An Operational Semantics for FOOPS

by

Paulo Borba
Joseph A. Goguen

Technical Monograph PRG-115

November 1994

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

Contents

An Operational Semantics for FOOPS

Paulo Borba* Joseph A. Goguen†

November 1994

Abstract

FOOPS is a 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, dynamic binding, concurrency, nondeterminism, atomic execution, evaluation of expressions as background processes, and object protection.

1 Introduction

FOOPS is a concurrent object-oriented specification language with an executable subset [18, 40]. FOOPS includes a functional language derived from OBJ [21], 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, dynamic binding, and many additional features.

Here we consider a natural extension of FOOPS for specifying systems of concurrent, distributed, and autonomous objects [12, 35]. Essentially, this extension allows the definition of non terminating (autonomous) methods, and has explicit constructors (called method combiners) for expressing concurrency, nondeterminism, atomic execution, and evaluation of method expressions as background processes. Furthermore, these constructors may be used to define more complex ones, by using the facilities for method combiner definition. Those features are necessary for modelling processes in a natural way.

This extension of FOOPS also provides a mechanism for object protection; in general terms, it's possible to specify which objects are allowed to directly invoke methods of a protected object. In fact, this can be used for defining private references (object identifiers); that is, references that

*Supported by CAPES, Brazil, Grant 2184-91-8, and the CEC under ESPRIT-2 BRA Working Group 6071, IS-CORE (Information Systems COrrectness and REusability). Electronic mail: Paulo.Borba@prg.oxford.ac.uk.

†Supported in part by the Science and Engineering Research Council, the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems COrrectness and REusability) and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), Fujitsu Laboratories Limited, and a contract under the management of the Information Technology Promotion Agency (IPA), Japan, as part of the Industrial Science and Technology Frontier Program "New Models for Software Architectures," sponsored by NEDO (New Energy and Industrial Technology Development Organization). Electronic mail: Joseph.Goguen@prg.oxford.ac.uk.

can only be used by a particular object. This mechanism seems to be essential for specifying and reasoning in a practical way about systems consisting of arbitrary object graphs [26].

All those aspects are considered by the structural operational semantics [39] 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 [28].

Along with the semantic description, we give comments that clarify many concepts and phenomena related to object-oriented languages. In particular, we show how the semantics suggests an appropriate programming style for FOOPS, and indicates how inconsistencies in FOOPS specifications may be avoided. We also justify the semantics adopted for some constructs, relating it to alternatives. In special, we discuss the "truly concurrent" and interleaving semantics for parallel composition. It turns out that these approaches are equivalent in the context of FOOPS, given some mild assumptions on the notion of equivalence of programs used for the language.

We adopt a special approach for modelling states of the operational semantics. Following some ideas from [18], we use order sorted theory presentations [19, 16] to represent states. This has the advantage of using all the power of the theory of ATDs for defining operations on states and reasoning about them. 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) [19].

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 formal definitions and operations for FOOPS signatures, specifications, and runtime database states. Lastly, we gradually describe the semantics; we give rules for function, method and attribute evaluation, followed by rules for method combiners, creation and deletion of objects, and a mechanism for object protection.

2 Overview of Order Sorted Algebra

We introduce some notation, definitions and basic results of order-sorted algebra (OSA), as originally presented in [19]. Most of the material in this section is copied from [19], [16], and [15]; here we also introduce some extra notation. This general overview of OSA is necessary because it is the mathematical theory supporting FOOPS functional level, and it is used to define the semantics described in this text.

OSA is a mathematical theory supporting multiple inheritance, overloading, polymorphism, error handling, partial functions, and multiple representation in an algebraic framework. The main idea to solve these problems is the definition of a partial order on the set of sorts of a given specification. This is interpreted as subset inclusion in the algebras that are models of this specification. More motivation and the history of OSA can be found in [19, 16]. Here we directly introduce the basic concepts.

2.1 Signatures

Signatures indicate the sorts and operations in a specification. This notion is formalized in this section.

The notation of sorted (also called "indexed") sets greatly facilitates the technical development of OSA. 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 $\{A_s \mid s \in S\}$. Similarly, given S-sorted sets A and B, an **S-sorted function** $f : A \to B$ is an S-sorted family $f = \{f_s : A_s \to B_s \mid s \in S\}$. 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; that is, $|A| = \bigcup_{s \in S} A_s$. Also, $e \in A$ is an abbreviation for $e \in |A|$.

In order-sorted algebra, S is a **partially ordered set, or poset**, i.e., there is a binary relation $\leq$ on S that is reflexive, transitive, and antisymmetric. 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 $1 \leq i \leq n$. Similarly, $\leq$ extends to pairs $\langle w, s\rangle$ in $S^* \times S$ by $\langle w, s\rangle \leq \langle w', s'\rangle$ iff $w \leq w'$ and $s \leq s'$.

Definition 2.1 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 $\{\Sigma_{w,s} \mid w \in S^*$ and $s \in S\}$. Elements of (the sets in) Σ are called operation (or function) symbols, or for short, **operations**. 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.

An order sorted signature is **monotone** iff the operations satisfy the following **monotonicity condition**,

$$\sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2} \text{ and } w1 \leq w2 \text{ imply } s1 \leq s2.$$

When the sort set S is clear, we write Σ for (S, Σ), and when the poset $(S, \leq)$ is clear, we write Σ for $(S, \leq, \Sigma)$. When $\sigma \in \Sigma_{w,s}$ we say that σ has **rank** $\langle w, s\rangle$, **arity** w, and (value, result, or coarity) **sort** s. A special case is $w = \lambda$, the empty string; then $\sigma \in \Sigma_{\lambda,s}$ is a **constant symbol** of sort s. Notice that the monotonicity condition excludes overloaded constants, because $\lambda = w1 = w2$ implies $s1 \leq s2$. □

An important observation is that the theory of OSA can be developed without the monotonicity condition for signatures [16]. In fact, it's necessary to avoid this condition in order to model FOOPS database states, as we will see later (see [16] for more motivation for not enforcing the monotonicity condition).

Given a signature $(S, \leq, X)$, we say that X is a **ground signature** iff it is formed only by distinct constant symbols; that is, $X_{\lambda,s} \cap X_{\lambda,s'} = \emptyset$ whenever $s \neq s'$, and $X_{w,s} = \emptyset$ unless $w = \lambda$. For a signature Σ, the notation $\Sigma(X)$ abbreviates $\Sigma \cup X$, if X is a ground signature disjoint from Σ (i.e., $X \cap \Sigma = \emptyset$). In this case, we may call X a **Σ-variable family**.

2.2 Algebras

We now turn to the models that provide actual functions to interpret the operation symbols in a signature.

Definition 2.2 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 $A_w = A_{s1} \times \cdots \times A_{sn}$ 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 $s \leq s'$ implies $A_s \subseteq A_{s'}$. When the sort set S is clear, (S, Σ)-algebras may be called **many-sorted Σ-algebras**; similarly, when $(S, \leq)$ is clear, $(S, \leq, \Sigma)$-algebras may be called **order-sorted Σ-algebras**.

We say that Ω is a **signature of non-monotonicities** for Σ if $\Omega \subseteq \Sigma$. Then, an order sorted Σ-algebra A is **monotone except on** Ω iff

> $\sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2}$ and $w1 \leq w2$ and $s1 \leq s2$ imply $A_\sigma : A_{w1} \to A_{s1}$ equals $A_\sigma : A_{w2} \to A_{s2}$ on A_{w1}, unless $\sigma \in \Omega_{w1,s1}$

An order sorted Σ-algebra A is **monotone** iff it is monotone except on the empty signature. □

Definition 2.3 Let (S, Σ) be a many-sorted signature, and let A and B be (S, Σ)-algebras. Then an (S,Σ)**-homomorphism** $h : A \to B$ is an S-sorted function $h = \{h_s : A_s \to B_s \mid s \in S\}$ satisfying the following **homomorphism condition**

(1) $h_s(A_\sigma^{w,s}(a)) = B_\sigma^{w,s}(h_w(a))$ for each $\sigma \in \Sigma_{w,s}$ and $a \in A_w$,

where $h_w(a) = \langle h_{s1}(a1), \ldots, h_{sn}(an)\rangle$ when $w = s1 \ldots sn$ and $a = \langle a1, \ldots, an\rangle$ with $ai \in A_{si}$ for $i = 1, \ldots, n$ when $w \neq \lambda$. If $w = \lambda$, condition (1) specializes to

(1′) $h_s(A_\sigma^{\lambda,s}) = B_\sigma^{\lambda,s}$.

When the sort set S is clear, a (S,Σ)-homomorphism may be called a **(many-sorted)** Σ**-homomorphisms**

Let $(S, \leq, \Sigma)$ be an order-sorted signature, and let A and B be order-sorted $(S, \leq, \Sigma)$-algebras. Then an $(S, \leq, \Sigma)$**-homomorphism** is any (S, Σ)-homomorphism. If A and B are monotone, a **monotone** $(S, \leq, \Sigma)$-homomorphism $h : A \to B$ is an (S, Σ)-homomorphism satisfying the following **restriction condition**:

(2) $s \leq s'$ and $a \in A_s$ imply $h_s(a) = h_{s'}(a)$.

When the poset $(S, \leq)$ is clear, $(S, \leq, \Sigma)$-homomorphisms are also called **(order-sorted)** Σ**-homomorphisms**. □

2.3 Terms

The algebra whose carrier sets are formed by the terms we can construct from a given signature Σ is called the **term algebra**; it's denoted by $\mathcal{T}_\Sigma$. In this section we describe an inductive construction defining the term algebra. For an order-sorted signature $(S, \leq, \Sigma)$, the term algebra is the least family $\{\mathcal{T}_{\Sigma,s} \mid s \in S\}$ of sets satisfying the following conditions:

- $\Sigma_{\lambda,s} \subseteq \mathcal{T}_{\Sigma,s}$ for $s \in S$;
- $\mathcal{T}_{\Sigma,s'} \subseteq \mathcal{T}_{\Sigma,s}$ if $s' \leq s$,
- if $\sigma \in \Sigma_{w,s}$ and if $ti \in \mathcal{T}_{\Sigma,si}$ where $w = s1 \ldots sn \neq \lambda$, then (the string) $\sigma\underline{(}t1 \ldots tn\underline{)} \in \mathcal{T}_{\Sigma,s}$.

Also,

- for $\sigma \in \Sigma_{w,s}$ let $\mathcal{T}_\sigma : \mathcal{T}_w \to \mathcal{T}_s$ send $t1, \ldots, tn$ to (the string) $\sigma\underline{(}t1 \ldots tn\underline{)}$.

Thus we can write $\sigma(t1,\ldots,tn)$ for $\sigma(\underline{t1\ldots tn})$. It's now easy to check that $\mathcal{T}_\Sigma$ is an order-sorted Σ-algebra.

The terms considered above are **ground terms**, in the sense that they involve no variables. In fact, terms with variables can be seen as a special case of ground terms, by enlarging the signature with new constants that correspond to variable symbols. Given an order-sorted signature $(S,\leq,\Sigma)$ and a Σ-variable family X, we can obtain a new order-sorted signature $(S,\leq,\Sigma(X))$ and form $\mathcal{T}_{\Sigma(X)}$. This can be viewed as an order-sorted Σ-algebra, by forgetting the constants in X; let's denote this algebra $\mathcal{T}_\Sigma(X)$. This gives the algebra of Σ-terms with variables in 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' such that $s \leq s'$. A condition on signatures called **regularity** guarantees that every term has a well defined least sort [19]. Here is the formal definition:

Definition 2.4 An order sorted signature $(S,\leq,\Sigma)$ is **regular** iff it is monotone, and given $\sigma \in \Sigma_{w1,s1}$ and $w0 \leq w1$, there is a least rank $\langle w,s\rangle$ such that $w0 \leq w$ and $\sigma \in \Sigma_{w,s}$. □

So, given a regular order-sorted signature $(S,\leq,\Sigma)$, for any $t \in \mathcal{T}_\Sigma$ there is a least $s \in S$ such that $t \in \mathcal{T}_{\Sigma,s}$; this is called the **least sort** of t and it's denoted by $LS(t)$.

In practice, regularity is not a strong restriction since non regular signatures can be translated into regular ones where the rank of an operation is considered to be part of its name.

Considering Σ-variable families X and Y, given an S-sorted map $a : X \to \mathcal{T}_{\Sigma(Y)}$, there is a unique Σ-homomorphism $\overline{a} : \mathcal{T}_{\Sigma(X)} \to \mathcal{T}_{\Sigma(Y)}$ which **substitutes** a term $a(x)$ for each variable $x \in X$ into each term t in $\mathcal{T}_{\Sigma(X)}$, yielding a term $\overline{a}(t)$ in $\mathcal{T}_{\Sigma(Y)}$ (see [19]). Hence a is called a **substitution** and $\overline{a}(t)$ denotes the application of this substitution to a term t. Usually, we use the alternative notations

$$t(x_1 \leftarrow t_1, x_2 \leftarrow t_2, \ldots, x_n \leftarrow t_n) \text{ and } a(t)$$

instead of $\overline{a}(t)$, where $|X| = \{x_1,\ldots,x_n\}$ and $a(x_i) = t_i$, for $i = 1,\ldots,n$. Moreover, we omit the pair $x_i \leftarrow t_i$ whenever $t_i = x_i$.

The key for developing OSA without the monotonicity condition for signatures is to consider typed (parsed) terms; that is, terms together with their sort information. We introduce the term algebra $\mathcal{P}_\Sigma$ of fully parsed terms associated to Σ. We let $\mathcal{P}_\Sigma$ be the least S-sorted set such that $\sigma \in \Sigma_{w,s}$ and $ti \in \mathcal{P}_{\Sigma,si}$, for $i = 1,\ldots,n$, where $w = s1\ldots sn$, and $s \leq s'$ imply $\sigma.ws(\underline{t1,\ldots,tn}) \in \mathcal{P}_{\Sigma,s'}$. The definitions introduced so far and the ones to come can be extended in an obvious way for parsed terms, but for simplicity we only consider unparsed terms. Moreover, parsed terms have least sorts even if the related signature is non monotonic or not regular.

Now, given a regular signature Σ, we can define a parsing function $\rho_\Sigma : \mathcal{T}_\Sigma \to \mathcal{P}_\Sigma$ which transforms an untyped term t into a fully typed term t' such that the sort of t' is the least sort of t. When not confusing, we drop the subscript from ρ. Here is the formal definition:

$$\rho(\sigma(e_1,\ldots,e_n)) = \sigma.wu(\rho(e_1,\ldots,e_n)),$$

where $\rho(e_1,\ldots,e_n) = \rho(e_1),\ldots,\rho(e_n)$, $u = LS(\sigma(e_1,\ldots,e_n))$, and w is the least sequence of sorts of size n such that $\sigma \in \Sigma_{w,u}$ and $LS(e_1),\ldots,LS(e_n) \leq w$. It's easy to extend ρ to equations (see Section 2.4), 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 associated signature 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.

2.4 Equations

In this section we give a formal definition for equations.

Definition 2.5 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 (for example, if X contains just the variables that occur in t and t', with sorts that are uniquely determined or have been previously declared) we allow it to be omitted[2]; that is, we allow unquantified notation for equations. We also say that an equation is **unquantified** if $X = \emptyset$.

Order-sorted conditional equations generalize order-sorted equations in the usual way, i.e., they are expressions of the form $(\forall X)\ t = t'$ **if** C, where the **condition** C is a finite set of unquantified Σ-equations involving only variables in X (when $C = \emptyset$, conditional Σ-equations are regarded as ordinary Σ-equations) □

For conciseness, sometimes we use variations on the notation for equations: (X, l, r, C) stands for $(\forall X)\ l = r$ **if** C; (l, r, C) is used when $X = \emptyset$; and we write $l = r$ if $X = C = \emptyset$.

2.5 Order-sorted Equational Deduction

This section gives rules of deduction for OSA with conditional equations. This yields a construction for initial and free order-sorted algebras as quotients of term algebras by the congruence generated by the rules of deduction from given equations. The details can be found in [19]; here we just introduce the rules of deduction.

Given an order-sorted signature $(S, \leq, \Sigma)$ and a set Γ of conditional Σ-equations, we consider each unconditional equation in Γ to be **derivable**. The following rules allow deriving further (unconditional) equations:

(1) *Reflexivity*. Each equation of the form
$$(\forall X)\ t = t$$
is derivable.

(2) *Symmetry*. If
$$(\forall X)\ t = t'$$
is derivable, then so is
$$(\forall X)\ t' = t.$$

(3) *Transitivity*. If the equations
$$(\forall X)\ t = t',\ (\forall X)\ t' = t''$$
are derivable, then so is
$$(\forall X)\ t = t''.$$

(4) *Congruence*. If $\theta, \theta' : X \to \mathcal{T}_{\Sigma}(Y)$ are substitutions such that for each $x \in X$, the equation

[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 [31].

$(\forall Y)\ \theta(x) = \theta'(x)$

is derivable, then given $t \in \mathcal{T}_\Sigma(X)$, the equation

$(\forall Y)\ \theta(t) = \theta'(t)$

is also derivable.

(5) *Substitutivity*. If

$(\forall X)\ t = t'$ if C

is in Γ, and if $\theta : X \to \mathcal{T}_\Sigma(Y)$ is a substitution such that for each $u = v$ in C, the equation

$(\forall Y)\ \theta(u) = \theta(v)$

is derivable, then so is

$(\forall Y)\ \theta(t) = \theta(t')$

Although these rules are rather compactly formulated, they correspond exactly to intuitions that we feel should be expected for equational deduction. Of course, there are many possible variations on this rule set; for example, see [41].

Given a set of equations Γ, there is a congruence $=_\Gamma$ relating two terms iff we can prove that they are equal from the equations in Γ and applications of the rules above. 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 (this can always be freely chosen without problems [20], so we do not give any more details of its definition).

Lastly, note that the concepts introduced here can be easily extended to considerparsed terms.

2.6 Theory Presentations

Specifications are modelled by the concept of theory presentation.

Definition 2.6 An **order-sorted theory presentation** (hereafter, presentation) is an ordered 4-tuple, $(S, \leq, \Sigma, \Gamma)$, where $(S, \leq, \Sigma)$ is an order-sorted signature and Γ is a set of Σ-equations. □

For a presentation $P = (S, \leq, \Sigma, \Gamma)$, we let $t =_P t'$, $[t]_P$, and $[\![t]\!]_P$ respectively mean $t =_\Gamma t'$, $[t]_\Gamma$, and $[\![t]\!]_\Gamma$. Also, given a signature $(S, \leq, \Sigma')$, we use $P \cup \Sigma'$ for the presentation $(S, \leq, \Sigma \cup \Sigma', \Gamma)$.

Now, we extend the definition of presentation to allow non-monotonic operations.

Definition 2.7 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 such that $\Sigma - \Omega$ is monotone. □

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 related operations don't necessarily agree on the intersection of their arities). The default equations are in the form: $\sigma.ws(\bar{x}) = \sigma.w's'(\bar{x})$, 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 $i = 1 .. k$, $\bar{x}$ stands for $x_1, \ldots, x_k$, and $w = LS(x_1), \ldots, LS(x_k)$. We let Γ^* be the union of Γ with default equations.

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

3 Overview of FOOPS

FOOPS extends OBJ with some concepts from object-oriented programming. This motivates two central design decisions (see [22, 18, 40, 42] for more details about FOOPS design): data elements are not objects, and classes are not modules.

The first distinction is based on the fact that data elements (e.g., natural numbers) are stateless, but objects (e.g., buffers) have an internal state that can change with time. In this way, FOOPS provides different constructs for defining abstract data types and classes of objects. Consequently, there are two constructs for specifying inheritance. In fact, overloading, polymorphism, and inheritance are also available for the specification of ADTs, by the definition of subsorts (subtypes).

The second design decision recognizes the necessity to have a construction where related classes and abstract data types can be defined together. In FOOPS, this is provided by modules, which are the main programming unit of the language. This is one of the main aspects of FOOPS (also derived from OBJ); it includes a powerful module interconnection language, supporting parameterized modules with semantic interface requirements, which allows the programming style known as "Parameterized Programming" [10].

Further justification for both decisions is given in [22], where this approach is compared with others.

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. Essentially, programs are written in modules and specifications are written in theories. Furthermore, theories are also used to specify the syntactic and semantic restrictions that must be satisfied by the actual arguments of a parameterized module. In order to specify how a theory is interpreted (satisfied) by another theory or module—necessary, for example, when instantiating a parameterized module—the language provides views, which are bindings indicating how the classes, sorts, and operations symbols of a theory are interpreted in another theory or module.

The acronym FOOPS stands for Functional and Object-oriented Programming System, but we usually use it for the language provided by the system. FOOPS was first presented in [18], but [40, 42] describes the language in detail, including some ideas about different approaches for its formal semantics (reflective semantics based on order-sorted algebra [18, 19], hidden order-sorted algebra [11, 17], and sheaf theory [44, 14]).

Here we briefly describe the functional level of FOOPS and some of its parameterized programming features. Following this, we give a detailed description of the object level and intuitions about its operational semantics.

3.1 Functional Level

The functional level of FOOPS is a syntactical variant of OBJ. At this level it is possible to define abstract data types, which are sets of data elements together with associated operations. A FOOPS functional module defines one or more abstract data types, where the keywords **sort** and **fn** respectively introduce the name of the set of data elements, and the associated operations (functions) symbols.

A very simple functional theory is

```
fth TRIV is
```

```
    sort Elt .
  endfth
```

It introduces the sort `Elt`, but it has no constraints about the operations associated to it. Hence, the only requirement that actual arguments to a module parameterized by `TRIV` must satisfy is to have a defined sort.

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

```
  fmod LIST[E :: TRIV] is
    pr NAT .
```

This module is parameterized by the sort of the elements in a list (parameter `E`). In order to define an operation giving the number of elements in a list, we use a built-in module defining natural numbers: the keyword `pr` indicates that the module `NAT` is imported and we don't add or identify data elements of the sorts defined in the module.

The following declaration introduces a sort for nonempty lists and another for lists,

```
    sorts NeList List .
    subsorts Elt < NeList < List .
```

where elements are considered singleton (nonempty) lists and nonempty lists are, ofcourse, lists, as indicated by the subsort relationship (<). This is what specifies inheritance at the functional level: for example, as all elements of `Elt` are elements of `List`, all functions associated to `List` can also be used for the elements of `Elt`.

The empty list is represented by the constant `nil`, and `_._` denotes the function that concatenates two lists.

```
    fn nil : -> List .
    fn _._ : List List -> List [assoc id: nil] .
    fn _._ : NeList List -> NeList .
    fn _._ : NeList NeList -> NeList .
```

The underscores in `_._` serve as placeholders for the arguments of this function. Hence,

```
  nil . nil
```

is a well formed term; that is, the application of `_._` to `nil` and `nil`. Note that `_._` is overloaded, and concatenation of nonempty lists results in a nonempty list. As indicated by the attributes, this function is associative (`assoc`) and has `nil` as identity (`id: nil`).

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. These are introduced by the following declarations:

```
    fn head : NeList -> Elt .
    fn tail : NeList -> List .
    fn #_ : List -> Nat .
```

The functions **head** and **tail** are only defined for nonempty lists. This gives the effect of partial functions, by defining them as total on specific subsorts restricting their domain.

The meaning of those functions 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:

```
      var E : Elt .
      var L : List .
      ax head(E . L) = E .
      ax tail(E . L) = L .
      ax # nil = 0 .
      ax # (L . E) = # L + 1 .
   endfmod
```

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

Instead of writing the first axiom for **#_**, we could have written the equivalent conditional axiom (**cx** indicates that the axiom is conditional):

```
      cx # L = 0 if L == nil .
```

where the condition for which the axiom is valid (or may be applied) follows **if**. Note that every module in FOOPS automatically imports a built-in module of booleans containing the usual operations, and the overloaded equality (_==_) and inequality operations (_=/=_).

3.1.1 Retracts

An interesting point of FOOPS is how expressions (terms) such as **head(tail(l))**, for a given term **l** of sort **NeList** (written **l:NeList**), are parsed. In fact, **tail(l):List**, but **head** requires an argument of sort **NeList**. Thus, we should conclude that **head(tail(l))** is not a well formed expression. However, as **tail(l)** may be equal to an element of **NeList** (when **l** has more than one element), FOOPS is flexible enough to allow us write this kind of expression which is actually parsed as

```
      head(r:List>NeList(tail(l)))
```

where **r:List>NeList** is a special function, called retract, which lower the sort of an expression of **List** to the required subsort **NeList**. It is defined by

```
      fn r:List>NeList : List -> NeList .
      var NL : NeList .
      ax r:List>NeList(NL) = NL .
```

In this way, an expression formed by the application of a retract is only reduced if the argument of the retract has the required subsort. Otherwise, the retract remains as an error message, indicating that the expression is not well parsed. In FOOPS, retracts are automatically defined between related sorts, and inserted in expressions whenever necessary. In a similar way, retracts are also available at the object level.

3.2 Object Level

At tbe object level, it is possible to define classes, which are collections of (potential) objects with same attributes and methods. Attributes correspond to properties of objects, they represent the internal state of objects. Methods are operations that objects can perform; they modify the state of objects. In addition to modifying states, methods may also yield results.

Attributes are atomically evaluated. Methods may be atomically evaluated or not: invocation of methods is synchronous and can be understood as remote procedure calls. An object can be evaluating many non atomic methods at the same time, including different instances of the same method. Naturally, there are operators for controlling the interference of methods executing in parallel.

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. In this way, methods and attributes have at least one object identifier as argument. indicating which method is going to execute the corresponding operation.

An object level module defines one or more classes and related abstract data types. In addition to that, abstract data types defined in functional modules can be imported by object modules. This is how the two levels are integrated. Let's consider an object module BUFFER defining a class of bounded buffers. Tbis module is parameterized by the capacity of buffers (a positive natnral number), specified by the functional requirements theory MAX:

```
fth MAX is
  pr NAT .
  fn max : -> NzNat .
endfth
```

where the sort of positive natnral numbers is represented by NzNat (it's defined in NAT). The module BUFFER is also parameterized 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 bnffer are stored in a list; so, it is necessary to import the functional module LIST, instantiating it with the argument module giving the sort of elements. In this instantiation, no view is specified since there is a trivial interpretation—the identity—from the theory constraining the arguments of LIST (i.e., TRIV) to the theory constraining E.

The class Buffer of bounded buffers is introduced by the declaration

```
  class Buffer .
```

Inheritance could also be defined at the object level, by a subclass declaration (similar to subsort). This implies that any attribnte or method associated to a class is also available to its subclasses, since objects of a subclass are also objects of an associated superclass.

Attributes are defined as operations from an object identifier to a value that denotes a current property of the related object. Multi-argument attributes have other argnments in addition to an identifier; this means that this attribute's associated property depends on the extra arguments. Objects of Buffer have tbe attribute elems, corresponding to the list of elements in a buffer.

```
    at elems_ : Buffer -> List [hidden] .
```

As indicated by the declaration `[hidden]`, the attribute **elems** is only visible inside **BUFFER**; so, clients of the objects of **Buffer** cannot directly look at the elements stored in buffers. Alternatively, we could have added the declaration:

```
  hidden elems_ : Buffer -> List .
```

In addition to **elems**, two more attributes are associated to **Buffer**:

```
    at empty?_ : Buffer -> Bool .
    at full?_ : Buffer -> Bool .
```

The attribute **empty?** indicates whether the buffer is empty, whereas **full?** indicates whether 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 the identifier of the object that performs it. For objects of **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. The following declarations introduce those methods:

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

The last one is hidden 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 considered a stored attribute.

For **Buffer**, we define **elems** as a stored attribute. The others are derived; so, we introduce the following equations:

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

This indicates that the buffer is empty if the list of the elements stored in it is empty; also, the buffer is full if the size of its associated list is **max**.

Equations defining attributes can only contain functions, attributes, and object identifiers. This kind of equation is 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 band side (LHS) indicates its associated attribnte and method, whereas its right hand side (RHS) is an expression specifying the new value for the attribute to be npdated. For instance, the behavior of **reset** is given by tbe DMA

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

which specifies that after the execution of **reset** 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 metbods are only executed if the (enabling) conditions are satisfied; otherwise, the evaluation is suspended. 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 doesn't update that attribute. This is called the frame assumption; it avoids writing equations indicating that some attributes are not updated.

The evaluation of methods specified by DMAs is atomic and yields the identifier of the object which executes the method; only this objects is modified, and its attributes are updated as specified. As for attribute equations, 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 metbods. IMAs are equations that specify how a metbod is defined in terms of other operations; this is indicated by a method expression, i.e., an expression formed by methods, attributes, functions, object identifiers, and method combiners (operators on method expressions). For example, the method **get** is specified by the IMA

```
    acx get(B) = result head(elems B) ; del(B) if not(empty? B) .
```

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

Similarly to DMAs, no method symbol or method combiner is allowed in an IMA's condition. IMAs are interpreted as left-to-right rewrite rules. Whereas the evaluation of the IMA's condition is atomic, the evaluation of the IMA's RHS is not atomic and may be interfered by the execution of other methods. However, atomicity can be achieved by using the atomic evaluation operator `[_]`, which atomically executes its argument, without interference from the execution of other methods. We assume that IMAs introduced by the keyword **acx** (or **aax**) have their condition and RHS atomically evaluated. In fact, an IMA in the form

```
  aax m(O) = e .
```

for `m : C -> C'`, is an abbreviation for

```
  ax m(O) = [e] .
```

and an IMA like

```
acx m(O) = e if c .
```

stands for the following declarations:

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

where m' is a new symbol. This is necessary if an expression has to be evaluated without interference from others. Sometimes, non atomic methods are useful, mainly when efficiency is essential; but they shouldn't be arbitrarily used because it's very difficult to reason about programs consisting of the parallel execution of many non atomic methods (it's necessary to reason about all possible interleaved interferences caused by those methods).

Lastly, we introduce a (weak) class invariant to **Buffer**; that is, a condition that must be satisfied for all objects of the class, independently of their state. In order to express that all bounded buffers can have at most **max** elements, we introduce the declaration

```
  inv #(elems B) <= max .
endomod
```

In fact, this kind of invariant is also considered valid if all attributes used in the predicate are not defined. For example, immediately after an object of **Buffer** is created, **elems** has no associated value (the built-in object creation operation doesn't initialize attributes, see Section 3.2.3); even so, we consider that the invariant is valid in this initial state

After creating a buffer, the only possible operation is **reset** because **elems** is not defined; the method **reset** clearly enforces the invariant, since **# nil** is **0**. It's also easy to check that the other operations related to **Buffer** preserve this invariant

A stronger kind of invariant requires all attributes used in the predicate to be defined. (This can be introduced by the keyword **str-inv**, instead of **inv**.) For example, the predicate

```
  #(elems B) <= max
```

isn't a *strong* invariant for **Buffer**, since the object creation operation doesn't respect it. In this case, we would have to hide this creation operation and introduce a customized operation that enforces that invariant. Note that in order to check whether a strong invariant is preserved for an object **o**, we should consider the effect caused by the deletion of other objects in the system, since some attributes of **o** might be undefined after that.

Constraints like class invariants are just annotations, they have no effect for the semantics of a FOOPS module. In fact, they just document properties of a given specification. They can be seen as proof obligations which, if discharged, might help a lot to reason about specifications.

3.2.2 Method Combiners

In addition to **result_;_** and **[_]**, FOOPS provides other method combiners: sequential composition, _;_ (interleaving) parallel composition, _||_; (external) nondeterministic choice, _[]_; background evaluation, **_&_**; and conditional, **if_then_else_fi**.

The semantics of these combiners is given later. Here we informally describe some of them; we suppose the reader has a general intuition about the others, since they are usually available in other programming languages.

Result

The resnlt method combiner (**result_;_**) fully evaluates its first argument (from left to right) and then evaluates the second. When both arguments are fully evaluated, the first one is given as resnlt.

This operator is mainly useful when an expression should yield a specific value, but this value has to be evaluated before other operations are executed. For example, consider the method **get**, defined in Section 3.2.1. Its behaviour could not be easily expressed without **result_;_**

Indeed, **result_;_** can be used to simulate some of the behaviour provided by the **return** statement in languages such as C and C++, and special conventions for variables names in Pascal and Eiffel for indicating the value to be returned by a function For instance, the C++ code corresponding to the "FOOPS like" method definition

```
m(O) = e ; X := f ; g ; return X .
```

and the Eiffel code corresponding to

```
m(O) = e ; Result := f ; g .
```

could be represented in FOOPS by

```
m(O) = e ; result f ; g .
```

where e, f. and g are method expressions.

Method Combiner Definition

New metbod combiners may be introduced as abbreviations for complex method expressions. This can be done by equations. For example, the internal nondeterministic choice operator **_Or_** is defined in terms of external choice by the axiom

```
ax P Or Q = (skip ; P) [] (skip ; Q) .
```

where P and Q are variables, and **skip** is any functional constant. In this axiom, the arguments of _[]_ may be immediately evaluated; so, the external choice will be nondeterministic. As desired this implies that the internal choice doesn't depeud whether its arguments are ready for evaluation or not.

Evaluation in the Background

Here we introduce a method combiner that resembles the UNIX operator **&** for evaluation of a program in the *background*. This means that the operator starts the evaluation of an expression but doesn't wait until it terminates. Instead, expressions following this operator are evaluated concurrently to the expression in the background. Also, the result generated by the expression in the backgronnd is discarded; this expression is only executed for its side-effects.

The FOOPS metbod combiner **_&_** starts the evaluation of its second argument (from left to right) in the background, and then yields its first argument. In fact, the UNIX unary postfix operator **_&** may be defined by

```
ax P & = skip & P .
```

In FOOPS, `P &` is a method expression (not a command or program, in UNIX terminology [5]), so it must yield a value; that's the role of the dummy constant **skip** in the axiom above.

The main application of this operator is to start the execution of non terminating methods. For example, if `m` is non terminating, invoking `m` like in `m(o) ; n(o)` is not very useful because `n(o)` will never be evaluated. Instead, we can use `m(o) & n(o)`. In this way, the evaluation of `m(o)` starts and `n(o)` is concurrently evaluated.

3.2.3 Object Creation and Deletion

Dynamic object creation and deletion are respectively provided in FOOPS by the following operators:

- `new.C() : -> C`,
- `new : C -> C`, and
- `remove : C -> C`,

for each class `C`.

For a given class `C`, the operator `new.C()` creates an object of `C` with a nondeterministically choosen identifier that is not already being used for another object This identifier is given as the result of the evaluation of the operator.

The operator `new` creates an object of the same class as the object identifier given as argnment, 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.

The operator `remove` receives an object identifier as 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. Contrasting to `new`, the argument of `remove` might be an arbitrary expression, it doesn't have to be an object identifier; however, it's supposed to yield an identifier.

The operators for object creation don't assign initial values for attributes Hence attributes should be explicitly set by special methods, since a non initialized attribute cannot be evaluated. Automatic initialization is not provided here because it can be easily simulated by the operators introdnced above together with method combiners and methods for setting attributes. For example, suppose that a class `C` has two stored attributes `a` and `a'`, and methods `set-a` and `set-a'` for assigning values to those attributes. A creation operation for `C` that also initializes those attribntes is given by the method combiner `create`, defined by

```
ax create(O) = [new(O) ; set-a(O,v) ; set-a'(O,v')] .
```

where `O` is a variable of class `C`, and `v` and `v'` are chosen initialization values for the respective attributes. The atomic evaluation operator guarautees that the created object can only be accessed after its attributes are initialized. Similarly to `new`, the operator `create` is not executed by an object; in fact, it should be executed even if its argument is an object identifier that is not in the state. So, it is modelled as a method combiner Also, in order to behave properly, `create` should only be invoked with an object identifier as argument.

Contrasting to the simplicity of the approach used above, it might be problematic to initialize objects of recursive classes, multi-argnment attributes, and to find default values for attributes in general. In fact, it might be the case that some attributes cannot be automatically initialized. That's another reason for not trying to automatically initialize attributes.

We can also easily simulate creation operations having attribute initialization values as arguments. For instance, the method combiner

```
mc create(_,a = _,a' = _) : C S S' -> C .
```

can be used to create objects of class C, assigning the values received as arguments to the attributes a and a' (respectively assumed to be of types S and S') This is formalized by the following axiom:

```
ax create(O,a = V,a' = V') =
      [new(O) ; set-a(O,V) ; set-a'(O,V')] .
```

where O:C, V:S, and V':S'.

Auto-methods

Auto-methods are automatically invoked in the background when objects of their corresponding classes are created. They may be used to define initialization operations for objects, but their main application is the specification of autonomous (active) objects: that is, objects that automatically perform some operations, instead of waiting for requests from other objects. In fact, autonomous objects can simulate (non terminating) processes in an object-oriented framework.

Here auto-methods can be modeled by standard methods and the operator for background evaluation. For this, we have to provide a customized operation for creation of objects, based on the pre-defined operation **new**. Basically, this customized operation should invoke **new** with the related auto-methods as expressions to be evaluated in the background.

For example, suppose that we want to define m, associated to class C, as an auto-method. The customized creation operation could be defined by

```
ax create(O) = new(O) & m(O) .
```

where O is a variable of class C. This operation creates an object of C with the identifier given as argument and then invokes m. This method may be an initialization operation or a non terminating method like

```
ax m(O) = n(O) ; m(O) .
```

In this case, the object behaves as a process which is always executing n.

Clearly, this corresponds to the intuitions about auto-methods discussed above.

3.2.4 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 [40].

First, FOOPS uses the convention that method and attribute applications are evaluated bottom-up. This means that a method or attribute can only be executed if its arguments are fully evaluated, i.e., the arguments cannot contain any attribute, method, retract, or method combiner symbol. They must be real values: evaluated functional terms or object identifiers. Other evaluation strategies are not appropriate because symbolic method or attribute execution does not make sense for objects: a method or attribute can only be executed when it has real arguments. However, the order in which the argnments of a method or attribute are evaluated is not fixed. Observe that this is a source of nondeterminism, since tbe arguments may be evaluated in different contexts.

An 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 tbe new operation symbol and rank (i.e., arguments and result type). As FOOPS provides dynamic binding, objects of the subclass use the new version of the operation, unless explicitly stated that the original version is desired; this 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 arity of the specialized version must be smaller or equal to the arity of the original version, and the resnlt of the first must be greater or eqnal to the result of the second (see [42] for details) A specialized version of a redefinition of an operation is considered to be a redefinition as well.

Lastly, objects may be introdnced together with the definition of their associated class, where values for their stored attribntes are specified. These are called specified objects and are particnlarly useful when defining classes of recursive data strnctnres such as stacks and linked lists. Specified objects have the same status as objects created at rnntime; this means that they can be modified and removed.

3.2.5 Protected Objects

In FOOPS, we can create objects that are protected from some other objects, in the sense that a protected object only executes methods directly invoked by a specific and selected group of objects. Roughly, this corresponds to the behaviour provided by hiding and abstraction mechanisms in process algebras, where a process might not be allowed to access some protected channels.

Object protection facilitates programming and reasoning with references (like object identifiers, and pointers in procedural languages) by reducing possible interferences to objects; this is done by restricting the objects that are allowed to request the execution of methods of a protected object. Also, by having arbitrary interference usually one cannot provide full encapsulation of complex objects nor the desired system behaviour; so, the system specification should include explicit, artificial code for avoiding nndesirable interferences. However, it seems more appropriate to directly support a mechanism for object protection.

For instance, object protection is quite nseful for defining linked lists of cells representing a sequeuce, because the intermediate cells in the list shonld only be accessed by their respective previous cell [26]. In fact, only the first cell in the list should accept arbitrary interference. The intermediate cells shonld be protected. Another example is a simple communication protocol, consisting of two agents and a channel used for communication between them. In this case, the channel should only be accessed by the two agents; it should he protected from other objects, which could disrupt the communication The channels are truly encapsulated only if they are protected; only in this case the protocol can be seen as a "black box" and then reused without restrictions about tbe environment where it's going to be used.

In particular, one application of object protection is the definition of constant objects; that is, objects that always in the same state. This can be obtained by creating an object that is protected from any other object. In this way, constant objects cannot be removed as well. This might be useful for the definition of recursive data structures like linked lists and stacks, where a constant object representing the empty list or stack is nsually necessary.

In order to use the mechanism for object protection, we should indicate which objects are allowed to directly request the execution of methods associated to a protected object. This is done at object creation time, by giving a set of object identifiers as argnment to **new** The empty set means that the created object cannot exccute any method. Alternatively, **any** may be given as argument, meaning that any object can directly invoke methods of the created object.

Specified objects have a default object protection statns that cannot change: no object can invoke metbods of a specified object. Iu fact, specified objects are constant objects.

For supporting object protection, tbere are special object creation operations:

- `new.C : Univ -> C`, and
- `new : C Univ -> C`,

for a class `C`, where `Univ` is the type associated to the sets of object identifiers given as argument to the creation operations. In fact, the operators for object creation introduced in Section 3.2.3 can be seen as abbreviations for the operators introdnced in this section. Indeed, `new.C()` corresponds to `new.C(any)`, and `new(o)` is the equivalent of `new(o,any)`.

For indicating the desired protection, the following syntactic constructors are available: `_++_`, `{}`, and **any**, where the first one may be used for adding an element to a set, and the second one denotes the empty set. For example, the expression `o ++ o' ++ {}` denotes the set formed by the identifiers `o` and `o'`.

New objects may be dynamically added to the group of objects that is allowed to invoke methods of a protected object, if the object that requests this operation is part of this gronp. For doiug that, there is a special operation: `addpr : C Univ -> C`, for any class `C`, which adds the objects specified by its second (from left to right) argnment to the collection of objects that can invoke methods of the object identified by its first argument. The second argument to the operatiou above should be constructed with the syntax constrnctors nsed for indicating the desired object protection for the creation operation.

A special case of the mechanism for object protection introduced here is provided by languages supporting composite objects (i.e., objects that incorporate others objects, instead of having refereuces to them). Composite objects can be modelled in FOOPS by indicating that the incorporated objects can only be accessed by the object that iucorporates them. Also, the notation introdnced in [26] snpports private refereuces, which can be nsed by only one object, giving a similar effect to composite objects. (In particular, [26] emphasizes the essential role played by private references for assertiug invariants about object graphs and reasoning abont them.) Our mechanism for object protection is clearly more general than the mechanism for private references.

3.2.6 Aspects of FOOPS Operational Semantics

Here we informally describe some aspects about FOOPS operational 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 evalnatiou of attribntes, and modified by the execution of methods or by the deletion and creation of objects. Modifying this information changes the database state.

Motivated by [18], here we represent a state of the FOOPS database, for a specification Sp, by an order-sorted presentation (with a signature of non-monotonicities) formed by the following components: 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 and subsort relationships in Sp. Using this abstract representation for states, typical operations on states are defined in a natural and simple way.

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, part of a 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 when 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 following declarations indicate that the class Buffer has three objects in this state (identified by b1, b2, and b3):

```
  fns b1 b2 b3 : -> Buffer .
  ax elems b1 = nil .
  ax elems b2 = (1 . 2) .
endfth
```

where the equations specify values for their stored attributes. Note that b3 hasn't been initialized.

Using this representation, typical operations on states can be easily defined. For instance, 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, this can be deduced by equational reasoning, from the equations in STATE1.

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

```
fth STATE2 is
  ⋮
  ax elems b1 = 5 .
```

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

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

Also, adding the object **b4** to **STATE2** results in a state with one more constant of sort **Buffer**:

```
  fth STATE3 is
    ⋮
    fns b1 b2 b3 b4 : -> Buffer .
    ax elems b1 = 5 .
    ax elems b2 = (1 . 2) .
  endfth
```

On the other hand, removing the object **b2** from **STATE3** yields a state in the form

```
  fth STATE4 is
    ⋮
    fns b1 b3 b4 : -> Buffer .
    ax elems b1 = 5 .
  endfth
```

The object and its related equations were removed from the database.

Remember that the value of a redefined attribute usually doesn't agree with the value of its original version for objects of the subclass. So, the same should be allowed for the functions modelling those attributes in database states. That's why we nse order-sorted presentations with a signature of non-monotonicities to model database states; just order-sorted presentations are not adequate for doing that in an elegant way. For example, a specification containing

```
  pr NAT .
  subclass C < C' .
  at a : C' -> Nat .
  at a : C -> Nat [redef] .
  var X' : C' .
  var X : C .
  ax a(X') = 0 .
  ax a(X) = 1 .
```

should have database states in the form

```
  ex NAT .
  subsort C < C' .
  fn a : C' -> Nat .
  fn a : C -> Nat [redef] .
  var X' : C' .
  var X : C .
  ax a.C'Nat(X'.C') = 0.Nat .
  ax a.CNat(X.C) = 1.Nat .
```

where parsed equations are used to avoid ambiguity. This allows both versions of `a` to have different definitions without generating any inconsistency. On the other hand, if states were represented by order-sorted presentations without a signature of non-monotonicities, unparsed equations would be used, being possible to prove that `0` is equal to `1` using the equations defining `a`. This is obviously not desirable; it would also mean that the semantics of the functional level is affected by the semantics of the object level.

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) that specify properties of the elements of the related signature.

Later, we show that signatures should also provide information about method combiners and other features supported by FOOPS. Now, we just give a simplified definition which will be extended when necessary.

Definition 4.1 A FOOPS **signature** consists of

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. The sort `Bool` (for boolean) is in S.

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 $\Sigma = F \cup A \cup M$, where F, A, and M respectively contain names for functions, attributes, and methods. Functions are related to sorts: $F_{w,u} = \emptyset$ if $wu \notin S^+$; F has the standard boolean operations; attributes have one class parameter at least: $A_{w,u} = \emptyset$, if $w \in S^*$; specified objects are related to classes: $M_{\lambda,u} = \emptyset$, if $u \in S$; methods have a class parameter: $M_{w,u} = \emptyset$, if $w \in S^+$; and there are retracts

   $$\texttt{r:A>B : A -> B} \in Retr_{\texttt{A},\texttt{B}},$$

   if $\texttt{A} \cong \texttt{B}$, where $Retr \subseteq (F \cup A)$. Lastly, a method and an attribute with related ranks cannot have the same name, in order to avoid mixing methods up with attributes (i.e., for any $\sigma \in A_{w,u}$, there are no w' and u' such that $wu \leq w'u'$ or $w'u' \leq wu$, and $\sigma \in M_{w',u'}$).

4. A family $R \subseteq A \cup M$ formed by names of redefined methods and attributes. As specialized versions of redefined operations are considered to be redefined, if $\sigma \in R_{w,u} \cap \Sigma_{w',u'}$ and $w'u' \leq wu$ then $\sigma \in R_{w',u'}$.

□

Sometimes, we use Σ to denote the signature $(U, \leq, \Sigma, R)$. Furthermore, we rely on the fact that a FOOPS signature Σ can be seen as the order-sorted signature $(U, \leq, \Sigma)$. In this way, the notation and concepts related to order-sorted signatures (e.g., terms, least sort, equations, etc.) are available for FOOPS signatures as well.

The constraints imposed on the components of a signature correspoud to some of the restrictions enforced on FOOPS modules. For instance, as a module defining the abstract data type of booleans is automatically included in any FOOPS module (for allowing conditional equations), signatures must have a sort `Bool` with its associated operations. Also, the restriction on R is enforced on the operations of FOOPS modules, in order to avoid the problem discussed at Section 3.2.4.

In some cases, a more general approach is used, by not imposing restrictions on signatures components. So, the semantics of some constructs is indirectly given, by translation to more general constructs. As long as this approach doesn't complicate the semautic definition, we use it and indicate how the translation can be done. For example, if booleans weren't assnmed to be in signatures, we would only be able to give the semantics of equations having a set of pairs of terms as a condition (following OSA), instead of a boolean expression (as actually supported by FOOPS). In this case, we would have to specify how the second kind of equation can be seen as a particular case of the first.

The generalization mentioned in the example above would slightly complicate the semantics. Hence, we don't use it. However, the semantics for qnalification notation for redefined operatious (i.e., `m.C` where `m` is redefined and `C` is a class name) is indirectly given and doesn't affect the semantics. Basically, we don't assnme that signatnres include a special (qualified) operation name for each redefined operation. Instead, we consider that the FOOPS signature correspouding to a module containing

```
subclass A < A' .
subclass B < B' .
subclass C < C' .
me m : A' B' -> C' .
me m : A B -> C [redef] .
```

is the same as the signature associated to a module with the declarations above plus the following one:

```
me m.A' : A' B' -> C' .
```

which provides a qualified notation for `m`. We assume that symbols containing “ ” cannot be used as operation names in FOOPS modnles (unless it corresponds to the qualified notation of some operation). Hence, this additional declaration doesn't introduce any conflict and `m.A'` can be used to access the original version of `m`.

The same technique can be used to support qualified notation for attributes. However, note that it's meaningless to ask for the original version of a redefined stored attribute for an object of the subclass, since it has no associated value (it's neither directly stored in the state nor necessarily equal to the specialized version). The same happens for original versions of derived attributes defined in terms of redefined stored attributes. Hence, we don't need to provide a qualificatiou notation for this kind of attribute.

An obvious motivation against allowing methods and attributes to have the same name and related ranks is that two operations cannot be distinguished if they have the same name and rauk. A less obvious motivation is illnstrated by the following signature:

```
pr NAT .
subclass A < A' .
```

```
at a : A' -> Nat .
me a : A -> Nat .
```

In this way, a may be interpreted as a method or as an attribute, depending on its argument. Indeed, in some expressions, it might be the case that a is parsed as an attribute and, after some argument evaluation, it's parsed as a method. Besides being confusing, this has bad consequences, mainly for expressions in the RHS of DMAs and conditions of equations, where no methods are allowed. For instance, for a method m, the following

```
var X : A' .
cx m(X) = ... if 0 <= a(X) .
```

is a valid equation because a is parsed as an attribute in a(X). However, if m is not redefined, the evaluation of m(o), for o of class A, requires the evaluation of the condition 0 <= a(o) which requires one method invocation, since a is parsed as a method in a(o). But this is not supported by FOOPS—there is no reasonable semantics for that. The same problem happens if this confusion occurs in the RHS of a DMA, or if an attribute updated by a DMA is sometimes parsed as a method.

In fact, this problem can also happen with pathological non regular signatures satisfying the restriction discussed above, but not with regular signatures (the ones that are of interest for specifications, as we will see later). Also, observe that there is no confusion between functions and methods (or attributes) because functions ranks are not related (by $\leq$) to method (or attribute) ranks, since they only have sorts.

Lastly, we could have added one more constraint on signatures: an operation should only be in R if it's overloaded by another with a greater rank. However, by analysing the semantic rules given here, we conclude that this constraint doesn't affect the semantics; that is, the meaning of a non overloaded operation is the same whether it's considered redefined or not. Hence, we don't introduce this restriction.

Now, we give a definition of specification.

Definition 4.2 A FOOPS **specification** is formed by a signature Σ and a set E of Σ-equations containing standard equations for the boolean operations (as given in [21], for example) and retract equations `r:A>B(X) = X`, for any types `A` and `B` in the same connected component of U (with respect to $\leq$), and a variable `X` of least type `B`. □

Hereafter, we use A to refer to the family of attributes in a specification S, when S is clear from the context; otherwise, we use the notation $A(S)$. The same convention is used for the other components: E, S, C, F, etc. Also, FE and AE respectively denote the set of functional and attribute equations in E. The set of functional equations is defined as the largest set of F-equations included in E, whereas AE is defined as the largest set of $F \cup A$-equations included in E and disjoint from FE.

Implicitly, the last definition requires Σ to be regular, since E cannot be empty and (unparsed) equations only make sense for regular signatures (see Section 2.4). As we have seen in Section 2.3, regularity is not a big restriction. Furthermore, this guarantees the least type (parse) for method expressions; so, we can work with untyped terms without ambiguity, leading to a conciser and simpler semantic description.

Observe that the definition above does not restrict the form of axioms. However, by the operational semantics that is described here (as we will see later), only DMAs and IMAs determine the behaviour of methods; other kinds of method axioms are irrelevant for the semantics

It's also important to note that equations may be non sort (or class) decreasing. This flexibility is usually desired for specificatious in general. It affects the semantics because the evaluation of an expression might then lead to an expression of a greater sort. If this happens and the expression is the argument of an operation that cannot be applied to an argument of greater sort, the evaluation will result in a non well formed term. The solution for this problem is given in later sections, bnt it basically consists of iuserting retracts (in order to lower the sort of expressions) during the evaluation of arguments and some pre-defined method combiners

Even if *method* and *attribute* equations were sort decreasing, retracts would have to be included in every specification, in order to avoid the problem discussed above. This happens because attribnte evaluation is specified in terms of an operation which gives the representative of an equivalence class (see Section 2.5); for specifications in general, we cannot guarantee that an eqnivalence class has a term with a smaller type than all other terms. Hence, attribute evaluation might yield a term of greater type, or even of a non related type

On the other hand, retracts could be avoided if *all* equations were sort decreasing. For this to work, representatives should be of a least type This would be feasible because we would consider that functional equations were sort decreasing as well.

Sometimes we use specification (siguature) when we refer to a FOOPS module. In those cases, we actually mean the specification (signature) corresponding to that module

5 Database States

As discussed in Section 3.2.6, 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 S there is an associated C-sorted family I_S (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 identifiers of objects 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. From an implementation point of view, this is essential for static type checking of expressions. Here we assume that I is provided by the FOOPS system.

In order to ensure least parse of terms, we assume that identifiers cannot have the same name as fuuctional coustants (formally, $|I| \cap F_{\lambda,u} = \emptyset$, for any $u \in S$). Lastly, the family I must contain the identifiers of the objects specified in S: $M_{\lambda,u} \subseteq I_u$, for any $u \in C$. Hereafter I may 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 $u \notin C$.

Definition 5.1 For a specification S, 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.

2. A signature $\Omega = R \cap A$ of non-mouotonicities, containing redefined attributes.

3. A set $DE = FE \cup AE \cup IdE$ 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[3].

□

As in the convention for specifications, we use D, DE, and Id for the corresponding components of a database state $\mathcal{D}$, when it is clear from the context; otherwise, we use $D(\mathcal{D})$, etc.

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 create, remove, or change the state of objects.

Strictly speaking, signatures of presentations representing database states should have a universal type, in order to guarantee local filtering—which is necessary to imply that equational satisfaction is closed under isomorphism [19]. However, any useful subset of FOOPS should include method combiners, which require the existence of a universal type (see Section 7). Hence, we assume that this is provided for any specification. Otherwise, it could be simply added to the signature of the definition above.

The restriction on the equations of database states is important to guarantee that constructs from the object level don't interfere with the semantics of the functional level. Otherwise, the functional theory associated to states would not be related to the specified functional theory. This would mean that the results yielded by expressions evaluated in states would not have the same meaning as the corresponding elements of the specified functional theory. If those restrictions on equations cannot be satisfied, the specification won't have any associated database state. This might happen if the equations in AE are contradictory. For instance, a specification including

```
pr NAT .
class C .
at a : C -> Nat .
var X : C .
ax a(X) = 0 .
ax a(X) = 1 .
```

where a is not redefined, violates the restriction because the attribute equations relate two functional expressions (0 and 1) which are not related by NAT (assuming this is a specification of the abstract data type of natural numbers). In fact, this specification has no associated database state.

Observe that some attributes may have no associated value in a particular database state (this implies that they cannot be evaluated). The restrictions on database states don't prevent this. An advantage of this flexibility is that new doesn't have to initialize attributes; as briefly discussed before, default values for attributes might not even be available for specifications such as

```
pr NAT .
class List .
at val_ : List -> Nat .
at next_ : List -> List .
⋮
```

[3]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 $Alg \mid F$.

which specifies a recursive class of linked lists. Also, the operation **remove** doesn't need to assign an *ad hoc* **nil** or **void** value to attributes containing the identifier of the object to be removed, in order to avoid dangling identifiers. Instead, after the execution of **remove**, those attributes have no associated value. For example, the deletion of **12** from a state in the form

```
  ⋮
fns l1 l2 : -> List .
ax val l1 = 4 .
ax next l1 = l2 .
```

simply yields a state in the form

```
  ⋮
fn l1 : -> List .
ax val l1 = 4 .
```

where the attribute **next l1** is simply not defined.

We let D_S be the family of all database states for a given specification S; that is, the family of all presentations (with a signature of non-monotonicities) that satisfy the requirements in Definition 5.1. for a fixed S. Note that D_S is not necessarily the family of all database states reachable from the initial one by execution of method expressions. Naturally, this family is contained in D_S.

Lastly, for a database state $\mathcal{D}$ and some $t \in \mathcal{T}_D$, we assume that 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 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

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. Here we introduce them. First, we define an operation that updates databases. Later, we give operations for adding and removing objects from databases.

5.1.1 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".

Definition 5.2 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). □

Assuming that l is the application of an attribute to argnments, $\Gamma \oplus (l, r)$ 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 5.3 A set Γ of unquantified, nnconditional Σ-eqnations is **contradictory** if it contains two different equations (l, r) and (l', r') snch that $l \equiv l'$ or $l \equiv r'$. □

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 syutactical in a similar sense. This is appropriate for our purposes in this text.

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

Definition 5.4 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. □

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 npdate database states.

Definition 5.5 The **overwriting** of a presentation (with a signature of non-monotonicities) $P = (S, \leq, \Sigma, \Omega, \Gamma)$ by a non contradictory finite set of unqnantified. nnconditional Σ-equations Γ'. denoted $P \oplus \Gamma'$, is the presentation $(S, \leq, \Sigma, \Omega, \Gamma \oplus \Gamma')$. □

5.1.2 Adding Objects to Databases

The operation $\cup$ adds some operation symbols to the signature of a presentation (see Section 2.6). So, it can be used to add objects to a database, withont any initialization, if the symbols represent object identifiers. In this way, $\mathcal{D} \cup Id$, for a $U^* \times U$-sorted family Id of object identifiers, adds the identifiers in Id to the database $\mathcal{D}$ of a specification S.

5.1.3 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 don't have any associated value in the resulting database. Here is the formal definition:

Definition 5.6 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. □

6 Methods, Attributes, and Functional Expressions

Now, we start to describe a structural operational semantics for the object level of FOOPS. In this section, we concentrate on the semantics of functional expressions, methods, and attributes. In the following sections, we progressively give the semantics of other language features.

Here we use the approach for operational semantics introduced in [39]. 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 concepts. First, for a specification S, we define the family $T_S = T_{\Sigma \cup I}$ of method expressions (without variables). By the definition of I, $\Sigma \cup I$ is regular whenever Σ is. Regularity of Σ is guaranteed by the definition of specification. This implies the existence of a least type (parse) for method expressions. That's why we can use untyped (unparsed) terms to define the semantics. The typing functions LS and ρ can be used whenever some type information is necessary.

Second, for a specification S, the semantics is given by a transition relation

$$\rightarrow_S \subseteq Conf(S) \times Conf(S)$$

on configurations, where $Conf(S) = T_S \times D_S$. 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, $\rightarrow_S$ is the least relation satisfying these inference rules

Note that there is no fixed relation between the object identifiers used in a method expression and the ones in the database state where the expression is going to be evaluated. So, even expressions with identifiers of non-existing objects may be evaluated. Naturally, this will only be successful if these identifiers are not necessary for the evaluation of the expression

For conciseness, we use $\rightarrow$ for $\rightarrow_S$, when S is clear from the context. Also, $P \rightarrow P'$ stands for $(P, P') \in \rightarrow$, where P and P' are configurations.

6.1 Functional Expressions

The semantics of the functional level of FOOPS is basically given in [9] and [28]. Following these, we introduce one rule for evaluating functional expressions. But first, we give the following notation, for a specification S:

- database states $\mathcal{D}, \mathcal{D}' \in D_S$; and
- functional terms, $v_i \in T_F$, for $i = 1..k$, for some natural number k. Also, $\bar{v}$ is an abbreviation for $v_1, \ldots, v_k$.

The evaluation of a functional expression is performed in one transition and yields the evaluated form of this expression. This is only done if the expression is not already in its evaluated form. Here we consider that the evaluated form of a functional expression is the representative of its equivalence class with respect to its related functional theory. The Rule **Fun** (for *fun*ctional) formalizes those aspects.

Rule 6.1 (Fun) For any $f \in F$,

$$\frac{}{\langle f(\bar{v}), \mathcal{D}\rangle \longrightarrow \langle [\![f(\bar{v})]\!]_{FE}, \mathcal{D}\rangle}$$

if $f(\bar{v}) \not\equiv [\![f(\bar{v})]\!]_{FE}$. □

From the rule above, we can observe that a function cannot be evaluated unless its arguments are functional terms. This restriction is essential to ensure that the operational semantics motivates a reasonable equality on method expressions; one that preserves functional equality. In order to illustrate the need for this restriction, consider the following specification:

```
pr NAT .
fn f : Nat -> Nat .
fn g : Nat -> Nat .
var X : Nat .
ax f(X) = X + X .
ax g(X) = 2 * X .
```

By equational reasoning, we can prove that `f(n)` is equal to `g(n)`, for any term `n` of sort `Nat`. Now, suppose that we extend the above functional specification with

```
class C .
me m : C -> Nat .
```

and assume that `m` has side effects. If functions can be evaluated with non functional arguments, usually `f(m(o))` won't be equal to `g(m(o))`, for some `o:C`, contradicting what we proved before about `f` and `g`; so, functional equality is not preserved. This happens because `f(m(o))` may be evaluated to `m(o) + m(o)`, whereas `g(m(o))` may be evaluated to `2 * m(o)`. Clearly, the evaluation of the first resulting expression invokes `m` twice, whereas the evaluation of the second invokes `m` just once. Moreover, the side effects and the generated results may be different, for each invocation.

6.2 Attributes

Before giving the semantics of attribute evaluation, we introduce some notation. For a specification S, hereafter consider arbitrary

- sort and class symbols $u, u', u_i, u'_i \in U$, for $i = 1 \,..\, k$; and
- object identifiers and evaluated functional terms not having retracts: $v_i \in \mathcal{T}_{(F-Retr)\cup I}$, for $i = 1 \,..\, k$, where $LS(v_i)$ is u_i, and $v_i \in \mathcal{T}_F$ implies $v_i \equiv [\![v_i]\!]_{FE}$. Also, we let v and v' be in $\{v_1, \ldots, v_k\}$, where $LS(v) = u$.

We write "fully evaluated term" to refer to object identifiers and *evaluated* functional terms not having retracts.

For simplicity, we consider that the first argument (from left to right) of an attribute or method is the identifier of the object that will perform the associated operation. As we haven't imposed a corresponding restriction on signatures, we should show how the semantics of the more general case is derived from the semantics which assumes that simplification. But this can be easily done (by changing the position of parameters, for instance), so we omit the details.

Attributes can only be evaluated if its associated object exists; so the first argument of the attribute should be the identifier of an object in the database state being used for evaluation. Also, the other arguments must be fully evaluated before 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; otherwise, the evaluation is suspended. For example, non initialized attributes cannot be evaluated. The Rule **Att** (for *att*ribute) formalizes those aspects.

Rule 6.2 (Att) For any $a \in A$,

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

if $a(\bar{v}) \in \mathcal{T}_{F\cup A\cup I}$, $v_1 \in Id(\mathcal{D})$, and $[\![a(\bar{v})]\!]_{\mathcal{D}\cup I}$ is in $\mathcal{P}_{F\cup Id(\mathcal{D})}$. □

The first condition guarantees that $a(\bar{v})$ is an attribute application; note that $a \in A$ is not enough to guarantee that, since a might belong to M as well. The second condition assures that the attribute's associated object exists. The last condition checks if the attribute has an associated value in the database. As we use $\mathcal{D} \cup I$ (instead of $\mathcal{D}$) in $[\![a(\bar{v})]\!]_{\mathcal{D}\cup I}$, 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 these identifiers are irrelevant to determine the value associated to that attribute.

By the convention introduced in Section 2.3, $[\![a(\bar{v})]\!]_{\mathcal{D}\cup I}$ stands for $[\![\rho(a(\bar{v}))]\!]_{\mathcal{D}\cup I}$, since $\mathcal{D}$ is a presentation with a signature of non-monotonicities. As ρ gives the least parse of an expression and the equations in $\mathcal{D}$ are parsed, only the most specific equations (the ones associated to a particular type of an operation) of $\mathcal{D}$ are used to define the value of $a(\bar{v})$. This means that this attribute is dynamically bound to the specialized version of its associated operation; on the other hand, attributes used in attribute equations are statically bound. In fact, using the theory of OSA in this way, we can only obtain a partial form of dynamic binding for derived attributes, whereas we can get a full form of dynamic binding for stored attributes.

6.2.1 Qualified Notation for Attributes

In Section 4, we have discussed a technique to support qualified notation for redefined attributes. Now, we give its semantics, by showing what's the specification corresponding to a FOOPS module having redefined attributes. Basically, the specification should contain all declarations from the module plus the operations providing the qualified notation, and one equation for each redefined attribute. These equations stipulate that a given qualified attribute is equal to the related original one. For instance, a module with the following declarations

```
pr NAT .
subclass A < A' .
at a : A' -> Nat .
at a : A -> Nat [redef] .
```

is translated to a specification containing the declarations above plus the following:

```
at a.A' : A' -> Nat .
var X : A' .
ax a.A'(X) = a(X) .
```

Because the equation above is parsed as

```
ax a.A'.A'Nat(X.A') = a.A'Nat(X.A') .
```

we have that `a.A'` is actually equal to the original version of a as desired

6.3 Methods Specified by DMAs

In order to define the semantics of method evaluation, we introduce some new notation. Given a specification S, we assume arbitrary

- Σ-variable family X such that $|X| = \{x_1, \ldots, x_k\}$, $v_i \leq LS(x_i)$, and $x_i \equiv x_j$ implies $v_i \equiv v_j$, for $i, j = 1..k$;
- expressions formed by functions, attributes, variables, and identifiers of specified objects[4]: $g, h, c \in \mathcal{T}_{F \cup A \cup (M \cap I)(X)}$; and
- method symbol $m' \in \{m.LS(x_1), m\}$, for some $m \in M$ not in the form *symbol.class-name*.

As any specification contains the sort `Bool`, hereafter we write $(\forall X)\ l = r$ **if** c instead of

$$(\forall X)\ l = r \ \mathbf{if}\ \{(c, \mathtt{true})\},$$

for any tern c (as above) of sort `Bool` Also, $\tilde{x}$ abbreviates $x_1, \ldots, x_k$, and $\tilde{y}$ is a sequence of variables from X. Lastly, we write $(\tilde{x} \leftarrow \tilde{v})$ for $(x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k)$

Now, we give a formal definition of DMA.

Definition 6.1 For a specification S, a **DMA** is a Σ-equation in the form

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

where m is a method: $m(\tilde{x}) \in \mathcal{T}_{M(X)}$; the result of m belongs to its associated class: $LS(m(\tilde{x})) = LS(x_1)$; and a is an attribute $a(x_1, \tilde{y}) \in \mathcal{T}_{A(X)}$. If $\tilde{y}$ is the empty sequence then $a(m(\tilde{x}), \tilde{y})$ stands for $a(m(\tilde{x}))$. □

The evaluation of a method specified by DMAs is atomic and yields the identifier of the object that executes this method; this object must be in the state where the method is evaluated 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

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

respectively correspond to the updated attribute and its new value. These equations are derived from the DMAs related to the method (as required by tbe 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 is either associated to the class indicated by tbe qualified notation, or to a class greater or equal to u and smaller or equal to u', where u is the class associated to the version of tbe method being evaluated, and u' is the least class (greater or equal to u) redefining this method (if there is no snch u', any related DMA may he used). Lastly, the method is only evaluated if there are some attributes to update and the updates are not contradictory; otherwise, it's snspended.

All the aspects discussed above are considered by tbe following rule.

Rule 6.3 (DMA) For any $m \in M$, let Γ be the set of all equations in the form

$$a(v_1, \tilde{y})(\tilde{x} \leftarrow \tilde{v}) = [\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D} \cup I}$$

such that E has a DMA in the form

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

$\texttt{true}$ belongs to $[c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D} \cup I}$; $[\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D} \cup I}$ is in $\mathcal{P}_{F \cup Id(\mathcal{D})}$; and $m' = m$ and $m \in R_{w',v'}$, for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or equal to the least class greater than or equal to u_1 such that m is redefined.

If $v_1 \in Id(\mathcal{D})$ and Γ is a non empty and non contradictory set, then

$$\frac{}{\langle m'(\tilde{v}), \mathcal{D}\rangle \rightarrow \langle v_1, \mathcal{D} \oplus \Gamma\rangle}$$

□

Remember that the operation $\oplus$ overwrites a database state (see Section 5.1.1).

DMAs can only specify a deterministic hehaviour for methods. So, note that if there are contradictory (or even redundant) DMAs in E, their associated method cannot be executed because its specified behaviour is cousidered inconsisteut; the evaluation is suspended. Obviously, this should be avoided. In the rule above, this is reflected by requiring Γ to be non contradictory. Similarly, suppose that two versions of a method are defined, but the specialized version is not considered a redefinition. In this case, following the last rule, equations from both definitions may be nsed to evaluate the specialized version. So, unless both versions specify the same updates, they are considered inconsistent; again, Γ will be coutradictory and consequently the method won't be execnted.

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. Indeed, usually they don't. Thus, ouly the most specialized method equations can be used for execution. This corresponds to the semantics of dynamic hindiug. That's why the restriction on $LS(x_1)$ is necessary in Rule **DMA**. Also, if a qualified uotation is used, observe that the indicated class must be the same as the least class of x_1, by the restrictions on m'; so, only equations associated to the indicated class can be used for evaluation.

Rule **DMA** makes clear that the adoption of the variant syntactic rule (which is implied by regularity) for redefinitions may cause some anomalies which require attention and should be avoided. For instance, consider the following specification

```
pr NAT .
subclass A < A' .
subclass B < B' .
me m : A' B' -> A' .
me m : A B -> A [redef] .
at a : A' -> Nat .
var X'   A' .
var Y'   B' .
var X : A .
var Y : B .
ax a(m(X',Y')) = ··· .
ax a(m(X,Y)) = ··· .
```

where m is redefined in a valid way. Observe that the expression m(a,b'), for a:A and b':B', cannot be evaluated using the DMA associated to the specific version of m because the class of b' is not a subclass of the class of Y. Also, the DMA related to the original version cannot be used for evaluation because it's not the most specialized one (i.e., the class of X' is not the same as the class of a). This situation should be avoided because although m(a,b') is a valid expression, it cannot be evaluated; it will be suspended forever.

From the definition of $\oplus$, we conclude that this operation might not preserve the properties of database states if its arguments don't satisfy some conditions. If this is the case in the use of $\oplus$ in the last rule, no transition is possible (because the result of $\oplus$ is not a valid database state). This means that the method cannot be executed. So, this circumstance should also be avoided. In general, this might happen if the state has an equation specifying an attribute value and this equation is not removed when that attribute is updated. For instance, this happens if a DMA specifies an update for a derived attribute. In this case, the state resulting from the update is a presentation containing two equations determining two values for the same attribute (one corresponding to the original derived attribute equation and another associated to the update). If these values are not equal (with respect to the related functional theory), the resulting presentation is not a valid database state, since it relates two expressions which are not related by the associated functional theory.

A similar problem occurs when a stored attribute is redefined by a derived one. In fact, a method associated to the superclass might try to update the redefined, derived attribute. But we have already discussed that derived attributes should not be updated. For example, consider the specification

```
pr NAT .
subclass C < C' .
me m : C' -> C' .
at a : C' -> Nat .
at a : C -> Nat [redef] .
var X   C' .
var Y   C .
ax a(m(X)) = 1 .
ax a(Y) = 0 .
```

where m is not redefined. Note that the original version of a is a stored attribute, whereas the specialized version of a is a derived attribute. Thus, the execution of m(o), for some o:C, adds the equation

```
a.CNat(o.C) = 1.Nat
```

to the database. But this conflicts with the equation

```
a.CNat(Y.C) = 0.Nat
```

which must be in any state (it belongs to AE). This would violate the restriction on states because we would then be able to prove that 0 equals 1 from the equations in the resulting state.

Here we do not give a complete semantics for updating of multi-argument stored attributes. In fact, this cannot be done in a simple and abstract way if presentations are used to model database states. So, we just give the semantics of DMAs specifying the update of only one attribute, as in

```
ax a(m(O,X),X) = e,
```

where evaluating `m(o,x)` only updates the attribute `a(o,x)`. That's why in the last rule we require the variables in $\tilde{y}$ to be in $\tilde{x}$. This implies that after instantiation, the LHS, RHS, and condition don't have any variables. So, the resulting RHS and condition can be evaluated, and the instantiated LHS specifies the value of only one attribute

It's difficult to give the semantics of DMAs such as

```
ax a(m(O),X) = e,
```

where the execution of `m(o)` should update all attributes `a(o,x)`, for any `x:X`. In order to consider this kind of DMAs, states would have to keep a history of updates for each multi-argument stored attribute. This is needed because such updates might not completely invalidate the previous ones, since each update may determine values for an arbitrary range of attributes. The value associated to a specific attribute could then be computed by checking what's the last update that determines it. This approach could be represented in our model for states. However, it turns out to be very detailed.

6.4 Methods Specified by IMAs

In this section we give the semantics of methods specified by IMAs. First, we give a formal definition for IMA.

Definition 6.2 For a specification S, an **IMA** is a Σ-equation in the form

$$(\forall X)\ m(\tilde{x}) = expr \ \texttt{if}\ c$$

where m is a method (i.e., $m(\tilde{x}) \in \mathcal{T}_{M(X)}$) and $expr$ is a method expression with variables from X: $expr \in \mathcal{T}_{\Sigma(X)}$. □

The execution of a method specified by an IMA corresponds to the evaluation of this IMA's RHS (instantiated with the method arguments), even if the method's associated object is not in the database used for evaluation (methods specified in this way are seen as abbreviations for complex expressions). 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; otherwise the evaluation is suspended. Also, similarly to DMAs, only specialized IMAs should be used for evaluation. The following rule considers those aspects.

Rule 6.4 (IMA) For any $m \in M$, if the IMA

$$(\forall X)\ m(\tilde{x}) = expr \texttt{ if } c$$

is in E; $\texttt{true} \in [c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D} \cup I}$; and $m' = m$ and $m \in R_{w', u'}$, for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or equal to the least class greater than or equal to u_1 such that m is redefined, then

$$\frac{}{\langle m'(\tilde{v}), \mathcal{D}\rangle \rightarrow \langle expr(\tilde{x} \leftarrow \tilde{v}), \mathcal{D}\rangle}$$

□

Because of dynamic binding, only the most specialized IMAs can he used for evaluation. Contrasting to DMAs (see comments following Rnle 6.3), if two or more IMAs that don't agree on the specified behaviour are used to define the same method, this method has a nondeterministic behavicnr. Similarly, if two versions of a method are defined, but the specialized version is not considered a redefinition, the IMAs related to both versions may be used to evaluate the specialized version, which will probably be nondeterministic.

Lastly, observe that the same anomalies associated to methods defined by DMAs, due to the adoption of the variant syntactic rule for redefinitions, might also happen for methods specified by IMAs (see Section 6.3 for details).

6.5 Arguments

From the transition rules given so far, it can be observed that the least sort of an expression is always in the same connected component of U (with respect to $\leq$) as the least sort of the result yielded by the evaluation of this expression. However, the least sort of this resnlt might be greater, smaller or even not related (by $\leq$) to the least sort of that expression. This is dne to the flexibility of the FOOPS type system. For instance, this happens because axioms are not necessarily sort decreasing, like in

```
sorts A B C .
subsort A < C .
subsort B < C .
class D .
at a : D -> A .
var X : D .
ax a(X) = e .
```

where `e` is a constant of sort `B`. So, the evaluation of `a(o)`, for `o:D`, gives `e`. But this resulting expression has least sort `B`, whereas `a(o)` has least sort `A`, which is not related to `B`.

Another example where this may happen is

```
subclass C < C' .
subsorts A < A' .
at a : C' -> A' .
at a : C -> A .
var X  C' .
ax a(X) = e .
```

where `e` is a constant of sort `A'` and `a` is not redefined (there are no special equations for the specialized version of `a`). Hence, `a(o)`, for `o:C`, has least sort `A`, but evaluates to `e` which has least sort `A'`, greater than `A`.

This fact has to be considered when evaluating arguments because an operation may not be defined for the type of the result of the evalnation of one of its arguments. In this case, a retract should be introduced to give the right type to the result.

In order to produce results of interest, retracts should be eventually eliminated (evaluated). Otherwise, operations might block or return exceptional retracted valnes. This can be avoided if the retracts in axioms can be eventnally eliminated (a non sort decreasing axiom may be considered a sort decreasing axiom, by inserting an adequate retract to its RHS). A fnnctional retract can be eliminated if its argnment is evalnated to an element that is equal (with respect to the associated functional theory) to an element of the desired sort. Similarly, an object level retract is eliminated if its argument evaluates to an object identifier of the desired class.

Now we introduce the Rule **Arg** for *arg*ument evaluation. Hereafter, we consider arbitrary method expressions without variables: $e, f, e_i, e'_i \in \mathcal{T}_S$, for $i = 1 .. k$.

Rule 6.5 (Arg) For any $i \in \{1, \ldots, k\}$ and $op \in (F \cup A \cup M)_{w,s}$ such that $LS(e_1) \ldots LS(e_k) \leq w$,

$$\frac{\langle e_i, \mathcal{D}\rangle \rightarrow \langle e'_i, \mathcal{D}'\rangle}{\langle op(e_1, \ldots, e_i, \ldots, e_k), \mathcal{D}\rangle \rightarrow \langle op(e'_1, \ldots, \tau(e'_i), \ldots, e'_k), \mathcal{D}'\rangle}$$

where $e'_j = e_j$ if $j \neq i$, for $j = 1 .. k$; and $\tau(e'_i) = e'_i$ if $LS(e'_i) \leq LS(e_i)$, otherwise $\tau(e'_i) =$ `r:u'>u`(e'_i), where `u'` $= LS(e'_i)$ and `u` $= LS(e_i)$. □

Notice that there is no fixed order to evaluate arguments; the order is nondeterministically chosen. In fact, the result of the evaluation of the expression may be nondeterministic, if some arguments have side effects. Furthermore, the evaluation of one argument might be interleaved with the evaluation of the others. However, from the semantic point of view, a step (transition) in the evaluation of one argument cannot happen at the same time as a step in the evaluation of another argument. This could be supported by a "truly concnrrent" semantics. In fact, in Section 7.2.1, we argue that the viable approaches for a "truly concurrent" semantics for FOOPS turn out to be equivalent to the interleaving semantics which we adopt here.

7 Method Combiners

Now, we show how the semantics given in the previous section can be modified and extended to support method combiners. It's simpler to directly give the semantics of each FOOPS predefined method combiner independently, instead of trying to specify some of them in terms of others. Here we introduce transition rules giving the semantics of each combiner. Lastly, we give one rule which specifies how new combiners, defined in terms of the predefined ones, are evaluated.

Method combiners are conceptually different from methods and attribntes. 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 argnments, and they yield results depending whether some particnlar arguments are fully evalnated. 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: $MC \subseteq \Sigma$, formed by method combiners names. Moreover, we consider that method combiners are not mixed up (in the sense of Definition 4.1) with functions, methods, or attributes. The following operations, which are in MC, represent the predefined method combiners:

- `_;_ : U T -> T`;
- `_||_, _[]_ : U U -> U`;
- **`result_;_`** `: U T -> U`;
- **`if_then_else_fi`** `: T U U -> U`; and
- `[_] : U -> U`;

for any types `T`, `U` $\in \mathcal{U}$.

For supporting the parallel composition and choice of method expressions having unrelated types, we consider that a universal type `Univ` is in $\mathcal{U}$. This type includes any class or sort. That is, $u \leq$ `Univ`, for any $u \in \mathcal{U}$. 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`".

Now, we proceed to give the semantics of method combiners

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. Rule **Seq** (for *seq*uential) is used for evaluation of the left argument and it indicates that transitions from this argument provokes transitions from a sequential composition:

Rule 7.1 (**Seq**)

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

□

When the left argument is fully evaluated. there is a transition to start the evaluation of the argument on the right, as indicated by Rule **SeqE** (for *seq*uential composition *e*limination):

Rule 7.2 (**SeqE**)

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

□

Here we adopt a "waiting semantics" for method expressions; that is, if a method cannot be executed in a database state (because no axiom specifying its behavior has a valid condition) then the object requiring the corresponding service (the client) has to "wait" until the service can be provided. However, notice that this does not necessarily mean that the client will be blocked, since it may be executing other tasks coucurrently.

In the last rule, the adoption of the "waiting semantics" is reflected by ensuring that the right argument of the sequential composition operator is only executed when the left one is fully evaluated. This is also reflected in the rules for other method combiners in the following sections. In order to capture a "non waiting semantics", the evaluation of f should start as soon as e cannot be evaluated. Thus, if a service is not available, the client doesn't wait and proceeds to the execution of the next service.

The first alternative was chosen because it gives a usefnl synchronization mechanism between clients and servers. This would have to be simulated by some form of "busy waiting", if the non waiting semantics were used. Usually, this simulation complicates the code and it's quite inefficient. Ou the other hand, the "uon waiting" behaviour can always be uaturally and efficiently simulated in terms of the "waiting" behaviour. For example, suppose that the operation **put** inserts an element in a buffer only when the buffer is not full. Thus, adding the axiom

```
ax put(B,N) = B if full?(B) .
```

releases the client if the buffer is full; the client doesn't need to wait for a place in the buffer.

Furthermore, the "nou waiting semantics" approach is not uniform, since in this case the left argument of _;_ may be discarded (when its corresponding service cannot be provided), but the argument of an operation has to be eventually evaluated. For example,

```
put(b,5) ; put(b,4)
```

evaluates to `put(b,4)` in a state where the buffer `b` is full. On the other hand, no transition is possible from `put(put(b,5),4)` in the same state.

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 trausitions from the parallel composition, as shown by the Rnle **ParL** (for *par*allel composition *l*eft argument evaluation)

Rule 7.3 (ParL)

$$\frac{\langle e, \mathcal{D}\rangle \rightarrow \langle e', \mathcal{D}'\rangle}{\langle e \mid\mid f, \mathcal{D}\rangle \rightarrow \langle e' \mid\mid f, \mathcal{D}'\rangle}$$

□

and the symmetric **ParR** (for *par*allel composition *r*ight argument evalnation):

Rule 7.4 (ParR)

$$\frac{\langle f, \mathcal{D}\rangle \rightarrow \langle f', \mathcal{D}'\rangle}{\langle e \mid\mid f, \mathcal{D}\rangle \rightarrow \langle e \mid\mid f', \mathcal{D}'\rangle}$$

□

Also, the arguments of a parallel composition operator can be eliminated when they are fully evaluated. This is specified by the Rules **ParLE** (for *par*allel composition *l*eft argument *e*limination)

Rule 7.5 (ParLE)

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

□

and **ParRE** (for *par*allel composition *r*ight argument *e*limination):

Rule 7.6 (ParRE)

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

□

7.2.1 "True Concurrency"

The interleaving semantics doesn't consider the behaviour caused by simultaneous transitions from the arguments of the parallel composition operator. This behaviour is usually considered by a "truly concurrent" semantics. In fact, "true concurrency" seems more natural than interleaving, since objects might be part of a distributed system (where expressions might be simultaneously evaluated). Hence, let's consider the introduction of the "truly concurrent" parallel composition combiner: `_|||_ : U U -> U`, for any `U` $\in \mathcal{U}$.

First, remember that an attribute cannot be both read and written at the same time because of physical limitations. So, simultaneous transitions from the arguments of a "truly concurrent" operator are only possible if the attributes accessed (i.e. read and/or written) in one transition are different from the attributes written in the other. Also, if an object is removed or created in one transition, it cannot be accessed, removed, or created by a simultaneous transition.

Some of those constraints cannot be elegantly expressed here because they rely on information that is abstracted by our framework. (For example, there's no simple procedure for determining what attributes are read during a given transition, since the evaluation of attributes is specified in terms of the representative of an equivalence class modulo equations.) Hence, instead of using the constraints above for defining a rule considering simultaneous evaluation of arguments, we use a weaker condition: the updates made by one transition don't interfere with the updates made by a simultaneous transition. Formally, if

$$\langle e, \mathcal{D}\rangle \rightarrow \langle e', \mathcal{D}_1\rangle \text{ (1) and } \langle f, \mathcal{D}\rangle \rightarrow \langle f', \mathcal{D}_2\rangle \text{ (2)}$$

then

$$\langle e, \mathcal{D}_2\rangle \rightarrow \langle e', \mathcal{D}'\rangle \text{ and } \langle f, \mathcal{D}_1\rangle \rightarrow \langle f', \mathcal{D}'\rangle.$$

It's easy to verify that this is implied by the constraints mentioned at the beginning of this section. First, observe that $\mathcal{D}_1$ and $\mathcal{D}_2$ can be respectively represented in the forms

$$\mathcal{D} \ominus RO \cup NO \oplus \Gamma \text{ and } \mathcal{D} \ominus RO' \cup NO' \oplus \Gamma',$$

for some RO, NO, Γ, RO', NO' and Γ', indicating the changes made by e and f, where $RO \cap NO = \emptyset$ and $RO' \cap NO' = \emptyset$. So, if the transitions 1 and 2 are possible then $\langle f, \mathcal{D}_1\rangle$ leads to

$$\langle f', \mathcal{D}_1 \ominus RO' \cup NO' \oplus \Gamma'\rangle.$$

since the stronger condition assures that the first step in the evaluation of f doesn't access the changes made by the first step in the evaluation of e. A similar reasoning can be used to show that $\langle e, \mathcal{D}_2\rangle$ leads to

$$\langle e', \mathcal{D}_2 \ominus RO \cup NO \oplus \Gamma\rangle.$$

Lastly, it remains to check that the stronger condition implies that $\mathcal{D}_1 \ominus RO' \cup NO' \oplus \Gamma'$ is the same as $\mathcal{D}_2 \ominus RO \cup NO \oplus \Gamma$. This can be easily done; we omit the details.

Now we give the semantics of _|||_ using the weaker condition. Basically, _|||_ is defined by the rules for interleaving, replacing _||_ by _|||_, plus Rule **TCPar** (for *t*ruly *c*oncurrent *par*allel composition).

Rule 7.7 (TCPar)

$$\frac{\begin{array}{c}\langle e, \mathcal{D}\rangle \rightarrow \langle e', \mathcal{D}_1\rangle, \langle f, \mathcal{D}_1\rangle \rightarrow \langle f', \mathcal{D}'\rangle,\\ \langle f, \mathcal{D}\rangle \rightarrow \langle f', \mathcal{D}_2\rangle, \langle e, \mathcal{D}_2\rangle \rightarrow \langle e', \mathcal{D}'\rangle\end{array}}{\langle e \,|||\, f, \mathcal{D}\rangle \rightarrow \langle e' \,|||\, f', \mathcal{D}'\rangle}$$

□

In fact, the weaker condition used above allows more transitions than expected for a "truly concurrent" operator. However, this turns out to be enough for our purposes here: we are interested in proving that the interleaving and the "truly concurrent" operators are equivalent (with respect to some mild notion of observation equivalence) in our framework; basically, we want to show that the extra rule for simultaneous evaluation of arguments is redundant. So, if we prove that this is the case considering the rule above, it follows that this is also the case if we consider a rule with a stronger premise. This equivalence is what should be expected since it's desirable to specify systems of distributed objects without worrying whether computations are being carried out simultaneously.

Here we suppose that the notion of equivalence that we are interested doesn't distinguish a configuration C having the transitions

$$\begin{tikzcd} C \arrow[r] & C_1 \arrow[r] & C_2 \end{tikzcd}$$

from a configuration C' having the transitions

$$\begin{tikzcd} C' \arrow[r] \arrow[rr, bend left] & C'_1 \arrow[r] & C'_2, \end{tikzcd}$$

where C'_1 and C'_2 are respectively (observation) equivalent to C_1 and C_2; and any other transition from C' is matched, in a similar way, by some transitions from C, and vice versa.

This is a quite mild assumption on equivalences over configurations. Roughly, it says that a configuration that can lead to a resulting configuration in either one or two transitions is equivalent to a configuration which can reach an equivalent resulting configuration in two equivalent transitions. This should be valid for most reasonable and interesting observation equivalences because the configurations reached from C and C' are equivalent, and any sequence of observations that can be made on the intermediate states reached by one of the configurations corresponds to a possible sequence of observations from the other.

We let $\approx$ denote the equivalence on configurations that we are interested. Thus the following theorem establishes the equivalence of the two operators for parallel composition.

Theorem 7.1 For any database state $\mathcal{D}$and method expressions e and f,

$$\langle e \mid\mid f, \mathcal{D}\rangle \approx \langle e \mid\mid\mid f, \mathcal{D}\rangle. \qquad (\omega)$$

Proof: We split the proof in three cases. First, we consider that both e and f are fully evaluated. In this case, the possible transitions from $\langle e \mid\mid f, \mathcal{D}\rangle$ and $\langle e \mid\mid\mid f, \mathcal{D}\rangle$ are justified by Rules 7.5, 7.6, and the corresponding ones to $\_\mid\mid\mid\_$:

$$\begin{tikzcd}
 & \langle e \mid\mid f, \mathcal{D}\rangle \arrow[dl, "1"'] \arrow[dr, "2"] & & & \langle e \mid\mid\mid f, \mathcal{D}\rangle \arrow[dl, "3"'] \arrow[dr, "4"] & \\
\langle \epsilon, \mathcal{D}\rangle & & \langle f, \mathcal{D}\rangle & \langle \epsilon, \mathcal{D}\rangle & & \langle f, \mathcal{D}\rangle
\end{tikzcd}$$

From these diagrams, we can conclude that $\langle e \mid\mid f, \mathcal{D}\rangle$ is equivalent to $\langle e \mid\mid\mid f, \mathcal{D}\rangle$, since the transitions from the first (1 and 2) are matched by the transitions from the second (3 and 4), in the sense that they lead to equivalent configurations (as $\approx$ is an equivalence, it is reflexive).

Now we assume that f is fully evaluated but e is not. Thus, we have the following possible transitions:

$$\begin{tikzcd}
 & \langle e \mid\mid f, \mathcal{D}\rangle \arrow[dl, "1"'] \arrow[dr, "2"] & & & \langle e \mid\mid\mid f, \mathcal{D}\rangle \arrow[dl, "3"'] \arrow[dr, "4"] & \\
\langle \epsilon, \mathcal{D}\rangle & & \langle e' \mid\mid f, \mathcal{D}_1\rangle & \langle \epsilon, \mathcal{D}\rangle & & \langle e' \mid\mid\mid f, \mathcal{D}_1\rangle
\end{tikzcd}$$

whenever $\langle \epsilon, \mathcal{D}\rangle \longrightarrow \langle e', \mathcal{D}_1\rangle$. This is justified by Rules 7.3. 7.6, and the corresponding ones to $\mid\mid\mid$. In the diagrams above, transitions 1 and 3 clearly match. In the meantime, let's assume that

$$\langle e' \mid\mid f, \mathcal{D}_1\rangle \approx \langle e' \mid\mid\mid f, \mathcal{D}_1\rangle. \qquad (\xi)$$

So, we can infer that transitions 2 and 4 match, what proves ω for this case

We can use a similar reasoning if e is fully evaluated but f is not.

Lastly, if neither e nor f is fully evaluated, the possible transitions from both configurations (as defined by Rules 7.3, 7.4, the corresponding ones to $\_\mid\mid\mid\_$ and Rule 7.7) are the following:

$$\begin{tikzcd}
 & \langle e \mid\mid f, \mathcal{D}\rangle \arrow[dl, "1"'] \arrow[dr, "2"] & \\
\langle e' \mid\mid f, \mathcal{D}_1\rangle \arrow[dr, "3"'] & & \langle e \mid\mid f', \mathcal{D}_2\rangle \arrow[dl, "4"] \\
 & \langle e' \mid\mid f', \mathcal{D}'\rangle &
\end{tikzcd}$$

and

$$\begin{tikzcd}
& \langle e \,|||\, f, \mathcal{D}\rangle \arrow[dl, "5"'] \arrow[dr, "6"] \arrow[dd, "7"] & \\
\langle e' \,|||\, f, \mathcal{D}_1\rangle \arrow[dr, "8"'] & & \langle e \,|||\, f', \mathcal{D}_2\rangle \arrow[dl, "9"] \\
& \langle e' \,|||\, f', \mathcal{D}'\rangle &
\end{tikzcd}$$

whenever $\langle e, \mathcal{D}\rangle \rightarrow \langle e', \mathcal{D}_1\rangle$ and $\langle f, \mathcal{D}\rangle \rightarrow \langle f', \mathcal{D}_2\rangle$. By Rule 7.7, transition 7 is only possible if $\langle f, \mathcal{D}_1\rangle \rightarrow \langle f', \mathcal{D}'\rangle$ and $\langle e, \mathcal{D}_2\rangle \rightarrow \langle e', \mathcal{D}'\rangle$; this justifies transitions 3, 4, 8, and 9. Transitions 1 and 2 respectively match 5 and 6 if we assume that

$$\langle e' \,||\, f, \mathcal{D}_1\rangle \approx \langle e' \,|||\, f, \mathcal{D}_1\rangle \qquad (\eta)$$

and

$$\langle e \,||\, f', \mathcal{D}_2\rangle \approx \langle e \,|||\, f', \mathcal{D}_2\rangle. \qquad (\kappa)$$

Also, by the assumption we made about $\approx$, transition 7 (together with transition 5 followed by 8) is matched by transition 1 followed by 3. This proves ω for this case.

Now, we have only to check propositions ξ, η, and κ. They can be informally justified by a similar reasoning as the one used for ω. However, this can only be formally verified if we are able to use the formal definition of $\approx$ with an associated proof technique. For example, this may be done using the notion of equivalence given in [3] with its related proof technique; we omit the details here. □

From this theorem and assuming that $\approx$ is preserved by _||_, we can guarantee that an expression not containing _|||_ is equivalent to an expression obtained from the first by substituting _|||_ for _||_. Similarly, _|||_ can be replaced by _||_. Hence, we conclude that there is no need to introduce the operator _|||_: it is semantically equivalent to _||_ and has a more complicated semantic definition. The extra rule for _|||_ is redundant.

Furthermore, the theorem above indicates that the implementation of interleaving may be "truly concurrent", in the sense that two expressions might be simultaneously evaluated, given some mild conditions. Fortunately, as we have shown, this doesn't generate any behaviour that cannot be observed from the interleaving of the evaluation of the two expressions

As mentioned before, simultaneous transitions are only possible if the attributes accessed in one transition are different from the attributes written in the other. This is a realistic restriction if attributes are directly implemented in terms of memory cells and transitions correspond to the execution of atomic transactions, which imply that attributes might be blocked. However, this restriction could be relaxed if the implementation provides a copy of each attribute; that is, one copy for reading (access) and another for writing (access). The reading copy could be used by many clients at the same time, whereas the write copy could be blocked by only one client at a given time. In this case, a complex mechanism is necessary to keep the consistency between the two copies. On the other hand, the reading copy may be read at the same time that the writing copy is being updated. Also, considering that an atomic transaction would only block the attributes that might be updated, a transaction could write to an attribute being read by a simultaneous transaction.

Clearly, this approach doesn't seem to be practical. It's likely that the efficiency gain obtained with the simultaneous exeeution is not worth because of the burden related to the mauagement of copies and extra memory space necessary to keep a copy of each attribute. Despite this, disregarding implementation issues, we superficially explore the consequences of this approach, from the semantic point of view.

First, let's assume that the operator `_//_ : U U -> U`, for any `U` $\in \mathcal{U}$, is defined by rules like the ones related to `_||_` plus one rule that allows simultaneons transitions whenever the attributes written in one transition are different from the attributes written in the otber. Also, an object removed or created in one transition cannot be accessed, removed, or created in a simultaneous transition. (So, the only difference between `_|||_` and `_//_` is that the second allows attributes read in one transition to be written in a simultaneous transition.) As in the definition of `_|||_`, some of the conditions necessary to formalize a rule considering simultaneous transition cannot be elegantly expressed in our framework. However, we can still argue that `_//_` is not equivalent to interleaving. This means that the extra behavionr associated to `_//_` cannot be expressed by the interleaving of two transitions.

First, consider the following specification defining a class of memory cells for storing natural numbers

```
pr NAT .
class Cell .
at v : Cell -> Nat .
me _:=_ : Cell Nat -> Cell .
me _:=_ : Cell Cell -> Cell .
vars C C' : Cell .
var N : Nat .
ax v(C := N) = N .
ax v(C := C') = v(C') .
```

where each object of `Cell` has an attribute `v` (for *value*) which stores a natural number, and two methods for changing the contents of a cell.

Now suppose that `X` and `Y` are identifiers of cells; the evaluation of the expression

```
X := Y // Y := X
```

in a database state $\mathcal{D}$ where `v(X) = 0` and `v(Y) = 1` is illustrated by the diagram in Figure 1, where $\mathcal{D}_1$ and $\mathcal{D}_2$ are respectively the states

$$\mathcal{D} \oplus \texttt{v(X) = 1} \text{ and } \mathcal{D} \oplus \texttt{v(Y) = 0}.$$

Note that transition 5 wouldn't be possible if `_|||_` (or `_||_`) were used instead of `_//_`. As can be seen, it's not equivalent to 1 followed by 3, or 2 followed by 4. This shows that the two operators are not equivalent, siuce the resulting states are clearly not (observation) equivalent.

The operators are not equivalent, but there are weaker relations among the different operators for parallelism. For example, the expression analysed above is eqnivalent to

`X := v(Y) || Y := v(X)` and `X := v(Y) ||| Y := v(X)`.

$$\begin{tikzcd}
 & & \langle \mathtt{X := Y // Y := X}, \mathcal{D} \rangle \arrow[dl, "1"'] \arrow[dr, "2"] \arrow[ddd, "5"] & & \\
 & \langle \mathtt{X // Y := X}, \mathcal{D}_1 \rangle \arrow[dl, "3"'] & & \langle \mathtt{X := Y // Y}, \mathcal{D}_2 \rangle \arrow[dr, "4"] & \\
\langle \mathtt{X // Y}, \mathcal{D}_1 \oplus \mathtt{v(Y) = 1} \rangle & & & & \langle \mathtt{X // Y}, \mathcal{D}_2 \oplus \mathtt{v(X) = 0} \rangle \\
 & & \langle \mathtt{X // Y}, \mathcal{D} \oplus \mathtt{v(X) = 1} \oplus \mathtt{v(Y) = 0} \rangle & &
\end{tikzcd}$$

Figure 1: Transitions generated by `X := Y // Y := X`.

since `v(X)` and `v(Y)` have to be evaluated before the assignment can be executed (this is not atomically done). Also, we can say that `e // f` is refined by `e ||| f` (or `e || f`), for any expressions `e` and `f`, because any behaviour observed from the evaluation of the first can also be observed from the evaluation of the second (but not the other way around).

Lastly, it's important to observe that the new operator is sensitive to the group of attributes updated by its arguments. For example, consider the following observation equivalent expressions:

`[ X := Y ; Y := Y ]` and `X := Y`.

Also, we have that `Y := X` is equivalent to `Y := X`. Now note that the expression

(1) `Y := X // X := Y`

is not equivalent to

(2) `Y := X // [ X := Y ; Y := Y ]`

since the (sub)expressions of 1 can be executed at the same time (they write to different attributes), whereas this is not possible for 2 (both arguments update `Y`). In fact, expression 1 might behave in a way that cannot be simulated by 2.

From the previous example, we conclude that it is not possible to find a reasonable notion of equivalence that is preserved by this operator. This essentially means that `_//_` cannot be used for compositional software development. So, it's not very useful in practice. That's another reason for choosing an interleaving semantics for FOOPS parallel composition.

7.3 Nondeterministic Choice

The choice operator (`_[]_`) nondeterministically chooses one of its arguments for evaluation. Using process algebra's terminology (see [24, 32, 23]), 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 (see Section 3.2.2)

Essentially, the nondeterminism of an external choice is partly resolved by the environment where the choice is evaluated. In fact, an argument may only be chosen if it can be evaluated in the current environment, or if it's already fully evaluated. If both arguments may be chosen, the operator autonomously (internally) resolves the nondeterminism.

Transitions from the choice between two expressions correspond to transitions from one of the expressions. This is specified by the Rule **ChoiceL** (for *choice left* argument evaluation)

Rule 7.8 (ChoiceL)

$$\frac{\langle e, \mathcal{D}\rangle \to \langle e', \mathcal{D}'\rangle}{\langle e \,[]\, f, \mathcal{D}\rangle \to \langle e', \mathcal{D}'\rangle}$$

□

and the similar **ChoiceR** (for *choice right* argument evaluation)

Rule 7.9 (ChoiceR)

$$\frac{\langle f, \mathcal{D}\rangle \to \langle f', \mathcal{D}'\rangle}{\langle e \,[]\, f, \mathcal{D}\rangle \to \langle f', \mathcal{D}'\rangle}$$

□

Moreover, a fully evaluated argument may be chosen by the choice without changing the database. This is expressed by the Rule **ChoiceLE** (for *choice left* argument *elimination*)

Rule 7.10 (ChoiceLE)

$$\frac{}{\langle v \,[]\, e, \mathcal{D}\rangle \to \langle v, \mathcal{D}\rangle}$$

□

and the symmetric **ChoiceRE** (for *choice rigth* argument elimination)

Rule 7.11 (ChoiceRE)

$$\frac{}{\langle e \,[]\, v, \mathcal{D}\rangle \to \langle v, \mathcal{D}\rangle}$$

□

It might be difficult to efficiently implement the kind of choice discussed here because it tries to "guess" whether an argument can be evaluated or not. This contrasts with an internal choice operator, which can be simply and efficiently implemented. Also, the internal choice is likely to be more useful in FOOPS specifications. However, we have adopted an external choice for two main reasons. First, it's more fundamental and can be used to define internal choice in a very simple way. Second, choice is usually used in FOOPS as an abstraction tool for writing specifications (when it's desirable to abstract from the reasons which determine one behaviour or another), rather than as an operator to construct implementations (as usually necessary in process algebras).

7.4 Result

As probably expected, the rules giving the semantics of the result method combiner should be similar to the rules for sequential composition. This is confirmed by the Rule **ResultL** for evaluation of the left argument of the result combiner.

Rule 7.12 (ResultL)

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

□

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. Those aspects are described by the Rule **ResultR** (for *result right* argument evaluation):

Rule 7.13 (ResultR)

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

□

Finally, when both arguments are fully evaluated, the left one is given as result, as indicated by Rule **ResultE** (for *result* elimination):

Rule 7.14 (ResultE)

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

□

7.5 Conditional

Besides conditional axioms, FOOPS has a method combiner that may be used for specifying conditional behaviour. In fact, this combiner provides a more general mechanism than conditional axioms because its condition may be an arbitrary method expression, whereas the latter cannot have conditions containing method symbols or method combiners.

First, the conditional method combiner (`if_then_else_fi`) fully evaluates its condition. After that, based on the result, one of the alternatives is choosen. The Rule **IfCond** specifies the evaluation of the condition:

Rule 7.15 (IfCond) For any $c, c' \in \mathcal{T}_\Sigma$,

$$\frac{\langle c, \mathcal{D}\rangle \rightarrow \langle c', \mathcal{D}'\rangle}{\langle \texttt{if}\ c\ \texttt{then}\ e\ \texttt{else}\ f\ \texttt{fi}, \mathcal{D}\rangle \rightarrow \langle \texttt{if}\ c'\ \texttt{then}\ e\ \texttt{else}\ f\ \texttt{fi}, \mathcal{D}'\rangle}$$

□

When the condition is fully evaluated, the first alternative is given as result, if the condition is true, as indicated by Rule **IfCondT**:

Rule 7.16 (IfCondT) If $v =_{FE}$ `true` then

$$\frac{}{\langle \texttt{if } v \texttt{ then } e \texttt{ else } f \texttt{ fi}, \mathcal{D}\rangle \rightarrow \langle e, \mathcal{D}\rangle}$$

□

If the condition is false, the conditional yields the second alternative:

Rule 7.17 (IfCondF) If $v =_{FE}$ `false` then

$$\frac{}{\langle \texttt{if } v \texttt{ then } e \texttt{ else } f \texttt{ fi}, \mathcal{D}\rangle \rightarrow \langle f, \mathcal{D}\rangle}$$

□

It might seem that this method combiner could be equivalently defined as a function, in terms of equations at FOOPS functional level. However, remember that functions can only be evaluated if their arguments are functional terms or object identifiers. This implies that the alternatives of a conditional would have to be evaluated before one of them is choosen. In general, that's not the desired behaviour because the evaluation of the alternatives might change the state.

7.6 Atomic Evaluation

Intuitively, the atomic evaluation of an expression corresponds to its evaluation in only one step (transition). In fact, this means that the attributes accessed by an expression being atomically evaluated cannot be modified by other expressions being concurrently evaluated. Atomic evaluation might is necessary when an expression has to be evaluated without interference from others; that's why FOOPS provides the atomic evaluation operator (`[_]`). Following those intuitions, we introduce the next rule:

Rule 7.18 (Atomic)

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

where $\rightarrow^{*}$ denotes the transitive, reflexive closure of $\rightarrow$[5]. □

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 previous to the beginning of the transaction; that's why there is no transition if the expression cannot be fully evaluated.

This approach is really useful for database systems, but it might be quite inefficient from the implementation point of view. So, if efficiency is essential, the atomic evaluation operator should only be used for expressions that can be fully evaluated in the contexts where they are used.

[5] We omit the obvious rules necessary for defining $\rightarrow^{*}$.

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 (intermediate states reached during the evaluation of an atomic expression cannot be observed). The following rule reflects those comments:

Rule 7.19 (Diverge) If $\langle e, \mathcal{D}\rangle$ is non terminating,

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

□

where

Definition 7.1 A configuration is **terminating** if there is no infinite sequence of $\rightarrow$-transitions from it. □

From the rule above, we conclude that practical applications should not use the atomic evaluation operator for expressions whose evaluation, in the contexts where they are used, might not terminate.

7.7 Method Combiner Definition

During evaluation, method combiners defined by the user are simply replaced by the expression that they abbreviate, as described by Rule **MCDef** (for *m*ethod *c*ombiner *def*inition):

Rule 7.20 (MCDef) For any $mc \in MC$ and $expr \in \mathcal{T}_{\Sigma(X)}$, if the axiom

$$(\forall X)\ mc(\bar{x}) = expr$$

is in E and $mc(\bar{x}) \in \mathcal{T}_{MC(X)}$,

$$\frac{}{\langle mc(\bar{e}), \mathcal{D}\rangle \rightarrow \langle expr(\bar{x} \leftarrow \bar{e}), \mathcal{D}\rangle}$$

where $\bar{e}$ stands for $e_1, \ldots, e_k$, and $LS(e_i) \leq LS(x_i)$, for $i = 1..k$. □

As can be seen from the rule above, the arguments of a method combiner don't have to be fully evaluated before the combiner is replaced by the expression that it abbreviates. In particular, they might contain method and method combiner symbols. This is necessary for most applications. In this way, conditional axioms cannot be used to define method combiners, since arguments would have to be evaluated before the evaluation of the condition.

Observe that introducing more than one axiom for the same method combiner gives a non-deterministic behaviour for it, since all axioms can be applied. In particular, adding rules for a predefined method combiner might completely change its behaviour. In fact, this cannot be done in FOOPS modules.

8 Object Creation and Deletion

Dynamic object creation and deletion are respectively provided in FOOPS by the operators **new** and **remove**. In this section, we describe the semantics of both operators. First, we consider object creation. Later, we introduce object deletion. 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. Hereafter we consider that signatures contain the following special combiners:

- `new.C() : -> C`,
- `new : C -> C`, and
- `remove : C -> C`,

for each class $C \in C$. (This extends the definition of signature, in the same way as done in Section 7.) Observe that the class name is used to form the operation name of the first creation operation; this is important to indicate the class of the objects to be created. A class name is not necessary to distinguish the different versions of the other creation operation because this information is already provided by the class of its argument (identifiers have a fixed and predefined class; see Section 5).

8.1 Object Creation

For a given class C, the operator `new.C()` creates an object of C having an identifier nondeterministically choosen from I_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 8.1 (Creation) For any class $C \in C$ and any identifier $v \in I_C$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin Id(\mathcal{D})$),

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

where V is a $U^* \times U$-sorted family containing v only; that is, $V_{\lambda,C} = \{v\}$ and $V_{w,u} = \emptyset$ if $w \neq \lambda$ or $u \not\equiv C$. □

Remember that the operation ∪ adds objects to a database state, according to the identifiers given as arguments, without setting their attributes (see Section 5.1.2).

Note that the creation operation directly introduces unbounded nondeterminism to FOOPS, if the family I of object identifiers is formed by infinite sets. In this case, infinitely many identifiers may be choosen for creating an object.

The operator **new** creates an object of the same class as the identifier given as argument, if this identifier is not already 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 Rule **CreationId** (for object *creation* with *id*entifier):

Rule 8.2 (CreationId) For any class $C \in C$ and any identifier $v \in I_C$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin Id(\mathcal{D})$),

$$\frac{}{\langle \texttt{new}(v), \mathcal{D}\rangle \rightarrow \langle v, \mathcal{D} \cup V\rangle}$$

where V is a $U^* \times U$-sorted family containing v only, as defined in Rule 8.1. □

From this rule, we conclude that in order to create an object of a given class, we have to know what identifiers are related to this class. Only these identifiers may be nsed for creating and accessing objects of this class. In practice, this is not a big restriction. For example, each family in I can be choosen (by the FOOPS system) to contain only names prefixed by the name of the class associated to the family. So, we can easily know what are the identifiers associated to a given class.

Observe that the operators for object creation don't assign initial valnes for attributes.

8.2 Object Deletion

The operator **remove** receives an object identifier as 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 suspended. The following rule formalize those aspects:

Rule 8.3 (Deletion) For any class $C \in C$ and any identifier $v \in I_C$ associated to an object in $\mathcal{D}$ (i.e., $v \in Id(\mathcal{D})$).

$$\frac{}{\langle \mathtt{remove}(v), \mathcal{D}\rangle \rightsquigarrow \langle v, \mathcal{D} \ominus V\rangle}$$

where V is a $U^* \times U$-sorted family containing v only, as in Rule 8.1. □

Remember that the operation $\ominus$ removes from a given database state all references to a particular object (see Section 5.1.3). This is necessary to avoid dangling identifiers. This operation might seem extremely centralized and inefficient, contrasting with the notion of distributed objects: bnt, in fact, it can be implemented in an efficient and decentralized way.

As the argument of **remove** doesn't have to be an object identifier, we need one more rnle (**DeletionArg**, for evaluation of the *argument* of the *deletion* operation) indicating how the argnment should be evaluated.

Rule 8.4 (DeletionArg)

$$\frac{\langle e, \mathcal{D}\rangle \rightsquigarrow \langle e', \mathcal{D}'\rangle}{\langle \mathtt{remove}(e), \mathcal{D}\rangle \rightsquigarrow \langle \mathtt{remove}(e'), \mathcal{D}'\rangle}$$

□

9 Protected Objects

In this section we give the semantics of object protection. An informal description of this mechanism was given in Section 3.2.5. First, we assume that signatures have the combiner **addpr** for changing the protection status of objects, and new object creation operations:

- `addpr : C Univ -> C`,
- `new.C : Univ -> C`, and
- `new : C Univ -> C`.

for each class `C` $\in C$. Also, the following combiners should be in signatures, for representing the object permission given as argument to **new**:

- `_++_ : Univ Univ -> Univ`,
- `{} : -> Univ`, and
- `any : -> Univ`.

Using those combiners we can create terms in the following forms: **any**, `{}`, and `o ++ s`, which respectively denote the set of all identifiers, the empty set, and the set containing the identifier `o` and the identifiers in the set denoted by s, where s is a term in one of the forms shown above. Only terms in those forms denote an object permission.

Second, we have to modify the structure of configurations to incorporate information about object permission. For a specification S, this information is represented by a finite mapping (referred as the *permission mapping*), belonging to $Perm(S) = |I_S| \mapsto \mathbb{P}|I_S|$, which maps an object identifier o to the set of identifiers of the objects that can directly invoke methods associated to o. So, we let configurations be represented by the elements of

$$PrConf(S) = \mathcal{T}_S \times Perm(S) \times D_S.$$

We drop the subscripts when not confusing, and we write $\langle e, (\varrho, \mathcal{D})\rangle$ for $(e, \varrho, \mathcal{D}) \in PrConf(S)$. For convenience, the elements of $Perm(S) \times D_S$ may also be called database states (containing object permission information).

We define the semantics of FOOPS with object protection following the same approach used in Section 6. Basically, for a specification S, we introduce the relation $\rightarrow_S \subseteq PrConf(S) \times PrConf(S)$. It's defined by the rules given in the previous sections—except Rules 6.3, 6.4, 8.1, 8.2, and 8.3—and some extra rules to follow, assuming that

- database states have some extra information; that is, the variables $\mathcal{D}$ and $\mathcal{D}'$ range over $Perm(S) \times D_S$, rather than over D_S;
- the operations on D_S are composed with the projections and constructors associated to $Perm(S) \times D_S$ (let ϱ and db respectively give permission mappings and database states). For example, for $\mathcal{D} \in Perm(S) \times D_S$, we now assume that $\mathcal{D} \oplus \Gamma$ stands for

  $$(\varrho(\mathcal{D}), db(\mathcal{D}) \oplus \Gamma)$$

  (similarly for $\cup$ and $\ominus$), and $[\![e]\!]_{\mathcal{D}}$ is an abbreviation for $[\![e]\!]_{db(\mathcal{D})}$ (similarly for $[e]_{\mathcal{D}}$ and $Id(\mathcal{D})$).

Note that the rules introduced in previous sections are still meaningful after the modification on the structure of states, since we have also modified the operations used to access states to consider the new structure.

In order to indicate the object requesting a particular service, we introduce the method combiner `_!_ : T U -> T`, for any types `T`, `U` $\in U$. The second argument of this operator is the identifier of the object requesting the evaluation of the first argument. The second argument of `_!_` and a permission mapping are enough to give the semantics of object protection, by modifying some of the transition rules introduced in previous sections (Rules 6.3, 6.4, 8.1, 8.2, and 8.3) and including new ones.

First we specify how `_!_` is propagated to subexpressions. This is necessary because a method can only be evaluated if there is an indication of the object which invoked it. For this, we have to add some new rules. Hereafter, we consider an arbitrary object identifier $o \in I$.

9.1 Attributes, Functions, and Identifiers

The evaluation of attributes, functions, and object identifiers doesn't update objects; hence, it doesn't depend on the mechanism for object protection, as formalized by the following rule:

Rule 9.1 (PrE) For any $e \in \mathcal{T}_{F \cup A \cup I}$,

$$\frac{}{\langle e \ ! \ o, \mathcal{D}\rangle \rightarrow \langle e, \mathcal{D}\rangle}$$

□

9.2 Arguments

The next rule shows how the mechanism for object protection is propagated to the arguments of methods, if they do not already have it. In the following rules assume that $k \geq 1$.

Rule 9.2 (PrM) For any $m \in M_{w,u}$ such that $LS(e_1), \ldots, LS(e_k) \leq w$, if e_i, for some $1 \leq i \leq k$, is not in the form $e \ ! \ f$ then

$$\frac{}{\langle m(\bar{e}) \ ! \ o, \mathcal{D}\rangle \rightarrow \langle m(\bar{e} \ ! \ o) \ ! \ o, \mathcal{D}\rangle}$$

where $\bar{e} \ ! \ o$ stands for $e_1 \ ! \ o, \ldots, e_k \ ! \ o$. □

The evaluation of method combiners is independent of the mechanism for object protection. Combiners just propagate this mechanism, if necessary, as specified by the next two rules:

Rule 9.3 (PrMCP) For any $mc \in MC_{w,u}$ such that $LS(e_1), \ldots, LS(e_k) \leq w$, if e_i, for some $1 \leq i \leq k$, is not in the form $e \ ! \ f$ then

$$\frac{}{\langle mc(\bar{e}) \ ! \ o, \mathcal{D}\rangle \rightarrow \langle mc(\bar{e} \ ! \ o), \mathcal{D}\rangle}$$

□

However, if the arguments already have the mechanism, it's not propagated, as formalized by the following rule.

Rule 9.4 (PrMC) For any $mc \in MC_{w,u}$ such that $LS(e_1), \ldots, LS(e_k) \leq w$, if all e_i for $i = 1..k$, are in the form $e \ ! \ f$ then

$$\frac{}{\langle mc(\bar{e}) \ ! \ o, \mathcal{D}\rangle \rightarrow \langle mc(\bar{e}), \mathcal{D}\rangle}$$

□

Now we give a rule for argument evaluation of expressions formed by the method combiner _!_.

Rule 9.5 (PrArg) For any $i \in \{1, \ldots, k\}$ and $op \in (F \cup A \cup M)_{w,s}$ such that $LS(e_1) \ldots LS(e_k) \leq w$,

$$\frac{\langle e_i, \mathcal{D}\rangle \rightarrow \langle e_i', \mathcal{D}'\rangle}{\langle op(e_1, \ldots, e_i, \ldots, e_k) \ ! \ o, \mathcal{D}\rangle \rightarrow \langle op(e_1', \ldots, \tau(e_i'), \ldots, e_k') \ ! \ o, \mathcal{D}'\rangle}$$

where $e_j' = e_j$, if $j \neq i$, for $j = 1..k$; and $\tau(e_i') = e_i'$, if $LS(e_i') \leq LS(e_i)$; otherwise, $\tau(e_i') =$ `r:u'>u`(e_i'), where `u'` $= LS(e_i')$ and `u` $= LS(e_i)$. □

Note that the original rule for argument evaluation is not replaced by the above rule, since it's still useful for evaluation of the arguments of attributes, for example.

9.3 Methods

Now we introduce new rules for method evaluation. Those are small modifications of the rules in Sections 6.3, 6.4, and 6.5. Here the behaviour of a method depends on the object that requested it. The method might be executed or suspended, depending whether the invocation is allowed by the object protection mechanism. For a given state $\mathcal{D}$, only the objects in $\varrho(\mathcal{D})(v)$ can directly invoke methods of v (when v is not in the domain of $\varrho(\mathcal{D})$, $\varrho(\mathcal{D})(v)$ denotes the empty set).

9.3.1 Methods Specified by DMAs

First we consider the evaluation of methods specified by DMAs.

Rule 9.6 (PrDMA) For any $m \in M$ let Γ be the set of all equations

$$a(v_1, \tilde{y})(\tilde{x} \leftarrow \tilde{v}) = [\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D} \cup I}$$

such that E has a DMA in the form

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

true belougs to $[c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D} \cup I}$; $[\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D} \cup I}$ is in $\mathcal{P}_{F \cup Id(\mathcal{D})}$; and $m' = m$ and $m \in R_{w',u'}$, for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or eqnal to the least class greater than or equal to u_1 such that m is redefined.

If $v_1 \in Id(\mathcal{D})$, $o \in \varrho(\mathcal{D})(v_1)$, and Γ is a non empty and non contradictory set then

$$\frac{}{\langle m'(\tilde{v}) \;!\; o, \mathcal{D}\rangle \rightarrow \langle v_1, \mathcal{D} \mathbin{\dot{+}} \Gamma\rangle}$$

□

9.3.2 Methods Specified by IMAs

Similarly to what was done in the last section, we modify the rule for evaluation of methods specified by IMAs. Note that the execution of a method m specified by an IMA sometimes requires the invocation of methods associated to other objects; those methods are directly invoked by the object that performs m.

Rule 9.7 (PrIMA) For any $m \in M$, if the IMA

$$(\forall X)\; m(\tilde{x}) = expr \;\mathtt{if}\; c$$

is in E. **true** $\in [c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D} \cup I}$; $m' = m$ and $m \in R_{w',u'}$. for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or equal to the least class greater than or equal to u_1 such that m is redefined; and $o \in \varrho(\mathcal{D})(v_1)$ then

$$\frac{}{\langle m'(\tilde{v}) \;!\; o, \mathcal{D}\rangle \rightarrow \langle expr(\tilde{x} \leftarrow \tilde{v}) \;!\; v_1, \mathcal{D}\rangle}$$

□

Contrasting to the original rule for evaluation of methods specified by IMAs, the rule above introduces a transition that is dependent on the state; that is, the transition is only possible if the object protection is not violated.

9.4 Object Creation and Deletion

Lastly, we modify the rules for object creation and deletion. Basically, the only difference from the creation operations introduced in this section and the operations introduced in Section 8 is that the former set the object permission.

First we have to define the function *set* for mapping the extra argument of **new** to the set of identifiers that it represents:

$$\begin{aligned} set(\{\}) &= \emptyset \\ set(\mathtt{any}) &= |I| \\ set(e \mathbin{++} f) &= \{e\} \cup set(f), \text{ if } e \in I \\ &= set(f), \text{ if } e \notin I \end{aligned}$$

Note that the arguments of _++_ that are not object identifiers are discarded by *set*.

The following rules describe the semantics of the operations for object creation, where

$$\mathcal{D} \oplus \{v \mapsto s\} \text{ denotes } (\varrho(\mathcal{D}) \oplus \{v \mapsto s\}, db(\mathcal{D})),$$

for any database state $\mathcal{D}$, $v \in I$, and $s \subseteq |I|$, the operation for mapping overwriting is represented by $\oplus$. Also, we assume that e is a term specifying an object permission, as described above. Any object is allowed to invoke the creation operations; that is, those operations don't depend on the mechanism for object protection, as indicated by the following rule:

Rule 9.8 (PrCreation) For any class $\mathtt{C} \in C$, and any identifier $v \in I_{\mathtt{C}}$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin Id(\mathcal{D})$),

$$\frac{}{\langle \mathbf{new}.\mathtt{C}(e) \mathbin{!} o, \mathcal{D}\rangle \rightarrow \langle v, \mathcal{D} \cup V \oplus \{v \mapsto set(e)\}\rangle}$$

where V is a $U^* \times U$-sorted family containing v only; that is, $V_{\lambda,\mathtt{C}} = \{v\}$ and $V_{w,u} = \emptyset$ if $w \neq \lambda$ or $u \not\preceq \mathtt{C}$. □

and

Rule 9.9 (PrCreationId) For any class $\mathtt{C} \in C$ and any identifier $v \in I_{\mathtt{C}}$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin Id(\mathcal{D})$),

$$\frac{}{\langle \mathbf{new}(v,e) \mathbin{!} o, \mathcal{D}\rangle \rightarrow \langle v, \mathcal{D} \cup V \oplus \{v \mapsto set(e)\}\rangle}$$

where V is a family containing v only, as in Rule 9.8. □

From those rules, we can see that by default an object doesn't have permission to invoke its own methods. If this is desired, as usually the case, this has to be specified at object creation time, by making sure that v is in $set(e)$.

We modify the semantics of **remove** in such a way that a deletion operation can only be performed if the object which asked for it is able to invoke methods of the object to be removed. Here is the new rule:

Rule 9.10 (PrDeletion) For any class $C \in C$ and any identifier $v \in I_C$ associated to an object in $\mathcal{D}$ (i.e., $v \in Id(\mathcal{D})$), if $o \in \varrho(\mathcal{D})(v)$ then

$$\frac{}{\langle \mathtt{remove}(v) \ ! \ o, \mathcal{D}\rangle \rightarrow \langle v, \mathcal{D} \ominus V \oplus \{v \mapsto \emptyset\}\rangle}$$

where V is a family containing v only, as in Rule 9.8. □

Note that the object protection information associated to an object is reset after its deletion.

Lastly, we define the semantics of the addpr(_,_) operator, which was informally described in Section 3.2.5.

Rule 9.11 (PrAdd) For any object v in $\mathcal{D}$ (i.e., $v \in Id(\mathcal{D})$), if $o \in \varrho(\mathcal{D})(v)$ then

$$\frac{}{\langle \mathtt{addpr}(v,e) \ ! \ o, \mathcal{D}\rangle \rightarrow \langle v, \mathcal{D} \oplus \{v \mapsto set(e) \cup \varrho(\mathcal{D})(v)\}\rangle}$$

□

9.5 Comparison with Other Approaches

Now we comment on some alternatives to the approach for object protection that we have used here. First, we consider the introduction of *root objects*, as in C++ and Eiffel. Basically, only root objects may be interfered in an arbitrary way; interference in non root objects is derived from interference in root objects. In FOOPS without object protection, any object is a root object. Usually, there is more than one root object in a distributed system, but only one in a sequential system.

This mechanism only protects non root objects from arbitrary interference. In fact, we have to check if the interference propagated by root objects doesn't violate the protection rules that we want to enforce for non root objects. In general, this might generate complicated proof obligations. Moreover, this only assures protection for a system in isolation, it doesn't guarantee protection if this system is included as part of a larger system, where more root objects might be available and additional propagated interference might occur. Clearly, this is not appropriate for a compositional development method.

In order to achieve compositionality with this approach it would be necessary to verify that the objects that should be protected cannot be accessed by new objects in a larger system. This can be enforced by making sure that the identifiers of the protected objects aren't yielded by any operation that may be invoked by objects in the larger system. Again, this might lead to complicated proof obligations. In fact, it seems simpler to support a mechanism that directly enforces object protection, rather than writing some specific code for guaranteeing that, and discharging complicated proof obligations.

As briefly discussed before, the second alternative for object protection is the support for private references [26]. However, it's not appropriate in general. In fact, this is a particular case of our mechanism for object protection. Considering the examples given in Section 3.2.5, private references are appropriate for modelling linked lists (since an intermediate cell should only be accessed by its precedent in the sequence); however, private references are too restrictive for modelling the communication protocol (since the channel should be protected, but shared by both agents). Hence, it seems useful to have a more general mechanism for object protection.

Another general approach for object protection was independently introduced by Hogg [25]. In our approach, one explicitly restricts the group of objects that can directly invoke methods of a protected object. In Hogg's approach, one explicitly indicate a so called *bridge object*, defining an associated group of protected objects (*island*); then any direct access to a protected object (an object in an island) must be indirectly derived from an access to an associated bridge object. There are subtle differences between the two approaches. Whereas Hogg's approach seems more abstract, it seems that a finer level of protection can be specified by our approach. In fact, more experiments would be necessary to give us more confidence that Hogg's approach shouldn't be supported by FOOPS.

10 Evaluation in the Background

In order to give the semantics of $\_\bigstar\_$ we use another structure for representing configurations. Now we consider that configurations also contain some information about the expressions being evaluated in the background. This reflects the conceptual distinction between both kinds of evaluation. Hereafter, for a specification S, configurations are represented by elements of

$$BgConf(S) = \mathcal{T}_S \times \mathcal{T}_S \times Perm(S) \times D_S,$$

where the first component of such a tuple is the "main" expression, the second is the expression in the background, and the last two correspond to the database state with object permission. We write $\langle e,\ f,\ (\varrho, \mathcal{D})\rangle$ or $\langle e,\ (f, \varrho, \mathcal{D})\rangle$ for $(e, f, \varrho, \mathcal{D}) \in BgConf(S)$.

We follow a similar approach to the one used in Section 9. First we define the relation $\hookrightarrow_S \subseteq BgConf(S) \times BgConf(S)$ by the rules used for the definition of $\rightarrow_S$ in the last section, assuming that

- $\rightarrow$ is replaced by $\hookrightarrow$, in all rules;
- database states also contain information about expressions being evaluated in the background; that is, the variables $\mathcal{D}$ and $\mathcal{D}'$ range over pairs of method expressions and database states with permission information (i.e., $\mathcal{D}, \mathcal{D}' \in \mathcal{T}_S \times Perm(S) \times D_S$), and
- the operations on database states are composed with the projections and constructors associated to $\mathcal{T}_S \times Perm(S) \times D_S$ (let ϱ, bg and db respectively give permission mappings, background expressions, and database states). For example, for $\mathcal{D} \in \mathcal{T}_S \times Perm(S) \times D_S$, we now assume that $\mathcal{D} \oplus \Gamma$ stands for

  $$(bg(\mathcal{D}), \varrho(\mathcal{D}), db(\mathcal{D}) \oplus \Gamma)$$

  (similarly for $\cup$ and $\ominus$), and $[\![e]\!]_{\mathcal{D}}$ abbreviates $[\![e]\!]_{db(\mathcal{D})}$ (similarly for $[e]_{\mathcal{D}}$ and $Id(\mathcal{D})$). Lastly, $\mathcal{D} \oplus \{v \mapsto s\}$ means $(bg(\mathcal{D}), \varrho(\mathcal{D}) \oplus \{v \mapsto s\}, db(\mathcal{D}))$

But we also need to introduce one rule indicating how $\_\bigstar\_$ is evaluated. Essentially, $\_\bigstar\_$ starts the evaluation of *its* second argument in parallel with the current background expression. The first argument is then given as result, as indicated by Rule **BgOp** (for *background operator*):

Rule 10.1 (BgOp) For any $b \in \mathcal{T}_\Sigma$ and $\mathcal{P} \in Perm(S) \times D_S$,

$$\frac{}{\langle e \bigstar f,\ b,\ \mathcal{P}\rangle \hookrightarrow \langle e,\ f \mid\mid b,\ \mathcal{P}\rangle}$$

□

Naturally, we assume that `_._ : U T -> U` is in MC, for any `T`, `U` $\in \mathcal{U}$.

The rule above finally defines $\hookrightarrow$. However, note that this relation only considers the evaluation of the main expression. We still have to specify how the expression in the background is evaluated; this is done by the relation $\rightarrowtail_S \subseteq BgConf(S) \times BgConf(S)$, defined by the following rule (**BgEval**, for *b*ack*g*round *eval*uation). A background expression is evaluated just as if it were a main expression, and any additional background expression resulting from this evaluation will also be evaluated in the background.

Rule 10.2 (BgEval) For any $b \in \mathcal{T}_\Sigma$, $\mathcal{P} \in Perm(S) \times D_S$, and a fully evaluated expression **skip**,

$$\frac{\langle b, \mathbf{skip}, \mathcal{P}\rangle \hookrightarrow \langle f, f', \mathcal{P}'\rangle}{\langle \epsilon, b, \mathcal{P}\rangle \rightarrowtail \langle \epsilon, b', \mathcal{P}'\rangle}$$

where b' is f, if $f' \equiv \mathbf{skip}$; otherwise, b' is $f \; || \; f'$. □

The transitions specified by $\rightarrowtail$ are called "internal transitions". They change the state without evaluating the main expression, which is what can be observed by an (external) user of a FOOPS system.

Finally, the semantics of FOOPS with evaluation of expressions in the background is given by the union of the relations $\hookrightarrow_S$ and $\rightarrowtail_S$. We use $\rightarrow_S \subseteq BgConf(S) \times BgConf(S)$ to denote this union; we omit the obvious rules defining it[6].

Using two different relations to define $\rightarrow_S$ might seem unnecessarily complicated. But this clearly separates transitions caused by the main expression from transitions generated by the expression in the background. This simplifies the definition of certain operators that should not consider internal transitions. For example, this is the case of `[_]` and `_[]_`. Using only one relation complicates the semantic definition. In fact, contrasting to what has been done here, it wouldn't be possible to define $\rightarrow$ using some of the rules introduced in the previous sections.

11 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, atomic execution, evaluation of expressions as background processes, auto-methods, non terminating methods, and a mechanism for object protection.

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 [40, 28]. Here we only consider the semantics of "flat" specifications; that is, specifications without module importation or generic parameters.

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. This approach uses all 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. It then becomes simple to define the basic operations

[6]Note that $\rightarrow_S$ is overloaded in this text.

on states. 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.

We have also justified the semantics of some constructs comparing to alternatives approaches. Perhaps surprising is the comparison between "true concurrency" and interleaving. We have argued that these approaches are equivalent in the context of FOOPS, given some mild assumptions on the notion of equivalence of programs adopted for the language. This has the interesting consequence that we can use the simpler interleaving model for reasoning about true concurrency. That's what should be expected since it's desirable to specify systems of distributed objects without worrying whether other computations are being carried out simultaneously. This result could probably be generalized for other concurrent object-oriented languages.

Along with the semantic description, we have clarified many concepts and phenomena related to object-oriented languages. In particular, the definition of the operational semantics raised the following technical points about FOOPS:

- there must be syntactical constraints on axiom conditions and DMAs RHS;
- the object creation operations shouldn't have arguments for automatic initialization of attributes; and
- invariants are just annotations;

We have also briefly discussed how the semantics suggests an appropriate programming style for FOOPS, and how to avoid introducing inconsistencies in specifications.

The semantics described in this text is part of a formal definition of FOOPS. So, it's useful as a formal basis for deriving implementations and tools for FOOPS (see [2] for the details on the derivation of a symbolic simulator 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 fact, 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 many applications [3]. The semantics is also useful for reasoning about general properties of FOOPS programs.

11.1 Related Work

Most of the FOOPS features considered here are not considered by other alternatives for the semantics of FOOPS, like the reflective semantics [18] and the sheaf semantics [44, 29, 4]. In fact, we are not aware of a formal semantics for a language integrating all those features. In particular, we haven't seen any proposal similar to the mechanism for object protection described here. Moreover, it seems that the semantics of evaluation in the background and atomic evaluation for a concurrent language hasn't been formalized before.

The basic idea about FOOPS reflective semantics, as described in [18], 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. Essentially, this defines an operational semantics for FOOPS using the functional part of the language, which has a denotational semantics based on order-sorted algebra [19] and an operational semantics given by order-sorted rewriting [28]. In other words, this can be seen as providing a simulator for FOOPS written in OBJ. Using this approach, reasoning about FOOPS programs is essentially reduced to order-sorted deduction [19], where part of the equational theory, given by the information stored in the database, changes with time. Comparing with FOOPS reflective semantics, in this text we use a more abstract approach [39], which is also more appropriate for giving the semantics of concurrency, nondeterminism, and other aspects of FOOPS not discussed in [18].

The most complete work on a sheaf theoretic semantics for FOOPS is [4]. In this work, only a subset of FOOPS is considered; in particular, inheritance, dynamic binding, and atomicity are not considered. In fact, contrasting to the semantics presented here, it seems that the semantics of the atomicity operator cannot be given in a simple and abstract way using the approach of [4]. Also, the semantic description using sheaf theory is much longer than an equivalent operational semantics description. The advantage of using sheaf theory for defining the semantic of FOOPS seems to be the rich mathematical structure associated to sheafs; however, it's not clear yet how this can be used.

A lot of effort has been done in order to give semantics for object-oriented languages [1, 6, 30, 7, 43, 8, 13, 27]. Most of the work in this area uses mathematical models based on set theory [7], metric spaces [1], category theory [30, 8], and hidden order-sorted algebra [13]. The exceptions are [6], which defines an assertional style proof system, and [43, 27], which give the semantics in terms of a process algebra based on an operational semantics [33]. By contrast, the work developed here is based on the simple frameworks of structural operational semantics [39] and OSA [19]. Similar approaches using some of those frameworks are used in works on process algebras [32, 36], imperative languages [37], and a general notation for giving the semantics of programming languages [34].

Because of the details associated to the semantics of object identification, and the lack of a fully abstract mathematical model for interleaving, operational semantics seems to be quite adequate for specifying the semantics of a language like FOOPS. In fact, this can be easily and concisely done, being still possible to reason about the semantics in an pragmatical way. Also, giving the semantics of FOOPS in terms of a process algebra is not worth, since it doesn't seem to be possible to use the algebra to reason about the semantics and programs; for that purpose, one has actually to use the (operational) semantics associated to the process algebra (see [27]).

11.2 Further Research

The language and semantics described here could be extended and revised in the following aspects:

- In addition to DMAs and IMAs, it would be interesting to consider the effects of other kinds of axioms; this might be useful for specifications in general.

- Perhaps it would be more appropriate to understand DMA and IMA conditions as *pre-conditions* (the meaning of an operation is undefined for a particular state if its unique related axiom has a pre-condition that is not valid in that state), rather than as *enabling conditions* (an operation blocks in a state where its unique related axiom enabling condition is not valid); this would provide a higher degree of underspecification to FOOPS specifications.

- Dynamic binding of derived attributes could be provided by adding a specific rule for evaluation of derived attributes, similar to the rule for evaluation of methods specified by IMAs; however, note that attributes should be atomically evaluated.

- A complete semantics for updating of multi-argument stored attributes should be developed.

- The mechanism for object protection could be extended; also, its suitability and expressiveness should be better explored by using it in practice.
- The development of large applications might suggest the addition of some application specific method combiners to the language.
- It might be worth investigating the consequences of adopting a more restricted computational model for FOOPS; for example, this could be achieved by allowing only one method to be executed in an object at a given time.

Acknowledgements

Thanks to Steve Schneider and Grant Malcolm for giving many suggestions which helped to simplify the semantics described here. The authors are grateful to Steve Schneider and Adolfo Socorro for carefully reading and correcting an early version of this text The first author 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, supported by the IS-CORE project. Special thanks go to José Meseguer and Răzvan Diaconescu for most of the text in Section 2, which was copied from [19], [16], and [15].

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. A symbolic simulator for FOOPS. Technical report, Oxford University, Computing Laboratory, Programming Research Group, May 1995. To appear.

[3] Paulo Borba and Joseph Goguen. Refinement of concurrent object-oriented programs Technical Report PRG-TR-17-94, Oxford University, Computing Laboratory, Programming Research Group, November 1994. To appear in the proceedings of the BCS/FACS workshop on formal aspects of object-oriented programming, London, December 1993.

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

[5] Stephen Coffin. *UNIX System V, Release 4, the Complete Reference*. Osborne/McGraw-Hill, 1991.

[6] 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.

[7] 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.

[8] Hans-Dieter Ehrich, Joseph Goguen, and Amilcar Sernadas. A categorial theory of objects as observed processes. In J.W. de Bakker, Willem P. de Roever, and Gregorz Rozenberg, editors, *Foundations of Object Oriented Languages*, pages 203–228. Springer-Verlag, 1991. Lecture Notes in Computer Science, Volune 489; Proceedings, REX/FOOL Workshopfli, Noordwijkerhout, the Netherlauds, May/June 1990.

[9] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[10] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE–10(5):528–543, September 1984.

[11] Joseph Goguen. An algebraic approach to refinement. In Dines Bjorner, C.A.R. Hoare, and Hans Langmaack, editors, *Proceedings, VDM'90: VDM and Z—Formal Methods in Software Development*, pages 12–28. Springer-Verlag, 1990. Lecture Notes in Computer Science, Volume 428.

[12] 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.

[13] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.

[14] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 11:159–191, 1992.

[15] Joseph Goguen. *Theorem Proving and Algebra*. to appear.

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

[17] Joseph Goguen and Grant Malcolm. Proof of correctuess of object representations. In A.W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*. Prentice Hall, 1994.

[18] Joseph Goguen aud 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.

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

[20] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementatioo of abstract data types. Technical Report RC 6487, IBM T.J. Watsou Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80–149.

[21] 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.

[22] 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.

[23] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.

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

[25] John Hogg. Islands: Aliasing protection in object-oriented languages. In *[38]*, 1991.

[26] Cliff Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, Department of Computer Science, University of Manchester, 1992

[27] 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.

[28] 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-Verlag, 1988. Lecture Notes in Computer Science, Volume 317.

[29] Grant Malcolm. A sheaf semantics for FOOPS. To appear, 1995. Oxford University, Computing Laboratory, Programming Research Group.

[30] José Meseguer. A logical theory of concurrent objects. In *Proceedings of ECOOP-OOPSLA 90 Conference on Object Oriented Programming*, pages 101–115. ACM, 1990

[31] 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.

[32] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[33] 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.

[34] Peter Mosses. *Action Semantics*. Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

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

[36] Elie Najm and Jean-Bernard Stefani. Object-based concurrency: a process calculus analysis. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, Theory and Practice of Software Development*, volume 494 (1) of *Lecture Notes in Computer Science*, pages 359–380. Springer-Verlag, 1991.

[37] Pawel Paczkowksi. *Annotated Transition Systems for Verifying Concurrent Programs*. PhD thesis, University of Edinburgh, April 1991.

[38] Andreas Paepcke, editor. *OOPSLA'91*. ACM, ACM Press, November 1991.

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

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

[41] Gert Smolka, Werner Nutt, Joseph Goguen, and José Meseguer. Order-sorted equational computation. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 299–367. Academic, 1989. Preliminary version in *Proceedings*, Colloquium on the Resolution of Equations in Algebraic Structures, held in Lakeway, Texas, May 1987, and SEKI Report SR-87-14, Universität Kaiserslautern, December 1987.

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

[43] 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.

[44] 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-Verlag, 1992. Proceedings, ECOOP'91 Workshop, Geneva, July 1991.