Algebraic Semantics of Object-Oriented Languages

Alexander Fronk

[1]Software-Technology, University of Dortmund, Germany

fronk@LS10.de

Abstract. *We develop an algebraic semantics of a sample core object-oriented language aggregating basic object-oriented features. We transform programs into differently structured algebraic specifications thereby maintaining the program's object-oriented structure. The semantics of these specifications, viz algebras, serve as a semantics for core object-oriented concepts. Static aspects are captured by these specifications, whereas dynamic ones are reflected on many-sorted algebras. We focus on the novelties of this approach, and end by discussing how it can be used to describe popular object-oriented languages.*

1. Introduction

Studying the semantics of programming languages has a long tradition in computer science. Various approaches, such as operational, denotational or axiomatic calculi use various formalisms with various objectives on several levels of abstraction. In the last two decades, algebraic specifications have frequently been used to study functional as well as imperative languages, and, in particular, object-oriented ones, thereby often focusing on specific aspects and concepts of this programming paradigm. The concept of *classes* introduces a structuring mechanism, and allows to talk about various relations between them, such as aggregation, locality, genericity, or inheritance. Class instances, i.e. *objects*, request for being shared, have attributes being updated, or can be used in different contexts, i.e. they allow for polymorphism. Hence, studying the semantics of object-oriented languages has to address both these static and dynamic aspects.

1.1. Σ-Algebra and Object-Orientation

In this paper, we develop an algebraic semantics of a sample core object-oriented language, *DoDL*, tailored to describe hyperdocuments in an object-oriented way. We employ the 1980's Σ-algebra style and adapt it to object-orientation. Our approach encompasses two steps. First, we develop an algebraic semantics of basic object-oriented concepts, into which, secondly, the semantics of the language's imperative parts is embedded. That is, we formalize classes, attributes and methods as well as class relations such as aggregation, locality, genericity, and inheritance through suitably structured algebraic specifications. Static aspects are captured on the specification level, whereas dynamic ones are reflected by many-sorted algebras. In the second step, we show how the algebraic formalization of imperative control structures can be integrated, and thus distinguish the object-oriented concepts of the language to structure code from the imperative ones to implement functionality and thus the algorithmic parts of the language. Since the algebraic definition of imperative languages has already been studied (cf. [Broy et al., 1987, Cousineau, 1980]), we concentrate on object-oriented aspects here, such as method invocation, or the *this* and *super* constructs.

1.2. Our Approach

The stringent integration of differently structured algebraic specifications allows to reflect the structuring mechanisms found in object-oriented languages. We use simple specifications to model simple classes, specifications with hidden symbols to model locality, parameterized specifications to model genericity, and hierarchical specifications to model inheritance. Dynamic aspects, such as method lookup or updates, are modeled by appropriate interpretation and hence by many-sorted algebras. We elaborate a ”transform-and-interpret” mechanism to strictly and uniformly cover object-oriented concepts in one formal approach based on the integration of differently structured algebraic specifications and their appropriate interpretation by intertwined algebras. To this extent, our approach distinguishes from others which use flat specifications. Moreover, elaborating rules for both transformation and interpretation allows to employ our approach for formalizing the semantics of ”real” object-oriented language, such as EIFFEL, BETA, JAVA, or C++, by adopting the rules to these languages.

Using algebraic semantics is further motivated by the fact that programs are usually developed from specifications by refinement [Ehrig and Kreowski, 1999]. Starting with a program and capturing its semantics by a specification, the latter can be refined to improve the implementation. We successfully applied this idea to *DoDL*-programs.

The language *DoDL* is a domain specific language for constructively describing hyperdocuments [Doberkat, 1996] using core object-oriented features and concepts. Details of the language and how to use it is elaborated in [Fronk, 2002a]. It was shown in this thesis that the semantics of *DoDL* can easily be formalized without considering hypermedial aspects. The present paper is a condensed version of this work with focus on the semantics of classes, inheritance, method lookup and updates. A more detailed version of this paper can be found in [Fronk, 2002b]. We use *DoDL* as a sample language to comprehensibly focus on our approach.

The semantics presented has been encoded within a compiler system where the transformation rules directly determine code generation [Fronk and Pleumann, 1999]. The compiler construction, however, is not subject to the present discussion. This paper is organized as follows: Section 2 discusses related work, algebraic preliminaries are given in Section 3 The language *DoDL* is introduced by examples. Selected semantic aspects of *DoDL* are studied in Section 4 The paper concludes in Section 5

2. Related Work

In the last two decades, algebraic specifications have frequently been used to study denotational semantics of functional [Broy and Wirsing, 1982, Williams, 1982] and imperative [Broy et al., 1987] languages. Algebraic specification languages inherently provide algebraic semantics and thus mathematical objects denoting syntactical constructs (cf. [Wirsing, 1995]). Algebraic semantics are also used in the field of abstract state machines to formalize the machine model underlying an operational semantics [Gurevich, 1991]. Based on this approach, a semantics for the C programming language is proposed in [Gurevich and Huggins, 1992].

In the context of algebraic specification languages, a variety of object-oriented aspects and concepts have been studied, and many results carry over to object-oriented programming languages as well. A survey of some interesting work discussing objects, their states, classes, and inheritance can be found in [Fronk, 2002b] giving a short overview on the appealing possibilities algebraic specifications offer for object-oriented issues.

3. Algebraic Preliminaries

For the reader's convenience, we introduce some preliminaries and fix the notation used in this paper. The reader familiar with algebraic specification may proceed to Section 4 Details on algebraic specifications can be found in text books such as [Klaeren, 1983].

Important Conventions: The collection of all specifications forms a class in set theory. In order not to conflict both with the notions of class and object used in object-orientation, as well as with fundamental mathematical questions, we restrict ourselves to a fixed universe of specifications, where the model classes defined in the following can be understood as sets. We further assume a specification NAT for data type `integer` with a sort nat, and a specification $BOOL$ for data type `bool` with a sort $bool$ together with the usual constants $true$ and $false$ to be given in each specification.

A **signature** is a tuple $\Sigma = \langle S, \Gamma \rangle$ where S is a set of **sorts**, $S = sorts(\Sigma)$, and Γ is a set of **operation symbols**, $\Gamma = opns(\Sigma)$. Each operation symbol is assigned a **characteristics** over $S^* \times S$. Variables are **S-sorted**. $X = \{X_s\}_{s \in S}$ denotes a **S-indexed** family of sets X_s of **variables** for each $s \in S$. The set of **S-sorted** Σ**-terms** over s, $\mathcal{T}(\Sigma, \mathrm{X})_s$ for short, is defined as usual. An **algebraic specification**, SP, is a tuple $\langle \Sigma, E \rangle$ consisting of a signature, $\Sigma = sig(SP)$, and a set of formulas over Σ, $E = axms(SP)$, called **axioms**. A specification $\langle \Sigma', E' \rangle$ is called **subspecification** of $\langle \Sigma, E \rangle$, $\langle \Sigma', E' \rangle \subseteq \langle \Sigma, E \rangle$ for short, if $\Sigma' \subseteq \Sigma$ and $E' \subseteq E$ hold. **Formulas** are defined as usual. We denote the **set of all well-formed formulas** by $WFF(\Sigma)$.

A Σ-**algebra**, $\mathcal{A}$, is a pair $(\{A_s\}_{s \in S}, \{f^{\mathcal{A}}\}_{f \in \Gamma})$ consisting of a family $\{A_s\}_{s \in S}$ of non-empty **carrier-sets**, A_s, for each $s \in S$, and a set $\{f^{\mathcal{A}}\}_{f \in \Gamma}$ of **operations** $f^{\mathcal{A}} : A_{s1} \times \ldots \times A_{sn} \rightarrow A_s$ for each $f : s_1 \times \ldots \times s_n \rightarrow s \in \Gamma$. $\mathcal{A}$ is a **model** of a specification, $\langle \Sigma, E \rangle$, if $\mathcal{A}$ satisfies each formula $e \in E$. The **set of all models** of $\langle \Sigma, E \rangle$ is denoted by $Alg(\Sigma, E)$. The **loose semantics** of a specification $SP = \langle \Sigma, E \rangle$, $Mod(SP)$ for short, is defined as the set $Alg(\Sigma, E)$.

Let $\Sigma = \langle S, \Gamma \rangle$ and $\Sigma' = \langle S', \Gamma' \rangle$ be two signatures with $\Sigma \subseteq \Sigma'$, and let $\mathcal{A}'$ be a Σ'-algebra. The Σ-algebra $\mathcal{A}' |_{\Sigma}$ is called Σ**-reduct** of $\mathcal{A}'$, if for each $s \in S$ the carrier-set $(\mathcal{A}' |_{\Sigma})_s$ is defined as A'_s, and for each $f \in \Gamma$ the operation $f^{\mathcal{A}'|_{\Sigma}}$ is defined as $f^{\mathcal{A}'}$.

Let SP and SP' be two specifications, such that all pairwise equal operation symbols have the same characteristics. The specification-building operation $import \ \_ \ into \ \_ : SPEC \times SPEC \rightarrow SPEC$ is defined by $sig(import \ SP \ into \ SP') := sig(SP) \cup sig(SP')$ and by $Mod(import \ SP \ into \ SP') := \{\mathcal{A} \in Alg(\Sigma_{import \ SP \ into \ SP'}) \mid \mathcal{A} |_{\Sigma_{SP}} \in Mod(SP)\}$.

Let $SP_A = \langle \Sigma, E \rangle$ be a specification, and let SP_B be a subspecification of SP_A. The pair $HS = \langle SP_A, SP_B \rangle$ is called **hierarchical specification**. SP_A is called **subclass specification**, SP_B is called **superclass specification** of HS. Let $sig(HS) = \Sigma$. The **loose semantics** of HS is defined as: $Mod(HS) := \{\mathcal{A} \in Mod(SP_A) \mid \mathcal{A} |_{\Sigma_{SP_B}} \in Mod(SP_B)\}$.

4. Algebraic Semantics

Our approach to an algebraic semantics works as follows. Transformation rules establish semantic functions for each syntactic concept offered by *DoDL*, and thus convert classes and their relations into structured algebraic specifications. The integration of simple algebraic specifications, hierarchical specifications, specifications with hidden symbols, and parameterized specifications allows formalizing these concepts as close as possible to the object-oriented paradigm. Thereby, our approach differs from flat specification approaches as usually used in the literature. With each algebraic specification we can obtain ”standard” interpretations which satisfy each axiom given. To establish a semantics reflecting the object-oriented paradigm, however, we define a set of interpretation rules and obtain specific many-sorted algebras as models for structured specifications. Notions such as *object* and *type*, *redefinition*, *method lookup* and *polymorphism* can be represented by these algebras. Hence, algebraic specifications can be understood as an intermediate language between a *DoDL*-program and its semantics.

Both transformation and interpretation rules can be formulated as an inference system. This may lead to a formal type system for *DoDL* (similar to an operational semantics of JAVA given by Drossopoulou and Eisenbach [Drossopoulou and Eisenbach, 1998]), or to the possibility to argue formally about properties of transformation and interpretation. For the time being, we do not exploit these advantages and prefer a natural language notation for simplicity. Mathematical definitions of the transformation rules can be found in [Fronk, 2002b].

4.1. Simple Classes

We introduce *DoDL* by example. A class definition encompasses the name of the class which can be generic or may inherit from another class. Simple classes are not generic, do not inherit from any class, and do not define local classes. A class provides three optional sections: a `declare`-section defines local classes; an `attributes`-section declares attributes, and a `construct`-section defines methods the bodies of which allow variable declaration and assignment, recursive method invocation, and the usual control structures such as sequences, alternatives together with `for`- as well as `while`-loops in a JAVA-like style. *DoDL* provides the types `nat`, `bool` and `string`, defined as usual. Further types can be self-defined using classes.

```
class grower is                  // a simple class
    attributes num: nat;
    construct
        nat get(void){ return num; }
        grower set(nat val){ num = val; }
        grower grow(void){ num = num + 1; }
end grower;
```

Transformation Rule 1 *A simple DoDL-class is transformed into a flat algebraic specification carrying the name of the class in capital letters.*

Let `c` be a simple *DoDL*-class and $C = \langle \Sigma_C, E_C \rangle$ its transformation. In the following, we always refer to this convention. Attributes can be understood as methods with arity zero. Hence, we transform each attribute declaration into a suitable operation:

Transformation Rule 2 *An attribute declaration of the form* `ident: type` *is transformed into an operation in $opns(C)$ of the form $ident : c \to type$. The identifiers c and $type$ are added to $sorts(C)$.*

The sort c corresponds to the name of the class being transformed. Method signatures contained in the `construct`-section are transformed as follows:

Transformation Rule 3

1. *An n-ary method declaration of the form* `type ident(type`$_1$ `ident`$_1$`, ..., type`$_n$ `ident`$_n$`)` *is transformed into a $(n+1)$-ary operation in $opns(C)$ of the form $ident : c \times type_1 \times \ldots \times type_n \to type$.*
2. *A unary method declaration of the form* `type ident(void)` *is transformed in to a unary operation in $opns(C)$ of the form $ident : c \to type$.*
3. *The identifiers c, $type$, and $type_i$, $i = 1, \ldots, n$ are added to $sorts(C)$.*

We apply the transformation rules on class `grower`. By Rule (1), we yield the flat specification shown below:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\Sigma_{GROWER} =} \\
\mathbf{sorts} & grower \\
\mathbf{opns} & num : grower \to nat, \\
 & get : grower \to nat, \\
 & set : grower \times nat \to grower, \\
 & grow : grower \to grower \\
\mathbf{vars} & g : grower, v : nat \\
\mathbf{axms} & num(g) = get(g), \\
 & get(set(g, v)) = v, \\
 & grow(g) = set(g, num(g) + 1)
\end{array}
$$

Sort $grower$ is introduced following Rule (2). The same rule is responsible for transforming the attribute declaration into an operation $num : grower \to nat$. The signatures of methods `get`, `set` and `grow` are transformed by Rule (3). For the time being, axioms are given *a posteriori* such that they reflect the semantics of these operations. Transformation of method bodies is discussed in Section 4.3 The methods `set`, which corresponds to variable assignment, and `grow` update attribute `num` and create a new object. This specification is interpreted by algebras serving as a mathematical model for simple *DoDL*-classes. Whenever an update takes place, a new interpretation of operation num is needed and thus a new algebra is created. Hence, we use the Hoare view on objects based on observability of updates [Hoare, 1993]. Sharing of objects is discussed in Section 5

We are not interested in initial or terminal models and use a loose semantics approach. A loose semantics allows to leave some carrier-sets uninterpreted, especially those used for technical reasons only; in the above example, sort $grower$ needs no specific interpretation. We define the semantics of simple *DoDL*-classes as follows:

Definition 1 *Given a simple DoDL-class, c, and its transformation, C, the* ***semantics*** *$[\![c]\!]$ of c is defined as the set $Mod(C)$ of all models of C. We assume that $[\![\_]\!]$ is a mapping from the set of all syntactically correct DoDL-classes into the set of all model sets.*

As an invariant, we can usually employ the axioms themselves to interpret operations. Other interpretation rules need to be defined explicitly. In the following, let $\mathcal{C}$ be a Σ_C-algebra. The sorts obtained by class name transformation are loosely interpreted:

Interpretation Rule 1 *Each sort $s \in sorts(C) \setminus \{nat, string, bool\}$ is interpreted arbitrarily in $\mathcal{C}$.*

The predefined types `nat`, `string`, and `bool` are interpreted in the usual way, that is, for each Σ_C-algebra $\mathcal{C}$ we define $C_{nat} = \mathbb{N}_0$, $C_{bool} = \{true, false\}$, and C_{string} as the free semi-group over letters and digits with concatenation. We can freely interpret transformed attribute declarations:

Interpretation Rule 2 *Each operation in $opns(C)$ of the form $ident : c \to type$, where $type$ is a sort in $sorts(C)$, and `ident: type` is an attribute declaration in `c`, is interpreted arbitrarily in $\mathcal{C}$.*

Let $\mathcal{G}$ be a Σ_{GROWER}-algebra. By Interpretation Rule (1), sort $grower$ is interpreted arbitrarily. Following Interpretation Rule (2), the operations num, $grow$, get and set are interpreted as follows for all $g, g' \in G_{grower}$ and for all $n \in G_{nat}$ (it is easy to prove that $\mathcal{G}$ is a model for $GROWER$):

$$num^{\mathcal{G}}(g) = get^{\mathcal{G}}(g) \qquad set^{\mathcal{G}}(g,n) = set(g,n)$$
$$get^{\mathcal{G}}(set(g,n)) = n, \qquad grow^{\mathcal{G}}(g) = set^{\mathcal{G}}(g, num^{\mathcal{G}}(g) + 1)$$

4.2. Inheritance

In *DoDL*, multiple inheritance is not allowed. We assume that attributes, methods and local classes of a superclass are replicated within the subclass. New attributes, methods and local classes can be added. If a method occurring in a subclass has the same signature as in its superclass, the method is redefined. Different parameter types or a different number of parameters lead to overloading.

Inheritance by specialization [Parisi-Presicce and Pierantonio, 1994], that is, inheritance modelled by model-inclusion on algebras, is proposed for hierarchical specifications by Wirsing [Wirsing et al., 1983]. We use them to express inheritance in *DoDL*. Further, we distinguish between **subclass relations** and **subtype relations**. We understand a subclass relation as a partial order on classes induced by subclass declaration, called *class inheritance*. A subtype relation, however, deals with class instances. A type is thereby understood as the set of all instances of a class, such that each instance of a subclass is an instance of its superclass. This property is called *substitution* and characterizes *type inheritance*, whereas polymorphism is characterized by different kinds of substitution [Cardelli and Wegner, 1985, Abadi and Cardelli, 1996]. In *DoDL*, class inheritance induces type inheritance. The designator `this` follows `self` in SMALLTALK.

We use a late binding semantics, i.e. the type of a caller `c` of a method `m` determines the class `m` is chosen from. This requires a suitable lookup mechanism. Looking up the correct method in a class hierarchy starts at that class `C` the caller is declared an instance of, and follows the hierarchy via superclass definition. Lookup stops at a class `D` that defines `m`, and for which no other class `D'` exists in the hierarchy that redefines `m` and is defined between `D` and `C`. With this strategy, methods called using `this` or `super` cannot always be determined at compile time, i.e. these designators cannot always be respected during the transformation process. Defining a semantics even happens prior to compilation. Therefore, we define suitable axioms for methods using `this` or `super`, and simulate a lookup mechanism on algebras.

4.2.1. Superclass Specification

Since any class can be used as superclass without being affected thereby, and due to the structure of hierarchical specifications, we define the following rule:

Transformation Rule 4 *A DoDL-class that is subclass to any other class* `c` *is transformed into a hierarchical specification, where the superclass specification is given by the transformation of* `c`*.*

4.2.2. Subclass Specification

It remains to consider how to obtain the subclass specification. A superclass may contain local classes. Then, the superclass is transformed into a specification with hidden symbols (see [Fronk, 2002a, Fronk, 2002b]). Local classes are inherited by subclasses. Thus, these subclasses must be represented by specifications with hidden symbols as well. Even if a superclass does not declare local classes, its subclasses may define them in their `declare`-sections. Then again, a subclass has to be transformed into a specification with hidden symbols. Only in case both a subclass and its superclass are simple (apart from inheritance), the transformation into a flat specification is sufficient. Nonetheless, each flat specification can be denoted as a specification with hidden symbols of the form $\langle \Sigma, \Sigma, E \rangle$. Since we do not discuss local classes in the present paper, we define the following rule:

Transformation Rule 5 *The subclass specification of a hierarchical specification is denoted as a flat specification. Symbols and axioms of the superclass specification are replicated and incremented by the subclass under consideration.*

The increment is formed by those local classes, attributes and methods that are defined in the subclass. The replication of the superclass is the identity if local classes are omitted. Henceforth, it is sufficient to consider the increment when a transformation is carried out. This leads to transformation rules analogous to those defined in the previous section. Nonetheless, some additional inheritance specific aspects have to be considered. In the following, let `subc` be a subclass of `c`, and let $SUBC = (SUBC', C)$ denote its ongoing transformation, where $SUBC'$ is the superclass replication the following transformation rules refer to. To ensure that C (the already transformed superclass) is a subspecification of $SUBC'$, it is sufficient to import C into $SUBC'$. Hence, the transformation of an inheriting class is anti-monotonous w.r.t. inclusion.

The signatures of inherited methods have to be adapted to the subclass in order to make these methods applicable to subclass instances.

Transformation Rule 6 *Any n-ary method declaration in* `c` *of the form* `type ident(type`$_1$ `ident`$_1$`, ..., type`$_n$ `ident`$_n$`)`, $n \geq 0$, *is transformed into a $(n+1)$-ary operation in $opns(SUBC')$ of the form $ident : subc \times type_1 \times \ldots \times type_n \rightarrow type$. The identifier $subc$ is added to $sorts(SUBC')$.*

We characterize polymorphism through substitution, and therefore each instance of a class must be an instance of its superclass. This leads to two operations which are vital for algebraic transformation (cf. [Cerioli et al., 1999]).

Transformation Rule 7 *The injection $inj : subc \rightarrow c$ and the projection $proj : c \rightarrow subc$ are added to $opns(SUBC')$. An axiom of the form $proj(inj(s)) = s$ is required in $axms(SUBC')$ for all $s \in subc$.*

4.2.3. Non-Redefined Methods

If a method is redefined, suitable axioms have to be given (see Sect. 4.3). If a non-redefined method `m` in `subc` makes use of a method redefined in `subc`, we call `m` **indirectly redefined**, and hence suitable axioms have to be given here as well. If a method is inherited and not (indirectly) redefined, its semantics carries over to the subclass. We respect different method types:

Transformation Rule 8

1. *For each non-redefined operation of the form $f : subc \times type_1 \times \ldots \times type_n \rightarrow c$ in $opns(SUBC')$, which does not use a method redefined in $subc$, there is defined an axiom in $axms(SUBC')$ for all $s \in subc$, $p_i \in type_i$, $i = 1, \ldots, n$, of the form $proj(f(s, p_1, \ldots, p_n)) = f(in(s), p_1, \ldots, p_n))$*
2. *For each non-redefined operation of the form $f : subc \times type_1 \times \ldots \times type_n \rightarrow type$ in $opns(SUBC')$ with $type \neq c$, which does not use a method redefined in $subc$, there is defined an axiom in $axms(SUBC')$ for all $s \in subc$, $p_i \in type_i$, $i = 1, \ldots, n$, of the form $f(s, p_1, \ldots, p_n) = f(in(s), p_1, \ldots, p_n)$*

The rule's first part delivers a subclass instance for non-redefined methods called by a subclass instance. The result of the method is by definition an instance of its superclass.

4.2.4. A Sample Transformation

We transform the following *DoDL*-class, `sponger`, which is a subclass of `grower`. A subclass definition in *DoDL* uses an `is-with` phrase.

```
class sponger is grower with                     // a subclass
    attributes value: nat;
    construct
        nat getV(void){ return value; }

        sponger set(val_1, val_2: nat){          // overloaded
            super.set(val_1); value = val_2; }

        sponger grow(void){ num = num + value; } // redefined

        nat sponge(val_1, val_2: nat){
            this.set(val_1, val_2);
            super.grow(); this.grow();
            return this.get(); }

        nat main(void){ this.sponge(7, 8); }
end sponger;
```

Class `sponger` introduces an attribute `value` and a method `getV`. Method `set` is overloaded since its signature has changed, and uses method `set` from class `grower`. Method `grow` is redefined, method `get` is not, and `main` calls `sponge` by value. The transformation of `sponger` yields a hierarchical specification of the form $SPONGER = (SPONGER', GROWER)$, where $SPONGER'$ is as shown on the next page.

With Transformation Rule (4), part 2, the superclass specification is as given in specification Σ_{GROWER}. Transformation Rule (6) rewrites the operations in $GROWER$

$\Sigma_{SPONGER'} = \textbf{import}\ \Sigma_{GROWER}\ \textbf{into}$

sorts	$sponger$	
opns	$num : sponger \rightarrow nat,$	(by Rule (6))
	$get : sponger \rightarrow nat,$	(by Rule (6))
	$set : sponger \times nat \rightarrow grower,$	(by Rule (6))
	$grow : sponger \rightarrow grower,$	(by Rule (6))
	$value : sponger \rightarrow nat,$	(by Rule (2))
	$getV : sponger \rightarrow nat,$	(by Rule (3))
	$set : sponger \times nat \times nat \rightarrow sponger,$	(by Rule (3))
	$sponge : sponger \times nat \times nat \rightarrow nat,$	(by Rule (3))
	$main : sponger \rightarrow nat,$	(by Rule (3))
	$inj : sponger \rightarrow grower,$	(by Rule (7))
	$proj : grower \rightarrow sponger$	(by Rule (7))
vars	$s : sponger, g : grower, n, n_1, n_2 : nat$	
axms	$num(s) = num(inj(s)),$	(by Rule (8).2)
	$get(s) = get(inj(s)),$	(by Rule (8).2)
	$proj(set(s, n)) = set(inj(s), n),$	(by Rule (8).1)
	$value(s) = getV(s),$	
	$getV(set(s, n_1, n_2)) = n_2,$	
	$grow(s) = inj(set(s, num(s) + value(s), value(s))),$	
	$sponge(s, n_1, n_2) = get(grow(grow(inj(set(s, n_1, n_2))))),$	
	$main(s) = sponge(s, 7, 8),$	
	$proj(inj(s)) = s$	(by Rule (7))

for usage with sort $sponger$. Rules (2) and (3) take care for the increment of class `grower`, i.e. the features defined in `sponger`, as usual. The injection and projection together with the required axiom are added by Rule (7). The attribute declaration `num` and method `get` are not redefined, and thus Rule (8).2 fixes the required axioms. The method `set` in class `grower` has method type `grower` and is rewritten by Rule (6). Therefore, projection applies and is fixed by Rule (8).1. The methods introduced in class `sponger` need fresh axioms explained for method `sponge`. Its sequence of method invocations can be rewritten as `(((this.set(val_1, val_2)).super.grow()).grow()).get()`. This expression in turn reads algebraically as: $get(grow(grow(super(set(n_1, n_2)))))$. To access the correct sorts and to reach the correct super-methods, we use injection to replace the `super` construct and consequently yield the given axiom.

4.2.5. Dynamic Aspects of Inheritance

In the following, let $\mathcal{S}$ be a Σ_{SUBC}-algebra, and $\mathcal{C}$ a Σ_C-algebra. Since a subclass is simple (apart from inheritance), the increment uses the same interpretation rules as defined in the previous section. Additionally, we need interpretation rules for injection and projection to realize dynamic method lookup.

Interpretation Rule 3 *For the carrier-sets S_{subc} and S_c, $S_{subc} \subseteq S_c$ holds.*

Sort $subc$ is interpreted as a subsort of c, and hence each instance of a subclass is an instance of its superclass. A detailed discussion on this aspect is given in [Fronk, 2002b].

Interpretation Rule 4 *For the injection and projection operations we define:*

1. $inj^{\mathcal{S}}(a) = a$, *for each* $a \in S_{subc}$
2. $proj^{\mathcal{S}}(a) = \begin{cases} a, & \textit{if } a \in S_{subc} \\ \perp, & \textit{else} \end{cases}$

Inherited methods have to be interpreted depending on whether they are (indirectly) redefined or non-redefined:

Interpretation Rule 5

1. *Each operation in* $opns(SUBC')$ *of the form* $f : subc \times s_1 \times \ldots \times s_n \to s$ *that is (indirectly) redefined in* $SUBC$ *is interpreted in* $\mathcal{S}$ *for all* $c \in S_c \setminus S_{subc}, a \in S_{s1} \times \ldots \times S_{sn}$ *as follows:*
$$f^{\mathcal{S}}(c, a) = f^{\mathcal{C}}(c, a)$$
2. *For values in* S_{subc}, f *is interpreted following the respective axioms given.*
3. *Any non-redefined operation in* $opns(SUBC')$ *of the same form, for which there exists an axiom in* $axms(SUBC')$ *of the form* $proj(f(s, a)) = f(inj(s), a)$, *with* $s \in \mathcal{S}_{subc}, a \in S_{s1} \times \ldots \times S_{sn}$, *is interpreted in* $\mathcal{S}$ *as follows:*
$$proj^{\mathcal{S}}(f^{\mathcal{S}}(s, a)) = f^{\mathcal{C}}(inj^{\mathcal{S}}(s), a)$$
4. *Any non-redefined operation in* $opns(SUBC')$ *of the same form, for which there exists an axiom in* $axms(SUBC')$ *of the form* $f(s, a) = f(inj(s), a)$, *with* $s \in \mathcal{S}_{subc}, a \in S_{s1} \times \ldots \times S_{sn}$, *is interpreted in* $\mathcal{S}$ *as follows:*
$$f^{\mathcal{S}}(s, a) = f^{\mathcal{C}}(inj^{\mathcal{S}}(s), a)$$

This rule establishes dynamic method lookup as follows. Inherited methods are overloaded in $opns(SUBC')$. Thus, two versions exist with different characteristics: $f : subc \times \ldots$ and $f : c \times \ldots$. In case f is not redefined, the interpretation of $f : subc \times \ldots$ cascades, i.e. it is "forwarded" to the interpretation of $f : c \times \ldots$ by the first part of the above rule. If it is redefined, new axioms are given in $SUBC'$ and search ends here. Furthermore, these interpretation rules have to maintain the semantics for hierarchical specifications. A proof for this property is given in [Fronk, 2002b].

4.2.6. A Sample Interpretation

Let $\mathcal{G}$ be a Σ_{GROWER}-algebra as given above. We interpret specification $SPONGER$ as follows. For non-redefined methods, we follow Rule (5), parts 3 and 4, yielding for each $s \in S_{sponger}, n \in S_{nat}$:

$$num^{\mathcal{S}}(s) = num^{\mathcal{G}}(inj^{\mathcal{S}}(s)) \qquad get^{\mathcal{S}}(s) = get^{\mathcal{G}}(inj^{\mathcal{S}}(s))$$
$$proj^{\mathcal{S}}(set^{\mathcal{S}}(s, n)) = set^{\mathcal{G}}(inj^{\mathcal{S}}(s), n)$$

The redefined method $grow$ is captured by Rule (5), part 1, if $s \in S_{grower} \setminus S_{sponger}$, and by part 2, otherwise:

$$grow^{\mathcal{S}}(s) = \begin{cases} grow^{\mathcal{G}}(s), & \text{if } s \in S_{grower} \setminus S_{sponger} \\ set^{\mathcal{S}}(s, num^{\mathcal{S}}(s) + value^{\mathcal{S}}(s), value^{\mathcal{S}}(s)), & \text{else} \end{cases}$$

By Rule (5), part 1, dynamic lookup is started. Assuming that class `grower` has a superclass defining method `grow` not redefined in `grower`, lookup further cascades to the respective algebras.

The increment, i.e. the features of class `sponger`, is interpreted following the axioms in specification $SPONGER$. For each $s \in S_{sponger}$ and $n, n_1, n_2 \in S_{nat}$ we have:

$$\begin{aligned}
value^{\mathcal{S}}(s) &= getV^{\mathcal{S}}(s) \\
getV^{\mathcal{S}}(set(s, n_1, n_2)) &= n_2 \\
set^{\mathcal{S}}(s, n_1, n_2) &= set(s, n_1, n_2) \\
sponge^{\mathcal{S}}(s, n_1, n_2) &= get^{\mathcal{S}}(grow^{\mathcal{S}}(grow^{\mathcal{S}}(inj^{\mathcal{S}}(set^{\mathcal{S}}(s, n_1, n_2))))) \\
main^{\mathcal{S}}(s) &= sponge^{\mathcal{S}}(s, 7, 8)
\end{aligned}$$

The interpretations of the set-operations determine the structure of the carrier-sets for S_{grower} and $S_{sponger}$. The former contains terms of the form $set(\ldots(set(g, n_1), \ldots), n_l)$, whereas the the latter have the form $set(\ldots(set(s, n_1, n_1'), \ldots), n_l, n_l')$, and are thus term-generated. Note that each value in $S_{sponger}$ is also in S_{grower}, i.e. $S_{sponger} \subseteq S_{grower}$, by Interpretation Rule (3).

Interpretation Rule (4) fixes injection and projection. With the above structure of $S_{sponger}$ and S_{grower}, we can resolve the else-case in $proj$ and obtain:

$$\begin{aligned}
\forall s \in S_{sponger}, n_1, n_2 \in S_{nat} &: inj^{\mathcal{S}}(set(s, n_1, n_2)) = set(s, n_1) \\
\forall s \in S_{sponger} &: proj^{\mathcal{S}}(s) = s \\
\forall s \in S_{sponger}, n_1, n_2, m \in S_{nat} &: proj^{\mathcal{S}}(set(inj(set(s, n_1, n_2)), m)) = set(s, m, n_2)
\end{aligned}$$

It is again easy to prove that $\mathcal{S}$ is a model for $SPONGER$.

4.3. Object Handling and Embedding Imperative Parts

Primitive statements and control structures are used within method bodies. We allow for declaration of variables, sequences, alternatives, as well as `for`- and `while`-loops. Additionally, object creation and updates, references, as well as method invocation and lookup together with the concepts of `this` and `super` are needed. The latter have already been discussed, objects need further investigation, and the method bodies are left to be transformed into suitable axioms. Exactly here, the semantics discussed so far and the semantics of the imperative parts are integrated. An approach to combine formal semantics of these different aspects is e.g. proposed in [Astesiano et al., 2000]. We utilize both algebraic term evaluation and denotational semantics of imperative languages.

4.3.1. Object Handling

In correspondence with the usual characterization of objects, we define an **object** as an *individually identifiable class instance*. Moreover, we use the designators given in attribute declarations to refer to an object. Each class hence describes a set of individuals through attributes and methods. The **state of an object** is given by the set of attribute/value pairs. Methods are responsible to controllably change this state, that is, methods allow to change values in attribute/value pairs and thus update an object. Its **behavior** paraphrases the conditions under which an objects is allowed to change its state. **Initialization** corresponds to the initial assignment of values to attributes, in *DoDL* this is done by set-methods. **Types** are given through model sets, and each instance of a *DoDL*-class `C` corresponds to a model in $Mod(C)$.

Objects are handled as follows. First, the notion of types is extended to pairs of an infinite set of identifiers and a model set, $(IDENT, MOD)$ for short. Identifiers correspond exactly to object names. Referencing is then defined as a partial function $ref : IDENT \,+\!\!\!\longrightarrow\, MOD$. If an object is updated at runtime, its representing model, m, achieves new axioms resulting in a new model m'. Therefore, the values of ref are changed for all instances i with $ref(i) = m$ to $ref(i) = m'$. This behavior corresponds to object references discussed in [Meyer, 1997]. Deletion of an object, i, yields undefined references simulated by an undefined value for $ref(i)$. Changing a reference corresponds to changing a value in ref. A `new`-operation creating an object obj of type t corresponds to adding a pair $(obj \mapsto \mathcal{M})$ to ref where $\mathcal{M}$ has to be a model in $Mod(T)$ (where T is the transformation of t), and obj is in $IDENT$. A technically more complicated yet more detailed approach is discussed in [Reggio, 1991]. So-called *entity algebras* are used to represent dynamic structures and thereby allow for references. Another approach is captured by *evolving algebras* [Gurevich, 1991].

4.3.2. Embedding Imperative Parts

The execution of a *DoDL*-program utilizes the mechanisms usually applied for imperative languages: assignments and method invocation form the primitive statements, control structures are responsible for the order of their execution. As soon as a *DoDL*-program is transformed and an appropriate interpretation is given, program execution reduces to evaluating these interpretations.

The transformation of a *DoDL*-program starts at its main class. Thereby, we follow the transformation rules. Based on these rules, we can define semantic functions, $[\![\_]\!]$, which denote a mathematical object to each syntactic construct. For example, the semantic function for a *DoDL*-program, `prg`, maps `prg` onto the model set $Mod(PRG)$ established by the interpretation of the transformation PRG. Local classes are introduced by a `declare`-section in *DoDL*. This section is given meaning through a function that maps such a section, $declsec$, onto a mapping from local classes, $\texttt{loc}_i$, to their transformations, LOC_i, i.e. $[\![declsec]\!](\texttt{loc}_i) = LOC_i$. Analogously, the `attribute`- and `construct`-section are given a meaning.

For method bodies, imperative constructs have to be formalized algebraically. We utilize denotational semantics for imperative languages. For example, the denotational semantics of an `if`-statement of the form $\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ is given as follows:

$$[\![\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!] = \begin{cases} [\![s_1]\!], & \text{if } [\![b]\!] = true \\ [\![s_2]\!], & \text{else} \end{cases}$$

Speaking algebraically, we have to evaluate the semantics of s_1, if the interpretation of b in a Σ_{BOOL}-algebra $\mathcal{B}$ yields $true$. Otherwise, we evaluate s_2, assuming a ternary operation $\mathsf{if\ \_\ then\ \_\ else\ \_} : bool \times s \times s \rightarrow s$ to be given for each sort $s \in S$ with the usual semantics if $true$ then x else $y =_s x$ and if $false$ then x else $y =_s y$:

$$[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!] := v^*(\textsf{if } b \textsf{ then } s_1 \textsf{ else } s_2) = \begin{cases} [\![s_1]\!], & \text{if } b^{\mathcal{B}} = true \\ [\![s_2]\!], & \text{else} \end{cases}$$

This shows precisely how an `if`-statement is transformed into an axiom. Axioms for s_1 and s_2 are established through transformation, and the semantic functions $[\![s_1]\!]$ and $[\![s_2]\!]$ are given by evaluating suitable axioms, thereby reducing the mapping $[\![\_]\!]$ stepwise to term evaluation v^* defined as usual on terms of the respective sort. For example, a method invocation of the form $\texttt{expr.ident(p}_1, \ldots, \texttt{p}_n\texttt{)}$ is transformed into a term $ident(expr, p_1, \ldots, p_n)$. With slight syntactic variations, method `sponge` of class `sponger` gives an example. The semantics of calling this method corresponds to the evaluation of its transformation. Since interpretations of operations satisfy the axioms given, we generally have

$$\begin{aligned}[\![\texttt{expr.m(p}_1, \ldots, \texttt{p}_n\texttt{)}]\!] &:= v^*(m([\![\texttt{expr}]\!], [\![\texttt{p}_1]\!], \ldots [\![\texttt{p}_n]\!])) \\ &= m^{\mathcal{C}}(v^*(expr), v^*(p_1), \ldots, v^*(p_n))),\end{aligned}$$

where `expr` is an expression denoting an instance of class `C`, and $\mathcal{C}$ is a Σ_C-algebra.

For variable declaration, sequences, and loops we can proceed analogously. Together with the evaluations given above, the semantics of a program is deduced by recursive transformation and evaluation. Hence, our algebraic semantics is denotational, and it can easily be calculated that the following semantics for `main` in class `sponger` holds (some equations are omitted):

$$\begin{aligned}[\![\texttt{main()}]\!] &= v^*(main(s)) = main^{\mathcal{S}}(s) = sponge^{\mathcal{S}}(v^*(s), 7, 8) \\ &= get^{\mathcal{S}}(grow^{\mathcal{S}}(grow^{\mathcal{S}}(in^{\mathcal{S}}(set^{\mathcal{S}}(v^*(s), 7, 8))))) \\ &= get^{\mathcal{G}}(set(set(v^*(s), 8, 8)), 8 + 8)) = 16\end{aligned}$$

5. Conclusion and Further Work

In this paper, we studied the algebraic semantics of a sample object-oriented language, *DoDL*. The approach is based on structured algebraic specifications used to denote the object-oriented concepts of *DoDL*.

First, we defined transformation rules to yield an adequate algebraic specification for each class definable in *DoDL*. Using interpretation rules, we established a mathematical model reflecting the semantics of classes. In a second step, algorithmic parts were embedded into the semantics by using the usual denotational approach to imperative languages. Our approach thus exploits the fact that an object-oriented language introduces code structuring mechanisms not found in imperative languages but extending them. Semantical aspects such as polymorphism and method lookup not expressible on the specification level were simulated on the respective algebras. Those concepts were also given a precise semantics by interpretation rules.

Depending on the specific concepts the language under consideration offers, its semantics may differ from the one presented for *DoDL*. For example, multiple inheritance as provided in C++ is not found in *DoDL* and hence not modelled in this paper;

a different inheritance mechanism as e.g. Beta is equipped with also requires a different formalization. This is done by adapting transformation and interpretation rules to these languages. The process of transformation and interpretation as well as the integration of differently structured algebraic specifications, however, is still applicable. Nonetheless, the goal of this paper was to develop such an adaptable approach to formally defining a semantics for arbitrary object-oriented languages.

Moreover, we have discussed how algebraic specification can be put to use for complex applications such as formalizing a language's semantics. In contrast to the usual task of specifications, we formalize an implementation instead of developing it from a specification. In fact, as soon as an algebraic semantics of a program is established it can be used to re-develop this program with the usual refinement steps.

The rules shown allow to generate mathematical models. A compiled program has to fulfil the properties of such a model. Hence, a model can be used as a test-base for the correctness of a compiled program.

Some concepts need further investigation. For example, abstract classes, virtual methods, interfaces and multiple inheritance are worth a formalization. What follows from references are mutually associated classes. We are confident that our approach is strong enough to capture these aspects to widen the range of the semantic issues discussed here. Our approach shows that algebraic specification and its model sets can be used to give an algebraic semantics to the object-oriented paradigm in a uniform manner.

Acknowledgements. We gratefully thank Prof. Dr. E.-E. Doberkat for supervising the author's PhD thesis and for his comments on this paper.

References

Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer.

Astesiano, E., Cerioli, M., and Reggio, G. (2000). Plugging data constructs into paradigm-specific languages: Towards an application to UML. In Rus, T., editor, *Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science (LNCS)*, pages 271 – 292. Springer.

Broy, M. and Wirsing, M. (1982). Algebraic definition of a functional programming language. *IEEE Transactions on Information Theory*, 17(2):137–161.

Broy, M., Wirsing, M., and Pepper, P. (1987). On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9(1):54 – 99.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522.

Cerioli, M., Mossakowski, T., and Reichel, H. (1999). From total equational to partial first-order logic. In Astesiano, E., Kreowski, H.-J., and Krieg-Brückner, B., editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, pages 31 – 104. Springer.

Cousineau, G. (1980). An algebraic definition for control structures. *Theoretical Computer Science*, 12(2):175–198.

Doberkat, E.-E. (1996). A language for specifying hyperdocuments. *Software - Concepts and Tools*, 17:163–172.

Drossopoulou, S. and Eisenbach, S. (1998). Towards an operations semantics and proof of type soundness for Java. Technical report, University of London, `http://www-dse.doc.ic.ac.uk/projects/slurp`.

Ehrig, H. and Kreowski, H.-J. (1999). Refinement and implementation. In Astesiano, E., Kreowski, H.-J., and Krieg-Brückner, B., editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, pages 201 – 242. Springer.

Fronk, A. (2002a). *Algebraische Semantik einer objektorientierten Sprache zur Spezifikation von Hyperdokumenten*. PhD thesis, Universität Dortmund, 2001, Shaker Verlag.

Fronk, A. (2002b). An approach to algebraic semantics of object-oriented languages. Memorandum 129, Lehrstuhl Software-Technologie, Fachbereich Informatik, Universität Dortmund.

Fronk, A. and Pleumann, J. (1999). Der *DoDL*-Compiler. Memorandum 100, Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie.

Gurevich, Y. (1991). Evolving algebras: An attempt to discover semantics. *EATCS Bulletin*, 43:264 – 284.

Gurevich, Y. and Huggins, J. K. (1992). The semantics of the C programming language. In Börger, E., Jäger, G., Kleine-Büning, H., Martini, S., and Richter, M., editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science (LNCS)*, pages 274 – 308. Springer.

Hoare, C. A. R. (1993). Algebra and models. *ACM SIGSOFT*, 12:1–8.

Klaeren, H. A. (1983). *Algebraische Spezifikation*. Springer.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, 2. edition.

Parisi-Presicce, F. and Pierantonio, A. (1994). Structured inheritance for algebraic class specifications. In Ehrig, H. and Orejas, F., editors, *Recent Trends in Data Type Specifications*, volume 785 of *Lecture Notes in Computer Science (LNCS)*, pages 295 – 309. Springer.

Reggio, G. (1991). Entities: An institution for dynamic systems. In Ehrig, H., Jantke, K. P., Orejas, F., and Reichel, H., editors, *Recent Trends in Data Type Specifications*, volume 534 of *Lecture Notes in Computer Science (LNCS)*, pages 246 – 265. Springer.

Williams, J. H. (1982). On the development of the algebra of funtcional programs. *ACM Transactions on Programming Languages and Systems*, 4(4):733 – 757.

Wirsing, M. (1995). Algebraic specification languages: An overview. In Astesiano, E., Reggio, G., and Tarlecki, A., editors, *Recent Trends in Data Type Specifications*, volume 906 of *Lecture Notes in Computer Science (LNCS)*, pages 81 – 115. Springer.

Wirsing, M., Pepper, P., Partsch, H., Dosch, W., and Broy, M. (1983). On hierarchies of abstract data types. *Acta Informatica*, 20:1–33.