (with a signature of non-monotonicities) that satisfy the requirements in Definition 4.1, for a fixed P. Note that D_{P} is not necessarily the family of all database states reachable from the initial one by the execution of method expressions. Naturally, this family is contained in D_{P}.

Hereafter we assume that, for a database state $\mathcal{D}$ and some $t \in \mathcal{T}_D$, the choice of the representative $[\![t]\!]_{\mathcal{D}}$ of the equivalence class $[t]_{\mathcal{D}}$ is a functional term or an object identifier whenever this is possible (i.e., $[\![t]\!]_{\mathcal{D}}$ is in $\mathcal{T}_{F \cup Id}$ if $|\mathcal{T}_{F \cup Id}| \cap [t]_{\mathcal{D}} \neq \emptyset$). We do not give any more details on the definition of representatives. Instead, we let it be defined when needed.

[7] Note that usually object identifiers are only visible to run-time systems; the programmer manipulates only variables containing such identifiers. So we can simply assume that the family I is defined by run-time systems.

[8] Strictly speaking, signatures of presentations representing states should have a universal type, in order to guarantee that equational satisfaction is closed under isomorphism [9]. However, it is just a small extension of this definition to assume that such a type exists, if it does not exist already in the language.

[9] Formally, for a given (D, DE)-algebra Alg, monotone except on Ω, there exists a monotone $(F \cup Id, FE)$-algebra Alg' and an injective $(F \cup Id)$-homomorphism from Alg' to $Alg \restriction F \cup Id$.

4.3 Design Decisions

Our approach for modelling states as OSA presentations has some particularities that are not so explicit but should be noticed. First, the restriction on the equations of database states is important to avoid inconsistances. For instance, a state like

```
fth STATE7 is
  pr NAT .
  class C .
  fn a : C -> Nat .
  fn c : -> C .
  ax a(c) = 0 .
  ax a(c) = 1 .
endfth
```

where `a` is not redefined, violates the restriction because the equations relate two functional expressions (`0` and `1`) which are not related by `NAT`. Allowing this kind of state would imply that the data types associated to states would not be the same used by Java or FOOPS programs, for example. This is clearly not desirable, since it would mean that the results yielded by expressions evaluated in states would not have the same meaning as the corresponding elements of the data types in the associated program.

Second, observe that some attributes may have no associated value in a particular database state; this implies that they cannot be evaluated or yield a default pre-defined value such as `null` in Java. The restrictions on database states do not prevent this flexibility, which has the advantage of not requiring object creation operations to initialize attributes; this is particularly useful for defining the semantics of languages, such as FOOPS, not necessarily having default values for attributes[10]. One further advantage is that object deletion operations do not need to assign an *ad hoc* value (usually `nil`, `void` or `null`) to attributes storing references to the object to be removed, in order to avoid dangling identifiers. Instead, after deletion those attributes might have no associated value, as illustrated in Section 3.1.

Lastly, contrasting to [8, 2], our approach opts for not representing *derived* attributes in specifications denoting states. Instead, the semantics of derived attributes should be given using operational semantics, similarly to how the semantics of methods is defined. In fact, representing derived attributes in specifications has the disadvantage of not supporting dynamic binding, in addition to not avoiding inconsistency problems in states similar to the ones that might happen when the restriction on the equations of database states are not observed.

4.4 Operations on States

In addition to the usual operations associated to presentations (e.g., $[\![\, e \,]\!]_{\mathcal{D}}$), some specific operations on database states (presentations with a signature of non-monotonicities) are necessary for defining the operational semantics. Those are introduced in this section. First, we define an operation that updates databases. Later, we give operations for adding and removing objects from databases.

4.4.1 Updating Databases

The update of a database $\mathcal{D}$ with equations Γ is denoted by $\mathcal{D} \oplus \Gamma$. Basically, this operation adds and removes some equations from a database. The added equations, denoted by Γ, establish "new" values for attributes. The removed equations are the ones that specify "old" values for the updated attributes.

First, we define the operation $\oplus$ for overwriting a set of equations by an unquantified, unconditional equation. Informally, for a set of equations Γ and an equation e, $\Gamma \oplus e$ is a set consisting of e and all equations in Γ whose LHS or RHS is not (syntactically) the same as the LHS of e. Note that we may refer to the term "equation" when we actually mean "parsed equation".

[10] Obviously, the same approach can also be used for defining the semantics of languages, such as Java, having default values for attributes.

Definition 4.2 The **overwriting** of a finite set of Σ-equations by an unquantified, unconditional Σ-equation is defined by the following equations:

- $\emptyset \oplus (l, r) = \{(l, r)\}$;
- $(\Gamma \cup \{(X, l', r', C)\}) \oplus (l, r) = \Gamma \oplus (l, r)$, if $l \equiv l'$ or $l \equiv r'$; otherwise,
- $(\Gamma \cup \{(X, l', r', C)\}) \oplus (l, r) = (\Gamma \oplus (l, r)) \cup \{(X, l', r', C)\}$,

for any set of equations Γ, and any Σ-equations (X, l', r', C) and (l, r). $\Box$

Assuming that l is the application of an attribute to arguments, $\Gamma \oplus (l, r)$ gives a set of equations derived from Γ by adding the equation (l, r), and deleting all equations specifying the value of the attribute denoted by l.

We need an auxiliary concept in order to extend the definition of overwriting for a set of equations. A set of unquantified, unconditional equations is called contradictory if it has two different equations composed by the same term. The following definition formalizes this.

Definition 4.3 A set Γ of unquantified, unconditional Σ-equations is **contradictory** if it contains two different equations (l, r) and (l', r') such that $l \equiv l'$ or $l \equiv r'$. $\Box$

Notice that this is a syntactical definition in the sense that a set containing two equations with the same LHS but different RHS is considered contradictory, even if the RHS are equivalent (modulo some equations). The definition of overwriting is also syntactical in a similar sense. This is appropriate for our purposes in this text.

Now, we can define overwriting for a set of equations.

Definition 4.4 Given two finite sets of Σ-equations Γ and $\Gamma' = \{e_1, \ldots, e_k\}$, for $k \geq 1$, if Γ' is a non contradictory set of unquantified, unconditional equations then the **overwriting** of Γ by Γ', denoted $\Gamma \oplus \Gamma'$, is defined as $\Gamma \oplus e_1 \oplus \cdots \oplus e_k$. Also, $\Gamma \oplus \emptyset$ is defined as Γ. $\Box$

Note that this uniquely defines the overwriting operation since Γ' is non contradictory, so $\Gamma \oplus e_i \oplus e_j$ is the same as $\Gamma \oplus e_j \oplus e_i$, for any $i, j \leq k$.

Lastly, we introduce the definition that can be used to update database states.

Definition 4.5 The **overwriting** of a presentation (with a signature of non-monotonicities) $P = (S, \leq, \Sigma, \Omega, \Gamma)$ by a non contradictory finite set of unquantified, unconditional Σ-equations Γ', denoted $P \oplus \Gamma'$, is the presentation $(S, \leq, \Sigma, \Omega, \Gamma \oplus \Gamma')$. $\Box$

4.4.2 Adding Objects to Databases

The operation $\cup$ adds some operation symbols to the signature of a presentation (see Section 2.1). So, it can be used to add (non initialized) objects to a database if the symbols represent object identifiers. In this way, $\mathcal{D} \cup Id$, for any $U^* \times U$-sorted family Id of object identifiers, adds the identifiers in Id to the database $\mathcal{D}$ of a specification P.

4.4.3 Removing Objects from Databases

The operation for removing objects from databases deletes object identifiers from the signature of a given presentation. Moreover, the equations formed by terms containing these symbols are removed as well. This means that all references to an object are removed after this object is deleted; that is, the attributes containing these references do not have any associated value in the resulting database. For a $U^* \times U$-sorted family Id of object identifiers and a database $\mathcal{D}$ of a specification P, the deletion is represented by $\mathcal{D} \ominus Id$. Here is the formal definition:

Definition 4.6 The **deletion** of a $S^* \times S$-sorted family Id of operation symbols from a presentation (with a signature of non-monotonicities) $P = (S, \leq, \Sigma, \Omega, \Gamma)$, represented by $P \ominus Id$, is the presentation $(S, \leq, \Sigma - Id, \Omega - Id, \Gamma')$, where Γ' is the set of all $(\Sigma - Id)$-equations in Γ. $\Box$

5 Conclusions

We presented a general approach for formally modelling states of object-oriented programs using OBJ specifications and associated order-sorted theory presentations. This approach uses the power of the theory of ATDs for defining operations on states and reasoning about them; in particular, the semantics of inheritance, and evaluation and dynamic binding of stored attributes is directly provided by OSA. This is the main advantage of using our approach for defining the semantics of object-oriented languages. It then becomes simple to define the basic operations on states. For example, the operation of reading an attribute's current value is simply defined by means of equational deduction or term rewriting in the OBJ module used for evaluation. Also, updating an attribute can be simply specified by replacing equations from the OBJ specification denoting a state. Similar straightforward operations on OBJ modules are used to define object creation and deletion.

Our approach has a direct impact on the simplicity of operational semantics definitions. In fact, a specialization of the approach presented here was extensively used to define an operational semantics for the concurrent object oriented language FOOPS [2], where it was confirmed the adequability of using specifications to model states of object-oriented languages; lots of complications could be avoided, a concise semantic definition was obtained, and many concepts usually confusing in other frameworks were clarified.

Another advantage of our approach is that specifications denoting states can be easily and directly defined from the associated object-oriented programs. Also, specifications denoting states can be easily and compositionally extended if the associated program is extended. This helps to understand and use the presented model to define operational semantics for concurrent object-oriented languages in special.

The approach presented in this paper could also be used to model states of *imperative* programs. Just observe that a variable can be trivially represented as an object having methods for assigning values to the variable and reading its current value. In general, this implies that imperative programs can be naturally viewed as object-oriented programs where the objects are simply variables. However, comparing with our approach, the use of mappings from variable names to their respective values seems more adequate to model states of *imperative* programs. In fact, mappings turn out to be a simpler and more natural model for that purpose.

References

[1] Pierre America and J.J.M.M. Rutten. A parallel object-oriefnted language: Design and semantic foundations. Technical Report CS-R8953, Centre for Mathematics and Computer Science, 1989.

[2] Paulo Borba and Joseph Goguen. An operational semantics for FOOPS. In Roel Wieringa and Remco Feenstra, editors, *International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94*. Vrije Universiteit, Amsterdam, September 1994. A longer version appeared as Technical Monograph PRG-115, Oxford University, Computing Laboratory, Programming Research Group, November 1994.

[3] Corina Cîrstea. A distributed semantics for FOOPS. To appear, 1995. Oxford University, Computing Laboratory, Programming Research Group.

[4] SUN Microsystems Computer Corporation. *The Java Language Specification*, 1.0 beta edition, October 1995.

[5] Frank de Boer. *Reasoning about dynamically evolving process structures —A proof theory for the parallel object-oriented language POOL*. PhD thesis, Vrije Universiteit, Amsterdam, 1991.

[6] David Duke and Roger Duke. Towards a semantics for Object-Z. In Dines Bjorner, C.A.R. Hoare, and Hans Langmaack, editors, *Proceedings, VDM'90: VDM and Z—Formal Methods in Software Development*, pages 242–262. Springer-Verlag, 1990. Lecture Notes in Computer Science, Volume 428.

[7] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In *Proceedings, Tenth Workshop on Abstract Data Types*. Springer, to appear 1993.

[8] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987.

[9] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 2(105), 1992.

[10] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988. Revised version to appear with additional authors José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud, in *Applications of Algebraic Specification using OBJ*, edited by Joseph Goguen.

[11] Cliff Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, Department of Computer Science, University of Manchester, 1993.

[12] José Meseguer. A logical theory of concurrent objects. In *Proceedings of ECOOP-OOPSLA90 Conference on Object Oriented Programming*, pages 101–115. ACM, 1990.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. Technical Report ECS-LFCS-89-85, 86, Laboratory for Foundations of Computer Science, Edinburgh University, 1989.

[14] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, September 1981.

[15] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical Report PRG-TR-28-92, Oxford University, Computing Laboratory, Programming Research Group, November 1992.

[16] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[17] David Walker. π-Calculus semantics of object-oriented programming languages. In *TACS'91 – Proceedings of the international Conference on Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991.

Contents

1 Introduction **1**

2 Order Sorted Algebra and OBJ **2**
2.1 Order Sorted Algebra 2
2.2 OBJ Specifications 3

3 States as Specifications **4**
3.1 States 4
3.2 Operations on States 7

4 States as Presentations **9**
4.1 Signatures and Specifications 9
4.2 States 10
4.3 Design Decisions 11
4.4 Operations on States 11
4.4.1 Updating Databases 11
4.4.2 Adding Objects to Databases 12
4.4.3 Removing Objects from Databases 12

5 Conclusions **13**