ON THE ALGEBRAIC SPECIFICATION OF DATABASES

Walter Dosch* Gianfranco Mascari** Martin Wirsing*

* Institut für Informatik - Technische Universität München - Arcisstrasse 21 - D-8000 Munich 2 Fed. Rep. of Germany

** Istituto di Automatica - Università degli Studi di Roma - Via Eudossiana 18 - I-00185 Rome Italy

ABSTRACT

For the conceptual level of database schemes a structured algebraic specification is presented. Within a uniform framework it comprises database abstractions, static and dynamic constraints, and a functional programming language for queries and updates. The specification is analysed w.r.t. syntactic and semantic aspects. Then the behaviour and the implementation of database specifications are discussed. Furthermore, algebraic conditions are presented that guarantee a sound specification according to criteria evolved from database theory.

0. INTRODUCTION

In the field of database specifications there is a growing interest in rigorous formal specifications (/Bjørner 80/, /Brodie, Zilles 81/, /Neuhold, Olnhoff 81/) which support a structured database design (/Ehrig, Fey 81/). Such specifications may essentially be classified according to the techniques used:

In constructive approaches a database scheme is specified by defining an abstract model, that is a specific mathematical structure onto which the universe of discourse is mapped.

In axiomatic approaches the behaviour is directly specified by means of a logic theory without resorting to a particular model. The abstract models of a constructive approach then provide for a concrete implementation.

The algebraic specification technique which combines tools from logic and universal algebra has proved to be a powerful and flexible tool for the formal definition of data structures (/Bauer, Wössner 82/) and programming language semantics. Therefore the development of algebraic database specifications (/Ehrig et al. 78/, /Paolini 81/) seems to be quite promising.

The algebraic specification technique may be employed for both approaches: It can serve to define specific abstract models (/Lockemann et al. 79/, /Hupbach 81/) . In this way the meaning of (semantic) data models which often is only partially described can be completely specified. But above all algebraic types can be used to specify database schemes directly.

For every specification technique it is necessary to develop criteria for a sound database definition (/Casanova et al. 81/). One guideline is a syntactic and semantic analysis of database specification in order to check whether the class of admitted models indeed meets the informal ideas in mind.

The algebraic specification technique offers a uniform framework for various subjects for which in database theory different mathematical tools, e.g. set theory, relations (/Codd 70/), logic (/Gallaire 81/), are used side-by-side. Many of the concepts developed as part of this technique seem to adequately meet the requirements of database specifications:

- The signature of an algebraic type specifies a syntactic interface by naming the available sorts, for example attributes or entities, and the operations with their arities.
- The axioms of an algebraic type, being arbitrary first order formulas, provide flexibility in expressing static and dynamic constraints.
- The use of partial functions captures finite errors, which may be induced by constraints, and infinite errors, that is the non-termination of queries or updates.
- In a hierarchy of algebraic specifications the visible behaviour of the nonprimitive parts is described by mapping them into primitive types.
- By an encapsulation mechanism the representation of the objects and operations can be hidden from the user.
- Algebraic specifications allow full parameterization. In this way the specification of database schemes gets possible with full generality.
- Algebraic specification languages, like CLEAR (/Burstall, Goguen 80/), and ASL (/Wirsing 82/), give precise semantics to flexible mani-

pulations with theories. For example, the specification of a large database scheme (/Albano, et al. 81/, /Hammer, Berkowitz 80/, /Ehrig et al. 78/) can be decomposed into parts of manageable complexity and, vice-versa, theories for small parts, like views of different users, may be put together to get an overall specification.

The paper is organized as follows: Section 1 contains a short overview on some basic notions of algebraic specifications. In section 2 external and conceptual database schemes are defined and related to each other by an abstract domain equation. In section 3 a conceptual database scheme is specified by a hierarchy of algebraic types using algebraically defined database abstractions. In section 4 a syntactic analysis of the specification leads to a classification of query and update operations as well as static and dynamic constraints. Section 5 to 7 provide tools for a semantic analysis: In section 5 criteria for the existence of Armstrong models are presented. In section 6 the behaviour of databases is formalized and the existence of behaviour models is discussed. In section 7 the implementation of database schemes is defined and some characteristic implementation steps are outlined. In section 8 a functional programming language for updates and queries is introduced. Finally in section 9 a notion of recursion complete language for updates and queries is algebraically defined, its equivalence to the "extended completeness " of /Chandra, Harel 80/ is outlined and the recursion completeness of the functional programming language (over set-like data structures) is shown.

This completes the aim of the paper to provide an abstract programming language model for database schemes. Several examples accompany the presentation. In the notation we largely follow the wide spectrum language CIP-L (/Bauer et al. 81/).

1. BASIC DEFINITIONS

Below we summarize some basic notions of algebraic specifications which extend the wellknown theory to partial algebras. For a more detailed treatment see for example /Wirsing et al. 80/ and /Wirsing 82/.

> Readers interested in an informal survey of this paper may skip this section.

1.1 SIGNATURES

A signature $\Sigma = \langle S, F\rangle$ comprises a set S of sorts and a set F of operation symbols together with their arities of $S^* \times S$. The union of signatures $\langle S_i, F_i\rangle$ means $\langle S_1 \cup S_2, F_1 \cup F_2\rangle$. A pair $\langle \alpha, \beta\rangle$ forms a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ if $\alpha : S_1 \to S_2$ is a total mapping for sorts and $\beta : F_1 \to F_2$ a family of mappings for operation symbols compatible with α. For a S-indexed family X of variables $W(\Sigma, X)$ denotes the set of all (finite) Σ-terms including the ground terms $W(\Sigma) = W(\Sigma, \emptyset)$.

1.2 PARTIAL ALGEBRAS

In a (partial) Σ-algebra $A = \langle (s^A)_{s\in S}, (f^A)_{f\in F}\rangle$ with each sort s of S a carrier set A_s and with each operation symbol $f : s1\times\ldots\times sk \to s \in F$ a (possibly partial) function $f^A : A_{s1}\times\ldots\times A_{sk} \to A_s$ is associated. A is finitely generated if every element of its carrier sets can be obtained by the interpretation t^A of a ground term $t \in W(\Sigma)$. The term algebra $W(\Sigma, X)$ and the ground term algebra $W(\Sigma)$ are total finitely generated Σ-algebras. For a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ and a Σ_2-algebra A^2 the σ-restriction $A^2/_\sigma$ is the Σ_1-algebra A^1 with $A^1_s = A^2_{\sigma(s)}$ and $f^{A^1} = \sigma(f)^{A^2}$ $(s,f \in \Sigma_1)$.

1.3 HOMOMORPHISMS, EXTREMAL ALGEBRAS

A family $(\varphi_s : A^1_s \to A^2_s)_{s\in S}$ of total mappings is called (strong) Σ-homomorphism $\varphi : A^1 \to A^2$ between two Σ-algebras A^1, A^2 if φ preserves the definedness (and undefinedness) of functions and is compatible with them.

In a class C of Σ-algebras the isomorphism classes I resp. Z of (strongly) initial resp. (strongly) terminal algebras are characterized by the existence of (strong) homomorphisms $\varphi : I \to A$ resp. $\varphi : A \to Z$ for all $A \in C$.

1.4 FORMULAS

Σ-formulas are all first-order formulas built from the definedness predicate $D_s(t)$ and the strong equality $t_1 =_s t_2$ as atomic formulas using the quantifiers $\forall$, $\exists$ and the connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$. The satisfaction $A \models \Phi$ of a Σ-formula Φ in a Σ-algebra A is defined as usual where for atomic formulas and allquantification

$A \models D_s(t)$ iff there exists an $a \in A_s : t^A = a$

$A \models t_1 =_s t_2$ iff t_1^A and t_2^A are both undefined or, both defined and equal $t_1^A = t_2^A \in A_s$.

$A \models \forall sx : P$ iff for all $a \in A_s : A \models P[a/x]$.

1.5 ALGEBRAIC TYPES

An algebraic type $T = \langle \Sigma, E\rangle$ consists of a signature Σ and a (countable) set E of closed Σ-formulas, called axioms. Its semantics $Mod(T)$ is the class of all Σ-algebras where all axioms $\Phi \in E$ are satisfied and $A \models \underline{true} \neq \underline{false}$. A consistent type , that is a type T with

$Mod(T) \neq \emptyset$, is called monomorphic, if $Mod(T)$ comprises only isomorphic algebras, and polymorphic elsewise. For a type T a Σ-formula Φ is provable ($T \vdash \Phi$), if Φ is deducible from E, valid ($T \models \Phi$), if all models of T satisfy Φ. A formula $\forall sx : \Phi(x)$ where Φ is quantifier-free is called maximal if for all ground terms t $T \models D(t)$ implies $T \models \Phi(t)$ or $T \models \neg \Phi(t)$.

1.6. HIERARCHICAL ALGEBRAIC TYPES

To structure the specification in a hierarchical algebraic type $T = \langle \Sigma, E, T^1 \rangle$ a primitive (sub-) type $T^1 = \langle \Sigma^1, E^1, T^2 \rangle$ is designated consisting of a primitive signature $\Sigma^1 \subseteq \Sigma$ of primitive sorts $S^1 \subseteq S$ and primitive operations $F^1 \subseteq F$, a subset $E^1 \subseteq E$ of primitive axioms and a (possibly empty) primitive subtype T^2.

The primitive terms $W(\Sigma^1, X^1)$ and the terms of primitive sort form subsets of $W(\Sigma, X)$. In a hierarchical type the non-primitive objects are specified by their visible behaviour under output operations, that is by operations leading from a nonprimitive sort into a primitive sort.

The visible behaviour should be specified sufficiently precise. T is called weakly sufficiently complete, if for all ground terms $t \in W(\Sigma)$ of primitive sort there exists a primitive term $p \in W(\Sigma^1)$ with $T \vdash D(t) \Rightarrow T \vdash t = p$, and sufficiently complete, if furthermore $T \vdash D(t)$ or $T \vdash \neg D(t)$. A related model-oriented notion is : A type $T = \langle \Sigma, E, T^1 \rangle$ is hierarchy persistent if for every model A^1 of the primitive type T^1 there is a model A of T such that $A/_{\Sigma^1}$ is A^1, and for every model B of T the reduct $B/_{\Sigma^1}$ is a model of T^1.

1.7 STRUCTURED ALGEBRAIC SPECIFICATIONS

In an algebraic specification language for the manipulation of algebraic types composed type expressions can be built from signatures and sets of axioms as atomic expressions using

- the quotient $T : E$ of a type expression T and a set E of axioms,
- the sum $T^1 + T^2$ of two type expressions T^1, T^2,
- the restriction $T/_\sigma$ of a type expression and a signature morphism $\sigma : \Sigma' \to \Sigma$. If $\Sigma' \subseteq \Sigma$ and σ is the inclusion operation, we write $T/_{\Sigma'}$ instead of $T/_\sigma$.
- the abstraction (PAR):T of a type expression and a type variable PAR yielding a parameterized type or type scheme with parameter type PAR,
- the application (instantiation) T(ARG) of a type scheme T to an argument type ARG,
- the constraint data(T) to finitely generated and minimally defined models of T.

This kernel language can be extended by notational variants like (data-)enrich by. The result of enriching a type $T = \langle \Sigma, E \rangle$ by $\langle \Sigma', E' \rangle$ is the type $T' = \langle \Sigma \cup \Sigma', E \cup E' \rangle$. A type T is data-enriched by $\langle \Sigma', E' \rangle$ if for every model A' of T' the reduct $A'/_\Sigma$ is a finitely generated minimally defined model of T.

The precondition notation

$$\underline{\text{funct}}\ (\underline{s}_1\ x : pre(x))\ \underline{s}\ f$$

in the signature of a type is an abbreviation for the declaration

$$\underline{\text{funct}}\ (\underline{s}_1)\ \underline{s}\ f$$

and the axiom

$$pre(x) = \underline{\text{false}} \Rightarrow D(f(x)) = \underline{\text{false}}\ .$$

2. EXTERNAL AND CONCEPTUAL LEVEL OF DATABASES

At the external level a database (scheme) may be seen as a black box. The only way to talk about the information contained in it is to put queries to it. There are different (groups of) users each interacting with the database using a specific set of queries and updates. In this situation both the complete database DB and the view V_i of a user may be described by hierarchical algebraic types. For each user there is a signature Σ_i naming the available sorts and operations. From the conceptual level DB an external view V_i is obtained by restricting DB to Σ_i ; conversely a database is the sum of its views where certain interferences among the views V_i are respected.

Definition A conceptual database scheme DB and an external database scheme or view V_i are parameterized hierarchical algebraic types such that

(1) $DB = (V_1 + \ldots + V_n) : Interference(V_1, \ldots, V_n)$
(2) $V_i = DB/_{\Sigma_i}$ for $i = 1,2,\ldots,n$ holds,

where $\Sigma_i = sig(V_i)$ are signatures and $Interference(V_1,\ldots,V_n)$ are $\Sigma_1 \cup \ldots \cup \Sigma_n$ - axioms.

This definition reflects two directions in the conceptual design:

a) First the external view V_i and their interferences are specified separately and then integrated into a conceptual database scheme DB. Algebraically this means to solve the abstract domain equation

(3) $DB = (DB/\Sigma_1 + \ldots + DB/\Sigma_n) :$
$Interference(DB/_{\Sigma_1}, \ldots, DB/_{\Sigma_n})$ $(n \geq 1)$
for the unknown type variable DB.

b) First the conceptual level DB is specified and then the different external views V_i are derived from it according to equation (2).

As it is well known from fixpoint theory of recursive functions the domain equation (3) in general has a lot of solutions. When using the method b) the specified conceptual scheme often contains more detailed information than needed for the different external schemes, then DB is not a minimal solution of equation (3). Furthermore when using database abstractions to construct a conceptual scheme, in general not a minimal solution of (3) is obtained.

3. SPECIFICATION OF THE CONCEPTUAL LEVEL

Below we specify the conceptual level of a database scheme by defining a hierarchy of algebraic types for its components.

3.1 OBJECT TYPES

The conceptual database specification as well as the views are modular composed (/Schiel et al. 82/) of various subtypes such as several object types $OBJ_1, \ldots, OBJ_n$ that define the data structures of the data items.

In general the basic object types OBJ_i are not monomorphic. Nevertheless the following requirements should be fulfilled:

Claim The hierarchical types OBJ_i for the basic objects should be specified such that

1) the definedness predicate is model-independent,
2) they are hierarchy persistent,
3) for their semantics only finitely generated models are regarded.

Condition 1) ensures that the definedness of terms is model independent, 2) allows to implement the overall type by using implementations of its primitive types, 3) ensures that every object has a finite denotation and can be finitely computed, i.e., there is "no junk" in the models.

3.2 DATABASE ABSTRACTIONS

In general the basic object types OBJ_i are themselves obtained in a hierarchical way based on other primitive types. To abbreviate type schemes frequently occurring in the construction of database specifications database abstractions have been introduced (/Hammer 76/, /Smith, Smith 77/, /dos Santos et al. 80/, /Brodie 81/). They are similar to generic mode constructors in programming languages (/Schmidt 80/). Below we specify aggregation, generalization, and correspondence.

3.2.1 AGGREGATION

Definition An aggregation is the type scheme

type AGGREGATION ≡ ($OBJ_1, \ldots, OBJ_n$, σ, PRE, Σ, E) :
data-enrich PROD($OBJ_1, \ldots, OBJ_n$)/$_\sigma$: PRE by Σ, E
endoftype ,

where the type scheme PROD forms the cartesian product of its parameter types (/Bauer, Wössner 82/)

type PROD ≡ (OBJ_1, ..., OBJ_n) :
  data-enrich OBJ_1, ..., OBJ_n, BOOL by
  sort prod ,
  funct (obj_1 x_1, ..., obj_n x_n :
      pre(x_1, ..., x_n)) prod mk,
  funct (prod) obj_i sel_i ,
  funct (obj_1, ..., obj_n) bool pre ,
  pre($x_1, \ldots, x_n$) ⇒ sel_i(mk($x_1, \ldots, x_n$)) = x_i
endoftype

If the precondition pre is constant true,

PRE = { pre(x_1, ..., x_n) = true }

then mk is a total operation; in this case we omit pre and PRE and write TPROD for PROD .

Example
A secretary may be characterized by the components name, age (over 18) and typing speed. Furthermore an operation to increase the age is provided.

type SECRETARY ≡ :
  data-enrich (secr, mksecr, sname, sage, speed,
      isadult) ≡
    PROD(NAME, AGE, TYPING_SPEED) :
    isadult(n, a, s) = (a ≥ 18)
  by funct (secr) secr increage ,
  increage(mksecr(n, a, s)) = mksecr(n, a+1, s)
endoftype

Here the notation

(secr, mksecr, ..., isadult) ≡
    PROD(NAME, AGE, TYPING_SPEED)

abbreviates the signature morphism σ of PROD/$_\sigma$ by establishing the correspondence

secr → prod, mksecr → mk, ..., pre → isadult .

Similarly one may specify a type DIRECTOR consisting of the component types NAME, AGE, and TELEPHONE.

3.2.2 GENERALIZATION

Definition A generalization is the type scheme

type GENERALIZATION $= (OBJ_1, \ldots, OBJ_n, \sigma, PRE, \Sigma, E)$:
  data-enrich $SUM(OBJ_1, \ldots, OBJ_n)/_\sigma$: PRE by Σ, E
endoftype ,

where the type scheme SUM defines the direct sum (disjoint union) of its parameters:

type SUM = (OBJ_1, ..., OBJ_n) :
  data-enrich OBJ_1, ..., OBJ_n, BOOL by
  sort sum ,
  funct (obj_i x_i: $pre_i(x_i)$) sum mk_i ,
  funct (sum) bool is_i ,
  funct (sum s : $is_i(s)$) obj_i pr_i ,
  funct (obj_i) bool pre_i ,
  $pre_i(x_i)$ ⇒ [$is_i(mk_i(x_i))$ = true ,
    $is_j(mk_i(x_i))$ = false , $(i \neq j)$
    $pr_i(mk_i(x_i))$ = x_i]
endoftype

Example

The staff of a company consists of either secretaries with excellent typing speed or directors older than 40 years. It additionally may have an operation to access the name of the staff directly.

type STAFF = :
  data-enrich (staff,
    sstaff, issecr, secr, quicktype,
    dstaff, isdir, dir, matureage) =
    SUM(SECRETARY, DIRECTOR) :
  quicktype(s) = (speed(s) ≥ 300) ,
  matureage(d) = (dage(d) ≥ 40)
  by funct (staff) name stname ,
  issecr(st) ⇒ stname(st) = sname(secr(st)) ,
  isdir(st) ⇒ stname(st) = dname(dir(st))
endoftype

3.2.3 CORRESPONDENCE

Many (semantic) data models widely use sets but nevertheless do not support the data structure set explicitly. Sets as objects are required in the conceptual modelling whenever sets have specific properties exceeding the properties of their elements.

Definition A correspondence is the type scheme

type CORRESPONDENCE = $(OBJ, \sigma, PRE, \Sigma, E)$:
  data-enrich $FINSET(OBJ)/_\sigma$: PRE by Σ, E
endoftype ,

where the type scheme FINSET defines finite sets of a member type OBJ (cf. /Wirsing et al. 80/) :

type FINSET = (OBJ) :
  data-enrich OBJ, BOOL by
  sort finset ,
  funct finset emptyset ,
  funct (finset s, obj x :
    preins(s, x)) finset insert ,
  funct (finset s, obj x :
    predel(s, x)) finset delete ,
  funct (finset) bool isempty ,
  funct (finset, obj) bool iselem ,
  funct (finset s : ¬ isempty(s)) obj choose ,
  funct (finset, obj) bool preins, predel ,
  iselem(emptyset, x) = false ,
  preins(s, y) ⇒
  iselem(insert(s,y), x)= (eq(x,y) ∨ iselem(s,x)),
  predel(s, y) ⇒
  iselem(delete(s,y), x)= (¬ eq(x,y) ∧ iselem(s,x)),
  isempty(s) ⇔ ¬ ∃ obj x : iselem(s, x),
  ¬ isempty(s) ⇒ iselem(s, choose(s)) = true
endoftype

Here eq denotes an equality operation on OBJ. Note that the result of the partial operation choose ("Give me some element of the set") is undefined for the empty set, uniquely determined for sets with one element, and ambiguously for sets with two or more elements. Furthermore, delete is specified as a constructor (cf. also section 5).

Example

A secretary belongs to a set of candidates when she submits to work and is no longer a candidate when she withdraws. This has to respect the following constraints: A secretary withdrawing (submitting) must (not) be a candidate.

type CANDIDATES = (SECRETARY) :
  data (cands, emptycand,csubmit,cwithdraw,cisempty,
    iscand,cchoose, precsubmit, precwithdraw)
    = FINSET(SECRETARY) :
    precsubmit(c, s) = ¬ iscand(c, s) ,
    precwithdraw(c, s) = iscand(c, s)
endoftype

3.2.4 AN EXAMPLE: EMPLOYMENT AGENCY

In this section for the well-known example of an employment agency (/Veloso et al. 81/) a conceptual scheme is specified by a hierarchy of algebraic types using database abstractions. It is based on the primitive types SECRETARY and COMPANY for secretaries and companies.

A secretary working for a company is an employee. The following type assumes no constraints, that is every secretary may work for every company.

type EMPLOYEE ≡ (SECRETARY, COMPANY) :
  (empl, < .,. >, secr, comp) ≡
    TPROD(SECRETARY, COMPANY)
endoftype

In the next step the set of employees is specified with the following constraints:

- A secretary cannot be hired by a company when she is already working for any company.
- She only can be fired by a company when she is working for this company.

type EMPLOYEES ≡ (EMPLOYEE) :
  data (empls, emptyempls, ehire, efire, eisempty,
    worksfor, echoose, preehire, preefire) ≡
    FINSET(EMPLOYEE) :
  preehire(e, <s, r >) =
    (∀ comp r_1 : ¬ worksfor(e, <s,r_1>)) ,
  preefire(e, <s,r>) = worksfor(e,< s,r>))
endoftype

Then an employment agency consists of a set of candidates and a set of employees respecting the following integrity constraint :

(PRE) No secretary can simultaneously be a candidate and an employee.

Moreover there are the following constraints:

(PRES) A secretary submitting can neither be a candidate nor be working for a company.

(PREH) A secretary can only be hired by a company if she is a candidate and does not work for any company.

(PREF) A secretary getting fired by a company must work for this company.

type EMPL_AGENCY ≡ (CANDIDATE, EMPLOYEE) :
  data-enrich (ag, <.,.>, cands, empls, preag) ≡
    PROD(CANDIDATES, EMPLOYEES) :

(PRE) preag(c, e) = ∀ secr s, comp r :
  [(iscand(c, s) ⇒ ¬ worksfor(e, < s,r>)) ∧
  (worksfor(e, <s,r>) ⇒ ¬ iscand(c, s))]
  by funct (ag a, secr s : presubmit(a, s))
    ag submit ,
  funct (ag, secr) bool presubmit ,
  funct (ag a, empl e : prehire(a, e)) ag hire,
  funct (ag a, empl e : prefire(a, e)) ag fire,
  funct (ag, empl) bool prehire, prefire,

  preag(c, e) ⇒

(PRES) [presubmit(< c, e>, s) = (¬ iscand(c,s) ∧
    ∀ comp r : ¬ worksfor(e, < s, r>)) ,

(S) presubmit(<c, e>, s) ⇒
    submit(<c, e>, s)= < csubmit(c, s), e >,

(PREH) prehire(<c,e>, <s,r >) =(iscand(c,s) ∧
    ∀ comp r_1 : ¬ worksfor(e, < s,r_1>)) ,

(H) prehire(<c,e>,<s,r>) ⇒ hire(<c,e>,<s,r>) =
    <cwithdraw(c,s), ehire(e, < s,r>)>,

(PREF) prefire(<c,e>,<s,r>) = worksfor(e,<s,r>) ,

(F) prefire(<c,e>,<s,r>) ⇒ fire(<c,e>,<s,r>) =
    < csubmit(c,s), efire(e, <s,r>)>]
endoftype

For the consistency proof it is necessary to check that the update operations submit, hire, and fire preserve the integrity constraint (PRE):

presubmit(<c,e>, s) ⇒ preag(csubmit(c,s), e) ,

prehire(<c,e >,<s,r>) ⇒
  preag(cwithdraw(c,s), ehire(e, <s,r>)) ,

prefire(<c,e>, <s,r>) ⇒
  preag(csubmit(c,s), efire(e, <s,r>))

This can be proved using the preconditions (PRE), (PRES), (PREH), and (PREF). For example, the application of submit keeps the integrity whereas csubmit may violate it. For the final specification of the employment agency all updates have to be hidden which possibly destroy the integrity:

type EMPLOYMENT_AGENCY ≡ (CANDIDATE, EMPLOYEE) :
  EMPL_AGENCY(CANDIDATE, EMPLOYEE)/$_\Sigma$
endoftype where
Σ ≡ sig(EMPL_AGENCY) ∖
    { csubmit, cwithdraw, ehire, efire }

For this type requirements of /Veloso et al. 81/ are provable (as postconditions), for example:

(POSTS) a = submit(a1, s) ⇒ iscand(cands(a), s) ∧
    ∀ comp r : ¬ worksfor(empls(a),<s, r>) .

More complex operations like

funct (ag, secr) bool works

('Does a secretary work?') may be specified at this level by an enrichment using descriptive formulations like

works(a,s) = ∃ comp r : worksfor(a, <s,r>)

or structural induction

works(<c, emptyempls>,s) = false ,

preag(c , ehire(e,<s_1,r>)) ⇒

works(<c, ehire(e, <s_1,r>)>, s_2) =

(eqs(s_1,s_2) ∨ works(<c,e>, s_2)),

preag(c, efire(e, <s_1, r>)) ⇒

works(<c, efire(e, <s_1, r>)>, s_2) =

(¬ eqs(s_1, s_2) ∧ works(<c, e>, s_2)).

Alternatively these operations may be defined in the next level using a programming language, see section 8.

4. SYNTACTIC ANALYSIS

In this section algebraic specifications for database schemes are analysed under syntactic aspects.

The hierarchical structure of algebraic types induces a syntactic classification of the operations as well as of the axioms:

Definition

For a database specification DB=<Σ,E,OBJ_1,...,OBJ_n> with primitive subtypes OBJ_i we call an operation symbol f of arity funct (s_1, ..., s_k) s query operation if the range s of f is a primitive sort, that is $s \in \bigcup_{i=1}^{n} sig(OBJ_i)$, and update operation otherwise.

Example

In the type SECRETARY sname, sage, speed, and isadult are query operations, whereas mksecr and increage are update operations.

In the type EMPLOYMENT_AGENCY for example iscand and worksfor are queries whereas hire, fire, and submit are updates.

Now let DB be a hierarchical algebraic type specifying a (view of a) conceptual database (scheme) which is based on the primitive object types OBJ_1, ..., OBJ_n. Then the signature

$$\Sigma\text{-DB} = sig(DB) \setminus \bigcup_{i=1}^{n} sig(OBJ_i)$$

contains all sorts and operation symbols of DB without the sorts and operations of the primitive object types.

Definition

An operation $f \in sig(OBJ_i)$ is called object operation, and an operation $f \in \Sigma$-DB is called holding operation.

These two definitions allow a two dimensional classification of the database operations into holding updates and holding queries as well as in object updates and object queries.

Example

In the EMPLOYMENT_AGENCY there are the object queries sname, sage, speed, the object update increage, the holding queries iscand and worksfor, and holding updates submit, hire, and fire.

This classification of the operations may also be used to discriminate two kinds of constraints.

Definition

In a database specification DB a formula Φ with DB ⊨ Φ is called static constraint iff there does not occur any operation f in Φ, which is a hcIding update. Otherwise Φ is called dynamic constraint.

Example

In the EMPLOYMENT_AGENCY the (provable) formula

mksecr(sname(s), sage(s), speed(s)) = s

as well as the axiom (PRE) are static constraints, whereas the preconditions (PRES), (PREH), (PREF) and the postcondition (POSTS) are dynamic constraints.

In the following sections the specification for database schemes is semantically analysed with respect to notions evolved from database theory. These notions are also related to algebraic concepts.

5. ARMSTRONG MODELS

In order to get an overview of the properties of a specification special models are studied.

Definition

An algebra A is called Armstrong model (/Makowsky 81/, /Fagin 82/) of a database specification DB, if all first order formulas which are provable in DB and which only contain terms the definedness of which is provable, are valid in A and no others.

The existence of Armstrong models is often accepted as a criterium for a sound specification since they help the database designer to see what he has implied by the axioms of his specification. Thus a major concern of research in database theory is to characterize the class of formulas for the description of data dependencies which allow or guarantee Armstrong models. For algebraic

types with partial functions there is the result:

Theorem

Let a database be specified by a consistent hierarchical type DB = <Σ, E, OBJ> where all axioms in E have positive conditional form

$$\forall \underline{s}\, x_1 \ldots \underline{s}_n x_n : \bigwedge_{i=1} D(p_i) \wedge p_i = q_i \Rightarrow C ,$$

where C is an atomic formula p = q or D(p).

(1) DB has an Armstrong model, iff DB is weakly sufficiently complete.

(2) Every Armstrong model is initial in DB.

(3) An Armstrong model is strongly initial in DB if for all ground terms $t \in W(\Sigma)$ either $DB \models D(t)$ or $DB \models \neg D(t)$.

Proof

See /Wirsing et al. 80/. □

For the database abstractions presented in section 3 the existence of Armstrong models is ensured whenever the preconditions and the parameter types behave well.

Proposition

For the database abstraction AGGREGATION (GENERALIZATION; CORRESPONDENCE) there exist Armstrong models, iff the operations pre (pre$_i$; preins, predel, choose) specified are weakly sufficiently complete and their parameter type(s) OBJ, ..., OBJ$_n$ (OBJ$_1$, ..., OBJ$_n$; OBJ) allow an Armstrong model.

Example

If cchoose is specified weakly sufficiently complete, then CANDIDATES has an Armstrong model I, where for example

(*) csubmit(emptycand, s) ≠

csubmit(cwithdrwaw(csubmit(emptycand,s),s),s)

holds. More generally, Armstrong models of type FINSET (if they exist) are based on sequences where multiplicity and ordering of the sequence elements is relevant for the equality. Also for implementing delete(s, x) the element x must be appended to the sequence s with a marker for being no longer present. Thus in I the candidate's history is completely reflected.

Also the type EMPLOYMENT_AGENCY has a (strongly initial) Armstrong model if cchoose and echoose are specified sufficiently complete. Similar to CANDIDATES in an Armstrong model of EMPLOYMENT_AGENCY the equality between objects of sort ag is determined by the sequence of updates submit, hire, fire leading to them. Thus Armstrong models contain redundant information which cannot be retrieved by queries.

In the partial algebra approach Armstrong models are minimally defined and defined terms are interpreted as different as possible. Thus the existence of Armstrong models is not endangered when adding inequalities such as (*) to a database specification as long as it remains consistent.

6. BEHAVIOUR OF DATABASES

In a view point dual to Armstrong models a database is significantly characterized by its behaviour, that is by all effects which are visible in the primitive types. Thus a model of a database specification is called a behaviour model if two elements are distinguished in the model only if they can be discriminated by queries. Such models can be defined using the behaviour view of a database which abstracts from the equations holding on its nonprimitive sorts.

For simplicity throughout section 6.1 and 6.2 let DB = <Σ, E, OBJ> be a database specification with exactly one nonprimitive sort db ∈ Σ where the primitive type OBJ contains one sort obj.

Definition

The behaviour view Beh(DB, V) of a database scheme DB = <Σ, E, OBJ> w.r.t. a subsignature V ⊆ Σ is given by

$$Beh(DB, V) = \Sigma + DB/_{sig(OBJ)} + OUT(V) : EQU(V)$$

where

$$OUT(V) = \{\underline{funct}\ \underline{obj}\ t_o \mid t \in W(V) \text{ of sort } \underline{obj}\}$$

$$EQU(V) = \{t = t_o \mid t \in W(V) \text{ of sort } \underline{obj}\}$$

Thus in a behaviour view of a database scheme DB the axioms of the sort db are neglected and all terms of W(V) of primitive sort obj are added as constants.

6.1 BEHAVIOUR MODELS

The identity of the nonprimitive parts of a database specification may be completely determined by their visible behaviour in the primitive object type OBJ. On the term level, one gets a visible behaviour of a nonprimitive object by putting the corresponding term into a context of a primitive sort.

Definition

a) A Σ-algebra A of a database specification DB = <Σ, E, OBJ> is called fully abstract, if for all t, t' ∈ W(Σ) of sort db

(1) $A \models D(t)$ iff there exists a context $c[x] \in W(DB, \{x\})$ of sort obj with $A \models D(c[t])$.

(2) $A \models t \neq t'$ iff there exists a context $c[x] \in W(DB, \{x\})$ of sort obj with $A \models c[t] \neq c[t']$.

b) A fully abstract model of Beh(DB, Σ) is called behaviour model of DB.

Proposition

a) If a database specification DB = <Σ, E, OBJ> is sufficiently complete, then all behaviour models of DB are isomorphic.
b) If furthermore all axioms of DB are of positive conditional form with maximal premisses and the definedness predicate D is model independent in DB, then the following is equivalent for an algebra A with signature of Beh(DB, Σ) :

(1) $A/_{\Sigma}$ is a strongly terminal model of DB .

(2) A is a strongly terminal model of Beh(DB, Σ)

(3) A is a behaviour model of DB .

Corollary

For the database abstractions AGGREGATION and GENERALIZATION Armstrong models and behaviour models coincide if the precondition operations pre resp. pre_i are sufficiently complete. In this case, AGGREGATION and GENERALIZATION are parameter monomorphic.

For the database abstraction CORRESPONDENCE all behaviour models are isomorphic if the operations preins, predel, and choose are sufficiently complete.

Example

The types SECRETARY and STAFF are parameter monomorphic.

The type CANDIDATES has behaviour models satisfying (in contrast to Armstrong models) for example the formula

csubmit(emptycand, s) =

csubmit(cwithdraw(csubmit(emptycand,s),s),s)

since its left and right hand side are not distinguishable by contexts of primitive sort.

6.2 DISTINCTION BY QUERIES

In a database specification the effects of update operations are visible to a user only by the change of results of queries. Thus internal states with different behaviour should be distinguishable by query operations only.

Definition

For a set Q of query operations a database specification DB = <Σ, E, OBJ > is called query distinctive, if any two states db, db' ∈ W(Σ) of sort db are distinguishable by queries, that is

$$Beh(DB, \Sigma)/_{\Sigma'} = DB'$$

where Σ' is the signature of

DB' = Beh(DB, <{db, obj}, Q>).

For database specifications that are sufficiently complete and query distinctive, behaviour models are completely characterized by queries.

Proposition

Let DB = <Σ, E, OBJ > be a database specification which is sufficiently complete and query distinctive. A Σ-algebra A is a behaviour model of DB iff for all db, db' ∈ W(Σ) of sort db

$A \models db \neq db' \leftrightarrow$ there exists a holding query $q \in \Sigma$ with $DB \models q(db) \neq q(db')$

$A \models D(db) \leftrightarrow$ there exists a holding query $q \in \Sigma$ with $DB \models D(q(db))$.

For the database abstractions of section 3 we get in particular:

Proposition

AGGREGATION, GENERALIZATION, and CORRESPONDENCE are query distinctive if the signature morphism σ does not forget any query operation.

Example

The type SECRETARY is query distinctive, since two objects of sort secr are different whenever they differ in at least one component NAME, AGE, or TYPING SPEED.

The distinction by queries is transitive with respect to parameter passing in parameterized types:

Proposition

If the database scheme DB(PAR) is query distinctive (w.r.t.its parameter type PAR) and ARG is query distinctive (w.r.t. its primitive type OBJ), then the database DB(ARG) is query distinctive (w.r.t. OBJ).

Corollary

If a database specification DB is built only by database abstractions then it is query distinctive if no query operations are forgotten by the signature morphisms used.

Example EMPLOYMENT AGENCY is query distinctive.

Finally, all composed data structures which do not allow to access all their components directly by a single operation are, in general, not query distinctive.

Example

Let the type FILE with parameter OBJ specify files using the updates emptyfile, rest, put, and the query operations get, eof. Then FILE is not query distinctive since files differing in the second, third, ... element cannot be distinguished by the queries get and eof. Note that the procedure get of PASCAL corresponds to the composition get•rest (see section 9): PASCAL-files are not query distinctive.

6.3 KEYS

In a database specification $DB=<\Sigma,E,OBJ_1,\ldots,OBJ_n>$ with various primitive types OBJ_i the queries leading to a distinguished primitive type $OBJi_0$ may be sufficient to discriminate the nonprimitive objects:

Definition

A primitive type OBJ_{i_0} of a database specification $DB = <\Sigma, E, OBJ_1, \ldots, OBJ_n>$ is called key if DB is query distinctive w.r.t. all operations with range in a sort of OBJ_{i_0}.

Example

Let secretaries be uniquely distinguishable by their names. Then NAME may serve as key since

$$SECRETARY/_{sig(SECRETARY)\setminus\{sage,\ speed,\ isadult\}}$$

is query distinctive, whereas

$$SECRETARY/_{sig(SECRETARY)\setminus\{sname\}}$$

will, in general, not be query distinctive.

7. IMPLEMENTATION AND BEHAVIOUR EQUIVALENCE OF DATABASES

At the beginning of a database design there stands the requirement analysis formalized as external or conceptual scheme. The conceptual scheme then is the starting point for a joint development towards an internal scheme. This development can be done for example by a stepwise refinement technique (cf. /Ehrig, Fey 81/) or by a series of transformation steps meliorating coherently algorithms and data structures (cf. /Bauer, Wössner 82/). The correctness of such refinement or transformation steps can be expressed and formalized by the notion of implementation. Each subsequent specification is an implementation of the previous one. The algebraic definition of an implementation does not depend on the structure of the specified database but only takes its behaviour into account (cf. /Ausiello et al. 80/).

Definition

A database specification $DB_1 = <\Sigma_1, E_1, OBJ_1>$ is called an implementation of $DB_2 = <\Sigma_2, E_2, OBJ_2>$ via a signature morphism

$$\kappa : OUT_2 \to OUT_1$$

if $\emptyset \neq Mod(Beh(DB_1,\Sigma_1)/_\kappa \subseteq Mod(Beh(DB_2,\Sigma_2)/_{OUT_2})$

where $OUT_i \equiv sig(OBJ_i) \cup OUT(\Sigma_i)$

(see also section 6) .

DB_1 and DB_2 are called behaviourally equivalent if they implement each other.

Informally DB_1 implements DB_2 if the visible parts of the models of DB_1 agree (after renaming) with the visible parts of models of DB_2. Therefore the semantics of queries is not affected by an implementation whereas the semantics of updates, in general, is.

When building hierarchies of data abstractions composite objects often share common components. In an implementation step these components are separated : The sum of two products with common subtypes can be implemented by the product of the common subtypes and the sum of the other components.

Proposition

Let $OBJ_1 \equiv PROD_a(OBJ, OBJ'_1)$ and $OBJ_2 \equiv PROD_b(OBJ, OBJ'_2)$.

Then $SUM(OBJ_1, OBJ_2)$ is implemented by $PROD(OBJ, SUM'(OBJ'_1, OBJ'_2))$ via the signature morphism κ induced by

$$\kappa(mk_1(mk_a(x, y_1)) = mk(x, mk'_1(y_1))$$
$$\kappa(mk_2(mk_b(x, y_2)) = mk(x, mk'_2(y_2))$$

Of course the implementation is more economic since the obj-component occurs only once in the product. However, as can be easily seen, both types are behaviourally equivalent.

Example

The type STAFF can be implemented by

PROD(NAME, AGE, SUM(TYPING_SPEED, TELEPHONE)) .

A quite different implementation step usually is applied to the set-like database abstraction CORRESPONDENCE. After introducing a key, tables (specified by the type scheme GREX below) may be used to implement the correspondence.

Definition An association is a type scheme

```
type ASSOCIATION ≡ (KEY, OBJ, σ, PRE, Σ, E) :
    data-enrich GREX(KEY, OBJ)/σ : PRE by Σ, E
endoftype where

type GREX ≡ (KEY, OBJ) :
  data-enrich KEY, OBJ, BOOL by
  sort grex ,
  funct grex init ,
  funct (grex g, key k, obj x :
                    preput(g, k, x)) grex put ,
  funct (grex g, key k : isin(g, k)) obj get ,
  funct (grex g,key k : predel(g,k)) grex cancel ,
  funct (grex, key) bool isin, predel ,
  funct (grex, key, obj) bool preput ,
```

isin(init, l) = false ,

preput(g, k, x) ⇒

isin(put(g,k,x), l) = (eqk(k,l) ∨ isin(g,l)),

isin(put(g,k,x), l) ⇒ get(put(g,k,x), l) =
if eqk(k,l) then x else get(g,l) fi ,

predel(cancel(put(g, k, x), l)) ⇒

cancel(put(g, k, x), l) =
if eqk(k, l) then if isin(g, l)
then cancel(g, l)
else g fi
else put(cancel(g,l),k,x) fi

endoftype

Proposition

Let OBJ = PROD(KEY, OBJ') such that for all key k, finset s there exists at most one o ∈ obj' with iselem(s, <k, o>). Then CORRESPONDENCE(KEY, OBJ) can be implemented by ASSOCIATION(KEY, OBJ) via the signature morphism κ induced by

$\kappa(insert(g, x)) = put(\kappa(g), sel_1(x), \ldots)$

$\kappa(delete(g, x)) = cancel(\kappa(g), sel_1(x))$

$\kappa(iselem(g, x)) = isin(\kappa(g), sel_1(x))$

and some particular operations getsome and isinit for κ(choose) and κ(isempty).

The particular choice of getsome such as "get an object with minimal key" often implies that CORRESPONDENCE and ASSOCIATION are not behaviourally equivalent.

Example

The type CANDIDATES can be implemented by the following type

type CANDIDATES' = (NAME, SECRETARY) :
data-enrich (cand', cinit', csubmit', cget',
cwithdraw', iscand', precwithdraw', precsubmit')
= GREX(NAME, SECRETARY)
by precsubmit'(g, i, x) = ¬ iscand'(g,i) ,
precwithdraw'(g, i) = iscand'(g, i)
by funct (cand') bool cisinit
funct (cand'c: ¬isinit(c)) secr getsome
...
endoftype

via the morphism induced by

κ(csubmit(g, x)) = csubmit'(κ(g), sname(x), x),

κ(iscand(g, x)) = iscand'(κ(g), sname(x)) .

8. AN ABSTRACT PROGRAMMING LANGUAGE MODEL

To express and perform complex database operations a database language is needed. The queries and updates of the database scheme serve as its primitive operations. Below we define a schematic implementation of a conceptual database scheme by a simple but adequate programming language. This completes an aim of this paper to propose an abstract programming language as a semantic database model. The programming language should be a scheme language: its semantics should be completely defined relative to an arbitrary database scheme providing the primitive operation symbols. The axioms for the programming language model are simple translations from the axioms of the database specification enriched by the axioms of the programming language constructs.

As an example we define a functional language (/cf. Buneman, Frankel 80/), in the FP-style of Backus to obtain recursion complete query and update languages.

Throughout this section for simplicity let DB = <Σ, E, OBJ > be a database specification with exactly one nonprimitive sort db and one primitive type OBJ with primitive sort pr .

8.1 THE BASIC DATABASE LANGUAGE

The basic database language B_DB_L is based on the types DB for the database specification and SEQU(OBJ) specifying sequences of objects. It comprises the sorts query and upd for query resp. update operations.

type B_DB_L = (DB, OBJ) :
data-enrich DB(OBJ), SEQU(OBJ) by
sort query, upd

Then the database scheme DB is "lifted" by the signature mapping α : sig(DB) → sig(B_DB_L) which relates sorts

α(db) = upd

α(obj) = query

α(pr) = query

and function symbols

$\alpha(funct(s_1, \ldots, s_n)\ s\ f) = funct\ \alpha(s)\ \overline{f}$.

Depending on its range s, a function symbol f yields an object $\overline{f}$ either of sort query or upd. Remembering the syntactic classification in section 4 we thus get by definition of α the following correspondence:

holding query f of DB	↦	funct query $\overline{f}$
object query f of OBJ	↦	funct query $\overline{f}$
holding update f of DB	↦	funct upd $\overline{f}$
object update f of OBJ	↦	funct query $\overline{f}$

Example

In B_DB_L(CANDIDATES, SECRETARY) the operations csubmit, cwithdraw, iscand, and precsubmit of the database CANDIDATES are translated into

funct upd csubmit, cwithdraw

funct query iscand, precsubmit .

8.2 A FUNCTIONAL DATABASE LANGUAGE

The basic database language is enriched by the language constructs Σ and axioms E describing a functional programming language.

type DB_L ≡ (DB, OBJ, ID) :
  data (NAT + ID + CONFIGURATION(DB, OBJ)) +
  enrich B_DB_L by Σ, E
endoftype

Functional programs are constructed using the conditional if . then . else . fi, the functional composition .∘. , a tuple constructor [.,.] with corresponding selector functions sel and (possibly recursive) function definitions recv and function calls call. For this purpose a type ID of identifiers is assumed. Thus for v ∈ {query, upd} the signature Σ comprises

funct (query, v, v) v if . then . else . fi ,
funct (v, v') upd .∘. (upd ∈ { v, v'}),
funct (query, query) query .∘., [.,.] ,
funct (nat) query sel ,
funct (id, v) v recv ,
funct (id) v call .

In the signature above function symbols with different arities are overloaded; for example there are actually four compositions

funct (query, upd) upd .∘. ,
funct (upd, query) upd .∘. ,
funct (upd, upd) upd .∘. ,
funct (query, query) query .∘. ,

three of which yield updates; they may change the state of DB. Note that (recursive) definitions of queries resp. updates have to be discriminated.

To define the semantics let a configuration consist of a database scheme and a sequence of input/output data:

type CONFIGURATION ≡ (DB, OBJ) :
  (conf, <.,.>, db, io) ≡ TPROD(DB, SEQU(OBJ))
endoftype

The following axioms of the semantic function

funct (v, conf) conf apply v ∈ {query, upd}

taken as left to right term rewrite rules describe an innermost (call-by-value) text substitution machine. The auxiliary function subst denotes the usual substitution, that is subst(x, f, g) replaces all free occurrences of call(x) in f by g ; its straightforward definition by structural induction is omitted.

$apply(\overline{f},<db,<o_1,\ldots,o_k>>)=<f(db,o_1,\ldots,o_k),<>>$
for holding updates $\underline{funct}(\underline{db},\underline{obj_1},\ldots,\underline{obj_k})\underline{db}$ f,

$apply(\overline{f},<db,<o_1,\ldots,o_k>>)=<db,f(db,o_1,\ldots,o_k)>>$
for holding queries $\underline{funct}(\underline{db},\underline{obj_1},\ldots,\underline{obj_k})\underline{obj_{k+1}}$ f

$apply(\overline{f},<db,<o_1,\ldots,o_k>>)=<db,<f(o_1,\ldots,o_k)>>$
for object operations $\underline{funct}(\underline{obj_1},\ldots,\underline{obj_k})\underline{obj_{k+1}}$ f

$io(apply(c,\pi))$ = <true> ⇒
  $apply(\underline{if}\ c\ \underline{then}\ v_1\ \underline{else}\ v_2\ \underline{fi},\pi) = apply(v_1,\pi)$

$io(apply(c,\pi))$ = <false> ⇒
  $apply(\underline{if}\ c\ \underline{then}\ v_1\ \underline{else}\ v_2\ \underline{fi},\pi) = apply(v_2,\pi)$,

$j \le k$ ⇒
$apply(sel(j),<db,<o_1,\ldots,o_k>>) = <db,<o_j>>$,

$apply([q_1,q_2],\pi) =$
  $<db(\pi), io(apply(q_1,\pi))\ \&\ io(apply(q_2,\pi))>$,

$apply(v_1 \circ v_2,\pi) = apply(v_1, apply(v_2,\pi))$,
$apply(recupd(f, v), \pi) =$
  $apply(subst(f, v, recupd(f, v)), \pi)$,

$\pi_2 = apply(subst(f, v, recquery(f, v)), \pi_1)$ ⇒
  $apply(recquery(f, v), \pi_1) = <db(\pi_1), io(\pi_2)>$

For brevity the axioms for context conditions like

  $j > k$ ⇒
  $D(apply(sel(j),<db,<o_1,\ldots,o_k>>)) = \underline{false}$

have been omitted. These axioms also specify that basic operations can only be applied according to their arity.

The functional database language has the following semantic properties:

Theorem

The type DB_L is weakly sufficiently complete and admits minimally defined behaviour models.

Proof See /Broy, Wirsing 80/. □

Note that the minimally defined models correspond to least fixed points and provide a mathematical

semantics for DB_L. The following example shows some properties of these models.

Example

In DB_L ≡ DB_L(CANDIDATES, SECRETARY, ID) the following equations hold (cf. section 3.2.3):

$apply(\overline{precsubmit}, \pi) = apply(\overline{not} \circ \overline{iscand}, \pi)$

$apply(\overline{precwithdraw}, \pi) = apply(\overline{iscand}, \pi)$

The operation apply is the only operation in DB_L with range in the primitive type CONFIGURATION; therefore all contexts of primitive sort for functional programs have the form apply(c[x], π). An induction on the minimally defined models of DB_L shows that $\overline{precsubmit}$ and $\overline{not} \circ \overline{iscand}$ as well as $\overline{precwithdraw}$ and $\overline{iscand}$ are not distinguishable by contexts of primitive sort. Thus

$$\overline{precsubmit} = \overline{not} \circ \overline{iscand}$$

$$\overline{precwithdraw} = \overline{iscand}$$

holds in all minimally defined behaviour models M of DB_L. Furthermore nonterminating queries or updates are undefined, for example

$$M \models D(recquery(x, call(x))) = \underline{false} .$$

Note that a database specification is implemented by its database language.

Proposition

For every database specification DB^1 the type $DB\_L(DB^1)$ is an implementation of DB^1 via the signature morphism κ, for example induced by

$$\kappa(f(db, o_1, \ldots, o_k)) = io(apply(\overline{f}, <db, <o_1, \ldots, o_k>>))$$

for all holding queries f of DB .

The following example shows how to use the functional database language.

Example

In DB_L ≡ DB_L(EMPLOYEES, EMPLOYEE, ID) the query $\overline{works}$ (cf. section 3.2.4) can be expressed by

recquery(w, body) ,

where

body ≡ if $\overline{eisempty}$
then $\overline{false}$
else if $\overline{eq} \circ [sel(1), \overline{secr} \circ \overline{echoose}]$
then $\overline{true}$
else call(w) ∘ $\overline{efire}$ ∘ $\overline{secr}$ ∘ $\overline{echoose}$
fi
fi

Then in DB_L it is provable that

$$apply(recquery(w, body), <db, <s>>) = <db, <works(db, s)>> .$$

where works is the operation specified in section 3.2.4.

Proposition

The application of recquery does not change the state of the database, that is

$$DB \models apply(recquery(f,q), <db_1, i_1>) = <db_2, i_2> \Rightarrow db_1 = db_2 .$$

From the full database languages an update language and a query language can be derived.

Definition

For a database language DB_L≡DB_L(DB,ID,SEQU(OBJ)) a query language Q_L may be given by

type Q_L ≡
[data-enrich DB_L by (v ∈ {query, upd})
funct (v, conf) seq(obj) qapply ,
qapply(v, π) = io(apply(v, π))]/$\overline{QUERY}$

endoftype

where

$$\overline{QUERY} = \{ recquery(f, v) \mid f \in \underline{id} \ \kappa \ v \in W(DB\_L) \} \cup \{ \underline{funct} (db, conf) \ \underline{sequ}(obj) \ qapply \} .$$

An update language UPD_L may be given by

type UPD_L ≡
[data-enrich DB_L by
funct (v, conf) db updapply ,(v ∈ {query, upd})
updapply(v, π) = db(apply(v, π))]/$\overline{UPDATE}$

endoftype

where

$$\overline{UPDATE} = sig(DB\_L) \cup \{ updapply \} \setminus \{ apply \} .$$

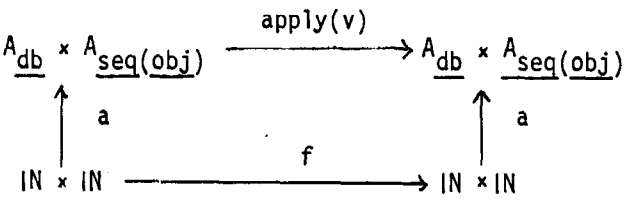
Thus programs of the query language do not change the state of the database, whereas programs of the update language do.

9. RECURSION COMPLETENESS

It has been advocated (/Chandra, Harel 80/) that the language to manipulate the database should have full computational power, i.e. that every partial recursive function can be computed.

Definition

A sig(DB_L)-algebra A is called recursion complete, if there exists a surjective recursive mapping $a : \mathbb{N} \times \mathbb{N} \to A_{db} \times A_{obj}$ such that for every partial recursive function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ there exists $v \in W(DB)$ of sort query or update such that $A \models apply(v, a(n, m)) = a(f(n, m))$, that is the diagram



commutes. A is called query resp. update recursion complete, if analogously for every partial recursive $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ there exists $q \in W(DB)$ of sort query such that

$$A \models qapply(q, a(n, m)) = a(f(n, m))$$

resp. an $u \in W(DB)$ of sort upd such that

$$A \models updapply(u, a(n, m)) = a(f(n, m))$$

holds.

Then for every database DB^1, $DB\_L(DB^1)$ is called query resp. update recursion complete if all minimally defined behaviour models of $DB\_L(DB^1)$ are query resp. update recursion complete.

Proposition

DB_L(FINSET) is query and update recursion complete.

DB_L is weakly sufficiently complete. Therefore the semantic operation apply - and thus also qapply - is a partial recursive function in every minimally defined model of Q_L(FINSET). Conversely the recursion completeness implies that every partial recursive relation can be simulated using qapply by

$$qapply(q) : obj_1 \times \ldots \times obj_k \to \{a(0), a(1)\} .$$

Both properties together show that Q_L(FINSET) is complete in the sense of /Chandra, Harel 80/.

Thus our notion of recursion completeness generalizes the completeness of Chandra, Harel ; it also generalizes their "extended completeness" since FINSET is parameterized with arbitrary object types.

10. CONCLUDING REMARKS

Algebraic types seem to be an interesting tool for the specification and analysis of database schemes. Such a specification may be considered as a stepwise development process where requirement analysis, external, conceptual, and internal schemes mark significant levels:

The transition from the requirement analysis to an external scheme can be seen as a refinement process where in each step the requirements are precized. The conceptual scheme is a solution of an "abstract domain equation" where the views of the external scheme have to be integrated. Then a joint development of data structures and algorithms leads to an internal scheme. In this step, for example keys are introduced and recursion can be removed.

The formal correctness of these transformations can be uniformly defined by an algebraic notion of implementation and thus algebraic methods can be used to support the specification of databases.

On the other hand for a sound database specification at each level it is necessary to analyse the properties of the algebraic types. Such an analysis may comprise the structure and behaviour of admissible models, the completeness of the specification, the complexity of the operations and the redundancy of information. This gives a guideline for the further development and a feedback with the informal ideas in mind.

When working with and reasoning about database specification a specification language seems to be needed to express flexibly various operations with algebraic types. But it would be very uneconomic (and to difficult for non-experts) to write down arbitrarily complex axiom systems. Therefore complementary to a specification language elaborated database abstractions may build the "skeleton" of a database-specification for which standard solutions and high level implementations are available.

ACKNOWLEDGEMENT

We gratefully acknowledge valuable discussions with Prof. G. Ausiello, C. Batini, Prof. F.L. Bauer, M. Brodie, Prof. E. Neuhold, and A. Pettorossi. Thanks go to R. Hyerle for reading a draft.

This research has been partially sponsored by the Sonderforschungsbereich 49, Programmiertechnik, München, and the Consiglio Nazionale delle Ricerche, Roma.

REFERENCES

/ALBANO et al. 81/
A. Albano, M.E. Occhiuto, R. Orsini: GALILEO: A conceptual language for database applications. A preliminary definition. CNR-PFI-DATAID Report No 13, November 1981

/AUSIELLO et al. 80/
G. Ausiello, C. Batini, M. Moscarini: Conceptual relations between databases transformed under join and projection. In: P. Dembinski (ed.): 10th Conf. on Mathematical Foundations of Computer Science. LNCS 88 (1980)

/BAUER, WÖSSNER 82/
F.L. Bauer, H. Wössner: Algorithmic language and program development. Berlin-Heidelberg-New York: Springer (1982)

/BAUER et al. 81/
F.L. Bauer, M. Broy, W. Dosch, R. Gnatz, F. Geiselbrechtinger, W. Hesse, B. Krieg-Brückner, A. Laut, T.A. Matzner, B. Möller, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, H. Wössner: Report on a wide spectrum language for program specification and development. Report TUM-I8104, Institut für Informatik, Technische Universität München (1981)

/BJØRNER 80/
D. Bjørner: Formalization of Data Base Models. In: D. Bjørner (ed.): Proc. Advanced Course on Abstract Software Specifications. LNCS 86 (1980)

/BRODIE 81/
M.L. Brodie: Association: A data base abstraction for semantic modelling. In P.P. Chen (ed.): Entity-Relationship Approach to Information Modelling and Analysis, ER Institute, Los Angeles, Oct. 1981, p. 583-608

/BRODIE, ZILLES 81/
M.L. Brodie, A.N. Zilles (eds.): Proc. Workshop on Data Abstraction, Data-Bases and Conceptual Modelling. SIGPLAN Notices 16 :1 (1981)

/BROY, WIRSING 80/
M. Broy, M. Wirsing: Algebraic definition of a functional programming language and its semantic models. Report TUM-I8008, Institut für Informatik, Technische Universität München, 1980. Also in RAIRO (to appear)

/BUNEMANN, FRANKEL 80/
O.P. Bunemann, R.E. Frankel: FQL - a functional query language. Proc. ACM SIGMOD, May 1979

/BURSTALL, GOGUEN 80/
R.M. Burstall, J.A. Goguen: The semantics of CLEAR: a specification language. In D. Bjørner (ed.): Proc. Advanced Course on Abstract Software Specification. LNCS 86 (1980)

/CASANOVA et al. 81/
M.A. Casanova, J.M.V. de Castilho, A.L. Furtado: Properties of conceptual and external database schemes. Technical Report DB108103, Depart. Informática, Rio de Janeiro, Brasil (1981)

/CHANDRA, HAREL 80/
A.K. Chandra, D. Harel: Computable queries for relational data bases, Journal of Computer and System Sciences 21, 156-178 (1980)

/CODD 70/
E.F. Codd: A relational model of data for large shared data banks. Comm. of the ACM 13:6, 377-387 (1970)

/EHRIG, FEY 81/
H. Ehrig, W. Fey: Methodology for the specification of software systems: from formal requirements to algebraic design specifications. In W. Brauer (ed.): Proc. 11th GI-Jahrestagung, München.Informatik-Fachberichte 50, Springer-Verlag (1981)

/EHRIG et al. 78/
H. Ehrig, H.-J. Kreowski, H. Weber: Algebraic specification schemes for data base systems. Proc. 4th Int. Conference on Very Large Data Bases, Berlin West (October 1978)

/FAGIN 82/
R. Fagin: Armstrong Databases. Proc. 7th IBM Symposium on Mathematical Foundations of Computer Science, Japan (May 1982)

/GALLAIRE 81/
H. Gallaire: Impacts of logic on data bases. Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981)

/HAMMER 76/
M.M. Hammer: Data abstractions for data bases. Proc. Conf. on Data: Abstraction, Definition and Structure. ACM FDT 8 (2)

/HAMMER, BERKOWITZ 80/
M. Hammer, B. Berkowitz: DIAL: A programming language for data intensive applications, SIGMOD 1980

/HAMMER, McLEOD 75/
M.M. Hammer, D.J. McLeod: Semantic integrity in a relational database system. Proc. 1st Int. Conf. on Very Large Data Bases, Framingham, Massachusetts (Sept. 1975)

/HUPBACH 81/
U. Hupbach: Modelling data bases by abstract data types. 4th Int. Datenbank Seminar, Schwerin, GDR (August 1981)

/LOCKEMANN et al. 79/
P.C. Lockemann, H.C. Mayr, W.H. Weil, W.H. Wohlleber: Data abstraction for data base systems. ACM Transactions on Database Systems, 4:1, 60-75 (1979)

/MAKOWSKY 81/
J.A. Makowsky: Characterizing data base dependencies. In: S. Even, O. Kariv (eds.): Proc. 8th Int. Colloquium on Automata, Languages and Programming. LNCS 115 (1981)

/NEUHOLD, OLNHOFF 81/
E.J. Neuhold, Th. Olnhoff: Building data base management systems through formal specifications. In: J. Diaz, I. Ramos (eds.): Formalization of programming concepts. LNCS 107(1981)

/PAOLINI 81/
P. Paolini: Abstract data types and data bases. In /Brodie, Zilles 81/

/dos SANTOS et al. 80/
C.S. dos Santos, E.J. Neuhold, A.L. Furtado: A data type approach to the entity-relationship model. In: P.P. Chen (ed.): Int. Conf. of the Entity-Relationship Approach to Systems Analysis and Design, North-Holland Amsterdam 1980, 103-119

/SCHIEL et al. 82/
W. Schiel, A.L. Furtado, E.J. Neuhold: Towards multi-level and modular conceptual schema specifications. Report 2/82, Institut für Informatik, Universität Stuttgart

/SCHMIDT 80/
J.W. Schmidt: Data Type Concepts for Databases. In: M. Atkinson (ed.): Data Design. Infotech State of the Art Report, Series 8, No 4 (1980)

/SMITH, SMITH 77/
J.M. Smith, D.C.P. Smith: Database Abstractions: Aggregation and Generalization: ACM Transactions on Database Systems 2 :2 (1977)

/VELOSO et al. 81/
P.A.S. Veloso, J.M.V. de Castilho, A.L. Furtado: Systematic derivation of complementary specifications. Proc. 7th Int. Conference on Very Large Data Bases, Cannes, France (September 1981)

/WIRSING 82/
M. Wirsing: Structured Algebraic Specifications. Proc. AFCET Symposium on Mathematics for Computer Science, Paris,March 1982

/WIRSING et al. 80/
M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy: On hierarchies of abstract data types. TUM-I8007. Institut für Informatik, Technische Universität München. Also Acta Informatica (to appear).