H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, Categorical Methods in Computer Science, LNCS 393, pages 142–156, Berlin, 1989. Springer

Objects, Object Types, and Object Identification

H.-D. Ehrich
Informatik/Datenbanken, TUBS, Postfach 3329, D-3300 Braunschweig, FRG

A. Sernadas C. Sernadas
Departamento de Matematica, IST, 1096 Lisboa Codex, PORTUGAL

Abstract - *The usefulness of category-theoretic concepts for a theory of object-oriented programming is advocated. Objects (in the latter sense) are defined as objects (in the category-theoretic sense) of a category* ***OB****. Colimits in* ***OB*** *are used to express aggregation of objects into complex objects as well as interaction between objects. Object types consist of an identification system, the object universe, and an instantiation system, describing the instances of the type. The main result of this paper gives a semantic basis for database-like identification by keys: the object universe can be specified uniquely (up to isomorphism) employing general principles of preservation of data, distinguishability by keys, and representability by keys.*

1. Introduction

Object-oriented approaches are becoming popular in programming, software design, databases and other fields of computer science. Essentially, an object reflects the idea of an encapsulated entity incorporating all concepts of a full-fledged computing system: data, storage, control, and communication with other objects. Moreover, in order to master large and varying collections of objects, they are organized into object types. Types are interrelated by a subtyping structure for which an appropriate inheritance mechanism is employed.

Many of these concepts were already built into the simulation language SIMULA (DMN67), but it took more than a decade of incubation to start the line of "object-oriented" languages and systems, as they are called now. This line began with Smalltalk-80 (GR83). Again, it took a while for object-oriented ideas to spread into other areas of computing, especially databases (Lo85, DD86, SW87).

Whereas the traditional styles of imperative and declarative programming are fairly well understood, with a considerable body of theory providing deeper insights into many aspects, this is not the case with the object-oriented style of programming (Am86). In this paper, we contribute to developing such a theory. The basis of our approach are *processes* (Ho85): they appear as sets of life cycles of objects, i.e. allowed sequences of events. Roughly speaking, our model of an object is such a process which can be *observed* along life cycles via attributes. For the notion of object *type*, we put special emphasis on object *identity* (KC86).

With emphasizing processes as basic building blocks of objects, we are going beyond the scope of current object-oriented languages like Smalltalk. Our purpose is to stimulate discussion on the theoretical and conceptual basis of object-orientation as such, with not too much bias towards systems as they are now, hopefully leading to insights which allow to evaluate current systems and help to develop more advanced ones in the future. While this paper concentrates on semantic foundations, there is related work on designing an object-oriented language for information systems specification and a methodology for using the language (SSE89, SFSE89).

Our approach is different from that of GM87, although there are also parallels, especially in adopting ideas and concepts from algebraic *data type* theory. Algebraic data type theory has been a source of inspiration for developing our approach, especially the usefulness of category-theoretic concepts. One particularly useful lesson from category theory is that it is not sufficient to look at the entities you want to study, but that it is indispensable to look at the *morphisms* between them. Indeed, the "aspect-of" and "part-of" relationships between objects generalize neatly to object morphisms, giving rise to a category ***OB*** with objects (in the sense of object-oriented programming) as objects (in the sense of category theory). ***OB*** is cocomplete, and colimits provide the right concept for studying aggregation of objects into complex objects as well as interaction between objects in a uniform way. In fact, a *society* of interacting objects (to be explained in section 3) can be viewed as a complex object with the members of the society as interrelated components.

Algebraic data types not only provide inspiration, they also appear as an integral part in our theory, even in two respects: as value domains of attributes, and as object *universes*, i.e. identification systems within object types. In an object universe, the elements act as object "surrogates", and the operations act as "naming" functions. An object *type* is such an identification system with an *instantiation* mapping, associating an object (acting as a template) with each object surrogate. The *instances* of the object type are all pairs of surrogates and their associated templates. Intuitively, this means that the template is "qualified" with the associated surrogate.

Generalizing the subtype relationship, there are also morphisms between object *types*. The resulting category of object types is *co*complete. This gives the basis for studying complex object types, i.e. object types built from other object types by generalization or some form of aggregation.

The main emphasis of this paper is on object *identity*. Employing database-like identification by keys, the question is how object universes can be *specified* abstractly, i.e. up to isomorphism. Our main result shows that this can be achieved by general, intuitively appealing principles of data preservation, distinguishability by keys, and representability by keys.

This main result could have been presented without the material in sections 2 and 3. In fact, it is a result about algebraic *data* type specification. We feel, however, that the context, i.e. our theory of objects and object types, is essential for appreciating the relevance of the problem and the result. Serving a more or less motivational purpose in this paper, sections 2 and 3 are kept in a somewhat narrative survey style. Part of the material is elaborated in ESS88, and other parts have yet to be elaborated.

2. Objects and Interaction

A prominent example of a data type is stack (BT88), and it 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*: $\longrightarrow S$, *top*: $S \longrightarrow E$, *push*: $S \times E \longrightarrow S$, *pop*:$S \longrightarrow S$, etc. which denote the empty sequence, the leftmost (or "topmost") entry of a sequence, adding a new entry to the left, and deleting the leftmost entry.

In an object-oriented view, a stack is not a *type*, but a single object *instance*. As an object, a stack has an internal state which can be changed and observed. 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, let us suppose that the events are *create*, *push*(e) for all $e \in E$, *pop* and *drop*, and that there is only one attribute *top*.

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 as *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 is a simple model of a *process* in the sense of (Ho85). We concentrate on deterministic processes here. 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* $\alpha : X^{*} \longrightarrow obs(A)$ associating observations with traces, in particular initial traces of life cycles.

Thus, 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, types can be arbitrarily complex, so we also admit, among others, set-valued and list-valued attributes.

Definition 2.1: An *observation* over A is a set of attribute-value pairs $y \subseteq \{ (a_1 : d_1), \ldots, (a_r : d_r) \}$ where $a_i \in A$ and $d_i \in type(a_i)$ for $1 \le i \le r$. The set of observations over A is denoted by *obs*(A).

An observation indicates values for some of the attributes. An attribute's value need not be unique, so each attribute may appear more than once in an observation. If its value is undefined, it does not appear. The empty observation expresses that all attributes are undefined.

Definition 2.2: 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 $\alpha : X^{*} \longrightarrow obs(A)$ 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 nonexisting object is always empty. Actually, α is only needed for (finite) prefixes of life cycles, but assuming α to be total simplifies matters.

An object can be viewed as the behaviour of a *state machine* with inputs X and outputs *obs*(A), realizing α as its input-output function and accepting ω-language Λ. For more details see ESS88.

Two kinds of relationship between objects are of fundamental importance. The first is that an object ob_1 is at the same time another object ob_2 or, to put it the other way round, ob_1 is an "aspect of" ob_2. The second is that an object is a "part of" another object.

As an example of the "aspect-of" relationship, consider a patient as an aspect of a person: a particalar patient is a particular person at the same time. As a person, he/she incorporates the potential to be a patient, thus having, among others, all events and attributes a patient has. In the patient aspect, only the special patient events and attributes are "visible".

Let $ob_i=(X_i,A_i,\Lambda_i,\alpha_i)$, i=1,2 . For the sets Λ_1 and Λ_2 of life cycles, we use the following notation. $\Lambda_1 \leq \Lambda_2$ means that, for each $\lambda_1 \epsilon \Lambda_1$, there is a $\lambda_2 \epsilon \Lambda_2$ such that $\lambda_1 \leq \lambda_2$, which in turn means that all event occurrences in λ_1 appear in λ_2, in the same order, but possibly interspersed with other events. $\downarrow X_1$ corresponds to the hiding operator (concealment) on processes and the restriction operator on traces (Ho85). On streams, it is defined by $\varepsilon \downarrow X_1 = \varepsilon$, $x\tau \downarrow X_1 = x(\tau \downarrow X_1)$ if $x \epsilon X_1$, and $x\tau \downarrow X_1 = \tau \downarrow X_1$ otherwise. On life cycle sets, it is defined by $\Lambda_2 \downarrow X_1 = \{ \lambda \downarrow X_1 \mid \lambda \epsilon \Lambda_2 \}$. The analogous operation $\downarrow A_1$ on observations $y \epsilon obs(A_2)$ is defined by $y \downarrow A_1 = \{ (a{:}d) \epsilon y \mid a \epsilon A_1 \}$.

Definition 2.3: ob_1 *is an aspect of* ob_2, formally $ob_1 \subseteq ob_2$, iff $X_1 \subseteq X_2$ and $A_1 \subseteq A_2$, and the following *inheritance conditions* hold:

(1) $\Lambda_1 \leq \Lambda_2$ (*life cycle inheritance*) ,

(2) $\alpha_1(\tau \downarrow X_1) = \alpha_2(\tau) \downarrow A_1$ for each trace $\tau \epsilon X_2^*$ (*observation inheritance*).

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 (patient) aspect. Although some object-oriented languages, e.g. Smalltalk, do not have this property, it is essential in our theory for a clean encapsulation of aspects as (sub-)objects in themselves.

The other 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. 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 instead of inclusions. This way, we obtain a morphism concept between objects.

Let $ob_i=(X_i,A_i,\Lambda_i,\alpha_i)$, $i=1,2$.

Definition 2.4: An *object morphism* $h: ob_1 \longrightarrow ob_2$ is given by an *event mapping* $h_X: X_1 \longrightarrow X_2$ and an *attribute mapping* $h_A: A_1 \longrightarrow A_2$ with $type(a)=type(h_A(a))$ for each $a \in A_1$, such that the following *generalized inheritance conditions* hold:

(1) $h_X(\Lambda_1) \le \Lambda_2$ (*generalized life cycle inheritance*)

(2) $h_X(\tau)=\rho \Downarrow h_X(X_1) \Rightarrow h_A(\alpha_1(\tau))=\alpha_2(\rho) \Downarrow h_A(A_1)$ for all $\tau \in X_1^*$ and all $\rho \in X_2^*$.
(*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.

The class of all objects with their object morphisms forms a category, called the *category **OB** of objects.*

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***. The following result is proved in ESS88.

Theorem 2.11: ***OB*** is cocomplete.

This theorem gives the background and general framework for studying object *aggregation*, i.e. putting objects together to form composite objects, as well as object *interaction*. In a general sense, interaction means to *share* something. Our model allows for rather general forms of *object sharing*, but in practice only special kinds of entities are shared: events or attributes.

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 foundation 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 may appear in several forms, for instance as synchronous message passing by sharing special *send* and *receive* events.

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. 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 $(\emptyset,\{a\},\{\varepsilon\},\lambda x.\emptyset)$.

Example 2.12: Let $\underline{stack}_0$ and $\underline{stack}_1$ be two stack objects isomorphic to the one given above, with all events and attributes of $\underline{stack}_0$ indexed by 0, and those of $\underline{stack}_1$ indexed by 1. Suppose we want to "synchronize" the *pop* events of the two stacks, i.e. $\underline{stack}_0$ and $\underline{stack}_1$ should share their *pop* events ($pop_0=pop_1$). To this end, we define a new ("global") event pop_{01}, also considered as an object $\underline{pop}_{01}=(\{pop_{01}\},\emptyset,\varepsilon,\lambda x.\emptyset)$, mapped to pop_0 and pop_1, respectively, by morphisms p_0 and p_1. Event sharing is described by these two morphisms.

$$\begin{tikzcd}
\underline{pop}_{01} \arrow[r, "p_0"] \arrow[d, "p_1"'] & \underline{stack}_0 \arrow[d, "q_0"] & \\
\underline{stack}_1 \arrow[r, "q_1"'] & \underline{stack}_{01} \arrow[r] & \underline{stack}_0 \parallel \underline{stack}_1
\end{tikzcd}$$

$p_0 : \mathit{pop}_{01} \longmapsto \mathit{pop}_0$

$p_1 : \mathit{pop}_{01} \longmapsto \mathit{pop}_1$

The colimit of these two morphisms (in this case a pushout) consists of an object $\underline{stack}_{01}$ and two morphisms to it, q_0 and q_1, as shown in the diagram above. Intuitively, $\underline{stack}_{01}$ consists of separate copies of $\underline{stack}_0$ and $\underline{stack}_1$, "glued" together at the *pop* events. In more detail, it has the following events (giving the copies the same names):

create_0 , drop_0 , $\mathit{push}_0(e)$ for all $e \in E$,
create_1 , drop_1 , $\mathit{push}_1(e)$ for all $e \in E$,
and pop^*_{01} (a new event representing the shared *pop* event)

The attributes are top_0 and top_1 . The morphisms q_0 and q_1 send pop_0 and pop_1 , respectively, to pop^*_{01}. All other events and the attributes are sent to themselves (or, rather, to their copies with the same names). The life cycles of $\underline{stack}_{01}$ are all life cycles of $\underline{stack}_0$ and $\underline{stack}_1$ with pop_0 and pop_1 replaced by pop^*_{01} wherever they occur. The observation mapping of $\underline{stack}_{01}$ is obvious: $\mathit{push}_0(e)$ and pop^*_{01} affect top_0, and $\mathit{push}_1(e)$ and pop^*_{01} affect top_1. The parallel composition $\underline{stack}_0 \parallel \underline{stack}_1$ (with sharing) is like $\underline{stack}_{01}$, but enriched by all interleavings of life cycles in the ranges of q_0 and q_1. This object can be characterized by a (sort of) universal and by a (sort of) couniversal property (cf. ESS88).

3. Object Types and Object Societies

An object type is given by an identification scheme and an instantiation scheme. Our semantic model for the former is an algebraic data type **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. Let OB be the class of objects in the category ***OB***.

Definition 3.1: An *object type* $OT=(\mathbf{U},\omega)$ consists of a *universe* $\mathbf{U}=(U,OP)$ of object surrogates U and naming operations OP, and an *instantiation* mapping $\omega : U \longrightarrow OB$.

Example 3.2: 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 $0 : \rightarrow \mathbb{N}$ and the successor function $\mathit{succ} : \mathbb{N} \longrightarrow \mathbb{N}$ may serve as "naming" operations. Let $\underline{stack}=(X,A,\Lambda,\alpha)$ 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 $\underline{stack}$, $\omega(n)=\underline{stack}$ for all $n \in \mathbb{N}$. An instance of type STACK is given by the template $\underline{stack}$ qualified by a surrogate $n \in \mathbb{N}$ (cf. definition 3.3 and example 3.4 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 object $(\emptyset,\emptyset,\{\varepsilon\},\lambda x.\emptyset)$. This corresponds to viewing d as a *constant* with its own value as "surrogate". Data elements cannot be changed by any event and cannot be observed via any attribute. Objects in general are like *variables*.

Object identification is studied in greater depth in the following sections. Object instantiation $\omega: U \rightarrow OB$ is based on object templates and qualification. Let $ob=(X,A,\Lambda,\alpha)$ be an object template and let $u \in U$ be an object surrogate.

Definition 3.3: The u-*instance* of ob is $u.ob=(u.X,u.A,u.\Lambda,u.\alpha)$ where $u.X=\{u.x \mid x \in X\}$, $u.A=\{u.a \mid a \in A\}$, $u.\Lambda=\{u.x_1\, u.x_2 \cdots \mid x_1 x_2 \cdots \in \Lambda\}$, and $u.\alpha(u.x_1 \cdots u.x_n)=\alpha(x_1 \cdots x_n)$.

Example 3.4: Referring to example 3.2 above, the n-instance of <u>stack</u>, n.<u>stack</u> 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.<u>stack</u> are those of <u>stack</u> with every event qualified by n. After any finite prefix of an n.<u>stack</u> life cycle, its observation is that of the corresponding <u>stack</u> trace obtained by omitting qualification.

For elementary object types (as in example 3.2), 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 cars as a subtype of CARs.

Let $OT_i=(U_i,\omega_i)$, $i=1,2$, be two object types, $U_i=(U_i,OP_i)$.

Definition 3.5: OT_1 is a *subtype* of OT_2 iff $U_1 \subseteq U_2$ and $\omega_1(u) \subseteq \omega_2(u)$ 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, however, obvious that the inclusion of U_1 in U_2 can be used as a key for U_1.

Generalizing this in an obvious way, we obtain the following notion of morphism between object *types*. Let OT_1 and OT_2 be as above.

Definition 3.6: An *object type morphism* $h: OT_1 \rightarrow OT_2$ consists of a mapping $h_U: U_1 \rightarrow U_2$ and a U_1-indexed family of object morphisms $h_\omega=\{h_\omega(u): \omega_1(u) \rightarrow \omega_2(h_U(u)) \mid u \in U_1\}$.

The class of all object types with their object type morphisms forms a category, called the *category* ***OT*** *of object types*.

As with objects, an essential categorial concept for studying object types is that of a colimit. Since the categories ***SET*** of sets and functions and ***OB*** of objects are cocomplete, the following theorem is evident.

Theorem 3.7: ***OT*** is cocomplete.

Coproducts of object types express *generalization*, for example LEGAL-PERSON = PERSON + COMPANY. Identification for generalized types is studied in the following sections. As to instantiation: each person or company instance is by itself a legal-person instance, with the same surrogate and with the same object template.

On the instance level, object *aggregation* is expressed by parallel composition which is related to colimits in ***OB***. Very generally speaking, composing an object type by generalization or aggregation is described by a parameterized data type (or data type constructor) by which a corresponding composite universe is built. Generalized instances are simply left unchanged, and aggregated instances are obtained as parallel compositions of a given sharing structure. Assuming for the moment being that there is no sharing, the aggregated instances are built by disjoint interleaving. We discuss generalization and the most useful forms of aggregation: tupling and grouping by means of sets and lists. The construction of the universes will be discussed in the following sections.

The object type $OT=OT_1+OT_2$ consists of the universe $\mathbf{U=U_1+U_2}$ and the instantiation mapping $\omega(u)=\omega_1(u)$ if $u\in U_1$ and $\omega_2(u)$ otherwise. Considering the injections consisting of those of the universes and the identities on the instance level, OT is a coproduct of OT_1 and OT_2 in the category ***OT***.

The object type $OT=OT_1\times OT_2$ consists of the universe $\mathbf{U=U_1\times U_2}$ and the instantiation mapping $\omega(u_1,u_2)=\omega_1(u_1)\parallel\omega_2(u_2)$.

The object type $OT=\mathbf{set}\,OT_1$ can be defined as follows. For the universe, we have $\mathbf{U=set\,U_1}$, 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 $OT=\mathbf{list}\,OT_1$ can be defined in a similar way. For the universe, we have $\mathbf{U=list\,U_1}$, 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.

By adding interaction information to one or several object types, an object *society* can be built: it is an object *instance*, constructed from all instances of the types and the interaction information expressed by object sharing, utilizing colimits and parallel composition.

We illustrate 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 can 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 the event sharing morphisms.

4. Complex Object Universes

Complex object types are composed from other object types. In this section, we study what this means for object *universes*. We study in particular, how complex object universes can be *specified*.

As composition operators for universes, we assume the following: + denotes disjoint union and is used to express generalization; × denotes cartesian product and is used to express one form of aggregation (tupling); **set** denotes the set of finite subsets and is used to express another form of aggregation (grouping). Other forms of aggregation like **list**, **bag**, **tree**, . . . can be introduced, but we will not do so, for the ease of presentation. Knowing how it works with **set**, it is not difficult to work out the details for the others.

Let S be some set of *base sorts*. Each sort $s \in S$ denotes a set $A(s)$.

Definition 4.1 : The set S^* of *sort expressions* over S is inductively defined as follows:

(1) each sort $s \in S$ is a sort expression,

(2) **0** and **1** are sort expressions,

(3) if α and β are sort expressions, so are $\alpha+\beta$, $\alpha\times\beta$, **set** α ,

(4) nothing else is a sort expression.

Like sorts, sort expressions are interpreted by sets. Their interpretation is completely determined by that of the base sorts, if we define

$A(\mathbf{0}) = \emptyset$	(empty set)
$A(\mathbf{1}) = \{\emptyset\}$	(one-element set)
$A(\alpha+\beta) = A(\alpha) + A(\beta)$	(disjoint union)
$A(\alpha\times\beta) = A(\alpha) \times A(\beta)$	(cartesian product)
$A(\mathbf{set}\,\alpha) = \mathbf{P}_{\text{fin}}A(\alpha)$	(finite subsets)

With these interpretations in mind and working "up to isomorphism", we assume that + and × are associative, and that + is also commutative. So we will write $\alpha_1+\cdots+\alpha_n$ or $\alpha_1\times\cdots\times\alpha_n$ without brackets , the former in arbitrary order. There are obvious isomorphisms $\alpha+\mathbf{0} \cong \alpha$ and $\alpha\times\mathbf{1} \cong \mathbf{1}\times\alpha \cong \alpha$. Thus, in a sense, we may view **0** and **1** as empty sum and product, respectively.

Like data types, object universes are algebras, so we have to consider what happens to the *operations* when composing the carrier sets. More precisely: given a set Ω of function symbols of the form $f:\alpha\to\beta$ (where, for the sake of generality, α and β are arbitrary sort expressions), which composite function symbols can be defined and interpreted reasonably? Clearly, $f:\alpha\to\beta$ is interpreted by a function $A(f): A(\alpha)\to A(\beta)$. In what follows, we will omit the $A(.)$; it will be clear from context what is meant.

From a set Ω of given *base functions*, we can compose new functions by parallel composition.

Definition 4.2 : The set Ω^* of *function expressions* over Ω is inductively defined as follows:

(1) each function $f:\alpha\to\beta$ is a function expression ,

(2) $\mathbf{0}:\mathbf{0}\to\mathbf{0}$ and $\mathbf{1}:\mathbf{1}\to\mathbf{1}$ are function expressions,

(3) If $f:\alpha\to\beta$ and $g:\gamma\to\delta$ are function expressions, so are $f+g:\alpha+\gamma\to\beta+\delta$, $f\times g:\alpha\times\gamma\to\beta\times\delta$, and **set** $f:$ **set** $\alpha\to$ **set** β .

(4) nothing else is a function expression.

0 denotes the empty function, and **1** denotes the one-element function sending the only element in 1 to the only element in 1 . For the other compositions we have

$$f{+}g(a) = \begin{cases} f(a) \text{ if a is of sort } \alpha \\ g(a) \text{ if a is of sort } \gamma \end{cases},$$

$$f{\times}g(a,b) = (\,f(a), g(b)\,)\ ,$$

$$\textbf{set}\ f(\{a_1,\ldots,a_n\}) = \{\,f(a_1),\ldots,f(a_n)\}\,.$$

With these interpretations in mind, we may assume that + and × are associative and that + is commutative, as in the case of sets. So we will write $f_1+\cdots+f_n$ or $f_1\times\cdots\times f_n$ here, too. Again, we have isomorphisms $f{+}\mathbf{0}\cong\mathbf{0}$ and $f{\times}\mathbf{1}\cong\mathbf{1}{\times}f\cong f$ so that we can view **0** and **1** as empty sum and product, respectively.

5. Object Identification

We assume that we have a family DATA of data types as a supply of values for attributes. Following the algebraic approach, DATA is a Σ_{DT}-algebra where $\Sigma_{DT}=(S_{DT},\Omega_{DT})$ is a data signature. A number of techniques are available to specify a specific algebra DATA (abstractly, i.e. up to isomorphism) within the category of all Σ_{DT}-algebras. We do not go into this issue here.

The object universe provides surrogates for all object instances and an identification system in terms of naming operations, ultimately based on data values. We would like to give semantically meaningful identification systems for complex object structures with intricate interdependencies, and such identification systems can be rather sophisticated.

To give a few examples, persons may be identified by simple keys like social security number or by name, address and birthdate. In some applications, however, it may be more convenient to identify persons by their name and their affiliation which is, say, another object of sort company (object-valued keys). To make things more complicated, in some contexts, persons may be identified by their name (etc.) and their father who is another object of the *same* sort person (recursive key). Even more complicated is an identification system for parts which may be atomic or composite, where atomic parts are identified, say, by part numbers, and composite parts are identified by the set of their components which are parts in turn. This involves recursive keys, generalization and complex (set-valued) keys.

In practice, there are often several keys for the same object class, like name, address, affiliation, etc., which *together* identify the objects. Formally, using product sorts, we can combine n such keys $k_1: \alpha \rightarrow \beta_1, \ldots, k_n: \alpha \rightarrow \beta_n$ into *just one* key $k: \alpha \rightarrow \beta_1\times\cdots\times\beta_n$ where k_i is recovered by k and subsequent projection on the i-th component.

It is quite common in real life to have several *alternative* keys where either one is sufficient to identify the object. In this paper, we do not go into this ramification. Rather, we assume that we have *exactly one* key for each object sort expression. Many-keyed objects can be handled as usual in the database field, picking one "primary" key and letting the others be attributes, appropriately equipped with constraints.

Let $\Sigma_{DT}=(S_{DT},\Omega_{DT})$ be a data signature, and let S_{OB} be a set of object sorts. Let $S=S_{DT}\cup S_{OB}$.

Definition 5.1 : A *key signature* $\Sigma_{KY}=(S_{OB},\Omega_{KY})$ *over* Σ_{DT} gives a set Ω_{KY} of function symbols $k[s]: s\rightarrow\alpha_s$, one for each object sort $s\epsilon S_{OB}$, with $\alpha_s\epsilon S^*$. The *extended* key signature of Σ_{KY} is $\Sigma^*_{KY}=(S^*-S_{DT},\Omega^*_{KY})$.

Clearly, if Σ_{KY} is a key signature, then Σ^*_{KY} is again a key signature. Intuitively, a key signature gives the object sorts and a single-keyed identification system for objects of these sorts.

Definition 5.2: Let Σ_{DT} be a data signature and Σ_{KY} be a key signature over Σ_{DT}. A *universe signature* $\Sigma_{UN}=\Sigma_{DT}+\Sigma_{KY}$ *over* Σ_{DT} *and* Σ_{KY} is an extension of Σ_{DT} by Σ_{KY}. The *extended* universe signature is $\Sigma^*_{UN}=\Sigma_{DT}+\Sigma^*_{KY}$.

We want to give a Σ^*_{UN}-algebra as a standard interpretation for the extended universe signature Σ^*_{UN} that can serve as a universe. In the data part, of course, the given standard semantics DATA of Σ_{DT} should be preserved, i.e. all data elements, and no additional data elements, should belong to the universe. Technically, this means that the intended universe **U** should have a Σ_{DT}-reduct which is isomorphic to DATA:

(U1) $\quad \mathbf{U}|\Sigma_{DT}\cong \text{DATA}$.

This requirement, of course, does not yet characterize the intended universe **U** uniquely (up to isomorphism), so we look for further conditions that **U** should reasonably satisfy.

Considerations of observability and constructivity suggest the following:

(U2) any two different objects in **U** should be distinguishable by key values,

(U3) any object in **U** should be representable by its key values.

We have to define what we mean by this. Two data elements or objects are distinguishable iff they are not indistinguishable in the following sense.

Definition 5.3 (*indistinguishability*):

(1) any two data elements are indistinguishable iff they are equal,

(2) tuples are indistinguishable iff their corresponding components are indistinguishable,

(3) finite sets are indistinguishable iff there is a 1-1 correspondence of indistinguishable elements,

(4) any two objects are indistinguishable iff their key values are indistinguishable.

Definition 5.4 (*representability*):

(1) every data element is representable (by itself),

(2) tuples are representable iff all their components are representable,

(3) finite sets are representable iff all their elements are representable

(4) objects are representable iff their key values are representable.

The three properties U1, U2 and U3 do not yet specify a universe uniquely (up to isomorphism) either, but we are close. We have to require in addition that **U** be maximal with these properties, i.e. **U** is not contained in any larger Σ_{UN}-algebra with the same properties.

Definition 5.5: Let Σ_{UN} be a universe signature. A *universe* for Σ_{UN} is a maximal Σ^*_{UN}-algebra satisfying U1, U2 and U3.

Thus, a universe provides a maximal set of surrogates for objects that can be represented and distinguished with the given key system. Our main result shows that such a universe exists and is (essentially) unique.

Theorem 5.6: For any universe signature Σ_{UN}, there is a unique (up to isomorphism) universe **U** for Σ_{UN} .

Proof: Within the initial-algebra framework of equational data type specification, a Σ^*_{UN}-algebra **U** satisfying U1, U2 and U3 can be specified along the following lines. The specification consists of Σ^*_{UN} and the following additional (hidden) operators and equations:

1. a "key generator" $k'[s]:\alpha_s \longrightarrow s$ as an inverse for each key operator $k[s]:s\longrightarrow\alpha_s$ in Ω_{KY}, together with equations $k'[s](k[s](x))=x$ and $k[s](k'[s](y))=y$.
2. operators and equations for specifying the desired interpretation of the structured sorts, involving injection operators for generating disjoint unions, construction operators for generating cartesian products, etc. Since initial specifications for these purposes are well known, we do not go into further detail here.
3. appropriate equations describing the desired interpretation of the composite key operators in $\Omega^*_{KY}-\Omega_{KY}$, using the operators in 2. We do not go into detail here either.

Let **U'** be an initial algebra of this specification, and let **U** be its Σ^*_{UN}-reduct. Clearly, **U** and **U'** have the same carriers.

Obviously, **U'** is specified as a conservative extension of DATA. Consequently, U1 holds for **U**. As an initial algebra, **U'** is a free extension of DATA. Thus, all elements in the carriers of object sorts are generated from data elements by the key generators and the generators for the structured sorts. Consequently, U3 holds for **U**. From the equations given in 1 above, it follows that each key operator must be interpreted by an injective function. Consequently, U2 holds for **U**.

Maximality and uniqueness (up to isomorphism) of **U** follow from the following

Proposition: For any Σ^*_{UN}-algebra **V** satisfying U1, U2 and U3, there is an injective morphism $h:\mathbf{V}\longrightarrow\mathbf{U}$.

For constructing h, we observe that there is an isomorphism from the data part of **V** to that of **U**. This is the basic building block for h. We define how h works on object "base" sorts, i.e. $s\in S_{OB}$. It is then obvious from definition 4.2 how h works on the remaining structured sorts $\alpha\in S^*$.

Let a be an element in the carrier of object base sort $s\in S_{OB}$. Consider its key value $b=k[s](a)$ of sort α_s. If b is mapped by h to h(b) in **U**, then a is mapped by h to that element h(a) such that $k[s](h(a))=h(b)$ holds. This h(a) exists in **U**: it is the element $k'[s](h(b))$ in **U'**.

Because of representability, this gives a well-behaved inductive definition of h. By construction, h is a Σ^*_{UN}-algebra morphism. By distinguishability, h is injective. Consequently, h(**V**) is a subalgebra of **U** containing DATA. □

Remark 5.7: In Eh86 and EDG86, a final algebra approach to constructing universes is given, restricted to keys without generalization and without complex objects. This final approach has been extended to generalized and complex keys in Wi87. The approach here is different: it is based on initial algebras (cf. SSE87). The universes of the final approach satisfy U1 and U2, but in general not U3, i.e. they are not necessarily representable. The universe described here is recovered in the final universe when restricting the latter to all representable objects.

6. Specialization

Besides generalization and aggregation into complex objects, there is another important mechanism for deriving new object sorts from old ones, namely *specialization*. For instance, the object sort **car** may be specialized to **sports car**, **Diesel car**, **compact car**, **midsize car**, etc. with the intention that the specialized sorts inherit their identification systems from **car**. Sorts specialized from the same sort need not denote disjoint object sets, as the above example shows, while generalization means disjoint union. And there can be objects not occurring in any specialization, which is not the case with generalization.

Specialization is easily included in our framework: we introduce a sort s specialized from α by defining the *inclusion* function $\mathrm{in}: s \hookrightarrow \alpha$ as a key in the key signature. Being a little sloppy, we can treat the composition $k[\alpha]\mathrm{in}$ as a "key" of s.

For example, if cars are identified by serial number $s\#$:**car**$\longrightarrow$**int**, then **sports cars** are also identified by serial number $s\#$: **sports car** $\hookrightarrow$ **car** $\longrightarrow$ **int**, too, and the same holds for **Diesel cars**, etc.

Using inclusions $\mathrm{in}: s \hookrightarrow \alpha$, specializations of arbitrary structured sorts can be defined, for instance **rescue vehicle** as a specialization of **car + aircraft**. In the universe **U** according to theorem 5.6 above, these inclusions can always be interpreted by *identity* functions. That means that α and its specialization s denote the *same* set of potential objects. And this is meaningful: in any *actual* population, we expect that the **sports cars** are among the **cars**, but not necessarily all of them. In the *universe*, however, not having any actual information, *all* (serial numbers of) **cars** are *potential* (serial numbers of) **sports cars**.

If we include key constraints, a sort s specialized from α may very well be more constrained than α. For example, we may know that **Diesel cars** are never produced by Rolls Royce, whereas this does not hold for **cars** in general, and also not for other specializations like **sports cars**. In this case, the set of potential objects of sort s (specialized from α) is properly contained in that of α.

On the other hand, any constraint on α is effective for its specialization s, too. This follows from the fact that the universe is an algebra: the inclusions have to be total functions.

The approach also works for the case where we have *one* specialization s for *several* different sorts $\alpha_1,\ldots,\alpha_r$ (a situation called "multiple inheritance"), for instance **motor planes** as **motor vehicles** and as **aircrafts**. We only have to make the (reasonable) assumption that these α's be subsorts of one common supersort β. Then $k(s): s \hookrightarrow \beta$ is the key of s, and everything works.

7. Constraints

With object universe specification by keys, there is another problem which we cannot discuss here in depth: in many applications, it is desirable to give *constraints* on keys in order to exclude surrogates for objects that are intuitively not possible in the model world of the application. As an example, consider persons whose birthdate precedes those of their parents.

Let C_{KY} be such a set of constraints, for instance in 1st-order predicate calculus. In general, the universe **U** for Σ_{UN} as constructed above will not satisfy C_{KY} , but C_{KY} will probably be valid in some subalgebras of **U** (containing all data elements).

Let SUB(**U**) be the set of all subalgebras of **U** with the same data part, say DATA. SUB(**U**) is partially ordered by inclusion, in fact, it forms a boolean lattice. Any set $Q \subseteq SUB(\mathbf{U})$ of subalgebras has a least upper bound, lub(Q). An obvious candidate for a universe for Σ_{UN} with key constraints C_{KY} would be the maximal subalgebra of **U** satisfying C_{KY}, provided that it is unique. Unfortunately, this need not be the case (cf. Eh86).

A sufficient condition that there is a unique such maximal subalgebra is the following: each constraint $\varphi \in C_{KY}$ that holds in all subalgebras in $Q \subseteq SUB(\mathbf{U})$ also holds in lub(Q), for all collections of subalgebras $Q \subseteq SUB(\mathbf{U})$.

For the case of keys without generalization or complex objects, a class of formulas with this property has been characterized in Eh86 (called *positive* formulas there). These results, however, do not carry over to the more general case studied here.

8. Concluding Remarks

This paper gives a sketchy outline of an algebraic theory of objects and object types, concentrating on object identity. The main technical result shows that universe signatures have canonical models which can be used as standard universes.

There are many problems with specifying objects, object types and object societies which are only briefly touched upon or not mentioned at all in this paper. E.g., object *instances* are observed *processes*, and here the wide field of processes and their specification comes in. There are many approaches, among them Petri nets, Hoare's CSP, Milner's CCs, and various forms of logic, for instance temporal logic, process logic, action logic,event logic, etc. Our process model, i.e. sets of streams as life cycles, is very simple and not powerful enough to capture all aspects of concurrency and nondeterminism. It was chosen to get the general idea clear, but it should be replaced by a more elaborate one, hopefully showing similar characteristics with respect to process morphisms and colimits.

Since objects are *observed* processes, not only the processes have to be specified, but also their effects on attributes, that is, roughly speaking, storage places containing elements from data types. It is a challenging problem to look for an appropriate blend of specification methods that can be put together, covering all aspects of objects and object types, with a nice model theory, useful deduction capabilities, and helpful operational aspects for analysis, implementation, verification and execution.

It is essential that the specification formalism allows for *abstract* descriptions so that it is possible to find out how an object - or a group of objects - behaves without having to "look inside". This will help to materialize one of the great potentials of the object-oriented approaches, namely to establish a methodology for producing not only correct and efficient, but also reusable code.

Acknowledgement Thanks are due to the anonymous referee for helpful comments.

References

Am86	America,P.: Object-Oriented Programming: A Theoretician's Introduction. EATCS Bulletin 29 (1986), 69-84
BT88	Bergstra,J.A.;Tucker,J.V.: The Inescapable Stack: an Exercise in Algebraic Specification with Total Functions. Report No. P8804, Programming Research Group, University of Amsterdam 1988
DD86	Dayal,U.;Dittrich,K.(eds): Proc. Int. Workshop on Object-Oriented Database Systems. IEEE Computer Society, Los Angeles 1986
DMN67	Dahl,O.-J.;Myhrhaug,B.;Nygaard,K.: SIMULA 67, Common Base Language, Norwegian Computing Center, Oslo 1967
Eh86	Ehrich,H.-D.: Key Extensions of Abstract Data Types, Final Algebras, and Database Semantics. Proc. Workshop on Category Theory and Computer Programming (D. Pitt et al, eds.), LNCS 240, Springer-Verlag, Berlin 1986, 412-433
EDG86	Ehrich,H.-D.;Drosten,K.;Gogolla,M.: Towards an Algebraic Semantics for Database Specification. Data and Knowledge, R.Meersman, A.Sernadas (eds.), North-Holland, Amsterdam 1988, 119-135
ESS88	Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: From Data Types to Object Types (to be published)
GM87	Goguen,J.A.;Meseguer,J.: Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In SW87, 417-477
Go79	Goldblatt,R.: Topoi, the Categorial Analysis of Logic. North-Holland Publ. Comp., Amsterdam 1979
GR83	Goldberg,A.;Robson,D.: Smalltalk 80: The Language and its Implementation. Addison-Wesley, Reading, Mass. 1983
Ho85	Hoare,C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs 1985
KC86	Khoshafian,S.N.;Copeland,G.P.: Object Identity. Proc. OOPSLA'86, ACM SIGPLAN Notices 21:11 (1986), 406-416
Lo85	Lochovski,F.(ed.): Special Issue on Object-Oriented Systems. IEEE Database Engineering 8:4 (1985)
SFSE89	Sernadas,A.;Fiadeiro,J.;Sernadas,C.;Ehrich,H.-D.: The Basic Building Block of Information Systems (to be published)
SW87	Shriver,B.;Wegner,P.(eds.): Research Directions in Object-Oriented Programming. The MIT Press, Cambridge, Mass. 1987
SSE87	Sernadas,A,;Sernadas,C.;Ehrich,H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. Proc. 13th VLDB, P.M.Stocker, W.Kent (eds.), Morgan-Kaufmann Publ. Inc., Los Altos 1987, 107-116
SSE89	Sernadas,A,;Sernadas,C.;Ehrich,H.-D.: Object-Oriented Language Features for Information Systems Specification (to be published)
Wi87	Winter,J.-H.: Zur Semantik von Schlüsselsignaturen mit Generalisierung und mengenwertigen Funktionen. Diplomarbeit, TU Braunschweig 1987