An Operational Semantics for FOOPS
Extended Abstract

Paulo Borba*
Programming Research Group
Oxford University Computing Laboratory

Abstract

FOOPS is a functional, concurrent, object-oriented language. We give a structural operational semantics for FOOPS, considering features such as: classes of objects with associated methods and attributes, object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding, concurrency, nondeterminism and atomic execution.

1 Introduction

FOOPS is a functional, object-oriented specification language with an executable subset [5, 18]. The functional part of FOOPS is derived from OBJ [8], which is a first order, purely functional language supporting an algebraic style for the specification, rapid prototyping and implementation of abstract data types.

FOOPS extends OBJ by providing a simple declarative style for object-oriented programming and specification using (conditional) equations. It supports classes of objects with associated methods and attributes, object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding, and many additional features.

Here we consider a natural extension of FOOPS for specifying systems of concurrent, distributed and autonomous objects [4, 16]. Essentially, this extension allows the definition of non terminating (autonomous) methods, and has explicit constructors (called method combiners) for expressing concurrency, nondeterminism and atomic execution. Those features are necessary for modelling processes in a natural way.

All those aspects are considered by the structural operational semantics [17] that we describe in this paper. However, we concentrate on the object level of FOOPS. An operational semantics for the functional level may be found elsewhere [12].

We adopt a special approach for modelling states of the operational semantics. Following some ideas from [5], we use order sorted theory presentations [6, 7] to represent states. As states are represented by an abstract structure, the semantics is defined in a simple way. In fact, lots of complications are avoided and a concise semantic definition can be obtained, although the language supports many features. Also, the use of this approach clearly facilitates the definition of the semantics, since the original FOOPS design used concepts from order-sorted algebra (OSA) [6, 7].

This text is structured in the following way. First, we give an overview of OSA. Second, we introduce most aspects of FOOPS. After that, we give the basis for the semantic definition; we introduce definitions and operations for FOOPS signatures, specifications, and runtime database states. Lastly, we gradually describe the semantics; we give rules for method and attribute evaluation, followed by rules for method combiners and creation and deletion of objects.

*The research reported in this paper has been supported by CAPES, Brazil, Grant 2184-91-8. This research is partially supported by Esprit Basic Research Action IS-CORE. Electronic mail: `Paulo.Borba@prg.oxford.ac.uk`.

2 Overview of Order Sorted Algebra

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

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

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

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

We now turn to the models that provide actual functions to interpret the operation symbols in a signature. Let $(\mathbf{S}, \Sigma)$ be a many-sorted signature. Then an **$(\mathbf{S}, \Sigma)$-algebra A** is a family $\{\mathbf{A_s} \mid \mathbf{s} \in \mathbf{S}\}$ of sets called the **carriers** of $\mathbf{A}$, together with a function $\mathbf{A}_\sigma : \mathbf{A_w} \to \mathbf{A_s}$ for each σ in $\Sigma_{\mathbf{w},\mathbf{s}}$ where $\mathbf{A_w} = \mathbf{A_{s1}} \times \ldots \times \mathbf{A_{sn}}$ when $\mathbf{w} = \mathbf{s1} \ldots \mathbf{sn}$ and where $\mathbf{A_w}$ is a one point set when $\mathbf{w} = \lambda$. Let $(\mathbf{S}, \leq, \Sigma)$ be an order-sorted signature. An **$(\mathbf{S}, \leq, \Sigma)$-algebra** is a many sorted $(\mathbf{S}, \Sigma)$-algebra $\mathbf{A}$ such that $\mathbf{s} \leq \mathbf{s}'$ in $\mathbf{S}$ implies $\mathbf{A_s} \subseteq \mathbf{A_{s'}}$. When $(\mathbf{S}, \leq)$ is clear, $(\mathbf{S}, \leq, \Sigma)$-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 is called the **term algebra** and it's denoted by $\mathcal{T}_\Sigma$. Similarly, $\mathcal{T}_\Sigma(\mathbf{X})$ is the algebra of Σ-terms with variables in the variable family $\mathbf{X}$.

A term may have many different sorts. In particular, if $\mathbf{t} \in \mathcal{T}_\Sigma$ has sort $\mathbf{s}$, then it also has sort $\mathbf{s}'$ for any $\mathbf{s}' \geq \mathbf{s}$. A condition on signatures called **regularity** guarantees that every term has a well defined least sort (see [6]). For a term $\mathbf{t}$, this is denoted by $\mathbf{LS}(\mathbf{t})$.

Given a term $\mathbf{t} \in \mathcal{T}_\Sigma(\{\mathbf{x}_1, \ldots, \mathbf{x_n}\})$ and terms $\mathbf{t}_1, \ldots, \mathbf{t_n}$, $\mathbf{t}(\mathbf{x}_1 \leftarrow \mathbf{t}_1, \mathbf{x}_2 \leftarrow \mathbf{t}_2, \ldots, \mathbf{x_n} \leftarrow \mathbf{t_n})$ denotes the term obtained from $\mathbf{t}$ by simultaneously substituting $\mathbf{t_i}$ for $\mathbf{x_i}$, for $\mathbf{i} = 1 \,..\, \mathbf{n}$.

We also introduce the term algebra $\mathbf{P}_\Sigma$ of fully parsed terms (i.e., terms together with their type information) associated to Σ. We let $\mathbf{P}_\Sigma$ be the least $\mathbf{S}$-sorted set such that $\sigma \in \Sigma_{\mathbf{w},\mathbf{s}}$ and $\mathbf{ti} \in \mathbf{P}_{\Sigma,\mathbf{si}}$ for, $\mathbf{i} = 1, \ldots, \mathbf{n}$, where $\mathbf{w} = \mathbf{s1} \ldots \mathbf{sn}$, and $\mathbf{s} \leq \mathbf{s}'$ imply $\sigma.\mathbf{ws}(\underline{\mathbf{t}1, \ldots, \mathbf{tn}}) \in \mathbf{P}_{\Sigma,\mathbf{s}'}$. 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 $\mathbf{t}$ into a fully typed term $\mathbf{t}'$ such that the sort of $\mathbf{t}'$ is the least sort of $\mathbf{t}$. 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 $\mathbf{t}$ abbreviates $\rho(\mathbf{t})$. In the same way, an unparsed equation might be used when a parsed equation is expected.

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

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 [6]). Furthermore, this congruence splits the term algebra into equivalence classes of terms modulo Γ. Hence, given a term $\mathbf{t}$, $[\mathbf{t}]_\Gamma$ denotes its equivalence class under Γ, and $[\![\mathbf{t}]\!]_\Gamma$ denotes the representative of this class.

An **order-sorted presentation with a signature of non-monotonicities** is an ordered 5-tuple, $(\mathbf{S}, \leq, \Sigma, \Omega, \Gamma)$, where $(\mathbf{S}, \leq, \Sigma)$ 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.

[1] Given a poset $(\mathbf{S}, \leq)$, let 5 denote the transitive and symmetric closure of $\leq$. Then 5 is an equivalence relation whose equivalence classes are called the **connected components** of $(\mathbf{S}, \leq)$.

[2] However, the reader should be aware that satisfaction of an equation depends crucially on its variable set [14].

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: $\sigma.\mathbf{ws}(\tilde{\mathbf{x}}) = \sigma.\mathbf{w}'\mathbf{s}'(\tilde{\mathbf{x}})$, for any $\sigma \in \Sigma_{\mathbf{w},\mathbf{s}} \cap \Sigma_{\mathbf{w}',\mathbf{s}'}$ such that $\mathbf{w} \leq \mathbf{w}'$ and $\sigma \notin \Omega_{\mathbf{w},\mathbf{s}}$, where $\mathbf{X}$ is a Σ-variable family, $\mathbf{x_i} \in \mathbf{X}$, for $\mathbf{i} = 1 \,..\, \mathbf{k}$, $\tilde{\mathbf{x}}$ stands for $\mathbf{x}_1, \ldots, \mathbf{x_k}$, and $\mathbf{w} = \mathbf{LS}(\mathbf{x}_1), \ldots, \mathbf{LS}(\mathbf{x_k})$. We let Γ^* be the union of Γ with the default equations.

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

3 Overview of FOOPS

FOOPS extends OBJ with some concepts from object-oriented programming. This motivates two central design decisions chosen for FOOPS (see [9, 5, 18, 19, ?] for more details about FOOPS design): data elements are not objects, and classes are not modules. In this way, FOOPS provides different constructs for defining abstract data types and classes of objects. Consequently, there are also two constructs for specifying inheritance.

Modules are the main programming unit of FOOPS. They can be used to define related classes and abstract data types. Furthermore, FOOPS has a powerful module interconnection language, supporting parameterized modules with semantic interface requirements.

This clear distinction between data elements and objects divides the language in two parts: the functional level and the object level. In each level, there are two kinds of modules: one of them is used to define executable code, and is simply called module; the other one, called theory, is used to specify properties about the operations of an abstract data type or a class.

3.1 Functional Level

The functional level of FOOPS is a syntactical variant of OBJ. A FOOPS functional module defines one or more abstract data types, where the keyword `sort` introduces the name of the set of data elements, and `fn` defines the associated operations (functions) symbols.

A very simple functional theory is:

```
fth TRIV is
  sort Elt .
endfth
```

It just defines the sort `Elt`.

As an example of a functional level module, we consider `LIST`, defining lists of elements of a given sort:

```
fmod LIST[E :: TRIV] is
  pr NAT .
  sorts NeList List .
  subsorts Elt < NeList < List .
  fn nil : -> List .
  fn _._ : List List -> List [assoc id: nil] .
  fn _._ : NeList List -> NeList .
  fn _._ : NeList NeList -> NeList .
  fn head : NeList -> Elt .
  fn tail : NeList -> List .
  fn #_ : List -> Nat .
  var E : Elt .
  var L : List .
  ax head(E . L) = E .
  ax tail(E . L) = L .
  ax # nil = 0 .
  ax # (L . E) = 1 + # L .
endfmod
```

This module is parameterized by the sort of the elements in a list. It introduces a sort for nonempty lists and another for lists, where elements are considered singleton (nonempty) lists and nonempty lists are, of course, lists. This is indicated by the subsort relationship (`<`).

The empty list is represented by the constant `nil`, and `_._` denotes the function that concatenates two lists. The underscores in `_._` serve as placeholders for the arguments of this function. Note that `_._` is overloaded. The attributes of this function specify that it is associative and that it has `nil` as identity.

Some other functions are: `head`, which gives the first element of a non empty list; `tail`, which maps a non empty list to one obtained by removing its first element; and `#_`, which gives the number of elements in a list.

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

3.2 Object Level

At the object level, it is possible to define classes, which are collections of (potential) objects with same attributes and methods. FOOPS has a general computational model where objects are naturally distributed and (internally) truly concurrent. Objects are dynamically created and deleted, and there are special operations for performing these actions. Furthermore, each object has an unique identifier, which is used by other objects for access.

As an example of an object level module, let's consider the object module `BUFFER`, defining a class of bounded buffers. This module is parameterized by the capacity (a natural number greater than `0`) of buffers, specified by the functional requirement theory `MAX`, and by the sort of the elements to be stored in buffers.

```
omod BUFFER[E :: TRIV, M :: MAX] is
  pr LIST[E] .
```

Here the elements of a bounded buffer are stored in a list. Hence, it is necessary to import the functional module `LIST`, instantiating it with the argument module giving the sort of the elements in buffers.

The class `Buffer` of bounded buffers is defined by the declaration

```
  class Buffer .
```

Attributes are defined as operations from an object identifier to a value that indicates a current property of the related object. Objects of `Buffer` have the following attributes:

```
  at elems_ : Buffer -> List [private] .
  at empty?_ : Buffer -> Bool .
  at full?_ : Buffer -> Bool .
```

The attribute `elems_` gives the list of the elements in a buffer. This attribute is private because clients should not directly look at the state of buffers. This is indicated by the declaration `[private]`. The other attributes are: `empty?` which indicates whether the buffer is empty; and `full?`, indicating if the buffer contains the maximum number of elements:

Like attributes, methods are defined as operations having an object of its class as parameter. They might also have some extra parameters. Methods either evaluate to a special result or to identifier of the object which performs it. For `Buffer`, the available methods are the following: `reset`, which removes all elements from a buffer; `get`, which removes the first element of a non empty buffer and gives it as result; `put`, which inserts an element at the end of a buffer, if it is not full; and `del`, which removes the first element of a non empty buffer.

```
  me reset : Buffer -> Buffer .
  me get : Buffer -> Elt .
  me put : Buffer Elt -> Buffer .
  me del : Buffer -> Buffer [private] .
```

The last one is private because we do not allow clients to remove an element from a buffer unless it is going to be used, what can be done with `get`.

3.2.1 Axioms

Attributes 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. Hence, one must specify how this is done. In FOOPS, we use equations for that. If no equation is given for an attribute, it is a stored attribute.

For `Buffer`, we define `elems_` as a stored attribute. The others are derived attributes, so we introduce the following equations for them:

```
var B : Buffer .
var E : Elt .
ax empty? B = (elems B) == nil .
ax full? B = #(elems B) == max .
```

Equations defining attributes can only contain functions, attributes and object identifiers. This kind of equations are interpreted as left-to-right rewrite rules, but attributes are atomically evaluated, without interference from the execution (evaluation) of methods.

The behavior of methods can be specified by two different kinds of axioms. A direct method axiom (DMA) specifies how a stored attribute is updated by a given method. In fact, a DMA is an equation such that its left hand side (LHS) indicates the respective attribute and method, and its right hand side (RHS) is an expression specifying the new value for the attribute to be updated. For instance, the behavior of `reset` is given by

```
ax elems(reset(B)) = nil .
```

indicating that after its execution by an object `B`, the value of `elems_`, for `B`, is `nil`.

Further examples of DMAs are:

```
cx elems(put(B,E)) = (elems B) . E if not(full? B) .
cx elems(del(B)) = tail(elems B) if not(empty? B) .
```

where the methods are only executed if the conditions (expression following `if`) are satisfied. The new value for the specified attribute is computed in terms of the method arguments and the current attribute values. If there is no axiom specifying the new value for a stored attribute after the execution of a given method, this method won't update this attribute.

The evaluation of methods specified by this form of axiom is atomic and yields the identifier of the object which executed the method. The attributes are modified as specified, and no other objects are modified. As for attributes, the axiom's RHS and condition must be formed by functions, attributes and object identifiers.

Alternatively to DMAs, indirect method axioms (IMAs) may be used for defining methods. IMAs are equations that specify how a method is defined in terms of other operations; this is indicated by a method expression, i.e., any expression formed by methods, attributes, functions, object identifiers and method combiners (operators on method expressions). For example, the method `get` is specified by

```
  a-cx get(B) = result head(elems B) ; del(B) if not(empty? B) .
endomod
```

where `result_;_` is a method combiner which first evaluates its first argument (from left to right) and then the second one, yielding the value resulting from the evaluation of the first argument.

As for DMAs, no method symbol is allowed in an IMA's condition. IMAs are interpreted as left-to-right rewrite rules. Whereas the evaluation of an IMA's condition is atomic, the evaluation of an IMA's RHS is non atomic and may be interfered by the execution of other methods. However, we assume that IMAs introduced by the keyword `a-cx` (or `a-ax`) have their condition and RHS atomically evaluated. In fact, an IMA in the form

```
a-cx m(O) = e if c .
```

stands for the declarations

```
me m´ : C -> C´ .
ax m´(O) = e if c .
ax m(O) = [m´(O)] .
```

where `m´` is a new symbol and `[_]` is the atomic evaluation operator, which atomically executes its argument, without any interference from the execution of other methods.

3.2.2 Other Aspects

Here we briefly describe some other aspects of FOOPS which are formally specified in other sections of this text. Details about these aspects can be found in [18, ?].

In addition to `result_;_` and `[_]`, FOOPS provides other method combiners. In this paper we give the semantics of sequential composition, `_;_`; (interleaving) parallel composition, `_||_`; and (external) nondeterministic choice, `_[]_`.

Dynamic object creation and deletion are respectively provided in FOOPS by the operators `new` and `remove`. They will be explained in Section 8.

One useful and flexible way of inheriting properties of a superclass is by redefining some of its methods and attributes. In FOOPS, this is indicated by writing `[redef]` after the declaration of the operation symbol and rank. In this way, objects of the subclass use the new version of the operation, unless explicitly stated that the original version is desired, what can be done using the qualified notation `op.C`, where `op` is the operation name and `C` is the name of the superclass having the original version. FOOPS adopts the variant syntactic rule for redefinitions; this means that the rank of the specialized version must be smaller or equal to the rank of the original version, and the result of the first must be greater or equal to the result of the second (see [19] for details).

3.2.3 Reflective Semantics

Operationally, a system implemented in FOOPS 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 and deletion or creation of objects. Modifying this information changes the database state.

The basic idea about FOOPS reflective semantics, as described in [5], is to represent FOOPS programs and database states as abstract data types (ADTs) in such a way that the queries and modifications to the database are encoded as functions of these data types.

Contrasting with this, here we use a more abstract approach [17] which is also more appropriate for giving the semantics of concurrency, nondeterminism, and other aspects of FOOPS not discussed in [5]. However, motivated by [5], here we represent a state of the FOOPS database, for a specification **Sp**, by an order-sorted presentation (with a signature of non-monotonicities) containing: the definition of the abstract data types of **Sp**; functions and sorts corresponding to attributes and classes defined in **Sp**; constants of the sorts representing classes, denoting objects; axioms of **Sp** specifying the meaning of derived attributes; and equations establishing the values of stored attributes for objects in the database. Also, the subsort relationships in these theories reflect the subclass relationships in **Sp**.

In order to illustrate the contents of a presentation representing a database state, let's consider the specification defined by the module `BUFFER[NAT,SIX]`, where `SIX` is a functional module defining a constant `max` equal to `6`. For this specification, one possible database state looks like:

```
fth STATE1 is
  ex LIST[NAT] .
  sort Buffer .
  fn elems_ : Buffer -> List .
  fn empty?_ : Buffer -> Bool .
  ...
  var B : Buffer .
  ax empty? B = (elems B) == nil .
  ...
```

where we represent a presentation with a signature of non-monotonicities with the same syntax of a FOOPS module (assuming that non-monotonic operations are indicated by the tag `[redef]`, and unparsed equations are used if there is no ambiguity). This state contains the functional part of the specification (i.e., `LIST[NAT]`), a sort corresponding to the class `Buffer`, functions representing attributes and their associated axioms.

In addition to that, the fact that the class `Buffer` has three objects in this state (identified by `b1`, `b2` and `b3`) is indicated by:

```
  fns b1 b2 b3 : -> Buffer .
  ...
```

```
    ax elems b1 = nil .
    ax elems b2 = (1 . 2) .
    ...
  endfth
```

where the equations specify values for the stored attributes of the objects in this state.

Given a database state, attributes are evaluated by reducing the corresponding expression in the module representing the database. For example, considering `STATE1`, the evaluation of

`elems(b2)` and `empty?(b1)`,

respectively results in `1 . 2` and `true`.

Method execution changes the state of the database. For example, executing `put(b1,5)` in `STATE1` changes the database to the state represented by a presentation in the form

```
  fmod STATE2 is
    ...
    ax elems b1 = 5 .
    ax elems b2 = (1 . 2) .
    ...
  endfmod
```

containing the same information as `STATE1`, except that the equation `elems b1 = nil` is replaced by `elems b1 = 5`.

4 Signatures and Specifications

A FOOPS module defines a signature and a specification. A FOOPS signature contains a sort and a class hierarchy, and names (together with typing and overriding information) of functions, methods and attributes. A FOOPS specification is formed by a signature and some axioms (equations) which specify properties of the elements of the related signature.

Definition 1 A FOOPS **signature** consists of:

1. A "sort set" $\mathbf{U} = \mathbf{S} \cup \mathbf{C}$, where $\mathbf{S}$ has sort names and $\mathbf{C}$ has class names.
2. A partial order $\leq$ on $\mathbf{U}$, which establishes the sort and class hierarchy. Since classes and sorts are not related: $\mathbf{u} \leq \mathbf{t} \Rightarrow \mathbf{u} \in \mathbf{S} \Leftrightarrow \mathbf{t} \in \mathbf{S}$, for any $\mathbf{t}, \mathbf{u} \in \mathbf{S} \cup \mathbf{C}$.
3. A $\mathbf{U}^* \times \mathbf{U}$-sorted family $\Sigma = \mathbf{F} \cup \mathbf{A} \cup \mathbf{M}$, where $\mathbf{F}$, $\mathbf{A}$ and $\mathbf{M}$ respectively contain names for functions, attributes and methods.
4. A family $\mathbf{R} \subseteq \mathbf{A} \cup \mathbf{M}$ formed by names of redefined methods and attributes.

□

Sometimes, we use Σ to denote a signature $(\mathbf{U}, \leq, \Sigma, \mathbf{R})$. Here we omit some natural restrictions on the components of signatures, such as: "a method and an attribute with related ranks cannot have the same name, in order to avoid mixing methods up with attributes". See [?] for the details.

Now, we give a definition of specification.

Definition 2 A FOOPS **specification** is formed by a signature Σ and a set Γ of Σ-equations. □

Hereafter, we use $\mathbf{A}$ to refer to the family of attributes in a specification $\mathbf{Sp}$, when $\mathbf{Sp}$ is clear from the context; otherwise, we use the notation $\mathbf{A}(\mathbf{Sp})$. The same convention is used for the other components: $\mathbf{S}$, $\mathbf{C}$, $\mathbf{F}$, etc. Also, we use $\mathbf{FA}$ for $\mathbf{F} \cup \mathbf{A}$. It is also useful to know that the set $\mathcal{F}$ of functional equations in Γ is the largest set of $\mathbf{F}$-equations included in Γ. Moreover, the set $\mathcal{A}$ of attribute equations is the largest set of $\mathbf{FA}$-equations included in Γ and disjoint from $\mathcal{F}$.

5 Database States

As discussed in Section 3.2.5, a FOOPS database state can be represented by a presentation with a signature of non-monotonicities. Here we make more precise what are the contents of such a presentation.

First, we assume that for any specification **Sp** there is an associated **C**-sorted family $\mathbf{I_{Sp}}$ (just **I**, when not confusing) of disjoint components; that is, $\mathbf{I_u} \cap \mathbf{I_{u'}} = ?$, if $\mathbf{u} \not\equiv \mathbf{u'}$. Each component is formed by symbols which can be used as object identifiers. In order to ensure least parse of terms, we consider that identifiers cannot have the same name as functional constants (formally, $|\mathbf{I}| \cap \mathbf{F}_{\lambda,\mathbf{u}} = ?$, for any $\mathbf{u} \in \mathbf{S}$). Lastly, the family **I** must contain the identifiers of the objects specified in **Sp**: $\mathbf{M}_{\lambda,\mathbf{u}} \subseteq \mathbf{I_u}$, for any $\mathbf{u} \in \mathbf{C}$. Observe that **I** can be seen as an $\mathbf{U}^* \times \mathbf{U}$-sorted family, if we consider that $\mathbf{I}_{\lambda,\mathbf{u}} = \mathbf{I_u}$, if $\mathbf{u} \in \mathbf{C}$; and $\mathbf{I_{w,u}} = ?$, if $\mathbf{w} \neq \lambda$ or $\mathbf{u} \notin \mathbf{C}$.

Definition 3 For a specification **Sp**, a **database state** is a presentation with a signature of non-monotonicities, consisting of:

1. A signature $(\mathbf{U}, \leq, \mathbf{D})$, where $\mathbf{D} = \mathbf{FA} \cup \mathbf{O}$, for some $\mathbf{O} \subseteq \mathbf{I}$ containing the identifiers of the objects in this state.
2. A signature $\Omega = \mathbf{R} \cap \mathbf{A}$ of non-monotonicities.
3. A set $\mathcal{D} = \mathcal{F} \cup \mathcal{A} \cup \mathcal{O}$ of parsed **D**-equations, for some finite set $\mathcal{O}$ of equations establishing the values for some of the stored attributes of objects in **O**. Actually, $(\mathbf{D}, \mathcal{D})$ has to be a conservative extension of $(\mathbf{F} \cup \mathbf{O}, \mathcal{F})$, in the sense that the equations in $\mathcal{D}$ should not relate functional expressions nor object identifiers that cannot be related by the equations in $\mathcal{F}$.

□

Observe that some attributes may have no associated value in a particular database state. The restrictions on $\mathcal{A}$ and $\mathcal{O}$ don't prevent this.

As in the convention for specifications, we use **D**, $\mathcal{D}$ and **O** for the corresponding components of a database state D, when it is clear from the context; otherwise, we use $\mathbf{D}(\mathrm{D})$, etc. We let $\mathbf{D_{Sp}}$ be the family of all database states for a given specification **Sp**; that is, the family of all presentations (with a signature of non-monotonicities) that satisfy the requirements in Definition 3, for a fixed **Sp**.

Lastly, for a database state D and some $\mathbf{t} \in \mathcal{T}_\mathbf{D}$, we assume that the choice of the representative $[\![\mathbf{t}]\!]_\mathrm{D}$ of the equivalence class $[\mathbf{t}]_\mathrm{D}$ is a functional term or an object identifier whenever this is possible. We don't give any more details on the definition of representatives. Instead, we let it be defined when needed.

5.1 Operations on Database States

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

The operation for adding objects to a database is given by the operation $\cup$ on presentations (see Section 2). In this way, $\mathrm{D} \cup \mathbf{O}$, for a $\mathbf{S}^* \times \mathbf{S}$-sorted family **O** of object identifiers, adds the objects in **O** to the database D of a specification **Sp**.

The operation for removing objects from databases deletes some operation symbols 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 don't have any associated value in the resulting database. For a database D and a family of objects **O**, this operation is represented by $\mathrm{D} \ominus \mathbf{O}$.

6 Methods and Attributes

Now, we start to describe a structural operational semantics for the object level of FOOPS. Here we use the approach for operational semantics introduced in [17]. We assume some familiarity with that. The semantics is given by a relation that indicates how an expression is evaluated in a database state. A pair formed by a method expression and a database state is called a configuration. The relation specifies the transitions from one configuration to another, according to how an expression is evaluated and how it changes the database; each transition corresponds to a computational step during the evaluation of an expression.

Let's formalize those ideas. First, for a specification $\mathbf{Sp}$, we define the family $\mathcal{T}_{\mathbf{Sp}} = \mathcal{T}_{\Sigma\cup\mathbf{I}}$ of method expressions (without variables). In fact, we can use untyped (unparsed) terms to define the semantics (see [?] for the details). The typing functions $\mathbf{LS}$ and ρ can be used whenever some type information is necessary.

Second, for a specification $\mathbf{Sp}$, the semantics is given by a transition relation

$$\rightarrow_{\mathbf{Sp}} \subseteq \mathbf{Conf_{Sp}} \times \mathbf{Conf_{Sp}}$$

on configurations, where $\mathbf{Conf_{Sp}} = \mathcal{T}_{\mathbf{Sp}} \times \mathbf{D_{Sp}}$. This relation is inductively defined over the syntax of method expressions by inference (transition) rules which indicate how we can infer that two configurations are related (i.e., there is a transition from one to the other), assuming that some others are related. Only transitions that can be deduced from the inference rules that we will give in this text are allowed. In other words, $\rightarrow_{\mathbf{Sp}}$ is the least relation satisfying these inference rules.

For conciseness, we use $\rightarrow$ for $\rightarrow_{\mathbf{Sp}}$, when $\mathbf{Sp}$ is clear from the context. Also, $\mathbf{P} \rightarrow \mathbf{P}'$ stands for $(\mathbf{P}, \mathbf{P}') \in \rightarrow$, where $\mathbf{P}$ and $\mathbf{P}'$ are configurations.

6.1 Attributes

Before giving the semantics of attribute evaluation, we introduce some notation. For a specification $\mathbf{Sp}$, hereafter consider:

- database states $\mathrm{D}, \mathrm{D}' \in \mathbf{D_{Sp}}$;
- sort and class symbols $\mathbf{u}, \mathbf{u}', \mathbf{u_i}, \mathbf{u_i'} \in \mathbf{U}$, for $\mathbf{i} = 1\,..\,\mathbf{k}$; and
- object identifiers and evaluated functional terms: $\mathbf{v_i} \in \mathcal{T}_{\mathbf{F}\cup\mathbf{I}}$, for $\mathbf{i} = 1\,..\,\mathbf{k}$, where $\mathbf{LS}(\mathbf{v_i}) = \mathbf{u_i}$, and $\mathbf{v_i} \in \mathcal{T}_{\mathbf{F}}$ implies $\mathbf{v_i} \equiv [\![\mathbf{v_i}]\!]_{\mathcal{F}}$. Also, we let $\mathbf{v}, \mathbf{v}'$ be in $\{\mathbf{v}_1, \ldots, \mathbf{v_k}\}$.

We use the term "fully evaluated term" to refer to object identifiers and evaluated functional terms.

For simplicity, we consider that the first argument (from left to right) of an attribute or method identifies the object which will perform the associated operation.

As an operation can only be performed if its associated object exists, methods and attributes can only be evaluated if their first argument is the identifier of an object in the database state being used for evaluation. So, as $\mathbf{v}_1$ and D are respectively used as the first argument of operations and the database state in the rules to follow, we assume that $\mathbf{v}_1 \in \mathbf{O}(\mathrm{D})$. Notice that other identifiers used as arguments are not necessarily associated with objects in the state. This is natural since these identifiers might not even be used for the evaluation of the operation.

The arguments of an attribute must be fully evaluated before its evaluation. This evaluation is atomic, only reads the database used for evaluation (so, the state does not change) and yields the value of that attribute in this particular state . This value is determined by the equations in that state. The evaluation is only possible if the attribute has an associated value in that state. For example, non initialized attributes— specially attributes of objects not in the current state— are not evaluated. The following rule formalizes those ideas.

Rule 1 For any $\mathbf{a} \in \mathbf{A}$,

$$\frac{}{\langle \mathbf{a}(\tilde{\mathbf{v}}),\, \mathrm{D}\rangle \rightarrow \langle [\![\mathbf{a}(\tilde{\mathbf{v}})]\!]_{\mathrm{D}\cup\mathbf{I}},\, \mathrm{D}\rangle}$$

if $\mathbf{a}(\tilde{\mathbf{v}}) \in \mathcal{T}_{\mathbf{FA}\cup\mathbf{I}}$ and $[\![\mathbf{a}(\tilde{\mathbf{v}})]\!]_{\mathrm{D}\cup\mathbf{I}}$ is fully evaluated. □

The first condition guarantees that $\mathbf{a}(\tilde{\mathbf{v}})$ is an attribute application. The second condition checks if the attribute has an associated value in the database. Observe that as we use $\mathrm{D} \cup \mathbf{I}$ (instead of D) in $[\![\mathbf{a}(\tilde{\mathbf{v}})]\!]_{\mathrm{D}\cup\mathbf{I}}$, we can evaluate the attribute expression even if it contains identifiers of objects not in the database. However, if this is the case, the evaluation is only performed if the attributes of the objects corresponding to these identifiers are irrelevant to determine the value associated to the attribute that we want to evaluate.

By the convention in Section 2, $[\![\mathbf{a}(\tilde{\mathbf{v}})]\!]_{\mathrm{D}\cup\mathbf{I}}$ stands for $[\![\rho(\mathbf{a}(\tilde{\mathbf{v}}))]\!]_{\mathrm{D}\cup\mathbf{I}}$, since D is a presentation with a signature of non-monotonicities. Hence, as ρ gives the least parse of an expression and the equations in D are parsed, only the most specific equations (the ones associated to a given type of an operation) of D are used to define the value of the given attribute. This is exactly what is required to support dynamic binding.

6.2 Methods Specified by DMAs

In order to define the semantics of method evaluation, we introduce some new notation. Given a specification $\mathbf{Sp}$, we assume:

- a Σ-variable family $\mathbf{X}$ such that $|\mathbf{X}| = \{\mathbf{x}_1, \ldots, \mathbf{x_k}\}$, $\mathbf{u_i} \leq \mathbf{LS}(\mathbf{x_i})$, and $\mathbf{x_i} \equiv \mathbf{x_j}$ implies $\mathbf{v_i} \equiv \mathbf{v_j}$, for $\mathbf{i}, \mathbf{j} = 1 \,..\, \mathbf{k}$. Also, $\tilde{\mathbf{x}}$ abbreviates $\mathbf{x}_1, \ldots, \mathbf{x_k}$. Lastly, we write $\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{v}}$ for $(\mathbf{x}_1 \leftarrow \mathbf{v}_1, \ldots, \mathbf{x_k} \leftarrow \mathbf{v_k})$.
- expressions formed by functions, attributes, variables and identifiers of specified objects: $\mathbf{g}, \mathbf{h}, \mathbf{c} \in \mathcal{T}_{\mathbf{FA}\cup(\mathbf{M}\cap\mathbf{I})(\mathbf{X})}$[3].
- a method symbol $\mathbf{m}' \in \{\mathbf{m}.\mathbf{LS}(\mathbf{x}_1), \mathbf{m}\}$, for some $\mathbf{m} \in \mathbf{M}$ not in the form *symbol.class-name*.

Also, as any specification contain the sort `Bool`, hereafter we write $(\forall\, \mathbf{X})\ \mathbf{l} = \mathbf{r}\ \mathtt{if}\ \mathbf{c}$ instead of $(\forall\, \mathbf{X})\ \mathbf{l} = \mathbf{r}\ \mathtt{if}\ \{(\mathbf{c}, \mathtt{true})\}$, for any term $\mathbf{c}$ (as above) of sort `Bool`.

The evaluation of a method specified by DMAs is atomic and yields the identifier of the object that executes this method. Some of the attributes of this object are updated. The resulting database state is the overwriting of the previous state by equations specifying those updates. The LHS and RHS of these equations respectively correspond to the updated attribute and its new value. These equations are derived from the DMAs related to the method (as required by the frame assumption).

However, only the related DMAs with the following properties are considered for evaluation: the DMA's condition, when instantiated with the method arguments, is valid in the current state; the DMA's instantiated RHS must be defined in the current state; and the DMA has to be as specialized as the method, unless a qualified notation is used or the method is not redefined. Lastly, the method is only evaluated if there are some attributes to update and the updates are non contradictory. All those aspects are considered by the following rule.

Rule 2 For any $\mathbf{m} \in \mathbf{M}$, let $\mathbf{E}$ be the set of all equations

$$\mathbf{a}(\mathbf{v}_1) = [\![\mathrm{g}(\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{v}})]\!]_{\mathrm{D}\cup\mathrm{I}}$$

such that Γ has a DMA in the form

$$(\forall\, \mathbf{X})\ \mathbf{a}(\mathbf{m}(\tilde{\mathbf{x}})) = \mathbf{g}\ \mathtt{if}\ \mathbf{c};$$

`true` belongs to $[\mathbf{c}(\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{v}})]_{\mathrm{D}\cup\mathrm{I}}$; $[\![\mathrm{g}(\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{v}})]\!]_{\mathrm{D}\cup\mathrm{I}}$ is in $\mathcal{P}_{\mathbf{F}\cup\mathbf{O}(\mathrm{D})}$; and $\mathbf{m}' = \mathbf{m}$ and $\mathbf{m} \in \mathbf{R}_{\mathbf{w}',\mathbf{u}'}$, for some $\mathbf{w}' \leq \mathbf{w}$, imply $\mathbf{LS}(\mathbf{x}_1) = \mathbf{u}_1$.

If $\mathbf{E}$ is a non empty, non contradictory set (i.e., if it doesn't have two different equations composed by the same term), then

$$\frac{}{\langle \mathbf{m}'(\tilde{\mathbf{v}}),\ \mathrm{D}\rangle \rightarrow \langle \mathbf{v}_1,\ \mathrm{D} \oplus \mathbf{E}\rangle}$$

□

DMAs can only specify a deterministic behaviour for methods. So, note that if there are contradictory (or even redundant) DMAs in Γ, their associated method cannot be executed because its specified behaviour is considered inconsistent. Obviously, this should be avoided. In the rule above, this is reflected by requiring $\mathbf{E}$ to be non contradictory.

The equations defining the versions of a redefined method associated to a class and a corresponding subclass do not necessarily have to agree on the behaviour that they specify. Thus, only the specialized method equations can be used for execution. That's why the restriction $\mathbf{LS}(\mathbf{x}_1) = \mathbf{u}_1$ is necessary in the last rule, if a specific version of a redefined method is not indicated (by a qualified notation). This corresponds to the semantics of dynamic binding.

6.3 Methods Specified by IMAs

The execution of a method specified by an IMA corresponds to the evaluation of this IMA's RHS (instantiated with the method arguments). Of course, the evaluation can only happen if the IMA's condition (instantiated with arguments) is satisfied in the state where the method is going to be evaluated. Also, the IMA should be as specialized as the method, unless a qualified notation is used or the method is not redefined. The following rule considers those aspects.

[3]Non constant method symbols are neither allowed in conditions nor in the RHS of DMAs.

Rule 3 For any $\mathbf{m} \in \mathbf{M}$, if the IMA

$$(\forall \mathbf{X})\ \mathbf{m}(\tilde{\mathbf{x}}) = \mathbf{g}\ \mathtt{if}\ \mathbf{c}$$

is in Γ, $\mathtt{true} \in [\mathbf{c}(\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{v}})]_{\mathrm{D}\cup\mathbf{I}}$, and $\mathbf{m}' = \mathbf{m}$ and $\mathbf{m} \in \mathbf{R}_{\mathbf{w}',\mathbf{u}'}$, for some $\mathbf{w}' \leq \mathbf{w}$, imply $\mathbf{LS}(\mathbf{x}_1) = \mathbf{u}_1$, then

$$\frac{}{\langle \mathbf{m}'(\tilde{\mathbf{v}}),\ \mathrm{D}\rangle \rightarrow \langle \mathbf{g}(\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{v}}),\ \mathrm{D}\rangle}$$

□

Because of dynamic binding, only the most specialized IMAs can be used for evaluation. Contrasting with DMAs, if two or more IMAs that don't agree on the specified behaviour are used to define the same method, this method will have a nondeterministic behaviour.

7 Method Combiners

Now, we show how the semantics given in the previous section can be modified and extended to support method combiners. We introduce transition rules giving the semantics of each combiner.

Method combiners are conceptually different from methods and attributes. Indeed, they don't correspond to operations related to objects. Hence, they have a special semantics. Among other particularities, they offer some control over the order of evaluation of their arguments, and their results are generated depending if some particular arguments are fully evaluated. On the other hand, similarly to methods, method combiners are neither allowed in the RHS of DMAs nor in conditions.

In order to give the semantics of method combiners, we assume that signatures contain one more component: $\mathbf{MC} \subseteq \Sigma$, formed by method combiners names. Moreover, we consider that method combiners are not mixed up (in the sense of Definition 1) with functions, methods or attributes. Lastly, the following operations, which are in $\mathbf{MC}$, represent the predefined method combiners: `_;_ : U T -> T`; `_||_, _[]_ : U U -> U`; `result_;_ : U T -> U`; and `[_] : U -> U`; for any types `T, U` $\in \mathbf{U}$.

In order to support the parallel composition and choice of method expressions with unrelated types, we consider that a universal type `Univ` is in $\mathbf{U}$. This type includes any class or sort. That is, $\mathbf{u} \leq$ `Univ`, for any $\mathbf{u} \in \mathbf{U}$. However, `Univ` is neither a class nor a sort. In this way, an expression `e || f` is well formed, even if the sorts of `e` and `f` are not related. This is possible because `_||_` can be parsed with the type `Univ Univ -> Univ`.

7.1 Sequential Composition

The argument on the left of the sequential composition operator (`_;_`) has to be fully evaluated before the evaluation of the other argument starts. Transitions from the argument on the left provokes transitions from a sequential composition. When the left argument is fully evaluated, there is a transition to start the evaluation of the argument on the right. Those aspects are indicated by the following rules:

Rule 4

$$\frac{\langle \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}',\ \mathrm{D}'\rangle}{\langle \mathrm{e}\ ;\ \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}'\ ;\ \mathbf{f},\ \mathrm{D}'\rangle}$$

□

Rule 5

$$\frac{}{\langle \mathbf{v}\ ;\ \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e},\ \mathrm{D}\rangle}$$

□

7.2 Parallel Composition

Here we give an interleaving semantics for parallel composition. So, transitions from the arguments of a parallel composition operator are interleaved and they cause transitions from the parallel composition, as shown below:

Rule 6

$$\frac{\langle \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}',\ \mathrm{D}'\rangle}{\langle \mathrm{e}\ ||\ \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}'\ ||\ \mathbf{f},\ \mathrm{D}'\rangle}$$

□

Rule 7

$$\frac{\langle \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \mathbf{f}',\ \mathrm{D}'\rangle}{\langle \mathrm{e}\ ||\ \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}\ ||\ \mathbf{f}',\ \mathrm{D}'\rangle}$$

□

Also, the arguments of a parallel composition operator can be eliminated when they are fully evaluated. This is specified by the the next two rules:

Rule 8

$$\frac{}{\langle \mathbf{v} \mid\mid \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e},\ \mathrm{D}\rangle}$$

□

Rule 9

$$\frac{}{\langle \mathrm{e} \mid\mid \mathbf{v},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e},\ \mathrm{D}\rangle}$$

□

7.3 Nondeterministic Choice

The choice operator (_[]_) nondeterministically chooses one of its arguments for evaluation. Using process algebra's terminology (see [10]), here we opt for an *external* choice operator rather than an *internal* choice operator. In fact, the latter can be simply defined in terms of the former.

Transitions from the choice between two expressions correspond to transitions from one of the expressions. This is specified below:

Rule 10

$$\frac{\langle \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}',\ \mathrm{D}'\rangle}{\langle \mathrm{e}\ \texttt{[]}\ \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}',\ \mathrm{D}'\rangle}$$

□

Rule 11

$$\frac{\langle \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \mathbf{f}',\ \mathrm{D}'\rangle}{\langle \mathrm{e}\ \texttt{[]}\ \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \mathbf{f}',\ \mathrm{D}'\rangle}$$

□

Moreover, a fully evaluated argument may be chosen by the choice, without changing the database. This is expressed by the next two rules:

Rule 12

$$\frac{}{\langle \mathbf{v}\ \texttt{[]}\ \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathbf{v},\ \mathrm{D}\rangle}$$

□

Rule 13

$$\frac{}{\langle \mathrm{e}\ \texttt{[]}\ \mathbf{v},\ \mathrm{D}\rangle \rightarrow \langle \mathbf{v},\ \mathrm{D}\rangle}$$

□

7.4 Result

As probably expected, the rules giving the semantics of the result method combiner should be similar to the rules for _;_. This is confirmed by the rule for evaluation of the left argument of the result combiner.

Rule 14

$$\frac{\langle \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}',\ \mathrm{D}'\rangle}{\langle \texttt{result}\ \mathrm{e}\ \texttt{;}\ \mathbf{f},\ \mathrm{D}\rangle \rightarrow \langle \texttt{result}\ \mathrm{e}'\ \texttt{;}\ \mathbf{f},\ \mathrm{D}'\rangle}$$

□

When the left argument is fully evaluated, the evaluation of the right one may start. But contrasting with sequential composition, the left argument is not eliminated. Finally, when both arguments are fully evaluated, the left one is given as result. Those aspects are described by the next rules:

Rule 15

$$\frac{\langle \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \mathrm{e}',\ \mathrm{D}'\rangle}{\langle \texttt{result}\ \mathbf{v}\ \texttt{;}\ \mathrm{e},\ \mathrm{D}\rangle \rightarrow \langle \texttt{result}\ \mathbf{v}\ \texttt{;}\ \mathrm{e}',\ \mathrm{D}'\rangle}$$

□

Rule 16

$$\frac{}{\langle \texttt{result}\ \mathbf{v}\ \texttt{;}\ \mathbf{v}',\ \mathrm{D}\rangle \rightarrow \langle \mathbf{v},\ \mathrm{D}\rangle}$$

□

7.5 Atomic Evaluation

Intuitively, the atomic execution of an expression corresponds to its evaluation in only one step (transition). In fact, this means that during the atomic evaluation of an expression, the attributes accessed by it cannot be modified by other expressions being concurrently evaluated. This might be necessary if an expression has to be evaluated without interference from others. Hence, FOOPS provides the atomic evaluation operator (`[_]`).

Following the intuitions discussed above, we introduce the next rule.

Rule 17

$$\frac{\langle \mathrm{e}, \mathrm{D}\rangle \rightarrow^* \langle \mathbf{v}, \mathrm{D}'\rangle}{\langle \texttt{[e]}, \mathrm{D}\rangle \rightarrow \langle \mathbf{v}, \mathrm{D}'\rangle}$$

where $\rightarrow^*$ denotes the transitive, reflexive closure of $\rightarrow$. □

Observe that the atomic evaluation only succeeds if the related expression can be fully evaluated. This corresponds to the semantics of atomic transactions in database systems, where the fact that the expression cannot be fully evaluated is considered a failure. Usually, after a failure, transactions recover the state before the beginning of the transaction. That's why there is no transition if the expression cannot be fully evaluated.

If the evaluation of the expression to be atomically evaluated doesn't terminate, the atomic expression doesn't terminate as well. In fact, it behaves as a divergent process (infinite loop) that doesn't modify the state, since the updates made by an atomic expression are only visible after its evaluation. The following rule reflects those comments.

Rule 18 If $\langle \mathrm{e}, \mathrm{D}\rangle$ is non terminating,

$$\frac{}{\langle \texttt{[e]}, \mathrm{D}\rangle \rightarrow \langle \texttt{[e]}, \mathrm{D}\rangle}$$

□

where a configuration is **terminating** if there is no infinite sequence of $\rightarrow$-transitions from it.

8 Object Creation and Deletion

Dynamic object creation and deletion are respectively provided in FOOPS by the operators `new` and `remove`. These operators are modelled as method combiners because they are associated to classes, not to objects; that is, their corresponding operations are not performed by objects. So, hereafter we consider that signatures contain the following special combiners: `new.C() : -> C`, `new : C -> C`, and `remove : C -> C`, for each class $\texttt{C} \in \mathbf{C}$. (This extends the definition of signature, in the same way as done in Section 7.)

For a given class `C`, the operator `new.C()` creates an object of `C` with an identifier nondeterministically chosen from $\mathbf{I}_{\texttt{C}}$, but that is not already being used for another object. This identifier is given as the result of the evaluation of the operator, as specified by the next rule.

Rule 19 For any class $\texttt{C} \in \mathbf{C}$ and any identifier $\mathbf{v} \in \mathbf{I}_{\texttt{C}}$ not associated to an object in D (i.e., $\mathbf{v} \notin \mathbf{O}(\mathrm{D})$),

$$\frac{}{\langle \texttt{new.C()}, \mathrm{D}\rangle \rightarrow \langle \mathbf{v}, \mathrm{D} \cup \mathbf{V}\rangle}$$

where $\mathbf{V}$ is a $\mathbf{U}^* \times \mathbf{U}$-sorted family containing $\mathbf{v}$ only; that is, $\mathbf{V}_{\lambda,\texttt{C}} = \{\mathbf{v}\}$ and $\mathbf{V}_{\mathbf{w},\mathbf{u}} = ?$ if $\mathbf{w} \neq \lambda$ or $\mathbf{u} \neq \texttt{C}$. □

The operator `new` creates an object of the same class as the identifier given as argument, if this identifier is not associated to another object (otherwise, the operation cannot be executed). This identifier is used for the created object and yielded by the operator. These aspects are formalized by the following rule:

Rule 20 For any class $\mathtt{C} \in \mathbf{C}$ and any identifier $\mathbf{v} \in \mathbf{I}_{\mathtt{C}}$ not associated to an object in D (i.e., $\mathbf{v} \notin \mathbf{O}(\mathrm{D})$),

$$\frac{}{\langle \mathtt{new(v)},\ \mathrm{D}\rangle \to \langle \mathbf{v},\ \mathrm{D} \cup \mathbf{V}\rangle}$$

where $\mathbf{V}$ is a $\mathbf{U}^* \times \mathbf{U}$-sorted family containing $\mathbf{v}$ only, as defined in Rule 21. □

argument, removes its associated object from the database state and yields this identifier. If this identifier doesn't correspond to an object in the state, the operation is not evaluated. The following rule formalize those aspects.

Rule 21 For any class $\mathtt{C} \in \mathbf{C}$ and any identifier $\mathbf{v} \in \mathbf{I}_{\mathtt{C}}$ associated to an object in D (i.e., $\mathbf{v} \in \mathbf{O}(\mathrm{D})$),

$$\frac{}{\langle \mathtt{remove(v)},\ \mathrm{D}\rangle \to \langle \mathbf{v},\ \mathrm{D} \ominus \mathbf{V}\rangle}$$

where $\mathbf{V}$ is a $\mathbf{U}^* \times \mathbf{U}$-sorted family containing $\mathbf{v}$ only, as in Rule 21. □

9 Conclusions

We have described a structural operational semantics for the object level of FOOPS, considering features such as: classes of objects with associated methods and attributes, object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding, concurrency, nondeterminism and atomic execution.

We have concentrated on the object level of FOOPS. The semantics of other aspects, like the functional level and the module system, are discussed elsewhere [18, 12]. Some of the features listed above are not considered by other alternatives for the semantics of FOOPS, like the reflective semantics [5] and the sheaf semantics [20, 13].

The usual features of object-oriented languages were explained in detail, in a simple and abstract way, by using a special approach for modelling states of the operational semantics. In fact, lots of complications were avoided, a concise semantic definition could be obtained and many concepts, usually confusing in other frameworks, were clarified. Indeed, this approach seems appropriate to define the operational semantics of other object-oriented languages as well.

Because of space reasons, we haven't considered other aspects of FOOPS such as evaluation of expressions as background processes and object protection. We have also omitted rules for evaluation of functional terms and arguments of operations. Those aspects are formalized in [?].

The semantics described in this text is also useful as a formal basis for deriving implementations and simulation tools for FOOPS. In addition, because of the simplicity of the semantic definition, the operational semantics establishes a framework to support the formal development of distributed software in an object-oriented language. In particular, it has been used to define a notion of refinement for FOOPS programs and specifications, together with an associated proof technique which seems to be appropriate for real applications.

This semantics is also useful for reasoning about general properties of FOOPS programs. Notably, it has been used to compare "true concurrency" with an interleaving semantics for parallel composition. We have concluded that both approaches are equivalent in the context of FOOPS, given some mild assumptions on the notion of equivalence used for the language. This has the interesting consequence that we can use the simpler interleaving model for reasoning about true concurrency.

Acknowledgements

I wish to thank Professor Joseph Goguen for supervision and many interesting advices and criticisms to this work. Thanks to Steve Schneider and Grant Malcolm for giving many suggestions which helped to simplify the semantics described here. I am grateful to Steve Schneider and Adolfo Socorro for carefully reading and correcting an early version of this text. I also had the opportunity to discuss in detail some of the aspects presented here with Grit Denker and Prof. Hans Dieter Ehrich, during an enjoyable visit to Braunschweig.

References

[1] Paulo Borba. An operational semantics for FOOPS. Technical report, Oxford University Computing Laboratory, June 1994.

[2] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology*. Joint System Development Corporation, Tokyo, Japan, January 1990.

[3] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, to appear.

[4] 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.

[5] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, to appear 1992.

[6] 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, Cambridge, 1992 (?).

[7] Joseph Goguen and David Wolfram. On types and FOOPS. In Robert Meersman, William Kent, and Samit Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 1–22. North Holland, 1991. Proceedings, IFIP TC2 Conference, Windermere, UK, 2–6 July 1990.

[8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[9] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and Aarturo Salomaa, editors, *Proceedings, 15th International Colloquium on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer, 1988. Lecture Notes in Computer Science, Volume 317.

[10] Grant Malcolm. A sheaf semantics for FOOPS, 1994. Programming Research Group, Oxford University.

[11] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[12] Ellen Munthe-Kaas, Joseph Goguen, and José Meseguer. Method expressions and default values for object-valued attributes. Technical Report to appear, SRI International, Computer Science Lab, 1989.

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

[14] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical report, Oxford University Computing Laboratory, November 1992. PRG-TR-28-92.

[15] Adolfo Socorro. *Design, Implementation, and Evaluation of a Declarative Object Oriented Language*. PhD thesis, Programming Research Group, Oxford University, 1993.

[16] David Wolfram and Joseph Goguen. A sheaf semantics for FOOPS expressions. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Object-Based Concurrent Computation*, pages 81–98. Springer, 1992. Proceedings, ECOOP'91 Workshop, Geneva, July 1991.