Restructuring hierarchical database objects

Serge Abiteboul, Richard Hull

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 615

RESTRUCTURING HIERARCHICAL DATABASE OBJECTS

Serge ABITEBOUL
Richard HULL

Février 1987

RESTRUCTURING HIERARCHICAL DATABASE OBJECTS[1]

RESTRUCTURATION D'OBJETS HIERARCHIQUES DANS LES BASES DE DONNEES

Serge Abiteboul
I.N.R.I.A.
Domaine de Rocquencourt
78153 Le Chesnay
FRANCE

Richard Hull[2]
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782
USA

16 Janvier 1987

Abstract: A class of hierarchical structures arising in Database Systems (complex objects) and Office Information Systems (forms) is studied. Two formalisms for restructuring are presented. The first focuses on a class of algebraic operators based on rewrite rules, and the second on structural transformations which preserve or augment data capacity. These transformations are related to a subclass of the rewrite operations which is closed under composition.

Résumé: Une classe de structures hiérarchiques utilisées dans les systèmes de gestion de bases de données (objets complexes), et dans les systèmes d'information en bureautique (formulaires) est étudiée. Deux formalismes pour restructurer l'information sont introduits. Le premier s'appuie sur une classe d'opérateurs algébriques fondés sur des règles de réécriture; le second sur des transformations structurelles préservant ou augmentant la puissance des structures de données. On montre une correspondance entre ces transformations structurelles, et une sous classe des opérations de réécritures fermée sous composition.

[1] An extended abstract of this paper appeared as [AH2].

[2] Work by this author supported in part by the National Science Foundation grant IST-83-06517, and IST-85-11541. Part of this work was performed while he was visiting I.N.R.I.A.

1. INTRODUCTION

A recent trend in the database field is to consider hierarchical data structures. Hierarchical data structures are central in the non-first-normal-form relational model [ABi,FT,RKS,SP...], and have been studied under the name formats [HY], and complex objects [ABe,BK]. These structures also arise naturally in semantic database models offering aggregation, grouping, and generalization [AH1,HK,HM,SS]. In fact, forms in Office Information Systems are similar kinds of structures [PFK,SLTC,T]. The purpose of this paper is to develop tools to manipulate typed hierarchical objects.

The focus of the paper is on typed hierarchical objects obtained using:

- tuple constructor (aggregation),
- set constructor (grouping), and
- union of types (generalization).

These objects and their underlying types are similar to those found in [HY,K,AH1]. Most other investigations have considered data structures involving the first two constructors only. The use of the third constructor allows to handle sets of objects of possibly different types. (In [BK], sets of objects of possibly different structures are considered without emphasizing the use of strict typing.) Another novel feature of the model is the utilization of particular constants which can serve as nonapplicable nulls, and are also used to model boolean and other finite domains.

One of the major research problems facing the database field is to understand how to manipulate hierarchical structures. Languages were already presented for typed objects built using the first two constructors [ABe,ABi,FT,J,KV,SP...]. Most of these investigations have focussed on operations to extract (selection/projection), or to combine information (union/difference), providing very limited capabilities for manipulating the structure of data (nest/unnest). The first theme of the paper is the presentation and study of an operation (called "rewrite operation") which (1) handles objects built using the three constructors, and (2) permits the specification of complex structural manipulations. We present some basic results on rewrite operations, and exhibit the subclass of "simple" rewrite operations which is closed under composition.

The second theme of the paper is to study the problem of data restructuring. In virtually all database models, it is possible to represent essentially the same data in different

ways. This notion of "data relativism" arises in a variety of contexts like database integration, view construction, form modification. It is thus crucial to understand data relativism at a fundamental level [H1,HY,HM], so that systems can effectively translate between alternative data representations.

Our study of data relativism is based on local structural transformations of types. Some of these transformations preserve data capacity, and lead to a characterization of "structural equivalence" which generalizes results of [HY]. Other transformations provide ways to augment the data capacity of a type. The effect of all these transformations can be achieved using only simple rewrite operations. If a user specifies the restructuring of the database using a sequence of transformations as defined here, the system can compute the new database state using a single rewrite operation (since simple rewrite operations are closed under composition).

The paper is organized as follows. Types are presented in Section 2 together with their corresponding objects. Section 3 introduces the rewrite operations, and a normal form for these operations. In Section 4, we show that the class of simple rewrite operations is closed under composition. Results concerning structural equivalence are presented in Section 5. Finally, augmentation is studied in Section 6. (A key lemma for Section 4 is proved in Appendix A. Motivating examples for the restrictions imposed on simple rewrite operations are gathered in Appendix B. Sketches of proofs for two results in Section 5 can be found in Appendix C.)

2. TYPES AND OBJECTS

The purpose of this section is to motivate and formally define types and objects.

We first present an example concerning two versions of information that might be stored in a personnel database. Consider the two templates shown in Figure 2.1. The structure of these forms is described by the types shown in Figure 2.2. Speaking roughly, a "type" is a tree with certain kinds of nodes. All leaf nodes of a type correspond to "basic types". This includes such types as "alpha", "9-dig", etc., and also some special type which has a one-element domain denoted by $\mathbf{1}$ (with a subscript). There are three kinds of internal nodes. Intuitively, $\times$-nodes correspond to the tuple constructor; $*$-nodes correspond to the

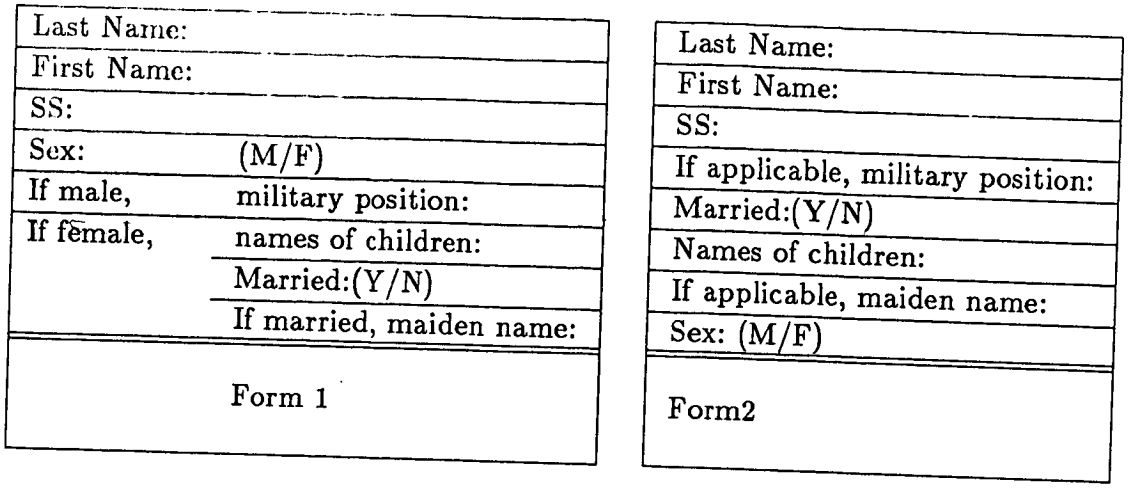

Figure 2.1: Two templates for a personnel database

set constructor; finally, +-nodes correspond to the union of types (or marked union) constructor.

Types as described here generalize the notion of "format" [HY], and the types of [AH1] to include basic types with one-element domain. As we shall see, these types can be used to represent nonapplicable nulls, and finite domains. Furthermore, they play an important role in our study of restructuring.

As mentioned in the introduction, types can serve as a formal model for representing data structures used in various disciplines. First, types subsume the family of form structures considered in Office Automation Systems [PFK,SLTC,T]. Furthermore, as indicated in the example, this model provides the possibility of faithfully representing forms in which different parts are to be filled out under different circumstances. (On the other hand, as described here the model does not directly incorporate the occurrence of dependent fields, such as one field which is to be the sum of two other fields). Also, types can be used to model classes of objects arising in Semantic Database Models [AH1,SS,HK,HM] which incorporate aggregation, grouping, and generalization. Finally, it should be noted that the structures of non-first-normal-form relations correspond to types in which tuple and set constructors alternate.

We now present the formal notions of type and object.

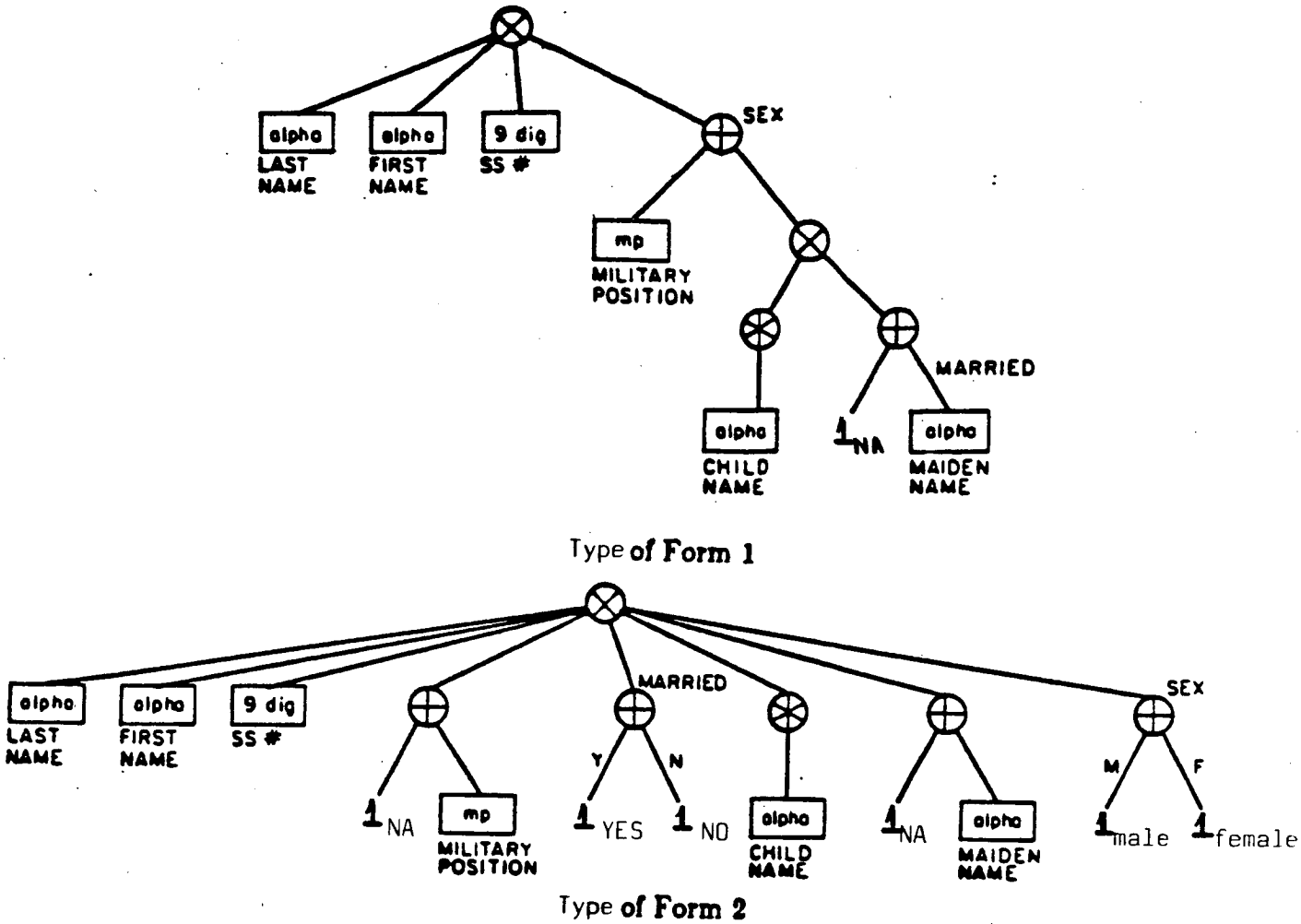

Figure 2.2: Types for Form1 and Form2

We assume the existence of infinite sets $dom_1,...,dom_n,...$ of *atomic values* called the *domains*. These sets are all countable and pairwise disjoint. We also assume the existence of a particular atomic value, denoted **1**, which is not in dom_i for any i. The set {**1**} will serve as the unique one-element *domain*.

We also assume the existence of an infinite set of symbols called *attributes*, which will be used as labels for the domains. To distingish between infinite and one-element domains, attributes with infinite domains are denoted with letters from the beginning of the alphabet; and attributes with one-element domain are denoted **1** with subscripts (e.g., $\mathbf{1}_f$, $\mathbf{1}_{yes}$...). Finally, we assume the existence of an infinite set of symbols called *tokens* which contains the set of attributes.

We use a syntactic representation for types where: square brackets ([,]) correspond to

tuple constructors, braces ({,}) correspond to set constructors, and angle brackets (<,>) correspond to union of types constructors. Formally, we have:

Definition: A *type* is an expression recursively defined as follows:

- if A is an attribute, $A{:}dom_i$ is a (basic) type for each i,
- if $\mathbf{1}_f$ is an attribute, $\mathbf{1}_f{:}\{\mathbf{1}\}$ is a (basic) type,
- if P is a non-attribute token, $P_1,...,P_n$ are distinct tokens, and $T_1{=}P_1{:}t_1,...,\ T_n{=}P_n{:}t_n$ are types, then $P{:}\{T_1\}$, $P{:}[T_1,...,T_n]$, and $P{:}\langle T_1;...;T_n\rangle$ are also types.

If P:t is a type, then the expression t is called a *structure* (i.e., a structure is a type deprived of its outermost token).

When domains are understood, we omit them from the specification of types. For instance, if it is understood that the domain of A and B is dom_1, we can use the expression $P{:}\langle Q{:}[A,B];\mathbf{1}_f\rangle$ to denote the type $P{:}\langle Q{:}[A{:}dom_1,B{:}dom_1];\mathbf{1}_f{:}\{\mathbf{1}\}\rangle$ (since the domain of $\mathbf{1}_f$ is by definition $\{\mathbf{1}\}$).

In the following, we assume that the order between tokens in a tuple or union of types constructor is irrelevant. For instance, we do not distinguish between the types $P{:}\langle Q{:}[A{:}dom_1,B{:}dom_1];\mathbf{1}_f{:}\{\mathbf{1}\}\rangle$ and $P{:}\langle \mathbf{1}_f{:}\{\mathbf{1}\};Q{:}[B{:}dom_1,A{:}dom_1]\rangle$. This motivates an alternative definition of types based on trees. Leaves of the trees denote atomic types, and are labelled by attributes. Internal nodes correspond to applications of the constructors, and are labelled by non-attribute tokens. Formally, we have:

Definition: A *type* is a rooted tree (V,E) where $V = V_+ \cup V_* \cup V_\times \cup V_b$ is the disjoint union of +-nodes, *-nodes, ×-nodes, and basic nodes such that

(i) a node is a basic node iff it is a leaf, and

(ii) each *-node has exactly one child.

The nodes of a type are labelled by tokens with the following restrictions:

(iii) a node is assigned an attribute iff it is a leaf, and

(iv) distinct siblings (i.e., distinct children of the same node) are assigned distinct tokens.

A domain is assigned to each leaf with the restriction that the one-element domain is

assigned to each leaf labelled $\mathbf{1}_f$ for some f.

The trees in Figure 2.2 correspond to the types in Figure 2.3 below. Note that some tokens are omitted in Figures 2.2 and 2.3. We will freely omit tokens when not necessary to the presentation.

It should be clear that this notion of type subsumes the concept of relational database schema. An example of relational database schema viewed as a type is given in Figure 2.4.

Form1:

[last_name,
first_name,
ss#,
sex: < male: military_position;
female: [children: {child_name},
married:
< $\mathbf{1}_{NA}$;maiden_name >] >]

Form2:

[last_name,
first_name,
ss#,
MP: < $\mathbf{1}_{NA}$; military_position >,
married: < $\mathbf{1}_{yes}$; $\mathbf{1}_{no}$ >,

children: { child_name },
MN: < $\mathbf{1}_{NA}$; maiden_name >,
sexe: < $\mathbf{1}_{male}$; $\mathbf{1}_{female}$ >]

Figure 2.3

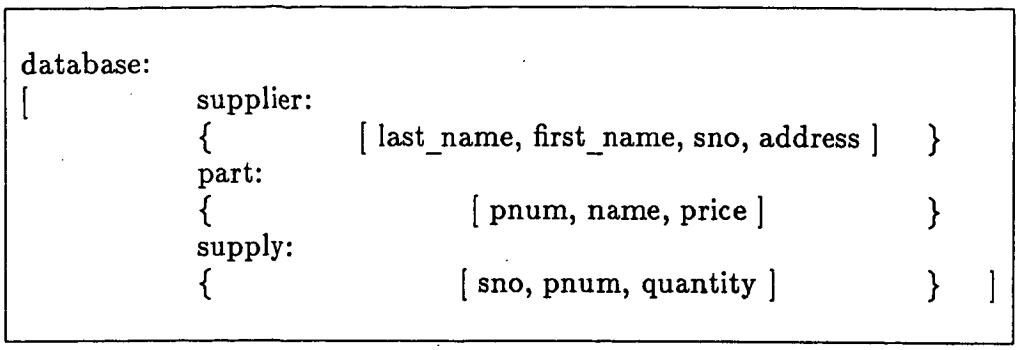

Figure 2.4

With each type, we associate a set of objects in the following way:

Definition: For each type T, the set of *objects* of type T, denoted obj(T), is defined recursively by:

- For each attribute[3] A, obj(A:dom_i)= {A:a | a in dom_i}, and for each index f, obj($\mathbf{1}_f$:{**1**})= {$\mathbf{1}_f$:**1**},
- obj(P:[T_1,...,T_n])= {P:[O_1,...,O_n] | $O_i \in$ obj(T_i) for each i in [1..n]},
- obj(P:{T'})= {P:{O_1,...,O_n} | $O_i \in$ obj(T') for each i in [1..n]}, and
- obj(P:<T_1;...;T_n>)= {P:<O_j> | $O_j \in$ obj(T_j) for some j in [1..n]}.

The set obj(T) is called the *domain* of T. If P:o is an object, then o is called the *value* of P:o (i.e., a value is an object deprived of its outermost token).

To illustrate the previous two definitions, we now give examples of objects, and their corresponding types:

(1) P:<B:13> and P:<R:[B:23, B':23]> are objects of type P:<B;R:[B,B']>.

(2) P:[Q_1:{R:[B:13,B':7,B'':105], R:[B:7,B':13,B'':13] }, Q_2:{}] is an object of type P:[Q_1: { R: [B,B',B''] }, Q_2: { S: [C,C'] }]

(3) P:{}, and P:{$\mathbf{1}_f$:**1**} are the only objects of type P:{$\mathbf{1}_f$}.

(where it is assumed that obj(B)= obj(B')= obj(B'') is the set of natural numbers.).

In our formalism, the convention of including tokens in objects is very strict. An advantage is that by inspection of an object, its type can (almost) be determined. This will turn to

be useful in our study of restructuring. Some other investigations may want to relax this constraint. On the other hand, new constraints on the naming convention may be introduced for query purposes, e.g., enforcing that two distinct nodes should be assigned distinct tokens.

When the type is understood, and an implicit ordering of siblings assumed, we can simplify the syntactic description of objects. For instance, the object of (2) can be written as [{[13,7,105], [7,13,13]}, {}].

3. REWRITE OPERATION

In this section, we introduce the rewrite operations on typed objects. These operations will be essential to our study of restructuring. We also exhibit a "decomposed form" for rewrite operations, and prove that every rewrite operation can be transformed into a decomposed one.

In order to manipulate data, we need a powerful operator. We use a notion of rule (inspired by the rules of [BK]) to describe how an object should be "rewritten". Indeed, a rewrite operation will be defined using *several* rules, each rule corresponding to some particular shape that can be taken by the operand.

We assume the existence of an infinite set **var** of symbols called *(unconstructed) variables*. Intuitively, variables will be place holders for values. The analogue for objects are the "terms" that are now defined.

Definition: For each type T, the *terms (of T)* are recursively defined by:

(i) each object of type T is a term of T,

(ii) if T = P:t for some structure t, and x is an unconstructed variable, P:x is a term of T,

(iii) if T = P:[T_1,...,T_n] is a type, and X_i a term of T_i for each i in [1..n], then P:[X_1,...,X_n] is a term of T,

(iv) if T = P:<T_1;...;T_n> is a type, and X_j a term of T_j for some j in [1..n], then P:<X_j> is a term of T, and

[3] Note that we distinguish between the *value* a, and the *object* A:a.

(v) if T = P:{T'} is a type, and X_i a term of T' for each i in [1..n], then P:{X_1,...,X_n} is a term of T.

If P:v is a term, then v is a *variable* (unconstructed if v is in **var**, constructed otherwise).

Note the strong analogy between (1) objects and values, (2) types and structures, and (3) terms and variables. (See Figure 3.1.) Note also that the definition of terms permits their specification with components of arbitrary granularity. For instance, P:x, P:[Q_1:y,Q_2:z], and P:[Q_1: {R:u_1,R:u_2}, Q_2:z] are terms of type

P:[Q_1: { R: [B,B',B"] }, Q_2: { S: [C,C'] }].

object	A:[B:12,C:7]	F:{D:0,D:7,D:12}
value	[B:12,C:7]	{D:0,D:7,D:12}
type	A:[B:int,C:int]	F:{D:int}
structure	[B:int,C:int]	{D:int}
term	A:[B:x,C:y]	F:x
variable	[B:x,C:y]	x

Figure 3.1

Given a type S, a term X of S, and a variable y appearing in X, a straightforward unification of X and S assigns[4] a structure t_y to y. The structure t_y will be called the *structure of* y *w.r.t.* X and S.

Rewrite rules will be used to specify object manipulation. We first present simple examples which show how rules can be used to perform simple selections and projections.

Example 3.1: Consider a type consisting of sets of Form1 as in Section 2. The following rules can be used on such forms:

renaming:

[4] Some problems arise if the variable y appears more than once in X. If the various occurences of y correspond to the same structure t, then y is assigned that structure. If this not the case, the structure of y w.r.t. X and S is undefined. We do not consider that case in the following.

[last_name:x, first_name:y, ss#:z, sex:w] → [nom: x, prenom:y, ss#:z, sexe:w]

ss# of persons with first name mary:

[last_name:x,first_name:'mary',ss#:z,sex:w] → ss#:z

persons with identical first and last names:

[last_name:x,first_name:x,ss#:z,sex:w] → [last_name:x,first_name:x,ss#:z,sex:w]

maiden_names (of married females):

[last_name:x,first_name:y,ss#:z,sex:<female:[children:u,married:<maiden_name:v>]>]
→ [last_name:x,maiden_name:v]

names of females with first name 'mary' having exactly one child:

[last_name:x,first_name:'mary',ss#:z,sex:<female:[children:{child:u},married:v]>]
→ last_name:x

Omitting certain tokens, the same queries are:

[x,y,z,t] → [nom:x,prenom:y,ss#:z,sexe:w]
[x,'mary',z,w] → z
[x,x,z,w] → [x,x,z,w]
[x,y,z,<[u,<v>]>] → [x,v]
[x,'mary',z,<[{u},v]>] → x

□

In the previous example, one rule is used to specify an operation. It is possible to use several rules to treat different options resulting from alternative structures of the operand as illustrated by the following example.

Example 3.2: Suppose that we want for each person, the set of children of that person. In our base of forms, males are not allowed to "have" children. Thus, one rule will be required for males, and one for females:

r_1 = [x,y,z,sex:<military_position:u>] → [x,{}]
r_2 = [x,y,z,sex:<female:[u,w]>] → [x,u]

The "rewrite expression", rew(r_1, r_2), will define a query on the database. □

We now formally present the rewrite rules and rewrite expressions. Other motivating examples are given afterwards.

Definition: Let X be a term of type S. *Rewrite rules* and *rewrite expressions* from X are

defined recursively in the following way:

(i) if y is a variable of structure t w.r.t. X and S,
(i-a) $X \rightarrow P{:}y$ is a rule from S to P:t,
(i-b) if ρ is a rewrite expression from Q:t to T' (and hence, t has set structure), $X \rightarrow \rho(y)$ is a rule from S to T',

(ii) (ii-a) if $X \rightarrow Y_1,...,X \rightarrow Y_n$ are rules from S to $T_1,...,T_n$, respectively, then $X \rightarrow P{:}[Y_1,...,Y_n]$ is a rule from S to $P{:}[T_1,...,T_n]$,
(ii-b) if $X \rightarrow Y$ is a rule from S to T_i for some i in [1..n], then $X \rightarrow P{:}\langle Y \rangle$ is a rule from S to $P{:}\langle T_1;...;T_n \rangle$,
(ii-c) if $X \rightarrow Y_1,...,X \rightarrow Y_n$ are rules from S to T', then $X \rightarrow P{:}\{Y_1,...,Y_n\}$ is a rule from S to P:{T'},
(ii-d) if $X \rightarrow Y_1,...,X \rightarrow Y_n$ are rules from S to T, and T is a set type, then $X \rightarrow Y_1 \cup ... \cup Y_n$ is a rule from S to T.

A rewrite expression from S=P:{S'} to T=Q:{T'} is an expression of the form Q:rew(Δ) where Δ is a set of rules from S' to T'.

The previous definition enforces several restrictions on the form of rewrite expressions. For instance, consider the type S = P:[A, B:{C}]. Then the expression A:{C:y} → E:y is not a rule from S since the left-handside does not correspond to S. On the other hand, the expression rew(r,r') where

$$r = P{:}[A{:}x, B{:}y] \rightarrow C{:}x$$
$$r' = P{:}[A{:}x, B{:}y] \rightarrow C{:}y$$

is not a rewrite expression from S since the right-handsides of the two rules conflict; one mappingg to $C{:}dom_1$, and the other to $C{:}\{dom_1\}$. Roughly speaking, if a set of rules yields a rewrite expression then (1) their left-handside can be "unified"; and (2) their right-handside can be "unified".

Suppose now that $X \rightarrow Y$ is a rewrite rule, and P:rew(Δ)(z) occurs nested in Y. Then for each rule $Z \rightarrow W$ in Δ, each unconstructed variable occurring in W necessarily occurs in Z.

The semantics of rewrite rules and expressions is given by the following definition:

Definition: An *assignment* is a partial mapping α from var to val (i.e., α maps variables into

values). The mapping is extended to terms, and rewrite expressions in the following way:

(0) $\alpha(O) = O$ for each object O,

(1) $\alpha(A{:}x) = A{:}\alpha(x)$ for each A and unconstructed variable x,

(2) $\alpha(P{:}[X_1,...,X_n]) = P{:}[\alpha(X_1),...,\alpha(X_n)]$,

(3) $\alpha(P{:}<X>) = P{:}<\alpha(X)>$,

(4) $\alpha(P{:}\{X_1,...,X_n\}) = P{:}\{\alpha(X_1),...,\alpha(X_n)\}$,

(5)[5] $\alpha(X_1 \cup ... \cup X_n) = \alpha(X_1) \cup ... \cup \alpha(X_n)$,

(6) $\alpha(P{:}rew(\Delta)(x)) = P{:}\{\alpha(V) \mid U \rightarrow V \in \Delta$, and $\alpha(U) \in \alpha(x)\}$,

Now we have:

Definition: Let ρ = Q:rew(Δ) from S, and O=P:o be an object of type S. Then the *effect* of ρ on O is defined by

$\rho(O) = Q{:}\{\beta(V) \mid$ for some β, and $U \rightarrow V$ in Δ, $\beta(U)$ in o$\}$.

We now present four examples of rewrite expressions. The first example illustrates the use of several rules to obtain different treatment of alternative structures resulting from the union of types constructor.

Example 3.3: Consider Form1 and Form2 of Section 2. It can be shown that a lossless mapping from sets of forms 1 into sets of forms 2 is obtained using the rewrite operation rew($\{r_1,r_2,r_3\}$) where

$r_1 = [x,y,z,<MP{:}w>] \rightarrow$

$$[x,y,z,<MP{:}w>,<\mathbf{1}_{no}>, \{\},<\mathbf{1}_{NA}>,<\mathbf{1}_{male}>]$$

$r_2 = [x,y,z,<female{:}[v,<\mathbf{1}_{NA}>]>] \rightarrow$

$$[x,y,z,<\mathbf{1}_{NA}>,<\mathbf{1}_{no}>, v,<\mathbf{1}_{NA}>,<\mathbf{1}_{female}>]$$

$r_3 = [x,y,z,<female{:}[v,<maiden\text{-}name{:}u>]>] \rightarrow$

$$[x,y,z,<\mathbf{1}_{NA}>,<\mathbf{1}_{yes}>, v,<u>,<\mathbf{1}_{female}>]$$

[5] Let $P{:}o_1$ and $P{:}o_2$ be two objects of type A:{T}. Then the union of $P{:}o_1$ and $P{:}o_2$ is defined in the obvious manner, i.e., $P{:}o_1 \cup P{:}o_2 = P{:}(o_1 \cup o_2)$.

The second example presents the utilization of union of rewrites.

Example 3.4: Consider the following two types:

pers:{R:[emp, male:{name}, female{name}]}

pers:{R:[emp, children:{ child:[name, sex:<$\perp_{male}$;$\perp_{female}$>]}]}

which might be used to store each employee along with his/her children differentiated by sex. We now give a rewrite expression which maps objects of the first type into objects of the second in the natural manner.

ρ_{male}= rew(name:w → [name:w, sex:<$\perp_{male}$>])

ρ_{female}= rew(name:w → [name:w, sex:<$\perp_{female}$>])

ρ= rew([emp:x, male:y, female:z] → [emp:x, children:ρ_{male}(y) ∪ ρ_{female}(z)])

The third example presents the use of rewrite rules in a nested way.

Example 3.5: Consider the following two types:

company:{ dept:[dname, pers:{R:[emp, male:{name}, female{name}]} }

comp:{ dept:[dname, pers:{R:[emp,children:{child:[name,sex:<$\perp_{male}$;$\perp_{female}$>]}]}

which might be used to store, for each department, each employee along with his/her children, differentiated by sex. We now give a rewrite expression which maps objects of the first type into objects of the second in the natural manner. (The corresponding mapping is data preserving.)

rew([dname:u, pers:v] → [dname:u, pers:ρ(v)])

where ρ is as in the previous example.

The last example illustrates the use of composition of rewrite expressions.

Example 3.6: Consider a type consisting of sets of Form1 as in Section 2. Suppose that we want to distinguish "mothers", i.e., females with at least one child. Then this can be computed using $\rho \circ \rho'$ where

ρ= rew([x,y,x,<female:[z,v]>] → [x,ρ"(z)])

for ρ"= rew(child_name:w → $\mathbf{1}_c$), and

ρ'= rew([x,{$\mathbf{1}_c$}] → [x,$\mathbf{1}_y$], [x,{}] → [x,$\mathbf{1}_n$]).

We now turn to a result showing how rewrite expressions can be put into a certain normal form called "decomposed". In particular, we show that for each rewrite expression

P:rew(Δ), there is an equivalent rewrite expression P:rew(Δ') such that all rewrite rules in Δ' are decomposed.

The notion of decomposed has the following particularly simple definition:

Definition: A term X is *decomposed* if each unconstructed variable occurring in X is either

(a) atomic; or

(b) a set variable whose structure is not $\{\mathbf{1}_f\}$ for any f.

A rewrite rule X $\rightarrow$ Y (or expression ρ) is *decomposed* if the premise of each rewrite rule occurring (possibly nested) in X $\rightarrow$ Y (or ρ, resp.) is decomposed[6].

Speaking roughly, a rewrite rule X $\rightarrow$ Y is decomposed if the term X (and all premises of nested rewrite rules in Y) makes explicit choices concerning which branch of each +-node to take. To articulate this more formally, we introduce two important notions. The first allows us to focus on the part of a type T lying "above" its $*$-nodes. (See Figure 3.2.)

Definition: The *$*$-frontier* of a type T is the set of nodes of T containing all leaves, and $*$-nodes of T which are not (proper) descendants of a $*$-node.

The second provides a natural partitioning of the domain of T to "choices" made at the +-nodes above the $*$-frontier, and at nodes with underlying structure $\{\mathbf{1}\}$. (See Figure 3.3.)

Definition: For a type T, a *choice tree* of T is a (partially labelled) subtree T' of T such that:

(i) each +-node occurring in T' above the $*$-frontier of T has exactly one child in T';

(ii) each $\times$-node p occurring in T' above the $*$-frontier of T has as children in T' all of the children that p has in T;

(iii) for each $*$-node p in T', the subtree in T' below p is precisely the subtree of T below p;

(iv) each $*$-node with a $\mathbf{1}$-child is labelled in T' by either $\{\}$ or $\{\mathbf{1}\}$ (indicating that the value associated with this $*$-node is $\{\}$ or $\{\mathbf{1}\}$, respectively).

The *domain* of a choice tree T' of T, denoted obj(T'), is the collection of objects O $\in$ obj(T) whose internal structure corresponds to T'.

[6] Note that if a rewrite expression ρ is decomposed, then each unconstructed variable occurring in the consequent of any (possibly nested) rewrite rule in ρ is atomic or a set variable whose structure is not $\{\mathbf{1}_f\}$ for any f.

It is clear that if $T_1,...,T_n$ are the choice trees of T, then the collection $\{\mathbf{obj}(T_1),...,\mathbf{obj}(T_n)\}$ is a partition of $\mathbf{obj}(T)$ into disjoint, non-empty sets. To illustrate this last remark and the previous definitions, consider the type:

$$T = A:<B:[C,D:\{D'\},E];F:<H;I:[J:<K:\{K':<K_1,K_2>\};L>,M:<N:\{\mathbf{1}_e\};O>]>>.$$

One can show that the $*$-frontier of T is {C,D,E,H,K,L,N,O}. (See Figure 3.2). The choice trees of T are:

$$A:<B:[C,D:\{D'\},E]>;$$

$$A:<F:<H>>;$$

$$A:<F:<I:[J:<K:\{<K_1,K_2>\}>,M:<N:\{\mathbf{1}_e\}>]>>, \text{ N marked by } \{\};$$

$$A:<F:<I:[J:<K:\{<K_1,K_2>\}>,M:<N:\{\mathbf{1}_e\}>]>>, \text{ N marked by } \{\mathbf{1}_e\};$$

$$A:<F:<I:[J:<K:\{<K_1,K_2>\}>,M:<O>]>>;$$

$$A:<F:<I:[J:<L>,M:<N:\{\mathbf{1}_e\}>]>>, \text{ N marked by } \{\};$$

$$A:<F:<I:[J:<L>,M:<N:\{\mathbf{1}_e\}>]>>, \text{ N marked by } \{\mathbf{1}_e\}; \text{ and}$$

$$A:<F:<I:[J:<L>,M:<O>]>>.$$

(One of these choice trees is shown in Figure 3.3.) Clearly, their corresponding domains are disjoint.

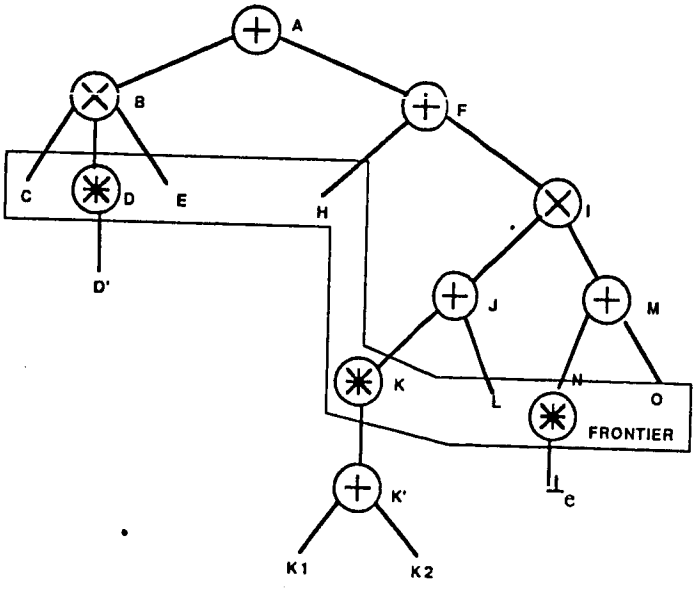

Figure 3.2: frontier

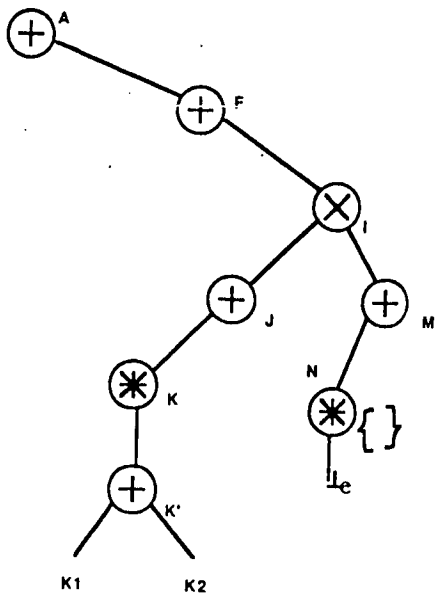

Figure 3.3: a choice tree

The following result is easily verified:

Lemma 3.6: If a term X over S is decomposed, then there is some choice tree S' of S such that $\{\alpha(X) \mid \alpha \text{ an assignment}\} \cap \mathbf{obj}(S) \subseteq \mathbf{obj}(S')$. □

We now have:

Lemma 3.7 (**Decomposition Lemma**): If $X \rightarrow Y$ is a rewrite rule from S to T then there is an $n \geq 1$ and a set $\{X_i \rightarrow Y_i \mid i \in [1..n]\}$ of rewrite rules from S to T such that

(i) each rule $X_i \rightarrow Y_i$ is decomposed;

(ii) for each object $O' \in \mathbf{obj}(P':\{S\})$, $\text{rew}(X \rightarrow Y)(O') = \text{rew}(\{X_i \rightarrow Y_i \mid i \in [1..n]\})(O')$; and

(iii) for each $i \in [1..n]$, there is a distinct choice tree S_i of S such that[7] $\{\alpha(X_i) \mid \alpha \text{ an assignment}\} \cap \mathbf{obj}(S) \subseteq \mathbf{obj}(S_i)$.

Thus, each rewrite expression is equivalent to a decomposed rewrite expression.

Proof: Let $X \rightarrow Y$ be a rewrite rule from S to T. We can assume inductively that each rule occurring within a rewrite expression of Y is decomposed, and that the collection of variables occurring within nested rewrite rules of Y is disjoint from the collection of variables occurring at the outermost level of X or Y. We now present an iterative construction for replacing unconstructed variables in X which violate the definition of decomposed by less complex unconstructed variables (or possibly $\{\}$ or $\{\mathbf{1}_f:\mathbf{1}\}$). In this construction, we begin with the set $\{X \rightarrow Y\}$ and generate sets Δ_1, Δ_2,.... These will have the property that $\text{rew}(\Delta_k)(O') = \text{rew}(X \rightarrow Y)(O')$ for each object $O' \in \mathbf{obj}(P':\{S\})$. If at some step some rule in Δ_k is not decomposed, we replace that rule by a set of rules according to the construction now described.

Suppose that $U \rightarrow V$ is a rewrite rule in Δ_k from S to T, and that w is a variable occurring in U which violates the definition of decomposed (i.e., w is unconstructed and: either it is of structure $\{\mathbf{1}_f\}$ for some f, or it is not atomic nor of set type.) Three cases arise:

a) Suppose w is a variable of structure $[P_1:s_1,...,P_n:s_n]$. In this case, let w_i be an unconstructed variable of structure s_i for $i \in [1..n]$, and form $U_1 \rightarrow V_1$ from $U \rightarrow V$ by replacing all

occurrances of w by $[P_1{:}w_1,...,P_n{:}w_n]$. It is easily verified that for each object $O \in obj(S)$, there is an assignment α such that $\alpha(U) = O$ iff there is an assignment α_1 such that $\alpha_1(U_1) = O$; and in this case $\alpha(V) = \alpha_1(V_1)$. It follows that for each $O' \in obj(P':\{S\})$, $rew(U \rightarrow V)(O') = rew(U_1 \rightarrow V_1)(O')$. Letting $\Delta_{k+1} = \Delta_k - \{U \rightarrow V\} \cup \{U_1 \rightarrow V_1\}$, we now have $rew(\Delta_k)(O) = rew(\Delta_{k+1})(O)$.

b) If w is of structure $<P_1{:}s_1;...;P_n{:}s_n>$ then let w_i be an unconstructed variable of structure s_i for $i \in [1..n]$, and construct rules $U_i \rightarrow V_i$, $i \in [1..n]$, from $U \rightarrow V$ by replacing each occurrance of w by $<P_i{:}w_i>$. It is clear from the construction of the U_i's that the families $\{\alpha(U_i) \mid \alpha$ an assignment$\}$ are disjoint. Also, it is easily verified that for each object $O \in obj(S)$, there is an assignment α such that $\alpha(U) = O$ iff there is some i and some assignment α_i such that $\alpha_i(U_i) = O$; and in this case $\alpha(V) = \alpha_i(V_i)$. It follows that for each $O' \in obj(P':\{S\})$, $rew(U \rightarrow V)(O') = rew(\{U_i \rightarrow V_i \mid i \in [1..n]\})(O')$. Letting $\Delta_{k+1} = \Delta_k - \{U \rightarrow V\} \cup \{U_i \rightarrow V_i \mid i \in [1..n]\}$, we have $rew(\Delta_k)(O') = rew(\Delta_{k+1})(O')$.

c) If w is of structure $\{\mathbf{1}_f\}$ for some attribute $\mathbf{1}_f$, let $w_1 = \{\}$ and $w_2 = \{\mathbf{1}_f{:}\mathbf{1}\}$. As in part (b), U_i for i = 1 or 2 is constructed from U by replacing all occurrances of w by w_i. If w does not occur as the variable of a rewrite expression of V, then V_1 and V_2 can be constructed as in part (b). Suppose now that w does occur as the variable of the rewrite expression $Q{:}rew(\Sigma)(w)$, where $\Sigma = \{X_j \rightarrow Y_j \mid j \in [1..m]\}$. Then X_j must be $\mathbf{1}_f{:}\mathbf{1}$ or $\mathbf{1}_f{:}z$ for each $j \in [1..k]$, and Y_j is a constant or a term whose only variable is z for each $j \in [1..m]$. In this case, when constructing V_1 replace $Q{:}rew(\Sigma)(w)$ by $Q{:}\{\}$; and when constructing V_2 replace $Q{:}rew(\Sigma)(w)$ by $Q{:}\{Y'_1,...,Y'_m\}$ where Y'_i is the result of replacing the variable in Y_i (if any) by **1**. It is easily verified that setting $\Delta_{k+1} = \Delta_k - \{U \rightarrow V\} \cup \{U_1 \rightarrow V_1, U_2 \rightarrow V_2\}$ satisfies the conditions of the construction.

It is clear from the construction that in the final family $\Delta = \{X_i \rightarrow Y_i \mid i \in [1..n]\}$, each rule is decomposed. Thus (i) holds. It is also clear that for each $O' \in obj(P{:}\{S\})$, $rew(X \rightarrow Y)(O') = rew(\Delta)(O')$. Thus (ii) holds. Parts (b) and (c) ensure that if $i \neq j$ then $\{\alpha(X_i) \mid \alpha$ an assignment$\} \cap \{\alpha(X_j) \mid \alpha$ an assignment$\} = \emptyset$. Since each X_i is decomposed, this and Lemma 3.6 implies that for each i, there is a distinct choice tree S_i such that $\{\alpha(X_i) \mid \alpha$ an

assignment} $\cap$ obj(S) $\subseteq$ obj(S_i). Hence (iii) also holds. The second sentence now follows easily from the definition of effect of rewrite expressions. □

At first glance, it appears in the above construction that for each i, there is a (not necessarily distinct) choice tree T_i of T such that $\{\alpha(Y_i) \mid \alpha \text{ an assignment}\} \cap \mathbf{obj}(T) \subseteq \mathbf{obj}(T_i)$. The following example illustrates that this need not be the case.

Example 3.8: Let S = P:{Q:<A;B>} and T = R:{$\mathbf{1}_f$}, and consider the rewrite rule

$$P{:}w \rightarrow R{:}\mathrm{rew}(Q{:}\langle A{:}x\rangle \rightarrow \mathbf{1}_f{:}\mathbf{1})(w).$$

On a given object O of type S, this rule yields R:{$\mathbf{1}_f$:**1**} if there is an object of type A in O, and yields R:{} otherwise. This rule is decomposed, but there is no choice tree of T which contains $\{\alpha(Y) \mid \alpha \text{ an assignment}\} \cap \mathbf{obj}(T)$. □

To conclude this section, we make two briefs remarks on (a) the design of a general query language for objects, and (2) the omission of tokens in terms.

- A calculus in the style of [J,ABe,RKS] can easily be designed for typed objects. An algebra in the spirit of [ABe] can be obtained by adding to the rewrite operations binary operations like union, intersection, difference, and cross product; and unary ones like power set, and set collapse. To obtain the power of the calculus with this algebra, we believe that *dynamic constants* such as in [SS] or [ABe] should be used. This would involve the possibility of using in the rules of an embedded rewrite operations variables from the outer levels.
- Some unexpected power comes from allowing the omission of tokens when they are implicit from the context. Consider the two types:

  $P{:}[\ P_1{:}\langle A_1;B_1\rangle,\ldots,P_n{:}\langle A_n;B_n\rangle\ ]$, and

  $Q{:}\langle\ Q_1{:}[A_1,A_2,\ldots,A_{n-1},A_n];$

  $Q_2{:}[B_1,A_2,\ldots,A_{n-1},A_n];$

  ...

  $Q_{2^n}{:}[B_1,B_2,\ldots,B_{n-1},B_n]\ \rangle$

  where n is some positive integer. Then objects of type P can be rewritten into objects of type Q using the rule

[7] if unconstructed variables were strictly typed, the condition would simply be $\{\alpha(X_i) \mid \alpha \text{ an assignment}\} \subseteq \mathbf{obj}(S_i)$.

$$[\ \langle x_1\rangle,\ldots,\langle x_n\rangle\ ] \rightarrow\ \langle [x_1,\ldots,x_n]\rangle.$$

However, a precise expression of this transformation would require 2^n different rules.

4. COMPOSITION OF SIMPLE REWRITE OPERATIONS

In this section, we introduce the "simple" rewrite operations. These operations will turn out to be central to our study of restructuring in the next two sections. We show that simple rewrite operations are closed under composition.

Speaking intuitively, the premise of an arbitrary rewrite rule acts as a filter, discriminating between objects which match the pattern of the premise and those that don't. In a simple rewrite expression, premises have very limited ability to discriminate between objects. Specifically, the most refined test that such premises can make is whether an object corresponds to a given choice tree or not. This intuition is realized formally by restricting the premises in a syntactic way, e.g., requiring that no repeated variables occur, and that no constants other than **1** occur.

It is convenient in this section to focus on individual rewrite rules, rather than rewrite expressions. Before providing a formal basis for this, we make a few intuitive remarks.

Suppose that $X \rightarrow Y$ is a rewrite rule from S to T (where S need not be a set type). Speaking intuitively, if $O \in \mathbf{obj}(S)$ and α is an assignment such that $\alpha(X) = O$, then $X \rightarrow Y$ associates $\alpha(Y)$ with O. If X is arbitrary, there may be more than one α such that $\alpha(X) = O$, in which case more than one value $\alpha(Y)$ is associated with O. As we shall see, this never occurs for simple rewrite rules, and so, each simple rewrite rule will define a (partial) single-valued function.

For technical reasons, it is convenient to include special rules of the form $X \rightarrow \Omega$, where Ω indicates the undefined value. This is needed to ensure that simple rewrite rules are closed under composition. For example, consider the rule $A{:}w \rightarrow Q{:}\langle A{:}w\rangle$ mapping $A{:}dom_1$ to $Q{:}\langle A{:}dom_1;B{:}dom_1\rangle$; and the rule $Q{:}\langle B{:}y\rangle \rightarrow B{:}y$ mapping $Q{:}\langle A{:}dom_1;B{:}dom_1\rangle$ to $B{:}dom_1$. Then, the composition of these two rules is $A{:}w \rightarrow \Omega$, which always yields the undefined value. Formally we have:

Definition: A *(generalized) rewrite rule* is defined as in Section 3, except that the following is added to part (i):

(i-c) $X \rightarrow \Omega$ is a rule from S to T, for any type T;
and each expression $Y, Y_1, \ldots, Y_n$ occurring in part (ii) of that definition is not permitted to be Ω.

At the level of rewrite expressions, rules $X \rightarrow \Omega$ do not affect results. More specifically,

Definition: Let ρ = P:rew(Δ) from S, and O be an object of type S. Then the *effect* of ρ on O is defined by

$$\{\beta(V) \mid \beta \text{ an assignment, } U \rightarrow V \text{ in } \Delta,\ V \neq \Omega, \text{ and } \beta(U) \text{ in } O\}.$$

We now define the family of simple rewrite rules and expressions:

Definition: A *simple* rewrite rule is a rewrite rule $X \rightarrow Y$ (where Y may be Ω) such that

(i) The only explicit set construction in the premise of any (possibly nested) rewrite rule is over a type of form P:$\{\mathbf{1}_f\}$;

(ii) No constants appear in the premise of any rewrite rule, except possibly $\mathbf{1}$;

(iii) There are no repeated variables in the premise of any rewrite rule; and

(iv) If $U \rightarrow \mathbf{1}_f:\mathbf{1}$ occurs in a (possibly nested) rewrite rule for some $\mathbf{1}_f$, then U is of type $\mathbf{1}_g$ for some $\mathbf{1}_g$.

A rewrite expression ρ is *simple* if each rewrite rule occuring in ρ is simple.

Because simple rewrite rules do not permit non-trivial explicit set construction in the premises, it is easily verified that

Lemma 4.1: If $X \rightarrow Y$ is simple from S to T, and if $O \in \mathbf{obj}(S)$, then there is at most one assignment α such that $\alpha(X) = O$. □

This permits:

Definition: Let $X \rightarrow Y$ be a simple rewrite rule from S to T. Then $[X \rightarrow Y]$ denotes the partial function from $\mathbf{obj}(S)$ to $\mathbf{obj}(T)$ where, for $O \in \mathbf{obj}(S)$, $[X \rightarrow Y](O) = \alpha(Y)$ if $Y \neq \Omega$ and there is some assignment α such that $\alpha(X) = O$; and $[X \rightarrow Y](O)$ is undefined otherwise.

If two expressions f and g denote the same partial function (i.e., for each object $O \in$ obj(S), either both f(O) and g(O) are defined and f(O) = g(O), or both are undefined), we say that f is *equivalent* to g, denoted $f \equiv g$.

We show that simple rewrite functions are closed under composition in two stages, first considering simple rewrite rules, and then simple rewrite expressions. (Examples are given in Appendix B which show that each restriction in the definition of simple is needed in order to ensure that simple rules are composable.) The next result deals with the composition of *decomposed, simple* rewrite rules. Its proof is rather involved, and relegated to Appendix A.

Lemma 4.2: If $W \rightarrow X$ from S to T and $Y \rightarrow Z$ from T to U are decomposed, simple rewrite rules, then there is a simple rewrite rule $W \rightarrow \hat{Z}$ such that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

Using this, we now have:

Proposition 4.3: Let ρ_i be a simple rewrite operation from R_i to R_{i+1} for each i in [1..n]. Then there exists a simple rewrite operation ρ such that $\rho_1 \circ \ldots \circ \rho_n = \rho$.

Proof: Clearly it suffices to show that there is a simple rewrite function ρ such that $\rho_1 \circ \rho_2 = \rho$. Suppose now that ρ_1 = Q:rew(Δ) maps P:{S} to Q:{T}, and ρ_2 = R:rew(Σ) maps Q:{T} to R:{U}. It is easily verified that if the construction of the proof of the Decomposition Lemma is applied to a simple rewrite rule, then it yields a set of simple rewrite rules. Thus, without loss of generality we may assume that both Δ and Σ contain only decomposed, simple rewrite rules.

Let $\Delta = \{W_i \rightarrow X_i \mid i \in [1..n]\}$ and $\Sigma = \{Y_j \rightarrow Z_j \mid j \in [1..m]\}$. By Lemma 4.2, there are expressions $\hat{Z}_{i,j}$ such that $W_i \rightarrow \hat{Z}_{i,j}$ is a simple rewrite rule and $[W_i \rightarrow \hat{Z}_{i,j}] \equiv [W_i \rightarrow X_i] \circ [Y_j \rightarrow Z_j]$ for $i \in [1..n]$ and $j \in [1..m]$. Let $\Gamma = \{W_i \rightarrow \hat{Z}_{i,j} \mid i \in [1..n], j \in [1..m]$ and $\hat{Z}_{i,j} \neq \Omega\}$. It easily follows that R:rew(Γ) is a simple rewrite expression from P:{S} to R:{U} such that R:rew(Γ) = (Q:rew(Δ)) $\circ$ (R:rew(Σ)). □

It should be clear from the definition of rewrite operations that each rewrite operation can be computed efficiently using parallelism. A consequence of the above proposition is that sequences of simple rewrite operations can also take full advantage of parallelism.

5. EQUIVALENCE PRESERVING TRANSFORMATIONS

The last two sections of this paper focus on restructuring of types. In particular, natural local transformations on types preserving data capacity (Section 5), and augmenting it (Section 6) are introduced. A fundamental result in this section (Theorem 5.3) states that the equivalence preserving transformations are "complete" in a formal sense. In both sections, the semantics of transformations are expressed using "simple" rewrite operations.

As noted in the introduction, data "relativism" refers to the phenomenon that two database schemas may hold essentially the same data. This arises in the important areas of user view definition, schema evolution, and schema translation. Previous work on data relativism [ABi,AABM,HM,H,HY,MB] suggests that an intuitively appealing formalism for comparing the data capacity of two structures can be based on local structural manipulations. This is substantiated in particular by results in [HY], which show that a family of 6 transformations and their inverses are "complete" for proving equivalence of information capacity between types (for which all domains are infinite). The results of this section generalize these results to include one-element domains, and relate them to the simple rewrite operations. The augmentations of the next section appear to provide a natural generalization for these transformations to increase the data capacity of types.

Before embarking on the formal development, we present a simple example which indicates how local manipulations might be applied in the context of database schema evolution. The example involves the two related types shown in Figure 5.1, which might be used to represent family units in some culture. Assume for a moment that in this culture, a family unit consists of either an adult female, or a married couple. (Unmarried adult males in this

Figure 5.1: The Polyandry Example

culture have no "legal" status). The type shown in part (a) can represent family units under this assumption: objects of this type consist in either a female or an ordered pair, with first coordinate a female and second coordinate a male. Suppose now that a new law has been enacted within this culture, which allows women to take more than one husband. Then the type in part (b) can be used to represent family unit. It is clear in this case that existing data stored in the structure of (a) can be translated into the structure of (b). This raises the question of whether the type (a) can be transformed into the type (b) using a sequence of capacity preserving and augmenting manipulations. As shown in Figure 5.2, the answer to this question is affirmative. (Note that the first four transformations here preserve data capacity.) Furthermore, as implied by Theorem 6.3, the corresponding mapping on objects is realized by a single simple rewrite operation.

We now define nine structural transformations on types. As we shall see, these transformations preserve the data capacity of types. The transformations are presented in five groups; the first three of these are essentially trivial, while the latter two are more provocative. The transformations are first defined as they occur at the root of a type, and then generalized to permit their occurrence at an arbitrary node of a type. Three simple examples of these transformations are shown in Figure 5.3 below.

Definition: The *capacity preserving transformations (cp-transformations)* are as follows[8]:

Renaming cp-transformations:

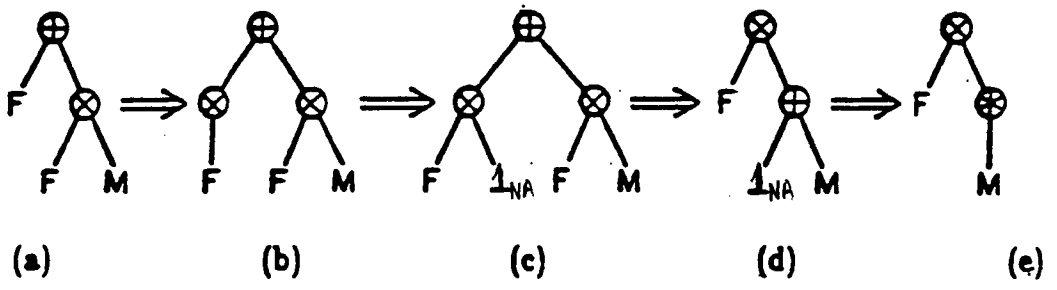

Figure 5.2: Structural transformation of a type which utilizes **1** types

[8] The result of a cp-transformation must be a type: in particular, siblings of a given node must have distinct tokens. We do not consider here this detail.

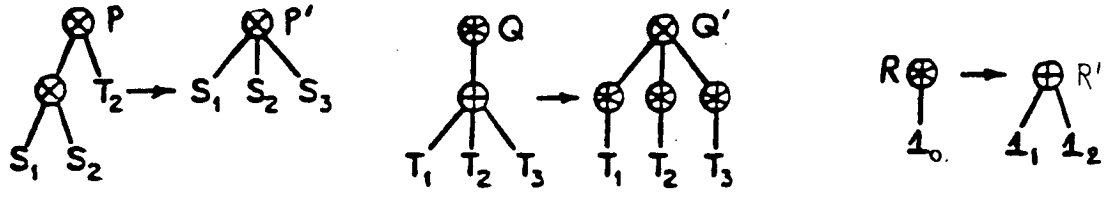

Figure 5.3: Examples of cp-transformations

(i) B:t is obtained from A:t by renaming.

Simple × cp-transformations:

(ii) replace $P:[T_1,...,T_{i-1},Q:[S_1,...,S_m],T_{i+1},...,T_n]$ by
$P:[T_1,...,T_{i-1},S_1,...,S_m,T_{i+1},...,T_n]$;

(iii) replace $Q:[T]$ by T;

Simple + cp-transformations:

(iv) replace $P:<T_1;...;T_{i+1};Q:<S_1;...;S_m>;T_{i+1};...;T_n>$ by
$P:<T_1;...;T_{i-1};S_1;...;S_m;T_{k+1};...;T_n>$;

(v) replace $Q:<T>$ by T;

Rising + cp-transformations:

(vi) replace $P:[T_1,...,T_{i-1},Q:<S_1,...,S_m>,T_{i+1},...,T_n]$ by
$P:<Q_1:[T_1,...,T_{i-1},S_1,T_{i+1},...,T_n];...;Q_m[T_1,...,T_{i-1},S_m,T_{i+1},...,T_n]>$;

(vii) replace $P:\{<T_1;...;T_n>\}$ by $P:[Q_1:\{T_1\},...,Q_n:\{T_n\}]$;

cp-transformations with **1**:

(viii) replace $P:[T_1,...,T_{i-1},\mathbf{1}_f:\mathbf{1},T_{i+1},...,T_n]$ by
$P:[T_1,...,T_{i-1},T_{i+1},...,T_n]$;

(ix) replace P:{$\mathbf{1}_f$:**1**} by P:<$\mathbf{1}_1$:**1**;$\mathbf{1}_2$:**1**>;

Suppose now that S is a type with a node p and the type R below it, that R' is constructed from R using a cp-transformation, and that S' is constructed from S by replacing R by R' at p. Then S' is the *result* of applying an cp-transformation to S, written S → S'. The reflexive, transitive closure of → is denoted by →*. □

With each of these transformations, one can associate a one-one, onto function from objects to objects. The corresponding functions are called *restructuring functions*. The semantics of most restructuring functions should be obvious. For instance,

- transformation (vi) maps an object P:[O_1,Q:<R_3:O'>] to the object P:<Q_3:[O_1,R_3:O']>; and
- transformation (ix) maps the object P:{} of type P:{$\mathbf{1}_f$:**1**} to P:<$\mathbf{1}_1$:**1**>, and the object P:{$\mathbf{1}_f$:**1**} to P:<$\mathbf{1}_2$:**1**>.

Most of the restructuring functions are realized by a simple rewrite rule, as illustrated by parts (a) and (b) of the next example. In the cases of transformations (iv), (vi) and (ix), a family of simple rewrite rules is needed, as illustrated in part (c) of the example. In these cases, the family of rewrite rules is "consistent", in the sense that they operate on the domains of distinct choice trees. If an cp-transformation occurs within a type with *-root, then a single simple rewrite operation realizes the restructuring function.

Example 5.1: Consider the cp-transformations of Figure 5.3. They are realized by the following rewrite rules.

(a) [[x,y],z] → [x,y,z];

(b) x → [ρ_1(x), ρ_2(x), ρ_3(x)] where
Q_i is the root token of T_i for each i and
ρ_1= rew(<Q_1:v>→v),
ρ_2= rew(<Q_2:v>→v), and
ρ_3= rew(<Q_3:v>→v); and

(c) {}→$\mathbf{1}_1$:**1**, and {**1**}→$\mathbf{1}_2$:**1**.

□

The straightforward proof of the following result is omitted.

Theorem 5.2: Let R and S be two set types such that R $\rightarrow$ S (resp., S $\rightarrow$ R), then there is a simple rewrite operation ρ which is a bijection from dom(R) to dom(S). Furthermore ρ defines the same mapping as the restructuring function from R to S (resp., the inverse of S $\rightarrow$ R). $\square$

The next result of this section implies that the application of cp-transformations is essentially Church-Rosser, transforming each type into a "normal-form" type which is unique up to relabelling of nodes. For this, we need two definitions: the first defines the normal form for types; and the second allows us to "ignore" the internal names in types.

Definition: A type S is in *normal form* if:

a. there is at most one +-node in S, in which case the root is the +-node and it has more than one child;

b. $\mathbf{1}_f$ is not a child of any $*$-node or $\times$-node for any $\mathbf{1}_f$;

c. no child of a $\times$-node is a $\times$-node; and

d. each $\times$-node has more then one child.

Definition: Two normal form types S and T are *isomorphic* (up to renaming), denoted S $\equiv$ T, if S can be transformed to T using only renamings.

The next result states that the application of the cp-transformations is essentially Church-Rosser. (This generalizes results of [HY]; a proof is sketched in Appendix C.)

Theorem 5.3: Let S be a type. Then there is a type T in normal form such that S $\rightarrow^*$ T. Furthermore, if T_1 and T_2 are normal form types such that S $\rightarrow^*$ T_1 and S $\rightarrow^*$ T_2, then T_1 $\equiv$ T_2. Also, if R is a type such that S $\rightarrow^*$ R, then R $\rightarrow^*$ T. $\square$

We now present two notions for comparing data capacity of database structures that were introduced by [HY], namely absolute dominance and equivalence. We will use these concepts to prove the "completeness" of our cp-transformations, thereby generalizing results of [HY] for formats.

Intuitively, absolute dominance roughly captures the intuition that natural database transformations should not "invent" data values. (See [H1] for more motivation.) To formally define this notion, we first define the "active domain" of objects.

Definition: If O is an object, the *active domain* of O, denoted by act(O), is the set of all atomic elements occuring in O. Also, if T is a type and X is a set of atomic elements, then $\mathbf{obj}_X(T)$ denotes $\{O \in \mathbf{obj}(T) \mid act(O) \subseteq X\}$.

We now have:

Definition: Let S and T be types. Then

- S is *dominated by* T *absolutely,* denoted $S \leq T$ (abs), if there is some k such that for each finite set X satisfying[9] $|X \cap dom_i| \geq k$ for each dom_i appearing in S or T, $|\mathbf{obj}_X(S)| \leq |\mathbf{obj}_X(T)|$; and
- S is *absolutely equivalent* to T, denoted $S \sim T$ (abs), if $S \leq T$ (abs) and $T \leq S$ (abs).

As a simple example, we note that $P:[A:dom_1,B:dom_1] \leq Q:\{A:dom_1\}$ (abs).

We now present a characterization of absolute equivalence between types which demonstrates that (i) the collection of the nine cp-transformations and their inverses is "complete" for absolute equivalence (Theorem 5.4), and (as a result) (ii) virtually any natural notion of equivalent data capacity is identical to it (Corollary 5.5). These generalize results of [HY], and their proofs are sketched in Appendix C.

Theorem 5.4: Let S_1 and S_2 be two types. Then $S_1 \sim S_2$ (abs) iff there is some normal form type T such that $S_1 \rightarrow^* T$ and $S_2 \rightarrow^* T$.

It follows from Theorems 5.3 and 5.4 that rewrite operations are "complete" for absolute equivalence. Also, it is decidable whether two types are absolutely equivalent (although it appears that testing this is co-NP). Perhaps the most important implication of Theorem 5.4 is the following, which implies that essentially all notions of capacity equivalence for types are identical to absolute equivalence.

[9] This condition is included to prevent certain combinatorial technicalities from having an impact.

Corollary 5.5: Let xxx-dominance be any reflexive, transitive binary relation on types such that

a. if $S \rightarrow T$ or $T \rightarrow S$, then $S \leq T$ (xxx), and

b. if $S \leq T$ (xxx) then $S \leq T$ (abs);

and let xxx-equivalence be defined from xxx-dominance in the natural manner. Then $S \sim T$ (xxx) iff $S \sim T$ (abs).

6. STRUCTURAL DOMINANCE

In this section, we define "structural dominance" (and "structural equivalence"), an intuitively appealing notion of dominance between types. This notion is based primarily on the transformations introduced in the previous section, together with three new transformations which augment data capacity. We prove that the new transformations also correspond to simple rewrite rules in a natural manner. We conclude (Theorem 6.2) that any finite sequence of transformations corresponds to a single simple rewrite operation.

To begin the formal development, we define the three data augmenting transformations, here called "augmentations", and use them to define the general notion of structural dominance.

Definition: The three *augmentations* on types are defined as follows:

(x) replace $P:<T_1;...;T_n>$ by $P:<T_1;...;T_n;\perp_f>$ for some $\perp_f$.

(xi) replace $P:<\perp_1;...;\perp_n>$ by A.

(xii) replace $P:<T;\perp_f>$ by $P:\{T\}$.

For types S and T, write $S \Longrightarrow T$ if one of the following holds:

(α) $S \rightarrow T$, or $T \rightarrow S$, or

(β) T is the tree obtained from S by replacing a subtree S' of S by one augmentation of S'.

The relation $\Longrightarrow^*$ is the reflexive, transitive closure of $\Longrightarrow$. Finally,

- S is *structurally dominated by* T, denoted $S \leq T$ (struct), if $S \Longrightarrow^* T$; and
- S is *structurally equivalent to* T, denoted $S \sim T$ (struct), if $S \leq T$ (struct), and $T \leq S$ (struct). □

A straightforward application of Theorem 5.4 shows that for each S and T, S $\sim$ T (abs) iff S $\sim$ T (struct).

With each of these transformations, one can associate an *(augmenting) restructuring function.* The semantics of these functions is now briefly presented:

(x) is the identity ;

(xi) maps P:<$\perp_i$:$\perp$> to A:a_i for each i in [1..n], and some distinct values a_1,...,a_n in dom(A); and

(xii) maps P:<O> to P:{O}, and P:<$\perp_f$:$\perp$> to P:{}.

We now state the easily verified result that each restructuring function is realized by a simple rewrite expression:

Theorem 6.1: Let R and S be two set types such that R $\Longrightarrow$ S, then there is a simple one-to-one rewrite operation ρ from obj(R) to obj(S). Furthermore ρ defines the same mapping as the restructuring function from R to S. □

Using Proposition 4.3, it follows that:

Theorem 6.2: Let R and S be two set types. If R $\leq$ S (struct) (i.e., R $\Longrightarrow^*$ S), there exists a one-to-one simple rewrite expression ρ from R to S. □

It remains open whether the converse of this result holds.

APPENDIX A

In this appendix, Lemma 4.2 is proved. The lemma is restated here for the reader's convenience.

Lemma 4.2: If $W \rightarrow X$ from S to T and $Y \rightarrow Z$ from T to U are decomposed, simple rewrite rules, then there is a simple rewrite rule $W \rightarrow \hat{Z}$ such that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

To prove Lemma 4.2, we perform an induction on the "$*$-height" of the type T:

Definition: For a type T, the *$*$-height* of T, denoted ht(T), is the maximum number of $*$-nodes in any branch of T.

For the induction, we assume now that $k \geq 0$ and state the following:

Inductive Assumption: If $W \rightarrow X$ from S to T and $Y \rightarrow Z$ from T to U are decomposed simple rewrite rules and $ht(T) < k$, then there is a decomposed simple rewrite rule $W \rightarrow \hat{Z}$ from S to U such that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

Note that if $k = 0$, then this is a vacuous assumption.

To advance the induction to types T whose $*$-height is k, we prove four lemmas. Speaking roughly, the first three lemmas perform an induction on the structure of X above the $*$-frontier of T, and the last lemma performs an induction on the structure of Z above the $*$-frontier of U. We state all four lemmas first, and then present their proofs.

Lemma A.1: Suppose that $W \rightarrow X$ from S to T and $Y \rightarrow Z$ from T to U are decomposed, simple rewrite rules such that

(a) $ht(T) = k$;

(b) Z is of basic type or a rewrite expression

(c) X is of basic type.

Then there is a variable $\hat{Z}$ such that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

Lemma A.2: Suppose that $W \rightarrow X$ from S to T and $Y \rightarrow Z$ from T to U are decomposed, simple rewrite rules such that

(a) $ht(T) = k$;

(b) Z is of basic type or a rewrite expression; and

(c) X is a rewrite expression.

Then there is a variable $\hat{Z}$ such that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

Lemma A.3: Suppose that $W \rightarrow X$ from S to T and $Y \rightarrow Z$ from T to U are decomposed, simple rewrite rules such that

(a) $ht(T) = k$;

(b) Z is of basic type or a rewrite expression.

Then there is a variable $\hat{Z}$ such that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

Lemma A.4: Suppose that $W \rightarrow X$ from S to T and $Y \rightarrow Z$ from T to U are decomposed, simple rewrite rules such that

(a) $ht(T) = k$;

Then there is a variable $\hat{Z}$ such that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

In the following proofs, we generally assume that S = P:s, T = Q:t and U = R:u, and that W = P:w, X = Q:x, Y = Q:y, and Z = R:z.

Proof of Lemma A.1: Suppose X = Q:x is of basic type. Then T = Q:t where $t = dom_i$ or **1**. It follows that z = y or z is a constant (possibly of a basic type other than T). If z = y, set $\hat{Z}$ = R:x, and otherwise set $\hat{Z}$ = Z. It is easily verified that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$. □

Proof of Lemma A.2: Let T = Q:{T'} and suppose that X = Q:rew(Δ)(w'), where w' is a variable occurring in W of type P':{S'} and $\Delta = \{W_i \rightarrow X_i \mid i \in [1..n]\}$. Note that ht(S') < k. Because Y = Q:y is of set type and Z is either of basic type or a rewrite expression, it follows that Z is either a constant or a rewrite expression. In the former case, setting $\hat{Z} = Z$ satisfies the lemma.

Now suppose that Z is a rewrite expression. Then U = R:{U'} for some U', and Z = R:rew(Σ)(y) for some $\Sigma = \{Y_j \rightarrow Z_j \mid j \in [1..m]\}$. By the inductive assumption, for each i in [1..n] and each j in [1..m] there is a variable $\hat{Z}_{i,j}$ such that $[W_i \rightarrow \hat{Z}_{i,j}] \equiv [W_i \rightarrow X_i] \circ [Y_j \rightarrow Z_j]$. Letting $\Gamma = \{W_i \rightarrow \hat{Z}_{i,j} \mid i \in [1..n] \text{ and } j \in [1..m]\}$ it easily follows that [P':w' →

R:rew(Γ)(w')] ≡ [P':w' → Q:rew(Δ)(w')] ∘[Q:y → R:rew(Σ)(y)]. It now follows that the lemma is satisfied by setting $\hat{Z}$ = R:rew(Γ)(w'). □

For the proof of Lemma A.3, we need the following easily verified result (proof omitted):

Lemma A.5: If X → Y from S to T is simple and decomposed, then there are choice trees S' and T' of S and T (resp.) such that

- {α(X) | α an assignment} ∩ obj(S) = obj(S'); and
- {α(Y) | α an assignment} ∩ obj(T) ⊆ obj(T'). □

Proof of Lemma A.3: For this lemma, we are assuming that T is an arbitrary type with *-height k, and X = Q:x is an arbitrary term (constructed from W) over T, but Z = R:z is still assumed to be of basic type or a rewrite expression. To prove the result, we essentially perform an induction on the structure of X.

For the basis of this induction, we must consider the cases where (a) x is a single variable, (b) x is a constant of basic type, (c) X is a rewrite expression, and (d) X is Ω. In case (a), since W → X is decomposed, x is either of basic type or of set type. If it is of basic type then Lemma A.1 guarantees that there is an appropriate $\hat{Z}$. If it is of set type, then T = Q:{T'} for some type T', and y is a variable of set type. Letting $\hat{Z}$ be the result of replacing all occurrances of y in Z by x now satisfies the lemma. Case (b) is handled by Lemma A.1, and case (c) is handled by Lemma A.2. Finally, if X = Ω then setting $\hat{Z}$ = Ω satisfies the lemma.

Before embarking on the induction, we address the special case (which arises only if k = 1) where T is of type Q:{$\mathbf{1}_f$} for some $\mathbf{1}_f$. We argue first that there is no rewrite expression occuring in X. [For suppose that some rewrite expression appears in X, say rew(Δ')(w'). Since T is of type Q:{$\mathbf{1}_f$}, the right-handside of rules in Δ' must be $\mathbf{1}_f$:**1**. Since [W → X] is simple, the left-handside must also be of the form $\mathbf{1}_g$:**1** by (iv) of the definition of simple. Thus w' is of type {$\mathbf{1}_g$}, which contradicts (b) of the definition of decomposed.] It follows that x is formed from the values {} and {$\mathbf{1}_f$:**1**}, combined using zero or more applications of the ∪ operation. Also, because [Y → Z] is decomposed, Y = Q:{} or Y = Q:{$\mathbf{1}_f$:**1**}. We consider two cases. First, suppose that {$\mathbf{1}_f$:**1**} does not occur anywhere in X. If Y = Q:{}, then

setting $\hat{Z} = Z$ satisfies the lemma, and if $Y = Q{:}\{\mathbf{1}_f{:}\mathbf{1}\}$ then $\hat{Z} = \Omega$ satisfies it. In the second case, $\{\mathbf{1}_f{:}\mathbf{1}\}$ occurs somewhere in X. Then set $\hat{Z} = Z$ if $Y = Q{:}\{\mathbf{1}_f{:}\mathbf{1}\}$ and set $\hat{Z} = \Omega$ if $Y = Q{:}\{\}$.

Turning to the induction, we must now consider the cases where X is built using a product construction, a union of types construction, a set construction, or a set-union ($\cup$) construction.

Suppose now that $X = Q{:}[X_1,...,X_n]$ where $X_i = Q_i{:}x_i$ is of type T_i for $i \in [1..n]$. Thus $T = Q{:}[T_1,...,T_n]$, and because $[Y \rightarrow Z]$ is decomposed, $Y = Q{:}[Y_1,...,Y_n]$ for some terms $Y_i = Q_i{:}y_i$, $i \in [1..n]$. By assumption, either $Z = R{:}z$ where R is an attribute token and z is either a variable or a constant, or $Z = R{:}rew(\Sigma)(z')$ for some unconstructed set variable z'. Because $Y \rightarrow Z$ is simple, if z (or z') appears in Y then it appears in at most one of the terms Y_i. Thus, let i be chosen so that z (or z') occurs in Y_i if it occurs in Y at all. By the inductive assumption, because X_i is simpler than X, there is some $\hat{Z}'$ such that $[W \rightarrow \hat{Z}'] \equiv [W \rightarrow X_i] \circ [Y_i \rightarrow Z]$.

At first glance, it would seem that $[W \rightarrow X] \circ [Y \rightarrow Z] \equiv [W \rightarrow X_i] \circ [Y_i \rightarrow Z]$ always holds. A subtlety here is that it is possible that $[W \rightarrow X_i] \circ [Y_i \rightarrow Z]$ yields a value on an object O in $\{\alpha(W) \mid \alpha \text{ an assignment}\}$ but that $[W \rightarrow X] \circ [Y \rightarrow Z]$ does not. By Lemma A.5, since each of the rules $[W \rightarrow X_j]$ and $[Y_j \rightarrow Z]$ is decomposed, the image Y_j is the domain of a choice tree of T_j, and the image X_j is a subset of the domain of a choice tree of T_j. If there is an integer $j \in [1..n]$ with $j \neq i$ such that the choice trees of X_j and Y_j are different, then $[W \rightarrow X_j] \circ [Y_j \rightarrow Z] \equiv [W \rightarrow \Omega]$, and more generally, $[W \rightarrow X] \circ [Y \rightarrow Z] \equiv [W \rightarrow \Omega]$. Thus, we define $\hat{Z}$ so that $\hat{Z} = \Omega$ if there is some $j \in [1..n]$ with $j \neq i$ such that the choice trees of X_j and Y_j are different, and $\hat{Z} = \hat{Z}'$ otherwise. It is now easily verified that this choice of $\hat{Z}$ satisfies the lemma.

Suppose now that $X = Q{:}\langle X'\rangle$ where $X' = Q'{:}x'$ is of type T'. This implies that $T = Q{:}\langle T_1;...;T_n\rangle$ for some types $T_1,...,T_n$ and $T_i = Q'{:}t_i$ for some particular $i \in [1..n]$. Furthermore, $Y = Q{:}\langle Y'\rangle$ where Y' is of type T_j for some $j \in [1..n]$. Two cases arise, depending on whether $i = j$ or not. If $i = j$, let $\hat{Z}$ be chosen so that $[W \rightarrow \hat{Z}] \equiv [W \rightarrow X'] \circ [Y' \rightarrow Z]$; and if $i \neq j$, let $\hat{Z} = \Omega$. This satisfies the lemma.

Suppose now that $T = Q:\{T'\}$ and that $X = Q:\{X_1,...,X_n\}$ for some $n \geq 0$. Because we addressed the possibility that T has type $Q:\{\mathbf{1}_f:\mathbf{1}\}$ above, we assume here that T' is not of type $Q:\{\mathbf{1}_f:\mathbf{1}\}$ for any $\mathbf{1}_f$. Because $Y \rightarrow Z$ is simple, this implies that y is an unconstructed set variable, and that Z is either constant (in which case set $\hat{Z} = Z$) or Z is a rewrite expression $R:rew(\Sigma)(y)$. For the latter case, suppose that $\Sigma = \{Y_j \rightarrow Z_j \mid j \in [1..m]\}$. For each pair i,j there is a term $\hat{Z}_{i,j}$ such that $[W \rightarrow \hat{Z}_{i,j}] \equiv [W \rightarrow X_i] \circ [Y_j \rightarrow \hat{Z}_{i,j}]$. It is now easily verified that setting $\hat{Z} = R:\{\hat{Z}_{i,j} \mid i \in [1..n]$ and $j \in [1..m]$, and $\hat{Z}_{i,j} \neq \Omega\}$ satisfies the lemma.

Finally, suppose that $X = X_1 \cup ... \cup X_n$. As in the previous paragraph, Z is either constant or set valued. If Z is a constant, set $\hat{Z} = Z$. Otherwise, $Z = R:rew(\Sigma)(y)$ for some Σ. By the inductive assumption, there are variables $\hat{Z}_i$ such that $[W \rightarrow \hat{Z}_i] \equiv [W \rightarrow X_i] \circ [Y \rightarrow Z]$ for $i \in [1..n]$. It is now straightforward to verify that $\hat{Z} = \hat{Z}_1 \cup ... \cup \hat{Z}_n$ satisfies the conditions of the lemma. □

Proof of Lemma A.4: We now assume that T has $*$-height k, that X is arbitrary, and that Z is arbitrary. We prove the lemma by inducting on the structure of Z.

For the basis, we must consider cases where Z is a basic constant, a basic type variable, a rewrite expression, or Ω. The first three of these are taken care of by Lemma A.3, and the last is satisfied by setting $\hat{Z} = \Omega$.

Referring to the definition of rewrite expression, we must now consider the cases where Z is build using a product construction, a union of types construction, a set construction, or a set-union ($\cup$) construction. Suppose now that $Z = R:[Z_1,...,Z_n]$ where Z_i is of type T_i for $i \in [1..n]$. Suppose inductively that for each i, $\hat{Z}_i$ has the property that $[W \rightarrow \hat{Z}_i] \equiv [W \rightarrow X] \circ [Y \rightarrow Z_i]$. It is now easily verified that setting $\hat{Z}$ to be $R:[\hat{Z}_1,...,\hat{Z}_n]$ satisfies the lemma.

Suppose now that $Z = R:<Z'>$ where Z' is of type T'. Assuming that $\hat{Z}'$ has the property that $[W \rightarrow \hat{Z}'] \equiv [W \rightarrow X] \circ [Y \stackrel{*}{\rightarrow} Z']$. It is now easily verified that setting $\hat{Z}$ to be $R:<\hat{Z}'>$ satisfies the lemma.

The cases where $Z = R:\{Z_1,...,Z_n\}$ and $Z = Z_1 \cup ... \cup Z_n$ are handled similarly. □

APPENDIX B

In this appendix, we exhibit examples to show that each restriction in the definition of simple is needed in order to ensure that simple rules are closed under composition.

a) Violation of (i). Let S = P:{P':<A:dom_1;C:dom_2>}, T = Q:{A:dom_1}, and U = R:{R':[A:dom_1,B:dom_1]}. Let W → X from S to T be

$$P{:}w \rightarrow Q{:}rew(P'{:}\langle A{:}y\rangle \rightarrow A{:}y)(w)$$

and let Y → Z from T to U be

$$Q{:}\{A{:}y,A{:}z\} \rightarrow R{:}\{R'{:}[A{:}y,B{:}z],R'{:}[A{:}z,B{:}y]\}$$

Thus, W → X has the effect of selecting from an object O ∈ obj(S) those elements of type P':<A>; and Y → Z yields a value only on one and two element objects. It can be shown that there is no rule $\hat{W} \rightarrow \hat{Z}$ such that $[\hat{W} \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$.

Suppose now that S' = P'':{S}, T' = Q'':{T} and U' = R'':{U}. It can also be verified that there is no finite set Δ of rewrite rules from S to U such that R':rew(Δ) ≡ (Q':rew(W → X)) ∘(R':rew(Y → Z)). Speaking intuitively, a set Δ which satisfies this equation is

$$\Delta = \{\ P{:}\{P'{:}\langle A{:}y_1\rangle,P'{:}\langle A{:}y_2\rangle,P'{:}\langle B{:}z_1\rangle,\ldots,P'{:}\langle B{:}z_n\rangle\}$$
$$\rightarrow R{:}\{R'{:}[A{:}y_1,B{:}y_1],R'{:}[A{:}y_2,B{:}y_1]\} \mid\ n \geq 0\ \},$$

but this set is infinite.

b) Violation of (ii). Let S = P:{P':<A:dom_1;B:dom_2>}, T = Q:{A:dom_1}, and U = A:dom_1; and let a_0 be a fixed element of the domain of A. Let W → X from S to T be

$$P{:}w \rightarrow Q{:}rew(P'{:}\langle A{:}u\rangle \rightarrow A{:}u)(w);$$

and let Y → Z from T to U be

$$Q{:}\{A{:}a_0\} \rightarrow A{:}a_0.$$

Then [W → X] ∘[Y → Z](O) is defined only if O contains the object P':<A:a_0> (and an arbitrary number of objects of type P':<B>). As above, it can be shown that there is no rule $\hat{W}$ → $\hat{Z}$ such that $[\hat{W} \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$. Also, there is no finite set Δ such that rew(Δ) ≡ rew(W → X) ∘rew(Y → Z).

In this example, the set constant Q:{A:a_0} is used. The result on composition would still hold if simple rewrite rules were permitted to have constants of basic types in premises.

c) Violation of (iii). Let

$$S = P:[P':\{K:[A:dom_1,B:dom_1]\},P'':\{K:[A:dom_1,B:dom_1]\}],$$

$$T = Q:[Q':\{K:[A:dom_1,B:dom_1]\},Q'':\{K:[A:dom_1,B:dom_1]\}],$$

and $U = R:\{K:[A:dom_1,B:dom_1]\}$. Let $W \rightarrow X$ from S to T be

$$P:[P':w_1,P'':w_2] \rightarrow Q:[Q':w_1,Q'':rew(K:[A:x,B:y] \rightarrow K:[A:y,B:x])(w_2)]$$

and let $Y \rightarrow Z$ from T to U be

$$Q:[Q':z,Q'':z] \rightarrow R:z.$$

Speaking informally, $[W \rightarrow X] \circ [Y \rightarrow Z](P:[P':w_1,P'':w_2])$ yields a value only if $\pi_{2,1}(w_2) = w_1$. Because this condition cannot be expressed in a single rewrite rule (even if repeated variables are permitted), it follows that there is no rule $\hat{W} \rightarrow \hat{Z}$ such that $[\hat{W} \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$. Furthermore, there is no finite Δ such that $rew(\Delta) \equiv rew(W \rightarrow X) \circ rew(Y \rightarrow Z)$.

In this example, a repeated variable of set type is used. The result on composition would still hold if simple rewrite rules were permitted to have repeated variables of basic type. (In that case, construction of the rule $\hat{W} \rightarrow \hat{Z}$ might involve a unification of such variables.)

d) Violation of (iv). Let $S = P:\{P':<A;B>\}$, $T = Q:\{\mathbf{1}_0\}$, and $U = R:<\mathbf{1}_1;\mathbf{1}_2>$. Let $W \rightarrow X$ from S to T be

$$P:w \rightarrow Q:rew(P':<A:v> \rightarrow \mathbf{1}_0:\mathbf{1})(w)$$

and let $Y \rightarrow Z$ from T to U be

$$Q:\{\} \rightarrow R:<\mathbf{1}_1:\mathbf{1}>$$

In this case, $W \rightarrow X$ tests for elements of type A. It returns $\{\mathbf{1}\}$ if there is at least one such element and $\{\}$ otherwise. Then $Y \rightarrow Z$ maps $\{\}$ to $<\mathbf{1}_1:\mathbf{1}>$, and is undefined otherwise. Again, there is no rule $\hat{W} \rightarrow \hat{Z}$ such that $[\hat{W} \rightarrow \hat{Z}] \equiv [W \rightarrow X] \circ [Y \rightarrow Z]$, and no finite set Δ such that $rew(\Delta) \equiv rew(W \rightarrow X) \circ rew(Y \rightarrow Z)$. □

APPENDIX C

In this appendix, proofs for Theorems 5.3 and 5.4 are presented. These are straightforward generalizations of results of [HY] to incorporate the one-element domains ($\mathbf{1}_f$) introduced in the present paper. For this reason, only sketches of the proofs are provided here.

Before demonstrating Theorems 5.3 and 5.4, certain functions on the natural numbers called "characteristic functions" are associated to types. In particular, Proposition C.2 below states that the characteristic functions of two normal form types are (essentially) equal iff the two types are isomorphic. The characteristic functions are defined by:

Definition: Let T be a type, and let $dom_1,...,dom_k$ be a listing of (infinite) domains which includes all (infinite) domains occurring in T. The *characteristic function* of T (relative to the listing $dom_1,...,dom_k$) is the function[10] α_T: $\mathbf{N}^k \to \mathbf{N}$ defined recursively by:

(a) if $T = P{:}dom_i$ then $\alpha_T(x_1,...,x_k) = x_i$;

(b) if $T = \mathbf{1}_f{:}\mathbf{1}$ for some $\mathbf{1}_f$ then $\alpha_T(x_1,...,x_k) = 1$;

(c) if $T = P{:}[T_1,...,T_n]$ then $\alpha_T(x_1,...,x_k) = \prod_{i=1}^{n} \alpha_{T_i}(x_1,...,x_k)$;

(d) if $T = P{:}\langle T_1;...;T_n\rangle$ then $\alpha_T(x_1,...,x_k) = \sum_{i=1}^{n} \alpha_{T_i}(x_1,...,x_k)$;

(e) if[11] $T = P{:}\{T_1\}$ then $\alpha_T(x_1,...,x_k) = \exp(\alpha_{T_1}(x_1,...,x_k))$.

Suppose now that T is a type over domains $dom_1,...,dom_k$, that X is a set of atomic elements, and that $x_i = |X \cap dom_i|$ for each $i \in [1..k]$. Then it is straightforward to show that $|\mathbf{obj}_X(T)| = \alpha_T(x_1,...,x_k)$.

Notation: If α and β are functions from $\mathbf{N}^k$ to $\mathbf{N}$, then $\alpha \approx \beta$ if there is some n such that $\alpha(x_1,...,x_k) = \beta(x_1,...,x_k)$ whenever $x_i \geq n$ for each $i \in [1..k]$.

The significance of the characteristic functions is given by the following easily verified result:

[10] Here $\mathbf{N}$ denotes the set of natural numbers.

[11] For $n \in \mathbf{N}$, exp(n) denotes 2^n.

Lemma C.1: For types S and T, S ~ T (abs) iff $\alpha_S \approx \alpha_T$. □

We now state:

Proposition C.2: Let S and T be normal form types. Then S is isomorphic to T iff $\alpha_S \approx \alpha_T$.

This proposition is proved after a series of four lemmas. The first three of these focus on the class of normal form types which have no +-node and which involve only one infinite domain.

Definition: Let dom_1 be a fixed (infinite) domain. Then a type T is *special* if

(i) T is in normal form;

(ii) T has no +-node; and

(iii) The only infinite domain occurring in T is dom_1.

Note that if T is special, then the characteristic function of T can be viewed as a function over one variable.

The first lemma used in the proof of Proposition C.2 provides a basic description of the characteristic functions of special types. (The straightforward inductive proof is omitted).

Lemma C.3: Let T be a special type. If ht(T) = 0, then $\alpha_T(x) = x^m$ for some $m \geq 0$. If ht(T) > 0, then

$$\alpha_T(x) = x^m \times \exp(\alpha_{T_1}(x) + \ldots + \alpha_{T_n}(x))$$

for some $n > 0$, some $m \geq 0$, and some special types T_i with $ht(T_i) < ht(T)$ for each $i \in [1..n]$, and with $ht(T_1) = ht(T) - 1$. □

To compare characteristic functions of special types we use:

Notation: If α and β are functions from **N** to **N**, then $\alpha < \beta$ if there is some n such that $\alpha(x) < \beta(x)$ for each $x \geq n$.

Using Lemma C.3, it is easily verified that:

Lemma C.4: If S and T are special types with ht(S) < ht(T), then for each $g \in \mathbf{N}$, $g\alpha_S < \alpha_T$. □

We now show that the characteristic functions of special types are totally ordered by <:

Lemma C.5: Let S and T be special types which are not isomorphic. Then either $g\alpha_S < \alpha_T$ for each $g \in \mathbf{N}$ or $g\alpha_T < \alpha_S$ for each $g \in \mathbf{N}$.

Proof (sketch): This is proved using an induction on max[ht(S),ht(T)]. If they have different $*$-heights, the result follows from Lemma C.4. Suppose that they have the same $*$-height r. If r = 0, the result is easily verified. If r > 0, by Lemma C.3, we know that

$$\alpha_S(x) = x^k \times \exp(\alpha_{S_1}(x) + \ldots + \alpha_{S_l}(x))$$

for some $l > 0$, some $k \geq 0$, and some special types S_i with $ht(S_i) < ht(S)$ for each $i \in [1..k]$, and with $ht(S_1) = r - 1$; and analogously

$$\alpha_T(x) = x^m \times \exp(\alpha_{T_1}(x) + \ldots + \alpha_{T_n}(x)).$$

By the inductive assumption, we can assume without loss of generality that the types $S_1,\ldots,S_k$ and $T_1,\ldots,T_n$ are listed in decreasing order under < of their characteristic functions. The lemma is now demonstrated by considering the different ways in which S and T are non-isomorphic: either there is some $i \leq \min[k,n]$ such that $S_i \not\equiv T_i$, or $k \neq n$ and $S_i \equiv T_i$ for each $i \leq \min[k,n]$. □

Using a similar argument to the above, it now follows that:

Lemma C.6: Let S and T be normal form types whose only infinite domain is dom_1. If S and T are not isomorphic then either $\alpha_S < \alpha_T$ or $\alpha_T < \alpha_S$. □

Proposition C.2 is the generalization of the above lemma to types involving more than one infinite domain.

Proof of Proposition C.2 (sketch): Let S and T be normal form types, and $dom_1,\ldots,dom_k$ be a listing of the domains occurring in S or T. If S and T are isomorphic then clearly $\alpha_S \approx \alpha_T$. For the converse, suppose that $\alpha_S \approx \alpha_T$. Let n be the maximum out degree of all ×-nodes of S or T. Following [HY], construct S' and T' from S and T by replacing each occurrance of

dom_i by the structure $\{P:[Q_1:dom_1,...,Q_{n+i}:dom_1]\}$. Now S' and T' are special and $\alpha_S \approx \alpha_T$, so by Lemma C.6, S' ≡ T'. It follows from the construction of S' and T' that S ≡ T as desired. □

Proposition C.3 ensures that two absolutely equivalent normal form types are isomorphic. The following result shows that any type can be transformed into a normal form type using the cp-transformations.

Proposition C.7: Let T be a type. Then there is no non-terminating sequence $T = T_0,T_1,T_2,...$ such that $T_i \rightarrow T_{i+1}$ for each $i \geq 0$.

Proof: Following [HY], we recursively define a function σ from types to **N** as follows:

(i) if $T = P:dom_i$ for some dom_i, then $\sigma(T) = 4$.

(ii) if $T = \mathbf{1}_f:\mathbf{1}$ for some $\mathbf{1}_f$, then $\sigma(T) = 4$.

(iii) if $T = P:[T_1,...,T_n]$, then $\sigma(T) = \exp(\prod_{i=1}^{n} \sigma(T_i))$.

(iv) if $T = P:<T_1;...;T_n>$, then $\sigma(T) = 2\times(\sum_{i=1}^{n} \sigma(T_i))$.

(v) if $T = P:\{T_1\}$, then[12] $\sigma(T) = hyp(\sigma(T_1))$.

As in [HY], it is straightforward to verify that $\sigma(T) < \sigma(S)$ whenever $S \rightarrow T$. (In particular, this holds for the two cp-transformations involving **1**.) The proposition now follows. □

We now have:

Proof of Theorem 5.3: Let S be a type. Note that if a type is not in normal form, then an cp-transformation can be applied to it. From this and Proposition C.7, it follows that there is a normal form T such that $S \rightarrow^* T$. Suppose now that T_1 and T_2 are normal form types such that $S \rightarrow^* T_1$ and $S \rightarrow^* T_2$. Then $T_1 \sim S \sim T_2$ (abs). By Lemma C.1 and Proposition C.2, it follows that T_1 is isomorphic to T_2 as desired. Finally, suppose that $S \rightarrow^* R$. Let T' be a normal form type such that $R \rightarrow^* T'$. Then $S \rightarrow^* T'$, and so T and T' are isomorphic. Thus, $T' \rightarrow^* T$ by renaming cp-transformations, and by transitivity, $R \rightarrow^* T$ as desired. □

[12] The *hyperexponentiation* function is defined recursively by hyp(0) = 1 and hyp(i+1) = exp(hyp(i)).

Proof of Theorem 5.4: Let S_1 and S_2 be types. If there is a normal form T such that $S_1 \rightarrow^* T$ and $S_2 \rightarrow^* T$ then clearly $S_1 \sim S_2$ (abs). For the converse, suppose that $S_1 \sim S_2$ (abs). Let T_i be a normal form type such that $S_i \rightarrow^* T_i$ for i = 1,2. Then $T_1 \sim T_2$ (abs). It follows from Lemma C.1 and Proposition C.2 that T_1 is isomorphic to T_2; and so $T_1 \rightarrow^* T_2$ by renaming cp-transformations. Letting $T = T_2$ now satisfies the theorem. □

REFERENCES

[ABe] Abiteboul, S., and C. Beeri, On the power of languages for complex objects, in preparation.

[ABi] Abiteboul, S. and N. Bidoit, Non first normal form relations: an algebra allowing data restructuring, Journal of Computer and System Sciences (1986) -extended abstract in PODS (1984)-

[AH1] Abiteboul, S. and R. Hull, IFO : a formal semantic database model, USC Technical Report 1984, to appear in TODS -extended abstract in PODS (1984)-

[AH2] Abiteboul, S. and R. Hull, Restructuring of complex objects and office forms, Intern. Conf. on Database Theory, Roma (1986)

[AABM] Atzeni, P. G. Ausiello, C. Batini and M. Moscarini, "Inclusion and equivalence between relational database schemata", Theor. Computer Science 19 (1982), 267-285.

[BK] Bancilhon, F., S. Khoshafian, A calculus for complex objects, PODS, Boston (1986).

[FT] Fisher, P., S. Thomas, Operators for non-first-normal-form relations, Proc. of the 7th International Comp. Soft. Applications Conf., Chicago (1983).

[HM] Hammer, M., D. McLeod, Data description with SDM: a semantic database model, TODS 6,3 (1981), 357-386.

[H1] Hull, R., Relative information capacity of simple relational database schemata, SIAM J. Computing (1986).

[H2] Hull, R., A survey of theoretical research in typed complex database objects, USC Technical Report (1986).

[HK] Hull, R., R. King, Semantic database modelling: survey, applications, and research issues, USC Technical Report (1986).

[HY] Hull, R., C.K. Yap, The format model: a theory of database organization, JACM 31,3 (1984), 518-537.

[K] Kuper, G.M., The logical data model: a new approach to database logic, Ph.D. thesis, Stanford University (1985).

[MB] Motro, A., P. Buneman, Constructing superviews, SIGMOD 1981.

[PFK] Purvy, R., J. Farrell, and P. Klose, The design of Star's records processing: data processing for the noncomputer professional, ACM Trans. on Office Automation

Systems 1, 1 (1983), 3-24.

[RKS] Roth M.A., H.F. Korth, A. Silberschatz, Theory of non-first-normal-form relational databases, Austin U. Internal Report (1985)

[SP] Scheck, H-J., P. Pistor, Data structures for an integrated data base management and information retrieval system, VLDB, Mexico (1982).

[SS] Smith, J., D. Smith, Database abstractions: aggregations and generalzation, ACM TODS (1977).

[SLTC] Shu, N.C., V.Y. Lum, F.C. Tung, and C.L. Chang, Specification of forms processing and business procedures for office automation, IEEE Trans. on Software Engineering SE-8,5 (1982), 499-512.

[T] Tsichritzis, D., Form management, Comm. ACM 25, 7 (1982), 453-478.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique