J. Inf. Process. Cybern. EIK 26 (1990) 1/2, 33–48
(formerly Elektron. Inf.verarb. Kybern.)

From Data Types to Object Types[1])

By *Hans-Dieter Ehrich, Amilcar Sernadas* and *Cristina Sernadas*

Abstract: A semantic basis for object-oriented approaches is presented. It provides a mathematical model for objects, object types, and societies of interacting objects, including static and dynamic aspects. The model covers complex objects and object types, providing a deeper understanding of specialization, inheritance, generalization, aggregation and interaction by object sharing, both in terms of structure and dynamics. The presented theory of object types draws some analogies from algebraic data type theory. Categories of objects and object types are defined. Both are shown to be cocomplete. Based on these results, structural relationsships as well as dynamic interaction between objects are expressed in a uniform way.

1. Introduction

Data types are sets of data elements which "belong together" in the sense that the same operations apply. Thus, it is widely accepted to view a data type as a set equipped with operations [20; 31]). A convenient mathematical model for a family of data types is a heterogeneous algebra ([19; 18]). Issues of specification, correctness, parameterization, modularization and implementation have been successfully clarified on this basis, and there are specification languages, design methodologies and design tools along these lines (see [12; 25] for recent textbooks).

Objects and object types in the sense of object-oriented programming are less well understood. A coherent theory is still missing ([1]). The purpose of this paper is to make some basic suggestions towards developing such a theory, drawing some analogies from algebraic data type theory. There are some parallels to ideas in [15], but there are also fundamental differences. The basis of our approach to object dynamics are processes ([21]) which bear some relationship to ω-languages ([30]), and we put special emphasis on object identity ([22]).

Object-oriented programming began as early as 1967 with the simulation language SIMULA ([7]), but the breakthrough is usually attributed to Smalltalk-80 ([17]). According to this development, objects are highly dynamic entities. They are not just data. And they are not just software modules with an interface of named operations and a hidden local state ([24]), although this aspect applies to objects, too. But there is more. Objects are organized into object types which display a sophisticated subtyping structure, together with an appropriate inheritance mechanism ([5]). Moreover, there is the whole world of object dynamics: objects may be created, changed and destroyed, and they may have an internal activity of their own. And, last but not least, there is some mechanism of communication between objects, e.g. by means of messages.

[1]) Extended version of a lecture given at the 4th Workshop "Mathematical Aspects of Computer Science", May 9–13, 1988 in Magdeburg, GDR.

Object-orientation has also entered the database field, but here the emphasis is more on structural aspects, especially complex object structures. Object dynamics is largely neglected ([23; 6]). In fact, there is some confusion about terms; a system with complex objects is not quite the same as an object-oriented system, although there are relationships. Object-oriented databases have attracted much interest (cf. [23; 6; 27]), but there is little concern about theoretical foundations. The suggestions made in this paper specifically aim at providing a theoretical basis for object-oriented databases.

In [29] and [28], we presented different aspects of an object-oriented approach to the design and specification of information systems, covering both structural and dynamic aspects. For our specification language OBLOG under development, we take some care to provide formal specification methods and a rigorous mathematical semantics. Special aspects of object identification by database-like keys have been studied in [9] and [10].

In this paper, we present a mathematical model for objects, object types, and societies of interacting objects. The model covers complex objects and object types, providing a deeper understanding of specialization, inheritance, generalization, aggregation and interaction, both in terms of object structure and object dynamics. Our theory of object types draws some analogies from algebraic data type theory. We define categories of objects and object types. Both are shown to be cocomplete. Based on these results, we express structural relationships as well as dynamic interaction between objects in a uniform way.

Our approach makes moderate use of a few category-theoretic notions. The reader may find it helpful to consult a textbook on category theory, for instance [2], [16] or [26].

2. Motivation and survey

The most prominent example of a data type is stack ([4]). As it happens, stack is also a good example to demonstrate some of the basic ideas about objects. As a data type, a stack with a set E of entries can be modelled as a set $S = E^*$ of finite entry sequences, together with operations like $empty: \rightarrow S$, $top: S \rightarrow E$, $push: S \times E \rightarrow S$, $pop: S \rightarrow S$, etc. with obvious meanings. It is well-known that data types with a "hidden" sort can be viewed as abstract machines ([14]). For a stack, the hidden sort is S representing the internal states, the operations *push* and *pop* represent state transitions, and *top* represents an output function. The constant *empty* represents the initial state.

In an object-oriented view, a stack is not a type, but a single object instance. As an object, a stack also has an internal state which can be changed and observed. So it can be modelled as a state machine ([15]). Concerning the operations on objects, it is natural to adopt an imperative style and distinguish between events which change the state and attributes which associate observations with states. In contrast to the data type view, we also have events for creating and destroying objects, moving them between states of existence and nonexistence. For stacks, the events are *create*, *push*(e) for all $e \in E$, *pop* and *drop*, and there is only one attribute *top*.

The "internal activity" of an object is performed in terms of events, but single events in isolation do not tell too much. What matters is which sequences of events are allowed. Especially informative about an object's behaviour are its complete life cycles, starting from nonexistence and either ending in nonexistence or running on forever. A valid life cycle for a stack is characterized as follows: it has to begin with

create and end, if ever, with *drop*, and each prefix of a life cycle has to have at most as many *pop*'s and *push*'s in order to avoid popping the empty stack. Life cycles can be infinite, corresponding to nonterminating event sequences. A set of finite and infinite sequences over an alphabet of events represents a *process* ([21]). For the moment being, we concentrate on deterministic processes. Formally, if X is a set of events, then $X^\sigma = X^* \cup X^\omega$ denotes the set of *streams* over X. The finite sequences $\tau \in X^*$ are called *traces* over X. The *life cycles* of an object are denoted by $\Lambda \subseteq X^\sigma$.

Life cycles alone do not tell everything about an object either. The behaviour of a stack is described satisfactorily if we describe what we can observe, depending on what happened to the stack before. The observations we can make about a stack are the values of its *top* entry. The *top* value is determined by the finite sequence of events that happened so far. Formally, if we have a set A of attributes and a set $obs(A)$ of observations over A (to be made precise below), the observable behaviour of an object is modelled by an *attribute observation mapping* $\alpha: X^* \to obs(A)$ associating observations with traces, in particular initial traces of life cycles.

Summing up, our process-oriented model for objects consists of a set X of events, a set A of attributes, a set $\Lambda \subseteq X^\sigma$ of life cycles, and an attribute observation mapping $\alpha: X^* \to obs(A)$.

The attribute observation mapping has the typical structure of an input/output function of a state machine with inputs X and outputs $Y \subseteq obs(A)$. In fact, a state machine is appropriate for giving a more concrete description of attribute observation, as mentioned above. Our process view, however, is more abstract and mathematically simpler, without losing essential information: from a process model, a corresponding state machine can be derived in a canonical way by well-known automata-theoretic constructions.

An object type is a collection of objects which "belong together", but in which sense? If all instances of an object type are dynamic and variable entities, some stable mechanism for referring to objects is badly needed. Thus, object identification is a crucial point for putting objects together in a type. Once we have an identification scheme, we need to know which object instance is associated with each identifier. Thus, the second crucial point for object typing is object instantiation.

Object identification can be rather sophisticated. It is common use to provide internal, system-generated "surrogates", e.g. natural numbers, for referring to objects, but we want semantically meaningful identification, and that is not always that simple. Not only may the identification system have an intricate structure, there may also be several identification systems at the same time (aliasing).

Considering database-like identification by keys, for instance, persons may be identified by simple keys like social security number or name, address and birthdate. In some applications, however, it may be more convenient to identify persons by their name and their affiliation which is another object of a different type (object-valued keys). To make things more complex, persons may be identified by their name etc. and their father who is another object of the same type (recursive key). Even more complicated is an identification system for parts which may be atomic or composite, where atomic parts are identified by part numbers and composite objects are identified by the list of their components which are parts in turn (recursive, generalized and complex key).

Instantiation of object types is based on object templates and qualification. An object template is like an object, from which individually named instances isomorphic to the template can be generated. For example, let a stack template be given by events $X = \{create, pop, \ldots\}$, attributes $A = \{top\}$, life cycles Λ as explained above, and attribute observation $\alpha: X^* \to obs(A)$. A stack instance with identifier s is

obtained as a "qualified copy" with events $s.X = \{s.create, s.pop, ...\}$, attributes $s.A = \{s.top\}$, life cycles $s.\Lambda$ and attribute observation mapping $s.\alpha$ defined in an obvious way.

For elementary object types, there is just one object template from which all instances are isomorphic copies. For complex object types constructed from other object types, instantiation may involve more than one template. Basically, instantiation of a complex object type is determined by instantiation of its component types and by the way it is composed.

Generally speaking, an object type is given by an identification scheme and an instantiation scheme. The intended semantics of the former is a universe of object "surrogates" with a set of "naming" operations defined on them. This is exactly what a data type is in the sense of algebraic data type theory! The identifiers are the terms, and aliasing can be accomplished by different terms evaluating to the same surrogate. The intended meaning of the instantiation scheme is to associate an object template with each object surrogate, describing that object instance obtained by qualifying the template with the surrogate. Formally, this is described by mapping the surrogates in the universe to the object templates of the type.

In object-oriented systems, object types are usually organized in type hierarchies, employing subtyping and inheritance. This way, specialization of object types can be expressed. For example, we may have an object type CAR with subtypes SPORTS-CAR, TURBO and DIESEL. The last two may have a common subtype TURBO-DIESEL, giving rise to multiple inheritance. Subtyping can be static as in this example where, for instance, a diesel remains a diesel throughout each of its possible life cycles. Subtyping can also be dynamic where objects can enter and leave the population of a subtype, driven by corresponding events. An example for this is PATIENT in a hospital as a subtype of PERSON: each person may become a patient as a phase in his/her life and (hopefully) leave that phase after a while. Such phases may occur repeatedly during a life cycle of an object.

In a type hierarchy, an object can belong to several types. For example, a patient is a person at the same time. The usual way of looking at it is that patients have all properties persons have, inheriting events and attributes from his/her existence as a person. As a patient, however, he/she may be subject to additional events and may have additional attributes, for instance those having to do with surgery, displaying a richer structure of life cycles and attribute observations. For reasons to become clearer later on, we prefer a dual view: a person "incorporates" all events and attributes of a patient as one of his/her potentials. A patient, then, has only his/her special events and attributes. In this sense, a patient is an aspect (or view) of a person. There are obvious compatibility requirements, called inheritance conditions here: the valid life cycles and observations of a person should "contain" all valid life cycles and observations of his/her existence as a patient.

Another important relationship between objects is the "part-of" relationship between a composite object and its components. A car, for instance, consists of a chassis, an engine, etc. In a way, the events, attributes, life cycles and observations of its engine are "contained" in those of a car. An engine event "gives rise" to a car event, but it is not really one in itself. Similarly, an engine attribute is observable when looking at a car as a whole, but it is not a car attribute in itself. There are analogous compatibility conditions here: the life cycles and observations of a car should "contain" only valid life cycles and observations of its engine. There is a common formalism behind the "aspect-of" and "part-of" relationships, as we will see.

Last but not least, objects must be able to communicate. Objects can only communicate if they have something in common, i.e. they share something. In databases, it

is quite common to employ a kind of attribute sharing by saying which attributes of which objects (e.g. tuples in the relational model) are "the same". In object-oriented systems, some form of event sharing is preferred since attribute sharing, i.e. shared variables, violates the locality principle of object encapsulation. Synchronous message passing (rendez-vous like) is one form of event sharing saying which *send* event is "the same" as which *receive* event. We favour a simple and powerful symmetric event sharing mechanism: any event may be shared by any number of objects. For example, buying a car is an event for a person, changing the set of cars he/she owns, and it is an event for the car, changing its owner, among others. Attribute sharing and event sharing are shown in this paper to be just special cases of a general mechanism of object sharing.

3. Objects

An object is given by its sets of events, attributes, life cycles and observations. We first make precise what we mean by an observation.

Let A be a set of *attributes*. For each attribute $a \in A$, we assume a data type $type(a)$ which determines the values a can have. Since object universes are also data types (see below), the case of object-valued attributes is included. Moreover, $type(a)$ can be an arbitrarily complex type, so we also admit, among others, set-valued and list-valued attributes.

Definition 3.1. An *observation* over A is a set of attribute-value pairs $y = \{(a_1 : d_1), \ldots, (a_r : d_r)\}$ where $a_i \in A$ and $d_i \in type(a_i)$ for $1 \leqq i \leqq r$. The set of observations over A is denoted by $obs(A)$.

An observation indicates values for some of the attributes. An attribute need not have a unique value, so it may appear more than once in an observation. Intuitively, this means that the attribute's value is not determined, it may be any of the associated values. If the attribute value is undefined, it does not appear. The empty observation thus expresses that all attributes are undefined.

Definition 3.2. An *object* $ob = (X, A, \Lambda, \alpha)$ consists of a set X of *events*, a finite set A of *attributes*, a set $\Lambda \subseteq X^\sigma$ of *life cycles* such that $\varepsilon \in \Lambda$, and a total attribute *observation mapping* $\alpha : X^* \to obs(A)$ such that $\alpha(\varepsilon) = \emptyset$.

The empty life cycle ε expresses that the object remains nonexistent, and that must be possible for any object. The observation of a nonexistent object is always empty. Actually, α will only be needed for (finite) prefixes of life cycles, but assuming α to be total simplifies matters.

In a more detailed and concrete view, an object can be modelled as a state machine $SM = (S, X, Y, \delta, \beta, s_0)$ where S, X and $Y \subseteq obs(A)$ are (possibly infinite) sets: the states, input alphabet and output alphabet. Since not every event may occur in every state, we assume the state transition function $\delta : S \times X \to S$ to be partial. Alternatively, we could work with the canonical one-point completion, introducing a special "error" state. Since the notion of observation already covers undefined values, we assume the output function $\beta : S \to Y$ to be total. $s_0 \in S$ denotes the initial state. For a trace $\tau = x_1 x_2 \ldots x_n \in X^*$ and a state $s \in S$, we say that τ is *defined at* s iff $\delta(\delta^*(s, x_1 \ldots x_{i-1}), x_i)$ is defined for each i, $1 \leqq i \leqq n$. δ^* denotes the usual extension of δ to input sequences.

Definition 3.3. The set of *life cycles* $\Lambda(\mathrm{SM})$ *of a state machine* SM is defined by

$$\Lambda(\mathrm{SM}) = \{\lambda \in X^* \mid \lambda \text{ is defined at } s_0 \text{ and } \delta^*(s_0, \lambda) = s_0\} \cup \{\lambda \in X^\omega \mid \text{for each prefix } \pi \text{ of } \lambda, \pi \text{ is defined at } s_0\} .$$

This definition reflects the idea that s_0 is the "state of non-existence" with which all life cycles begin and all finite life cycles end.

Definition 3.4. A state machine $\mathrm{SM} = (S, X, Y, \delta, \beta, s_0)$ is a *realization* of an object $\mathrm{ob} = (X, A, \Lambda, \alpha)$ iff $Y \subseteq obs(A)$, $\Lambda = \Lambda(\mathrm{SM})$, and $\alpha(\tau) = \beta(\delta^*(s_0, \tau))$ for each $\tau \in X^*$ which is defined at s_0.

This way, a state machine with $Y \subseteq obs(A)$ determines an object by its input/output function and the set of life cycles it "recognizes" according to Definition 3.3. In an abstract sense, it also works the other way round: for a given object, there may be many different realizations, but all of these are behaviourally equivalent. So well-known methods of automata theory can be applied to construct a canonical state machine, for instance the minimal one, for a given object. We do not go into further detail here.

Now we make precise what it means that an object ob_1 (e.g. a patient) is at the same time another object ob_2 (e.g. a person) or, to put it the other way round, ob_1 is an "aspect of" ob_2. Let $\mathrm{ob}_i = (X_i, A_i, \Lambda_i, \alpha_i)$, $i = 1, 2$.

Definition 3.5. ob_1 *is an aspect of* ob_2, formally $\mathrm{ob}_1 \subseteq \mathrm{ob}_2$, iff $X_1 \subseteq X_2$ and $A_1 \subseteq A_2$, and the following *inheritance conditions* hold:

1) $\Lambda_1 \leqq \Lambda_2$ (life cycle inheritance),
2) $\alpha_1(\tau \downarrow X_1) = \alpha_2(\tau) \downarrow A_1$ for each trace $\tau \in X_2^*$ (observation inheritance).

We have to explain notation. $\Lambda_1 \leqq \Lambda_2$ means that, for each $\lambda_1 \in \Lambda_1$, there is a $\lambda_2 \in \Lambda_2$ such that $\lambda_1 \leqq \lambda_2$, which in turn means that all event occurrences of λ_1 appear in λ_2, but possibly interspersed with other events. $\downarrow X_1$ corresponds to the hiding operator (concealment) on processes and the restriction operator on traces ([21]). On streams, it is defined by $\varepsilon \downarrow X_1 = \varepsilon$, $x\tau \downarrow X_1 = x(\tau \downarrow X_1)$ if $x \in X_1$, and $x\tau \downarrow X_1 = \tau \downarrow X_1$ otherwise. On life cycle sets, it is defined by $\Lambda_2 \downarrow X_1 = \{\lambda \downarrow X_1 \mid \lambda \in \Lambda_2\}$. The analogous operation $\downarrow A_1$ on observations $y \in obs(A_2)$ is defined by $y \downarrow A_1 = \{(a{:}d) \in y \mid a \in A_1\}$.

Motivations for the inheritance conditions have been given in the previous section. In more detail, life cycle inheritance expresses that each (possible) life cycle of object ob_1 (e.g. a patient) should be contained in some (possible) life cycle of ob_2 (e.g. a person), and observation inheritance says that any ob_2 (person) trace, restricted to an ob_1 (patient) trace by hiding the additional events, gives rise to the same observations in ob_1 and ob_2 when only considering the attributes of ob_1. This means that the additional (non-patient) events have no side effects on the values of attributes in the aspect (patient).

Here is the reason why we take the dual view of inheritance mentioned above: persons containing patients as an aspect. Disallowing side effects of non-patient events on patient attributes is ok, but it would not reflect reality appropriately to have it the other way round, not allowing for patient events having side effects on person attributes in general.

Formally, the "part-of" relationship between objects is a generalization of the "aspect-of" relationship. The events and attributes of the components are not "the same", but "give rise to corresponding" events and attributes in the composed object. This is appropriately modelled by mappings, including the special case of inclusions. This way, we obtain a morphism concept between objects (instances). Let $\mathrm{ob}_i = (X_i, A_i, \Lambda_i, \alpha_i)$, $i = 1, 2$.

Definition 3.6. An *object morphism* $h: \mathrm{ob}_1 \to \mathrm{ob}_2$ is given by an *event mapping* $h_X: X_1 \to X_2$ and an *attribute mapping* $h_A: A_1 \to A_2$ with $type(a) = type(h_A(a))$ for each $a \in A_1$, such that the following *generalized inheritance conditions* hold:

(1) $h_X(\Lambda_1) \leqq \Lambda_2$
(generalized life cycle inheritance),

(2) $h_X(\tau) = \varrho \downarrow h_X(X_1) \to h_A(\alpha_1(\tau)) = \alpha_2(\varrho) \downarrow h_A(A_1)$ for all $\tau \in X_1^*$ and all $\varrho \in X_2^*$
(generalized observation inheritance).

Here, h_X is extended to life cycles by mapping them event by event along the sequence, and this is in turn extended to sets by taking the set of images. Similarly, h_A is extended to observations by mapping attribute-value pairs $(a:d)$ elementwise to $(h_A(a):d)$. In the sequel, we will omit the subscripts X and A when no confusion can arise.

This notion of object morphism is rather general. By admitting arbitrary event and attribute mappings, we can take cases into account where different events and attributes of a part are considered "the same" in the greater context of the composed object.

Proposition and Definition 3.7. *The class of all objects with their object morphism forms a category, called the category* **OB** *of objects.* □

The proof is straightforward, so we omit it here. Composite objects are formed by putting objects together such that each of the latter is a "part-of" the former. Objects may have common parts, and objects with common parts may be composed again. This way, a rather involved part structure may arise. A useful categorial concept for studying this is that of a colimit.

Thus, we are interested in the existence of colimits in **OB**. As a special case, we first study coproducts of two objects (which corresponds to "disjoint union"). Let $\mathrm{ob}_i = (X_i, A_i, \Lambda_i, \alpha_i)$, $i = 1, 2$, be two objects. By $+$ we denote disjoint union, i.e. coproducts in the category **SET** of sets and functions. For the ease of notation, we treat the corresponding injections as inclusions. Let $\alpha_1 + \alpha_2$ be defined by $\alpha_1 + \alpha_2(\tau) = \alpha_1(\tau \downarrow X_1) + \alpha_2(\tau \downarrow X_2)$ for each $\tau \in (X_1 + X_2)^*$. Let $\mathrm{ob}_1 + \mathrm{ob}_2 = (X_1 + X_2, A_1 + A_2, \Lambda_1 + \Lambda_2, \alpha_1 + \alpha_2)$. Clearly, the injections $\mathrm{ob}_i \to \mathrm{ob}_1 + \mathrm{ob}_2$, $i = 1, 2$, defined by the respective injections of X_i in $X_1 + X_2$ and A_i in $A_1 + A_2$, are morphisms in **OB**. Notationally, we also treated them as inclusions.

Lemma 3.8. $\mathrm{ob}_1 + \mathrm{ob}_2$ *with the injections* $\mathrm{ob}_i \to \mathrm{ob}_1 + \mathrm{ob}_2$ *is a coproduct in* **OB**.

Proof. For $i = 1, 2$, let $h_i: \mathrm{ob}_i \to \mathrm{ob}_3$ be any object morphisms to some object $\mathrm{ob}_3 = (X_3, A_3, \Lambda_3, \alpha_3)$. Let $g: \mathrm{ob}_1 + \mathrm{ob}_2 \to \mathrm{ob}_3$ be given by the unique pair of functions $g: X_1 + X_2 \to X_3$, $g: A_1 + A_2 \to A_3$ such that $g \mid X_i = h_i$ and $g \mid A_i = h_i$, respectively. Since $h_i(\Lambda_i) \leqq \Lambda_3$ and $\Lambda_1 + \Lambda_2$ contains only life cycles in Λ_1 or Λ_2, it is evident that $g(\Lambda_1 + \Lambda_2) \leqq \Lambda_3$. This proves that g satisfies the generalized life cycle inheritance condition. In order to prove that g also satisfies the generalized observation inheritance condition, let $\tau \in (X_1 + X_2)^*$ and $\varrho \in X_3^*$ such that $g(\tau) = \varrho \downarrow g(X_1 + X_2)$. Let τ be an interleaving of $\tau_i \in X_i^*$, $i = 1, 2$. Then we have $\alpha_3(\varrho) \downarrow h_i(A_i) = h_i\alpha_i(\tau_i)$, and consequently $\alpha_3(\varrho) \downarrow g(A_1 + A_2) = g\alpha(\tau)$ where $\alpha = \alpha_1 + \alpha_2$. This proves that g is an object morphism which in turn proves the theorem. □

The coproduct object $\mathrm{ob}_1 + \mathrm{ob}_2$ reflects the choice operator on processes ([21]). Along the same lines, based on coproducts in **SET**, we obtain the following general result.

Lemma 3.9. **OB** *has coproducts.* □

It is well-known that a category has all colimits (is cocomplete) iff it has coproducts and coequalizers, so we now prove the following result.

Lemma 3.10. **OB** *has coequalizers.*

Proof. For $i = 1, 2$, let h_i: $\mathrm{ob}_1 \to \mathrm{ob}_2$ be two morphisms. Let $\mathrm{ob}_i = (X_i, A_i, \Lambda_i, \alpha_i)$ for $i = 1, 2, \ldots$ Let g: $\mathrm{ob}_2 \to \mathrm{ob}_3$ be determined by the **SET** coequalizers on the event and attribute sets, coincidentally defining X_3 and A_3. g is surjective on both sets. Let $\Lambda_3 = g(\Lambda_2)$, and let α_3 be given by $\alpha_3 g(\sigma) = g\alpha_2(\sigma)$ for each $\sigma \in X_2^*$. Clearly, g is an object morphism. We show that g is the coequalizer of h_1 and h_2 in **OB**.

Let f: $\mathrm{ob}_2 \to \mathrm{ob}_4$ be any object morphism such that $fh_1 = fh_2$. Since g is a coequalizer in **SET**, on events as well as on attributes, there are unique mappings h: $X_3 \to X_4$ and h: $A_3 \to A_4$ such that $f = hg$. It remains to prove that h is an object morphism. Concerning generalized life cycle inheritance, we have $h(\Lambda_3) = hg(\Lambda_2) = f(\Lambda_2) \leqq \Lambda_4$. Concerning generalized observation inheritance, let $\pi \in X_4^*$, and let $\varrho \in X_3^*$ such that $h(\varrho) = \pi \downarrow h(X_3)$. Let $\sigma \in X_2^*$ such that $g(\sigma) = \varrho$. Then we conclude

$$h\alpha_3(\varrho) = h\alpha_3 g(\sigma) = hg\alpha_2(\sigma) = f\alpha_2(\sigma) = \alpha_4(\pi) \downarrow f(A_2) = \alpha_4(\pi) \downarrow hg(A_2)$$
$$= \alpha_4(\pi) \downarrow h(A_3) \,.\ \square$$

Putting the last two lemmas together, we obtain the following result.

Theorem 3.11. **OB** *is cocomplete.* □

The basis for object aggregation in our model is the parallel composition of objects, appearing as interleaving of life cycles. Let $\Lambda_i \subseteqq X_i^\sigma$, $i = 1, 2$, and let

$$\Lambda_1 \,||\, \Lambda_2 = \{\lambda \in (X_1 + X_2)^\sigma \mid \lambda \downarrow X_1 \in \Lambda_1 \wedge \lambda \downarrow X_2 \in \Lambda_2\}$$

be the interleaving of Λ_1 and Λ_2.

Definition 3.12. The *parallel composition* of ob_1 and ob_2 is

$$\mathrm{ob}_1 \,||\, \mathrm{ob}_2 = (X_1 + X_2, A_1 + A_2, \Lambda_1 \,||\, \Lambda_2, \alpha_1 + \alpha_2) \,.$$

The only difference between choice (coproduct) $\mathrm{ob}_1 + \mathrm{ob}_2$ and parallel composition $\mathrm{ob}_1 \,||\, \mathrm{ob}_2$ is in the set of life cycles. Evidently, for $i = 1, 2$, the injections of X_i into $X_1 + X_2$ and A_i into $A_1 + A_2$ define morphisms $\mathrm{ob}_i \to \mathrm{ob}_1 \,||\, \mathrm{ob}_2$. Thus, there is a unique morphism

$$g\colon \mathrm{ob}_1 + \mathrm{ob}_2 \to \mathrm{ob}_1 \,||\, \mathrm{ob}_2$$

where g is the identity on $X_1 + X_2$ and $A_1 + A_2$. In fact, we have $\Lambda_1 + \Lambda_2 \leqq \Lambda_1 \,||\, \Lambda_2$. Moreover, consider the property

$$\forall \lambda' \in \Lambda' \;\; \exists \lambda_1 \in \Lambda_1 \;\exists \lambda_2 \in \Lambda_2 \colon\; \lambda_1 \leqq \lambda' \wedge \lambda_2 \leqq \lambda' \,. \qquad (*)$$

This means that each life cycle in Λ' contains both a life cycle of Λ_1 and one of Λ_2. This holds for $\Lambda' = \Lambda_1 + \Lambda_2$ (remember the empty life cycle!) and for $\Lambda' = \Lambda_1 \,||\, \Lambda_2$.

Most interestingly, parallel composition can be characterized by a sort of universal property: fixing the events, attributes and observation mapping as in $\mathrm{ob}_1 + \mathrm{ob}_2$ and $\mathrm{ob}_1 \,||\, \mathrm{ob}_2$, the latter object has the maximal set of life cycles satisfying property (*). More precisely, if $\mathrm{ob}' = (X_1 + X_2, A_1 + A_2, \Lambda', \alpha_1 + \alpha_2)$ is any object with Λ' satisfying (*), we have $\Lambda_1 + \Lambda_2 \leqq \Lambda' \leqq \Lambda_1 \,||\, \Lambda_2$, i.e. morphism g: $\mathrm{ob}_1 + \mathrm{ob}_2 \to \mathrm{ob}_1 \,||\, \mathrm{ob}_2$ factorizes uniquely over ob' (with morphisms which are again identities on events and

attributes). Intuitively speaking, there is a spectrum of objects ob′ representing any degree of interleaving, ranging between no interleaving ($\hat{=}$ choice) on the left end and full interleaving ($\hat{=}$ parallel composition) on the right end.

Parallel composition can also be characterized by a sort of couniversal property. For $\Lambda' = \Lambda_1 \,||\, \Lambda_2$, the following holds:

$$\forall \lambda_1 \in \Lambda_1 \;\; \forall \lambda_2 \in \Lambda_2 \;\; \exists \lambda' \in \Lambda' : \lambda_1 \leqq \lambda' \wedge \lambda_2 \leqq \lambda' \,. \qquad (**)$$

This means intuitively that any two life cycles from Λ_1 and Λ_2, respectively, are contained in some single life cycle in Λ'. If $\mathrm{ob}' = (X_1 + X_2, A_1 + A_2, \Lambda', \alpha_1 + \alpha_2)$ is any object such that Λ' satisfies property (**), then we have $\Lambda_1 \,||\, \Lambda_2 \leqq \Lambda'$, i.e. there is a unique morphism $\mathrm{ob}_1 \,||\, \mathrm{ob}_2 \to \mathrm{ob}'$ (with identities on events and attributes).

These characterizations can be generalized to non-disjoint parallel composition

$$\mathrm{ob}_1 \,||\, \mathrm{ob}_2 = (X_1 \cup X_2, A_1 \cup A_2, \Lambda_1 \,||\, \Lambda_2, \alpha_1 \cup \alpha_2) \,,$$

provided that α_1 and α_2 are compatible. We will refer to this generalization in Section 5 (Example 5.1), but we abstain from working out the details here.

These results and constructions provide the general framework for studying not only object aggregation, but also object interaction. In fact, these concepts are closely related: very generally speaking, to interact means to share something, and vice versa. We postpone the discussion of interaction by sharing to Section 5, where we also give an example to illustrate how this works.

In the next section, we first introduce object types, including mechanisms of aggregating object types by aggregating the instances in a uniform way.

4. Object types

An object type is given by an identification scheme and an instantiation scheme. Our semantic model for the former is an algebraic data type $\boldsymbol{U}$, called the universe of object surrogates. It consists of a carrier set U and naming operations OP. The instantiation scheme is a mapping from object surrogates to object templates. An object instance of that type is a pair consisting of an object surrogate and the template it is mapped to: the latter is qualified by the former. Let OB be the class of objects in the category **OB**.

Definition 4.1. An *object type* $\mathrm{OT} = (\boldsymbol{U}, \omega)$ consists of a *universe* $\boldsymbol{U} = (U, \mathrm{OP})$ of object surrogates U and naming operations OP, and an *instantiation* mapping ω: $U \to \mathrm{OB}$.

Example 4.2. An object type STACK of stacks might use natural numbers as stack surrogates. The universe then consists of the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers. The constant $0: \to \mathbb{N}$ and the successor function $succ: \mathbb{N} \to \mathbb{N}$ may serve as "naming" operations. Let **stack** $= (X, A, \Lambda, \alpha)$ be an object template displaying the structure and behaviour of stacks (cf. the informal description at the beginning of Section 2). The instance mapping of STACK is then the constant mapping sending all natural numbers to **stack**, $\omega(n) =$ **stack** for all $n \in \mathbb{N}$. An instance of type STACK is given by a surrogate $n \in \mathbb{N}$ and the template **stack** (cf. Example 4.5 below for details).

Please note that a data type can be considered a special case of an object type where ω sends each element d of the carrier to the empty (initial) object $(\emptyset, \emptyset, \{\varepsilon\}, \lambda x.\emptyset)$. This corresponds to viewing d as a constant with its value as surrogate. Data constants cannot be changed by events and cannot be observed via attributes. Objects in general are like variables.

We first study object identification in more detail. For describing universes, we have the well-developed theory of algebraic data types available. Especially, we take advantage of parameterized data types, also called data type constructors, for putting object types together ([3; 8; 11]), for instance

1. products $\boldsymbol{U} = \boldsymbol{U}_1 \times \boldsymbol{U}_2$ with carrier $U = U_1 \times U_2$, the naming operation (construction) $(.\,,.)\colon U_1 \times U_2 \to U$, and projections $\mathrm{pr}_i\colon U \to U_i$, $i = 1, 2$.
2. coproducts $\boldsymbol{U} = \boldsymbol{U}_1 + \boldsymbol{U}_2$ with carrier $U = U_1 + U_2$, naming operations (injections) $\mathrm{in}_i\colon U_i \to U$, $i = 1, 2$, and particularization $\mathrm{pcl}\colon U \to U_1 \cup U_2$.
3. sets $\boldsymbol{U} = \mathbf{set}\ \boldsymbol{U}_1$ with carrier $U = \boldsymbol{P}_{\mathrm{fin}} U_1$ (finite subsets of U_1) and naming operations like $\varepsilon\colon \to U$, *add*: $U \times U_1 \to U$, etc.
4. lists $\boldsymbol{U} = \mathbf{list}\ U_1$ with carrier $U = U_1^*$ and naming operations like $\varepsilon\colon \to U$, *append*: $U \times U_1 \to U$, etc.

There is one peculiarity here that is not studied in classic algebraic data type theory, namely database-like identification by keys ([9; 10; 29]). We give a short account of the problem. Let $\boldsymbol{U} = (U, \mathrm{OP})$ be a universe.

Definition 4.3. A *key* K of $\boldsymbol{U}$ is a set $K = \{k_1, \ldots, k_n\} \subseteq \mathrm{OP}$ of unary operations on U, $k_i\colon U \to U_i$ for $i = 1, \ldots, n$, where $U_1, \ldots, U_n$ are carriers of data types, such that, for any two elements $u, v \in U$, if $k_i(u) = k_i(v)$ for all $i = 1, \ldots, n$, then $u = v$.

For products, the projections form a key. For coproducts, particularization is a key. For sets $U = \boldsymbol{P}_{\mathrm{fin}} U_1$, if there is a one-element key $\{k\colon U_1 \to U_2\}$ for U_1, then its elementwise extension to sets $k\colon U \to \boldsymbol{P}_{\mathrm{fin}} U_2$ is a key. Similarly, if $\boldsymbol{U} = \mathbf{list}\ U_1$ and if there is a key $\{k\colon U_1 \to U_2\}$ for U_1, then its elementwise extension to finite sequences $k\colon U \to U_2^*$ is a key.

By separating the range types of keys into proper data types and universes for object types, and by giving a key as only operations for each universe sort in a (data and universe) signature, unique abstract universes can be specified. The standard model is characterized by general principles of data conservation, representability by data, distinguishability, and maximality [13]). This shows the close relationship between identification by (non-updateable) keys and surrogate-based identification (cf. the discussion in [22]).

Object instantiation $\omega\colon U \to \mathrm{OB}$ is based on object templates and qualification. Let $\mathrm{ob} = (X, A, \Lambda, \alpha)$ be an object template and let $u \in U$ be an object surrogate.

Definition 4.4. The *u-instance* of ob is $u.\mathrm{ob} = (u.X, u.A, u.\Lambda, u.\alpha)$ where $u.X = \{u.x \mid x \in X\}$, $u.A = \{u.a \mid a \in A\}$, $u.\Lambda = \{u.x_1\, u.x_2 \ldots \mid x_1 x_2 \ldots \in \Lambda\}$, and $u.\alpha(u.x_1 \ldots x.u_n) = \alpha(x_1 \ldots x_n)$.

Example 4.5. Referring to Example 4.2 above, the n-instance of **stack**, n.**stack** for $n \in \mathbb{N}$, has events $n.create$, $n.push(e)$ for all entries $e \in E$, $n.pop$ and $n.drop$, and attribute $n.top$. The life cycles of n.**stack** are those of **stack** with every event qualified by n. After any finite prefix of an n.**stack** life cycle, its observation is that of the corresponding **stack** trace obtained by omitting qualification.

For elementary object types (as in Example 4.2), the instantiation mapping will be a constant mapping associating the same object template with each surrogate, i.e. all instances are isomorphic copies of one fixed template. For complex object types, instantiation is more complicated: more than one template can be used for instantiation, even for the same surrogate. The former case occurs with generalization (see below), and the latter case occurs when we have subtypes.

If, for instance, PATIENT is a subtype of PERSON, then each patient is a specific person at the same time or, to be more precise, the former is an aspect of the latter. So they should have the same surrogate. In fact, assuming that every person may become a patient some time, the universes of PATIENT and PERSON should be equal. In general, however, we should take into account that not every surrogate of the supertype is in the subtype. As an example, consider DIESEL as a subtype of CAR.

Let $\mathrm{OT}_i = (\boldsymbol{U}_i, \omega_i)$, $i = 1, 2$, be two object types, $\boldsymbol{U}_i = (U_i, \mathrm{OP}_i)$.

Definition 4.6. OT_1 is a *subtype* of OT_2 iff $U_1 \subseteqq U_2$ and $\omega_1(u) \subseteqq \omega_2(u)$ for each $u \in U_1$.

Please note that the naming operations are not involved in this definition: we do not require that subtype and supertype have the same identification scheme. It is obvious, however, that the inclusion of U_1 in U_2 can be used as a key for U_1.

Generalizing this as in the object instance case, we obtain the following notion of morphism between object types. Let OT_1 and OT_2 be as above.

Definition 4.7. An *object type morphism* $h: \mathrm{OT}_1 \to \mathrm{OT}_2$ consists of a mapping $h_U: U_1 \to U_2$ and a U_1-indexed family of object morphisms

$$h_\omega = \{h_\omega(u): \omega_1(u) \to \omega_2 h_U(u) \mid u \in U_1\} .$$

Considering U_1 and U_2 as discrete categories, we can view h_U, ω_1 and ω_2 as functors, and this way we can model an object type morphism as a natural transformation $h: \omega_1 \dot{\to} \omega_2 h_U$. However, since U_1 has no morphisms, this is a rather uninteresting kind of natural transformation, so we prefer to stick to more elementary terminology.

Proposition and Definition 4.8. *The class of all object types with their object type morphisms forms a category, called the category* **OT** *of object types.* □

Proving the category properties is easy. As with objects, an essential categorial concept for studying object types is to utilize colimits. On the instance level, object aggregation is expressed by parallel composition, which is somewhat related to coproducts in **OB**. On the type level, we also have colimits in **OT**. Since **SET** and **OB** are cocomplete, the following theorem is evident.

Theorem 4.9. **OT** *is cocomplete.* □

We are only interested in coproducts so far, so we show how the coproduct $\mathrm{OT} = (\boldsymbol{U}, \omega)$ of two objects types $\mathrm{OT}_i = (\boldsymbol{U}_i, \omega_i)$, $i = 1, 2$, looks like: the universe is $\boldsymbol{U} = \boldsymbol{U}_1 + \boldsymbol{U}_2$, and instantiation ω is given by $\omega = \omega_1 + \omega_2$. The injections are those of the universes, together with the identity morphisms for objects.

Coproducts of object types express generalization, for example LEGAL-PERSON = PERSON + COMPANY. The surrogates of LEGAL-PERSON are given by the disjoint union of those of PERSON and COMPANY. Identification is provided by the injections, and particularization gives a key. Each person or company instance is by itself a legal person instance, with the same events and attributes. Thus, in generalized types, more than one object template is used for instantiation.

Very generally speaking, aggregation is described by a parameterized data type (or data type constructor) by which an aggregated universe can be built. The aggregated instances are then obtained by parallel composition, possibly involving sharing. If there is no sharing, the aggregated instances are built by disjoint interleaving. We discuss the most useful forms of aggregation: tupling and grouping by means of sets and lists.

The object type $\mathrm{OT} = \mathrm{OT}_1 \times \mathrm{OT}_2$ consists of the universe $\boldsymbol{U} = \boldsymbol{U}_1 \times \boldsymbol{U}_2$ and the instantiation mapping $\omega(u_1, u_2) = \omega_1(u_1) \,||\, \omega_2(u_2)$.

The object type $\mathrm{OT} = \mathbf{set}\ \mathrm{OT}_1$ can be defined as follows. For the universe, we have $\boldsymbol{U} = \mathbf{set}\ \boldsymbol{U}_1$, and for instantiation, $\omega(V)$ is the parallel composition of all instances $\omega_1(u)$ for $u \in V \subseteq U_1$: a set object instance has all events and attributes of its member objects, and its life cycles are all interleavings of those of the members.

The object type $\mathrm{OT} = \mathbf{list}\ \mathrm{OT}_1$ can be defined in a similar way. For the universe, we have $\boldsymbol{U} = \mathbf{list}\ \boldsymbol{U}_1$, and for instantiation, $\omega(L)$ is the parallel composition of all instances $\omega_1(u)$ for the elements u of list L over U_1. Thus, on the instance level, there is no difference between sets and lists, the only difference is on the surrogate and identification level.

Further examples can be constructed aggregating objects into bags (multisets), trees, etc. of objects.

5. Object societies

An object society is a — typically large — collection of objects which interact in one way or another. In a very general sense, interaction means to share something. In our model, there are three kinds of entities that can be shared: events, attributes and (component) objects. Actually, event and attribute sharing are special cases of object sharing.

A certain kind of attribute sharing is common in databases. For instance, in an order issued by a customer, the part ordered should be the same as the part shipped to the customer in fulfilling that order. In fact, attribute sharing is the basis for the natural join operation of relational databases, which in turn is the fundament of relational database design.

In object-oriented systems, event sharing is usually given preference over attribute sharing. The latter means to share memory which contradicts the locality principle of object encapsulation. Event sharing is a general way to express synchronous communication. It may appear in the special form of synchronous message passing by sharing *send* and *receive* events. We favour a general and symmetric form of event sharing: any event may be shared between any number of objects.

Whichever sharing mechanism is adopted, the underlying mathematics is the same, namely that of object sharing, expressed by object morphisms and colimits in the category **OB** of objects.

Please note that a single event e can be viewed as an object $(\{e\}, \emptyset, \{\varepsilon\}, \lambda x.\emptyset)$, and a single attribute a can be viewed as an object $(\emptyset, \{a\}, \{\varepsilon\}, \lambda x.\emptyset)$. Thus, object sharing generalizes event and attribute sharing and allows for mixed forms of both.

The following example illustrates how object morphisms and colimits in **OB** express event sharing. Attribute sharing and general object sharing follow the same pattern.

Example 5.1. Let STACK be the object type of Example 4.2. Suppose we want to "synchronize" the *pop* events of **0.stack** and **1.stack**, i.e. **0.stack** and **1.stack** should share their *pop* events ($0.pop \equiv 1.pop$). To this end, we define a new ("global") event pop_{01}, also considered as an object $\boldsymbol{pop}_{01} = (\{pop_{01}\}, \emptyset, \varepsilon, \lambda x.\emptyset)$. Event sharing is described by the pair of morphisms shown in Fig. 1. The colimit of these two morphisms (in this case a pushout) consists of an object **glue-01-stack** and two morphisms to it, q_0 and q_1 (see Fig. 2). Intuitively, **glue-01-stack** consists of separate copies of **0.stack** and **1.stack**, "glued" together at the *pop* event. In more detail, it has the following events (giving the copies the same names):

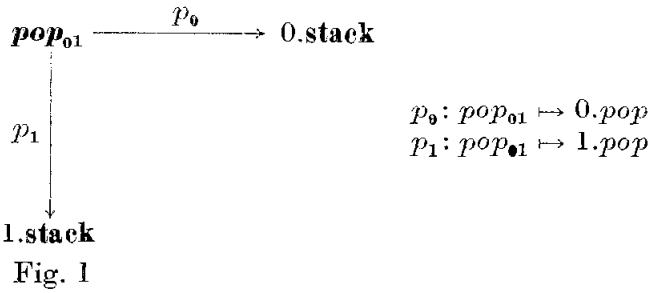

Fig. 1

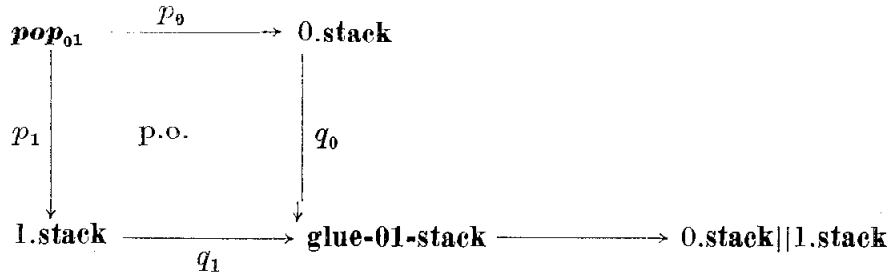

Fig. 2

0.*create*, 0.*drop*, 0.*push*(*e*) for all $e \in E$,
1.*create*, 1.*drop*, 1.*push*(*e*) for all $e \in E$, and
pop^*_{01} (a new event representing the shared *pop* event).

The attributes are 0.*top* and 1.*top*. The morphisms q_0 and q_1 send 0.*pop* and 1.*pop*, respectively, to pop^*_{10}. All other events and the attributes are sent to themselves (or, rather, to their copies with the same names). The life cycles of **glue-01-stack** are all interleavings of life cycles of **0.stack** and **1.stack** with 0.*pop* and 1.*pop* replaced by pop^*_{01} wherever they occur. The observation mapping of **glue-01-stack** is obvious: 0.*push*(*e*) and pop^*_{01} affect 0.*top*, and 1.*push*(*e*) and pop^*_{01} affect 1.*top*. The parallel composition **0.stack || 1.stack** with sharing can be characterized by the universal and couniversal properties outlined in the previous section, appropriately generalized to the non-disjoint case (in the case at hand, we are dealing with a pushout instead of a coproduct).

The right place to study object sharing is in the greater context of a composite object where components share components. For example, if the event of buying a car is shared by the car and its buyer, there should be an object like "car-market" or so, of which both the car and its buyer are components.

This way, composite objects with shared components can be described. In the previous section, we indicated how types of such objects can be created. Building an object society, however, works the other way round: it is an object instance constructed from an object type by adding interaction information, employing the operation of parallel composition.

We explain this by an example. When specifying the object society of a trader's world, we first define the object type structure involved, say CLIENT, ORDER, DEPOT, PRODUCT, SUPPLIER, STOCK, etc. By generalization, we may consider all instances of all these types assembled together in one type, say TRADE. Adopting event sharing, we can now introduce global events as objects with morphisms to those local events that are to be shared. The object society *trader's-world* now is the composite object built as the parallel composition of all instances of TRADE with overlapping defined by means of a colimit of all objects and all event sharing morphisms.

6. Concluding remarks

The theory of objects and object types outlined in this paper needs elaboration in several respects. Our goal is to develop a systematic methodology for object-oriented information system design, based on formal methods and a reliable theory. In this paper, we concentrate on semantic fundamentals, but semantics is only part of the theory needed. As a basis for specification, we need a logic calculus with a useful model theory and a consistent proof theory for proving properties like, for instance, correctness.

For algebraic data types, we know a lot about equational specification, its semantics and proof theory. For attributes and their constraints, it seems advisable to employ more general first-order predicate calculus. For specifying life cycles or observation sequences along life cycles, some sort of temporal or process logic is required. When considering the state machine model, the appropriate logic is some sort of dynamic or algorithmic logic. For covering all aspects of objects, object types and object societies, a useful selection of these calculi has to be put together.

Following the pattern of algebraic data type specification, we envisage abstract object types, i.e. isomorphism or some sort of equivalence classes of object types, as algebraic semantics of object type specification, parameterized specification and its algebraic semantics in terms of functors, and an algebraic notion of implementation, i.e. refining objects over objects. And we envisage an operational semantics in terms of deduction or replacement systems which can serve as a basis for early prototyping.

References

[1] *America, P.*: Object-Oriented Programming: A Theoretician's Introduction. EATCS Bull. 29 (1986), 69—84.

[2] *Arbib, M. A., E. G. Manes*: Arrows, Structures, Functors: The Categorical Imperative. Academic Press, New York 1975.

[3] *Burstall, R. M., J. A. Goguen*: Putting Theories together to make specifications. In: Proc. 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Mass., 1977; pp. 1045 to 1058.

[4] *Bergstra, J. A., J. V. Tucker*: The Inescapable Stack: an Exercise in Algebraic Specification with Total Functions. Report No. P8804, Programming Research Group, University of Amsterdam, 1988.

[5] *Cardelli, L., P. Wegner*: On Understanding Types, Data Abstraction, and Polymorphism. ACM Comput. Surv. 17 (1985), 471—522.

[6] Proc. Int. Workshop on Object-Oriented Database Systems; *U. Dayal, K. Dittrich* (eds). IEEE Computer Society, Los Angeles 1986.

[7] *Dahl, O.-J., B. Myhrhaug, K. Nygaard*: SIMULA 67, Common Base Language. Norwegian Computing Center, Oslo, 1967.

[8] *Ehrich, H.-D.*: On the Theory of Specification, Implementation, and Parameterization of Abstract Data Types. J. ACM 29 (1982), 206—227.

[9] *Ehrich, H.-D.*: Key Extensions of Abstract Data Types, Final Algebras, and Database Semantics. In: Proc. Workshop on Category Theory and Computer Programming; *D. Pitt* et al. (eds.); LNCS 240; Springer-Verlag, Berlin 1986; pp. 412—433.

[10] *Ehrich, H.-D., K. Drosten, M. Gogolla*: Towards an Algebraic Semantics for Database Specification. In: Proc. IFIP WG2.6 Working Conf. DS-2, Albufeira, 1986 (final proceedings to be published by North-Holland)

[11] *Ehrig, H., H.-J. Kreowski, J. W. Thatcher, E. G. Wagner, J. B. Wright*: Parameterized Data Types in Algebraic Specification Languages. In: Proc. 7th ICALP; *J. W. de Bakker, J. van Leeuwen* (eds.); LNCS 85; Springer-Verlag, Berlin 1980; pp. 157—168.

[12] *Ehrig, H., B. Mahr*: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Springer-Verlag, Berlin 1985.
[13] *Ehrich, H.-D., A. Sernadas, C. Sernadas*: Objects, Object Types, and Object Identity in Information Systems. In: Proc. Int. Workshop on Categorical Methods in Computing, Berlin, 1988 (to be published)
[14] *Goguen, J. A., J. Meseguer*: Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. In: Proc. 9th Int. Conf. on Automata, Languages and Programming; *M. Nielsen, E. M. Schmidt* (eds.); LNCS 140; Springer-Verlag, Berlin 1982; pp. 265—281.
[15] *Goguen, J. A., J. Meseguer*: Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In: [27]; pp. 417—477.
[16] *Goldblatt, R.*: Topoi, the Categorial Analysis of Logic. North-Holland, Amsterdam 1979.
[17] *Goldberg, A., D. Robson*: Smalltalk 80: The Language and its Implementation. Addison-Wesley, Reading 1983.
[18] *Goguen, J. A., J. W. Thatcher, E. G. Wagner*: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. Current Trends in Programming Methodology IV: Data Structuring; *R. Yeh* (ed.); Prentice-Hall, Englewood Cliffs 1978; pp. 80—149.
[19] *Goguen, J. A., J. W. Thatcher, E. G. Wagner, J. W. Wright*: Initial Algebra Semantics and Continuous Algebras. J. ACM **24** (1977), 68—95.
[20] *Guttag, J. V.*: The Specification and Application to Programming of Abstract Data Types. Tech. Report CSRG-59, Univ. of Toronto, Toronto, Ontario, Canada, 1975.
[21] *Hoare, C. A. R.*: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs 1985.
[22] *Khoshafian, S. N., G. P. Copeland*: Object Identity. In: Proc. OOPSLA '86; ACM SIGPLAN Notices **21** (1986) 11, pp. 406—416.
[23] Special Issue on Object-Oriented Systems; *F. Lochovski* (ed.). IEEE Database Engineering 8:4 (1985).
[24] *Parnas, D. L.*: A Technique for Software Module Specification with Examples. Comm. ACM **15** (1972), 330—336.
[25] *Reichel, H.*: Initial Computability, Algebraic Specification, and Partial Algebras. Akademie-Verlag, Berlin 1987.
[26] *Semadeni, Z., A. Wiweger*: Einführung in die Theorie der Kategorien und Funktoren. Teubner-Verlag, Leipzig 1979.
[27] Research Directions in Object-Oriented Programming; *B. Shriver, P. Wegner* (eds.). The MIT Press, Cambridge (Mass.) 1987.
[28] *Sernadas, A., J. Fiadeiro, C. Sernadas, H.-D. Ehrich*: Abstract Object Types: A Temporal Perspective. In: Proc. Colloq. on Temporal Logic and Specification; *H. Barringer* (ed.); Springer-Verlag (to be published)
[29] *Sernadas, A., C. Sernadas, H.-D. Ehrich*: Object-Oriented Specification of Databases: An Algebraic Approach. In: Proc. 13th VLDB; *P. M. Stocker, W. Kent* (eds.); Morgan-Kaufmann, Los Altos 1987; pp. 107—116.
[30] *Staiger, L.*: Research in the Theory of ω-languages. J. Inform. Process. Cybernet. **EIK 23** (1987) 8/9, 415—439.
[31] *Zilles, S. N.*: Algebraic Specification of Data Types. In: MIT Project MAC, Computation Structures Group Memo 119, MIT, Cambridge (Mass.), 1975; pp. 1—12.

Kurzfassung

Es wird eine semantische Grundlage für objekt-orientierte Ansätze beschrieben. Hierzu wird ein mathematisches Modell für Objekte, Objekttypen und Mengen miteinander kommunizierender Objekte angegeben, das sowohl statische als auch dynamische Aspekte abdeckt. Das Modell behandelt auch komplexe Objekte und Objekttypen und ermöglicht ein tieferes Verständnis für Spezialisierung, Vererbung, Generalisierung, Aggregation und Kommunikation durch "object sharing", d. h. gemeinsame Unterobjekte, wobei sowohl

die Struktur als auch die Dynamik berücksichtigt werden. Unsere Theorie der Objekte und Objekttypen verwendet einige Analogien aus der algebraischen Theorie der Datentypen. Wir definieren Kategorien von Objekten und Objekttypen. Beide werden als kovollständig nachgewiesen. Auf der Grundlage dieser Ergebnisse lassen sich strukturelle Beziehungen zwischen Objekten ebenso wie dynamische gegenseitige Beeinflussung von Objekten einheitlich darstellen.

Резюме

Описана некоторая семантическая основа объектно-ориентированных подходов. Для этого предложена математическая модель объектов, типов объектов и множеств между собой коммуницирующих объектов, которая учитывает как статические так и динамические аспекты.

Модель направлена на комплексные объекты и типы объектов и способствует более полному пониманию специализации, наследственности, обобщения, агрегации и коммуникации способом „деления объектов", т. е. способом совместных подобъектов. При этом учитывается как структура так и динамика. В нашей теории объектов и типов объектов имеют место некоторые аналогии с алгебраической теорией типов данных.

Мы определяем категории объектов и типов объектов. Оказывается, что обе являются ко-полными.

На основе этих результатов можно представить единым образом как структурные отношения между объектами так и динамическое влияние объектов друг на друга.

(received: first version October 6, 1988,
extended version January 10, 1989)

Authors' addresses:

Prof. Dr. H.-D. Ehrich
Technische Universität Braunschweig
Abt. Datenbanken
Postfach 3329
3300 Braunschweig
Federal Republik of Germany

Prof. A. Sernadas,
Prof. C. Sernadas
Departamento de Matematica
Inst. Superior Técnico
1096 Lisboa, Codex
Portugal