QUERY LANGUAGES FOR HIERARCHIC DATABASES
(Expanded and Revised Version of
TR#454, June 1987)

by

E. Dahlhaus and J.A. Makowsky

Technical Report #652
October 1990
Query Languages for Hierarchic Databases

E. Dahlhaus
Department of Computer Science
University of Bonn
Bonn, FRG

J.A. Makowsky
Department of Computer Science
Technion - Israel Institute of Technology
Haifa, Israel

Revised version October 1990

Abstract

We generalize relational data bases such as to include also hierarc-
chic structures in the form of directories of relations and directories of
directories. In this framework we study computable directory trans-
formations which generalize the computable queries introduced by A.
Chandra and D. Harel. We introduce a transformation language DL
and show its completeness. The language DL can serve as a basis for
specification and correctness of directory transformations and also as a
basis to study their complexity. The method developed can be seen also
in a broader context: It allows the general manipulation of "objects"
(as in SMALLTALK or SETL) and adds to it a construct for paral-
lelism (as in VAL). We also discuss the relationship of our approach to
various other models of hierarchic and object-oriented database mod-
els.
Contents

1 Introduction 3

2 The semantic model 6

3 Computable directory queries and relations 9

4 The directory query language $DL$ 11
   4.1 Syntax of $DL$ ................................................... 11
   4.2 Semantics of $DL$ ................................................. 12
   4.3 Queries expressed in $DL$ ....................................... 15
   4.4 Some useful facts ............................................... 18

5 The choice of basic constructs 19
   5.1 Independence of the constructs ............................... 19
   5.2 The impact of choosing other constructs as basic .......... 22
   5.3 Axioms of set theory and programming constructs ........ 23

6 Coding directories by files and the proof of the completeness theorem 24
   6.1 Construction of $tar.$ .......................................... 24
   6.2 The construction of $TAR.$ ................................. 26

7 Uniform $\Sigma_1$-definability and $DL$ 28

8 Modelling complex data and objects 30
   8.1 Extended directory queries .................................... 30
   8.2 Object-oriented databases and the cumulative hierarchy 32

9 Digression: Gandy Machines 33

10 Conclusions and further research 34
1 Introduction

The relational model for data bases was introduced as a means to describe an appropriate user interface. It served to give semantics to concepts from data bases without taking into account the way the data basis was represented in a computer. The relational model was extremely successful (cf. [Ul82],[Ma83]).

When this study was first undertaken ([DM86a], [DM86b], [DM87]), we were dealing with a file/directory system as well as a data basis. The question we studied was, if one can describe the resulting user interface in a similar way. We stated then that such a description might be particularly interesting for the design and specification of integrated systems such as

"Office by example" [Z82], mail handling software, or any directory restructuring programs. We also noted that our work could serve to model various approaches to hierarchical data bases, cf. [U182] or for specifying file systems, cf. [MS84].

The results of our work were paralleled by other authors (eg. [AB88], [ABGG89], [AK89], [ABDDMZ90], [HS88], [HS89], [SS86]), but the exact relationship between our original model and these later works was not spelled out.

In the present paper we attempt to extend the relational model for data bases to allow a cumulative hierarchic structure, as it is well known in set theory. Families of relations and families of families of ... of families of relations are introduced and are called in analogy to the practice in operating systems "directories". Directories are thus just sets of (sets of sets of ... of) relations, or, in the terminology of logic, higher order relations. We chose this approach, because, as in set theory, it is rich enough to model any other hierarchic structure. We shall discuss the issue of modelling other hierarchic approaches at the end of the paper, in section 8.

The formal definition of this extension of the relational model is presented in section 2. In [CH80] queries are (partial) functions mapping finite sequences of relations (the data base state) into a new relation (the answer to the query). In their framework it is not possible to express what is a restructuring of a data basis or to deal with hierarchies of relations. In our model the analogue of a query is a directory transformation which maps directories into directories. Queries will be special cases of directory transformations. Directory transformations will be called directory queries. Other special cases are directory manipulation programs such as tar in UNIX, system programs reorganizing the division of a disk, or any other restructuring of entire data base systems.
Programming languages which manipulate higher order relations have been considered in various other contexts before. Mostly, the motivation behind such set oriented languages stems from the need to implement readily, arbitrary, abstractly defined data structures. The purpose of very high level languages is to "provide high level abstract and the ability to select data representation in an easy and flexible manner" [SSS79]. The most prominent example is SETL introduced by J. Schwartz [Sch75]. Also "object oriented" programming can be viewed as set oriented. A prominent example of an object oriented programming language (or better environment) is Smalltalk (GR83) or (Ho82). The latter is also a good reference for concepts and implementations of programming languages. Our paper can also be viewed as a contribution to the theoretical foundations of set oriented programming.

In the above sense, relational data base query languages are also set oriented languages. It is clear that relations and operations on relations, as in relational calculus and more powerful query languages [CI82], can be readily implemented in a programming language like SETL. It could be shown that the introduction of the directory concept into relational data bases gives us a framework of equal flexibility, and, with the appropriate choice of programming primitives, of equal power as SETL (cf. [DM86b]).

R. Gandy, in [Ga80], discusses some philosophical aspects of Church's Thesis which are related to our framework. Gandy postulates four principles concerning models of computability from which, in contrast to Church's Thesis, it is provable that functions in these models are partially recursive. He also proves the minimality of those four principles in the sense that no three of them suffice to prove this result. The universe of discourse in [Ga80] are the hereditary finite sets with urelements (cf. [Ba75]), which also form the background of our work here. The computable queries, introduced in this paper, however, do not satisfy all of Gandy's principles. This shows, that not all computable functions satisfy Gandy's principles. But Gandy tries to capture mechanistic aspects of Computation machines, rather than to axiomatize the meaning of computability, as was initiated in [CR80].

The main problem we address in this paper is that of defining precisely the semantic notion of a computable directory query extending naturally the notion of computable queries. This is the context of section 2 and 3. With such a definition one can now define the semantics of various directory query languages. A directory query language $L$ is complete if for every computable directory query there is an expression (program) in $L$ corresponding to it.

In section 4 we define a directory query language $DL$ which is complete. $DL$ is an extension of $QL$ [CH80] with various directory handling constructs.
They correspond to the set theoretic operations union, complement, power set, singleton set and the replacement and induction principle. The induction principle also occurs in QL in the form of the while-construct. The replacement principle leads to a new programming construct

\text{mkdir } \text{y}_i \text{ from } \text{y}_j \text{ in } \text{y}_k \text{ by } P

This construct is very much in the spirit of parallel programming or of data flow languages. It is similar to the for all construct of VAL [cf. [Ho83]. It replaces the subdirectories of \text{y}_i simultaneously and puts them into the directory \text{y}_j. The construct also allows parallel query processing to be expressible in DL. As mentioned before, the programming language DL turns out to be an abstract and well defined sublanguage of SETL which is equivalent to SETL both in computing power and flexibility.

In section 5 we analyze the constructs of DL and exhibit an independent (non redundant) subset \text{DL}_o of DL which is of the same expressive power.

In section 6 we prove the completeness of DL. One way of doing this is to reduce the completeness proof of DL to the completeness proof of QL. This is achieved by showing that we can code each directory by a DL program as \textit{one relation}. After that we can use the completeness of QL to transform this relation into another relation which is a coded directory. The main problem is to guarantee that the coded directories can also be decoded by a program in DL. In other words we show the existence of a computable directory query corresponding to TAR in UNIX. The difference between TAR in UNIX and TAR here is that our coding function does not depend on the way relations and directories are implemented. However, this approach has been criticized in [AB88], as being artificial and misleading. An alternative proof of the completeness is given in section 7. In this section we discuss more generally the relationship between computable directory queries and various set theoretic definability concepts. This section is more of foundational interest than of computational relevance. It relates computability in hereditarily finite sets over urelements to \Sigma_1-definability in the sense of A. Levy [Le65]. It also gives an intrinsic proof of completeness of DL.

As in [CH80] we present our main results in a simplified framework in which neither tuples of the relations nor arbitrary members of directories can be named. It is easy (but tedious) to extend our framework to handle names and predefined objects similar to section 6 of [CH80]. This extension is called in [CH80] the extended query language. We shall discuss the analogue of the extended query language to directory queries informally in in the first part of section 8.
In the second part of section 8 we give a rather detailed discussion of the relationship between our paper and parallel work which appeared between 1986 and 1990.

In section 9 we digress and outline how to reconcile Gandy’s four principles describing computability by discrete mechanical devices with the theory of computable directory transformations and offer a formulation of Gandy’s thesis relating to complexity of parallel computations.

In section 10 we present conclusions and an outlook for further research.

Acknowledgments: We are indebted to C. Beeri for valuable remarks, to R.A. Rasson for his help in preparing this revised version, especially section 8, and to Y. Sagi for retyping the paper in LaTeX. We are also thankful to the editors and referees of this journal who agreed to publish this paper so late after its first circulation.

2 The semantic model

The purpose of this section is to define data bases of higher order. The traditional relational data bases are first order data bases containing only relations. Higher order relational data bases also contain finite sets of finite relations which are called simple directories. More complicated directories can be formed by allowing directories to contain finite sets of both relations and directories of lower order. Relations are just structured files. The approach here is a minor modification of the cumulative hierarchy of finite sets with urelements. In traditional set theory every element is also a set. Sometimes this is not very useful, and elements which have no elements themselves, but are not empty, are allowed. Such elements are commonly called urelements. A similar approach was proposed by D. Roedding in 1967 (cf. [Roe64], [Roe67]) as a general framework for computation with finite objects. We chose this approach for its universality and its simple set of primitives. As much as set theory is rich enough to model virtually all objects encountered in mathematics, the cumulative hierarchy of hereditarily finite sets is rich enough to model all finite objects one may encounter in computer science. However, sometimes the objects one wants to model have a rather awkward definition when written down in this set theoretic way. It would be advisable then to add a new primitive based on such a definition and treat it like a macro or subroutine call in a programming language. We do not advocate our model as being particularly user friendly and easy

6
to use. We rather want to stress, that the foundational question of what it means to compute new objects from a given finite set of objects can be adequately settled in this model.

We start our definition as in [CH80].

2.1 Definition Let $U$ denote a fixed countable set, called the universal domain. Let $D \subseteq U$ be finite and nonempty, and let $R_1, \ldots, R_k$ for $k > 0$, be relations such that, for all $i$, $R_i \subseteq D^m$.

(i) $B = (D, R_1, \ldots, R_k)$ is called a relational first order data base of type $a$, where $a = (a_1, \ldots, a_k)$. $R_i$ is said to be of rank $a_i$. We shall also call the relations directories of order 1.

(ii) Let $V_1(D)$ be the set of all directories of order 1, i.e.

$$V_1(D) = \bigcup_{i \in \mathbb{N}} P_{fin}(D^i)$$

where $P_{fin}(X)$ denotes the set of all finite subsets of $X$.

(iii) Let $V_{j+1}(D) = V_j(D) \cup P_{fin}(V_j(D))$ and

$$V(D) = \bigcup_{j \in \mathbb{N}} V_j(D).$$

$V(D)$ is the set of all directories and $V_j(D)$ is the set of directories of order at most $j$. The order of a directory $\delta \in V(D)$ is the smallest $j$ such that $\delta \in V_j(D)$. $V(D)$ is traditionally called the cumulative hierarchy.

(iv) A higher order data base (hodb) is an ordered tuple $B = (D, \Delta_1, \ldots, \Delta_k)$ where each $\Delta_i$ is a directory in $V(D)$.

2.2 Definition Two directories $\Delta \in V_n(D)$ and $\Delta^\# \in V_m(D^\#)$ over domains $D$ and $D^\#$ are similar if

(i) $\Delta$ and $\Delta^\#$ are of the same order;

(ii) If $\Delta \in V_1(D)$ then $\Delta$ and $\Delta^\#$ have the same rank.

(iii) Otherwise, there is a function $f : \Delta \rightarrow \Delta^\#$ which is 1-1, onto and such that for each $\delta \in \Delta$, $\delta$ and $f(\delta)$ are similar.
Each directory $\Delta \in V$ can be thought of as a directed acyclic graph with labeled leaves in the following way: The leaves are either relations (i.e. in $V_1(D)$) or the empty directory, which is in $V_2(D)$ and is denoted by $\emptyset_{\text{dir}}$. In the first case their label is the rank of the relation. In the other case the label is $-1$. Here we have to remark that for each natural number $k$ we have an empty relation $\emptyset_k$ of rank $k$. There is a directed edge from $\delta_1$ to $\delta_2$ iff $\delta_1$ is a member of $\delta_2$. Two directories are similar if their labeled graphs are isomorphic. Note that $\emptyset_{\text{dir}}$ is really just the empty set whose elements could be relations or directories. We distinguish it from the empty relations of various ranks.

2.3 Definition Let $B = (D, \Delta_1, ..., \Delta_k)$ and $B^\# = (D^\#, \Delta_1^\#, ..., \Delta_k^\#)$ be two hodb's and let $h : D \rightarrow D^\#$ a function between the two domains. We define an extension $h : V(D) \rightarrow V(D^\#)$ in the following way:

(i) For $\delta \in V_1(D)$ a n-ary relation

$$h(\delta) = \{(h(d_1), ..., h(d_n)) : (d_1, ..., d_n) \in \delta\}$$

So $h(\delta)$ is a n-ary relation in $V_1(D^\#)$.

(ii) For $\delta \in V_m(D)$ we put

$$h(\delta) = \{h(\sigma) : \sigma \in \delta\}.$$ 

2.4 Remark If $h$ is one-one then $h(\delta)$ is similar to $\delta$. This is not true in general because we think of directories as sets, not as multisets.

2.5 Definition

(i) $h$ is an isomorphism from $B$ into $B^\#$ if $h$ is one-one and onto and for $0 \leq i \leq k$ $h(\Delta_i) = \Delta_i^\#$.

(ii) Two hodb's $B = (D, \Delta_1, ..., \Delta_k)$ and $B^\# = (D^\#, \Delta_1^\#, ..., \Delta_k^\#)$ are similar if each $\Delta_i$ is similar to $\Delta_i^\#$.

(iii) Two similar hodb's $B = (D, \Delta_1, ..., \Delta_k)$ and $B^\# = (D^\#, \Delta_1^\#, ..., \Delta_k^\#)$ are isomorphic if there is an isomorphism $h : B \rightarrow B^\#$.

2.6 Remark In the case that each $\Delta_i$ is a relation, this notion of isomorphisms coincides with the usual notion of isomorphism of relational data bases. In general, it is a natural extension of this notion.
3 Computable directory queries and relations

3.1 Definition Let \( D \) be a finite set and \( V(D) \) be the set of directories over \( D \). An \( k \)-ary directory transformation is a function \( T : V(D)^k \rightarrow V(D) \) such that for every bijection \( h : D \rightarrow D \) and every \( b_1, ..., b_k \in V(D) \) we have

\[
T(h(b_1), ..., h(b_k)) = h(T(b_1, ..., b_k)).
\]

3.2 Remark If we replace \( V(D) \) by \( \text{Rel}(D) \), the set of all relations over \( D \), this is just the isomorphism invariance of queries in [CH80].

Since all the elements of \( V(D) \) are finite objects, it makes sense to speak of a "standard" coding of \( V(D) \) in the natural numbers \( \mathbb{N} \). This allows us to use freely the notion of computable functions over \( V(D) \).

3.3 Definition An \( k \)-ary directory transformation is computable if it is computable using the standard coding.

3.4 Examples The directory transformations in examples (ii)-(xi) are all computable directory transformations.

(i) The computable queries are computable directory queries: If \( B = (D, R_1, ..., R_k) \) is a relational data base state and \( q \) is a computable query producing a relation \( Q \) we just regard each \( R_i \) as a directory of order 1 and put \( T_q \) to be the obvious \( k \)-ary directory transformation.

(ii) Let \( \delta \) be a directory and let \{\( \delta \)\} be the directory containing \( \delta \) as its only subdirectory. Let \( T_{\text{singleton}} \) be the transformation which maps \( \delta \) into \{\( \delta \)\}.

(iii) Let \( \delta_1, \delta_2 \) be two directories and let \( \delta_1 \cup \delta_2 \) be the directory which contains exactly the subdirectories of \( \delta_1 \) and those of \( \delta_2 \) as its subdirectories. Let \( T_{\cup} \) be the transformation which maps \( \delta_1 \cup \delta_2 \) into \( \delta_1 \cup \delta_2 \).

(iv) Let \( \delta_1, \delta_2 \) be two directories and let \( \delta_1 \cap \delta_2 \) be the directory which contains exactly the subdirectories of \( \delta_1 \) which are not in \( \delta_2 \) as its subdirectories. Let \( T_{\text{difference}} \) be the transformation which maps \( \delta_1 \cap \delta_2 \) into \( \delta_1 \cap \delta_2 \).

(v) Let \( \delta \) be a directory and let \( \text{Pow}(\delta) \) be the directory containing exactly each subset of subdirectories of \( \delta \) as a subdirectory. Let \( T_{\text{power}} \) be the transformation which maps \( \delta \) into \( \text{Pow}(\delta) \).
(vi) Let $\delta$ be a directory and let $U(\delta)$ be the directory containing exactly each subdirectory of a subdirectory of $\delta$ as a subdirectory. Let $T_{\cup}$ be the transformation which maps $\delta$ into $U(\delta)$.

(vii) Let $R$ be an $n$-ary relation of cardinality $p$. We associate with $R$ a directory $\delta$ of order 2 containing $p$ $n$-ary relations each of which contains exactly one $n$-tuple of $R$ and such that each $n$-tuple of $R$ occurs in $\delta$.

(vii) Let $\delta$ be a directory and let $Files(\delta)$ be the directory containing exactly the relations of $\delta$ as its subdirectories. Let $T_{Files}$ be the transformation which maps $\delta$ into $Files(\delta)$.

(ix) Let $\delta$ be a directory and let $Flat(\delta)$ be the directory of order 2 containing exactly the relations which are leaves of $\delta$ as its subdirectories. Let $T_{Flat}$ be the transformation which maps $\delta$ into $Flat(\delta)$.

(z) Kuratowski pair: Let $K$ - Pair$(\delta_1, \delta_2) = \{\delta_1, \{\delta_1, \delta_2\}\}$

$K$-Pair is a computable directory transformation. When the context is clear we write also just $(\delta_1, \delta_2)$ instead of $K$-Pair$(\delta_1, \delta_2)$.

(xi) Let $\delta$ be a directory and let $HTC(\delta)$ be the set of all directories and relations, which are in its transitive closure under membership (the hereditary transitive closure).

(xii) Empty relations and directories: Remember that we distinguish between empty relations of rank (sometimes also called arity) 0, 1, 2, ... which are in $V_2(D)$ and are denoted by $\emptyset_0, \emptyset_1, \ldots$ respectively, the empty directory in $V_2(D)$ which we denote by $\emptyset_{dir}$. The unique non-empty 0-ary relation which has exactly one element, the empty sequence, is denoted by 1. The projection of 1 and $\emptyset_0$ is defined to be the empty relation of rank 0.

(xiii) As in QL we can use 1 as truth value true and $\emptyset_0$ as truth value false. This allows us to define computable predicates as directory queries whose value are true or false.
The examples (i)-(vii) will be among the basic constructs of our directory transformation language \( DL \), defined in the next section. The reader can easily find more examples. As an exercise for computable predicates we suggest comparison of relations via file length, rank of relations and testing whether a directory is in \( V \). 

4 The directory query language \( DL \)

The directory query language \( DL \) we define is essentially a programming language computing finite higher order objects (directories) over some finite domain. As for \( QL \) from [CL80], its access to a directory, however, is only through a restricted set of operations: the operations from \( QL \) augmented by the operations from examples (i)-(vi) in the previous section. Let us now define \( DL \) formally. We include also a definition of \( QL \) to make the paper more self-contained.

4.1 Syntax of \( DL \)

4.1 Definition (Atomic Terms of \( QL \))

(i) \( y_1, y_2, \ldots \) are variables of \( QL \).

(ii) \( E \) is a constant of \( QL \) denoting equality of elements of the domain.

The set of terms of \( QL \) is inductively defined as follows:

4.2 Definition

(i) \( E \) and the variables are terms of \( QL \);

(ii) if \( rel \) is a relation name then it is a term of \( QL \);

(iii) For any terms \( t_1, t_2 \) of \( QL \),

\[
(t_1 \cap t_2), -(t_1), t_1 \downarrow, t_1 \uparrow \text{ and } t_1^\prime
\]

are terms of \( QL \).

Next we define the terms of \( DL \) inductively:

4.3 Definition
4.2 Semantics of DL

To define the semantics of DL we have to define first assignments of variables and then a meaning function of programs.

4.4 Definition (Assignments for variables)

Let $B = (D, \Delta_1, ..., \Delta_k)$ be a hodb.

Let $z$ be a function from the variables $y_1, y_2, ..., y_m$ into $V(D)$, the set of directories over $D$. We call such a function a directory assignment over $D$ or assignment for short.

4.6 Remark We think of the set of all directory assignments over $B$ as the set of states for our directory query. We denote this set by States $(B)$. Note that $z$ is called an assignment, because it assigns values of $V(D)$ to the variables, as much as in propositional logic we assign boolean values to propositional variables.

(i) if dir is a directory name then dir is a term of DL;

(ii) All terms of QL are also terms of DL.

(iii) For any terms $t_1, t_2$ of DL

$$\{t_1, U(t_1), Pow(t_1), Singl(t_1), (t_1 \sim t_2), (t_1 \cup t_2)$$

are terms of DL.

The set of programs of DL (QL) is inductively defined as follows:

4.4 Definition

(i) If $t$ is a term of DL (QL) then $y_i := t$ is a program of DL (QL).

(ii) If $P_1, P_2$ is a program of DL (QL) then $(P_1; P_2)$ and while $y_i$ do $P_i$ are programs of DL (QL).

(iii) All programs of QL are also programs of DL.

(iv) If $P$ is a program of DL then

```
mkdir $y_i$ from $y_i$ in $y_k$ by $P(y_1, ..., y_m)$
```

is a program of DL. The variable $z_j$ occurs here as a bounded variable similar to $j$ in $\Sigma_j a_j$.

(i) if dir is a directory name then dir is a term of DL;

(ii) All terms of QL are also terms of DL.

(iii) For any terms $t_1, t_2$ of DL

$$\{t_1, U(t_1), Pow(t_1), Singl(t_1), (t_1 \sim t_2), (t_1 \cup t_2)$$

are terms of DL.

The set of programs of DL (QL) is inductively defined as follows:

4.4 Definition

(i) If $t$ is a term of DL (QL) then $y_i := t$ is a program of DL (QL).

(ii) If $P_1, P_2$ is a program of DL (QL) then $(P_1; P_2)$ and while $y_i$ do $P_i$ are programs of DL (QL).

(iii) All programs of QL are also programs of DL.

(iv) If $P$ is a program of DL then

```
mkdir $y_i$ from $y_i$ in $y_k$ by $P(y_1, ..., y_m)$
```

is a program of DL. The variable $z_j$ occurs here as a bounded variable similar to $j$ in $\Sigma_j a_j$. 

12
The meaning of program $P$ acting on $B$ is a partial function
\[ \mu(P) : States(B) \rightarrow States(B). \]
First we define for every term $t$ of $DL$ inductively the meaning function $\mu_0(t) : States(B) \rightarrow V(D)$ in the following way:

4.7 Definition For terms $t$ in $QL$, $\mu_0(t)$ is defined as in [CH80]. If $t_1$ and $t_2$ are terms in $QL$ then:

(i) $\mu_0(E)(z) = \{(z, z) : z \in D\}$
(ii) $\mu_0(y_i)(z) = z(y_i)$
(iii) $\mu_0(\text{rel.})(z) = \Delta_i$, if $\Delta_i$ is a relation;
(iv) $\mu_0(t_1 \cap t_2)(z) = \mu_0(t_1)(z) \cap \mu_0(t_2)(z)$, if $\mu_0(t_1)(z)$ and $\mu_0(t_2)(z)$ have the same rank, otherwise $\mu_0(t_1 \cap t_2)(z) = \emptyset$;
(v) $\mu_0(*)(z) = \emptyset$, if $\mu_0(t_1)(z)$ is a relation, otherwise it is $\emptyset$. * stands here for $\cap$, $\cup$ or $\cap$. The meaning $\emptyset'$ of * is complement, projection of all components except of the first, extension of the relation by one last component, or permutation of the last two components of the tuples respectively.

4.8 Definition For the other terms in $DL$, $\mu_0$ is defined inductively in the following way: Let $t_1$ and $t_2$ be terms in $DL$. Then for each $z \in States(B)$:

(i) $\mu_0(\text{dir.}) = \Delta_i$;
(ii) $\mu_0(\{t_1\})(z) = \{\mu_0(t_1)(z)\}$
(iii) $\mu_0(\text{Pow}(t_1))(z) = \text{Powset of } \mu_0(t_1)(z)$;
(iv) $\mu_0(U(t_1))(z) = \bigcup_{\mu_0(t_1)(z)} \{\mu_0(t_1)(z)\}$, if all subdirectories of $\mu_0(t_1)(z)$ are relations of the same rank or all its subdirectories are not relations, otherwise it is set to be $\emptyset$;
(v) $\mu_0(t_1 \cup t_2)(z) = \mu_0(t_1)(z) \cup \mu_0(t_2)(z)$, if $\mu_0(t_1)(z)$ and $\mu_0(t_2)(z)$ are both relations of the same rank or both not relations, otherwise it is set to be $\emptyset$. 

13
(vi) $\mu_0(Sing(t_1))(x) = \{\{z\} : x \in \mu_0(t_1)(x)\}$.

Next we define for every program $P \in DL$ inductively the meaning function $\mu(P)$ in the following way:

4.9 Definition

(i) If $P$ is of the form $y_i := t$ then we put $\mu(P)(z)(y_j) = z(y_j)$ if $j \neq i$ and $\mu(P)(z)(y_i) = \mu_0(t)(z)$ otherwise.

(ii) If $P$ is $P_1; P_2$ then $\mu(P)(z) = \mu(P_2)(\mu(P_1)(z))$. This is the usual composition of functions.

(iii) If $P$ is while $y_j$ do $P_1$ then $\mu(P)(z)$ is defined in the usual way on a sequence of states $z_{i+1} = \mu(P_1)(z_i)$ with $z_0 = z$. $\mu(P)(z)$ is the first $z_i$ such that $z_i(y_j)$ is not an empty relation or directory.

(iv) If $P$ is mkdir $y_i$ from $y_j$ in $y_k$ by $P_1(y_1, \ldots, y_m)$ then

$\mu(P)(z)(y_i) = \{\mu(P_1)(z_1)(y_j) : z_1(y_i) = z(y_i) \}$

for $l \neq j$ and $z_1(y_j) \in z(y_k)$, if for all $z_1$, s.t. $z_1(y_i) = z(y_i)$

for $l \neq j$ and $z_1(y_i) \in z(y_k)$, $\mu(P_1)(z_1)(y_j)$ is defined,

otherwise $\mu(P)(z)(y_i)$ is undefined.

The meaning of $P$ is mkdir $y_i$ from $y_j$ in $y_k$ by $P_1(y_1, \ldots, y_m)$ can be phrased in words as follows: for the case $m = j = 1$, the new directory $y_i$ is obtained in the following way: one applies in parallel to all the subdirectories $y_j$ of $y_k$ the program $P_1$ and puts into $y_i$ all the results so obtained. If $j > m$ the new directory contains exactly one subdirectory $\mu(P_1)(z)(y_i)$. Otherwise, the directories $y_1, \ldots, y_{j-1}, y_{j+1}, \ldots$ are free parameters. Remember that $y_j$ occurs here as a bounded variable. The reader acquainted with axiomatic set theory will easily recognize in this definition the replacement axiom of Zermelo-Fraenkel set theory.
4.3 Queries expressed in DL

Let $D = (D, \Delta_1, \ldots, \Delta_k)$ be a db and $z_{\text{initial}}$ be the assignment with $z_{\text{initial}}(y_i) = \Delta_i$ for all $i \leq k$ and $z_{\text{initial}}(y_i) = \theta_{-1}$ for all $i > k$. Given a program $P(y_1, \ldots, y_n) \in DL$ and a variable $y$, we look at the partial function $T_{p,j} : V(D)^k \rightarrow V(D) T_{p,j}(\Delta_1, \ldots, \Delta_k) = \mu(P)(z_{\text{initial}})(y_j)$.

4.10 Theorem: For every program $P \in DL$ and each variable $y_j$ the partial function $T_{p,j} : V(D)^k \rightarrow V(D)$ is a partially computable directory query.

Proof: For programs of the form $y_i := t$ this follows from the examples (i)-(iv) of section 3. For $P$ of the form $P_1; P_2$ or while $y_i$ do $P_1$ this follows from the closure properties of partial recursive functions. For the mkdir construct this follows from the following closure property of partial recursive functions:

Let $f$ be a partial recursive function from $N^n \rightarrow N$. We denote by $\langle f(a_1, \ldots, a_{j-1}, a, a_{j+1}, \ldots, a_m) : a < b \rangle$ the Goedel number of the set $\{ f(a_1, \ldots, a_{j-1}, a, a_{j+1}, \ldots, a_m) : a < b \}$. Let $g(a_1, \ldots, a_{j-1}, b, a_{j+1}, \ldots, a_m)$ be defined to be

$$
\langle f(a_1, \ldots, a_{j-1}, a, a_{j+1}, \ldots, a_m) : a < b \rangle.
$$

Then $g$ is a partial recursive function from $N^m \rightarrow N$. Now let $P$ be of the form

$$\text{mkdir } y_j \text{ from } y_i \text{ by } P_1(y_1, \ldots, y_m).$$

To complete the proof we note that $f$ corresponds to the program $P_1, g(b)$ to $y_i, b$ to $y_i$ and $a$ to $y_j$.

4.11 Theorem (Completeness Theorem): The directory query language DL is complete, i.e. for every computable directory query $T$ there is a program $P_T \in DL$ computing it.

The proof of this theorem will be presented in section 6. In the proof of 4.3 we shall use the main result of [CH80]:

4.12 Theorem: The query language $QL$ is complete, i.e. for every computable query $T : V_1(D)^n \rightarrow V_1(D)$ there is a program $P_T \in QL$ computing it.
The natural question arises to whether the set of basic constructs is minimal, and, if not, what are the exact interrelationships. It turns out that this is a rather delicate problem. In the following definition we introduce a sublanguage $DL_0$ of $DL$ which has an independent set of constructs. The proof of the independence will be presented in section 5.

4.13 Definition Let $DL_0$ be obtained from $DL$ by restricting its definition to the constructs while and mkdir together with

\[(t)\). ((t) 1), (t) 2), E, U(t), Sing(t), (t_1 \cdot \cdot \cdot t_n).\]

4.14 Remark Generally, as in [CH80], we can simulate the conditional statement by the while-construct. Consider

if $y_i = \emptyset$ then $P$ else $Q$.

Let $y_j$ and $y_k$ be variables not appearing in $P$ or $Q$. Then the following procedure does the same as the above conditional statement:

\[
y_j := 0; y_k := \emptyset;
\]

while $y_j = \emptyset$ do ($P; y_j := E; y_k := E$);

while $y_k = \emptyset$ do ($Q; y_k := E$).

Here we use only constructs of $DL_0$ if $P$ and $Q$ are in $DL_0$. Therefore to be empty is decidable in $DL_0$. Also we can replace the comparison with empty by any other predicate computable in $DL_0$, because the complement can be expressed in $DL_0$ by the term $(E) \downarrow \neg y$.

4.15 Remark Using the mkdir-construct we have what is called in set theory a replacement scheme in $DL$. In other words, let $P$ be a predicate which is expressable in $DL$. Then the function $G$ which maps each directory $\delta$ to the set of its subdirectories $\delta_1$, such that $P(\delta_1)$, is also expressable in $DL$: in the following way:

(i) Let $H$ be the function with maps each $\delta_1$ to its singleton \{\delta_1\} if $P(\delta_1)$ and to the empty set otherwise. $H$ is obviously expressable in $DL$ if $P$ is in $DL$.

(ii) $\cup \{H(\delta_1): \delta_1 \in \delta\}$ is the set $G(\delta)$. 

16
4.16 Lemma: There is a program $u - \text{pair}$ in $DL_0$ which computes for two directories $\delta_1, \delta_2$ the directory which contains exactly $\delta_1$ and $\delta_2$ as its subdirectories.

Proof: We consider the function $F : Singl(D^d) \times V(D) \rightarrow V(D)$ defined as follows:

$$F((x, y)), a, b) = a \text{ if } x = y, b \text{ otherwise}$$

By the remark above $F$ is computable in $DL_0$, because $x \neq y$ means $(x, y) \in E = \emptyset$. Using the mkdir-construct the function which computes for each $a$ and $b$ the set $\{F(u, a, b) = \{a, b\} \in \text{computable in } DL_0 \}.$

4.17 Lemma: There is a program $\text{join}$ in $DL_0$ which computes for two directories $\delta_1, \delta_2$ the directory which contains exactly the subdirectories of $\delta_1$ and those of $\delta_2$. (Note that $\text{join}$ here stands for the union of two directories and not for the "join"-operation for relations.)

Proof: $\text{join}(\delta_1, \delta_2)$ is the program $U(u - \text{pair}(\delta_1, \delta_2)).$ \[\Box\]

4.18 Lemma: There is a program $\text{Rel}$ in $DL_0$, which decides whether a directory is a relation or not.

Proof: $\text{join}(\delta, u - \text{pair}(\delta))$ is an empty directory or relation iff $\delta$ is a relation. \[\Box\]

Moreover we get the following

4.19 Lemma: There is a function computable in $DL_0$ which maps each relation $R$ to $D^k$, where $k$ is the rank of $E$, and which maps directories, which are not relations, to $\emptyset$.

Proof: By the remark and lemma 4.6, we have only to consider the case that the input $R$ is a relation. There we start with $Y_1 := 1$ and as long as $\text{join}(R, Y_1)$ is empty, we set $Y_1 := (Y_1) \downarrow$. After leaving this loop $Y_1$ is the wanted $D^k$. \[\Box\]

4.20 Proposition: Every program in $DL$ is expressible by a program in $DL_0$.

Proof: We have to show that the missing term operations can be expressed in $DL_0$. For intersection and complement for relations we use lemma 4.4-4.7. For $\{t\}$ we use
in the case \( j > m \).

To write a program for \( F(t) \) we first observe that the power set of a finite set is the smallest set containing all the singletons of its elements and which is closed under join. This can be easily converted into a program using \( \text{Singl}(t) \), \( \text{join} \), \( \text{U} \) and the constructs while, \text{mkdir}. \( \square \)

4.21 Remark From a complexity point of view, \( \text{Singl} \) is an operation which takes logarithmic space whereas the power set takes exponential space.

4.4 Some useful facts

We conclude this section with some easy propositions which serve as examples and which will be used over and over again in section 6.

4.22 Proposition: Let \( \delta = (\delta_1, \delta_2) \) be the Kuratowski pair of \( \delta_1 \) and \( \delta_2 \), and let \( \pi_1(\delta), \pi_2(\delta) \) be the projections (cf. example (z) of section 3). Then there are computable directory queries in DL computing the Kuratowski pair and its projections respectively.

Proof: Recall that \( \delta = \{ \{ x \}, \{ x, y \} \} \). Now singleton directories are in DL and therefore the union of two singletons, the unordered pair of two directories, is computable in DL. From this we can conclude that also the Kuratowski-pair is computable in DL.

The first projection \( \pi_1(p) \) is the intersection of the elements of \( p \). That is expressible in DL.

The second projection \( \pi_2(p) \) is \( U(p - \pi_1(p)) \). \( \square \)

4.23 Proposition: The hereditary transitive closure \( \text{HTC}(\delta) \) of a directory \( \delta \) is computable by a program of DL.

Proof: As first step set \( y := \{ \} \cup \delta \) and \( z := \delta \). As long as the set of elements of \( z \) not being in \( V_1(D) \) is not empty do \( P \),

where

\[
P \triangleq z_1 := U\{ z : x \notin V_1(D) \}; y_1 := y \cup z; z := z_1; y := y_1.
\]

The output \( y \) of this procedure is the hereditary transitive closure of \( \delta \). \( \square \)
5 The choice of basic constructs

5.1 Independence of the constructs

In this section we shall prove the independence of the constructs of $DL_0$, as announced in section 4. That means:

5.1 Theorem: For each construct $c$ of $DL_0$ there is a computable directory query $T$ which is not computable in $DL_0 - \{c\}$.

Proof: For each construct $c$ of $DL_0$ we will prove a lemma from which one can easily check that $DL_0 - \{c\}$ is not complete. □

5.2 Remark: It should be noted that the independence of the constructs in QL, as presented in [CH80], does not entail the independence of the same constructs in $DL$. It would be conceivable that some higher order construct would allow us to express constructs of QL in some way using this higher order construct.

The negation:

Let $h$ be a surjective map from a domain $D$ to a domain $D_1$. For each $k$-ary relation $r$ we define

$h( r ) = \{ ( h( x_1), \ldots, h( x_n ) ) : ( x_1, \ldots, x_n ) \in r \}$

and for directories $\delta$ we define $h(\delta) = \{ h(\delta_i) : \delta_i \in \delta \}$. We prove now the following

5.3 Lemma: For each directory query $T$ of $DL_0 - \{\sim\}$ and each surjective map $h : D \rightarrow D_1$:

$h( T( \delta_1, \ldots, \delta_n ) ) = T( h(\delta_1), \ldots, h(\delta_n) )$ (1)

Proof: This follows from the fact that each function of the base of $DL_0$ except $\sim$ has this faithfulness property (1) and for each directory $\delta$ we have $\delta$ is empty iff $h(\delta)$ is empty. By induction on the length of the program the lemma is easily checked. □

An immediate consequence of lemma 5.2 is that the complement of the diagonal $\{(x,y) : x \neq y\}$ is not computable in $DL_0 - \{\sim\}$. 

19
The operators $\uparrow$ and $\downarrow$:

5.4 Lemma:

(i) If $P$ is a program in $DL_0 - \{\uparrow\}$ then each leaf of each directory has at each state of the program a rank not exceeding the maximal rank of the leaves of the input.

(ii) Provided that each leaf of any directory of the input has a rank not less than 2, then for each directory generated by a program $P$ of $DL_0 - \{\uparrow\}$ its nonempty leaves have a rank not less than 2.

(i) and (ii) can be easily checked by induction on the length of the program.

The equality predicate:

Recall that $E$ is the equality predicate in $QL$ and is also a construct of $DL_0$. We get the following fact:

5.5 Lemma: All directories generated by a program of $DL_0 - \{E\}$ and an input with only empty leaves have only empty leaves.

Proof: All operations of $DL_0 - \{E\}$ preserve emptiness of each leaf. □

The permutation of the last two elements:

Recall that $\iota$ represents the permutation of the last two elements of a tuple.

As a construct in $DL$ it is also a construct in $DL_C$. We consider a program $P$ in $DL_0 - \{\iota\}$ with an unary relation as its only input. Let $(D, R)$ be the input structure with domain $D$. Let $I$ be a bijection from $D$ to $D$. Define

$$I_2(s) = \{(x_1, x_2, I(x_3), \ldots, I(x_n)) : (x_1, \ldots, x_n) \in s\}$$

$I_2$ is extended to directories in the canonical way. We apply the bijection here on all $k$-th components with $k > 2$. The first and the second component are not changed.

Then the following fact proves that $DL_0 - \{\iota\}$ is incomplete:

5.6 Lemma: For each $DL_0 - \{\iota\}$-computable function $T : (V(D))^k \rightarrow V(D)$ and each bijection $I : D \rightarrow D$ we have
The union operation:

Recall that $\text{Singl}$ represents the function which maps each relation or directory $\delta$ to the set \( \{ \{x\} : x \in \delta \} \). We can prove now the following:

\textbf{5.7 Lemma} Suppose $P$ is a program in $DL_0 - \{\text{Singl}\}$ and the input is $D$. Then each set generated by $P$ has as leaves the empty set and relations definable by boolean combinations of the equality (and therefore closed by any bijection on $D$).

\textbf{Proof}: The application of any elementary operation of $QL$ on relations preserve the properties stated in Lemma 5.6. The $\text{mkdir}$-construct is the only construct, which can map any directory in $V_i(D)$ to a directory in $V_{i+1}(D)$. Since all subdirectories generated by $P$ preserves the properties stated in this lemma and by the assumption that each subprogram of $P$ also fulfills the properties of this lemma, also the $\text{mkdir}$-construct preserves the definability of any relation by boolean combinations of the equality. If we apply any term of $DL$ on higher order directories, then it can happen, that the relations are not changed or some new relations appear as unions of old relations or as negations of old relations. In all these cases the definability by boolean combinations of the equality is preserved. □

The singleton operation:

Recall that $U$ represents the function mapping each set to the union of all its elements. Then the following fact is true:

\textbf{5.8 Lemma}: Let $P$ be a program of $DL_0 - \{U\}$ and all inputs of $P$ not be in $V_i(D)$. Then each relation generated by $P$ with this input is describable by a constant term in $QL$.

\textbf{Proof}: The only operation generating a relation from a nonrelational directory $\delta$ dependent on $\delta$ is the union. □
The while loop:

By induction on the length of the program, we get the following:

5.9 Lemma: For each function $T$ computable in $DL_0 - \{\text{while}\}$ there is a natural number $k$, s.t. for each $n$ we have: if $z_1, \ldots, z_m \in V_n(D)$ then $T(z_1, \ldots, z_m) \in V_{n+k}(D)$.

The parallel construct mkdir:

Recall that mkdir applies a program on all subdirectories of a directory and constructs in that way a new directory. Let $\{x\}^k$ be the set generated from $x$ by applying $k$ times the singleton operation $\{x\}$. We define $E$ to be $x$ itself. Let $I : D \rightarrow D$ be a bijection. Then a relation $r$ is preserved by $I$ iff $I(r) = r$, where $I$ applied to relations is defined canonically.

5.10 Lemma: Let $P$ be a program of $DL_0 - \{\text{mkdir}\}$ and $\delta_1, \ldots, \delta_k$ be an input. Then for each directory $\delta$ generated by $P$ and the $\delta_i$:

for each subdirectory $\delta$ of the transitive closure of $\delta$, each natural number $m, n$:

$$\{x \in D^m : \{x\}^n \in \delta\}$$

is preserved by automorphisms of $(D, \delta_1, \ldots, \delta_k)$.

Proof: The claimed property of $\delta$ is preserved by all operations of $DL_0$ except the mkdir-construct. □

For example, the power set is not computable in $DL_0 - \{\text{mkdir}\}$.

5.2 The impact of choosing other constructs as basic

Our model of hodb's is very set theoretic. Although the traditional membership relation $\in$ does not appear among the basic constructs, it can be easily implemented by the following macro: First we observe that $\{x\} \cap y \neq \emptyset$ iff $x \in y$. Here the construct $\{x\}$ is among the basic constructs, but intersection and comparison to the empty set are not. As union and set difference are among the basic constructs, it is clear how to define intersection. Recall that comparison to the empty set appears hidden in the definition of the while-construct and, therefore, also in the derived conditional statement (cf. Remark 4.14). Once we have implemented the comparison to the...
empty set we easily get also set comparison by the formula:

$$(A \setminus B) \cup (B \setminus A) = \emptyset$$

If $A \subseteq B$ then $A \cap B = A$. Therefore, we have the following Theorem:

5.11 Theorem (Universality Theorem)

Every object which is set theoretically definable by some boundedly quantified set theoretic formula can be described by some DL-program.

This theorem guarantees that every data structure or concept can be somehow (possibly in a rather complicated way) defined within the model of hodb's. To avoid notational complications, like in standard mathematical practice, those data structures and concepts are then introduced by definitional macros and added to the general framework without affecting the basic properties of computability and isomorphism invariance. The importance of Theorem 5.11 will be further discussed in section 8.

The choice of our basic constructs was guided by our continuation of the work by A. Chandra and D. Harel [CH80]. The question arises what would have happened had we chosen different basic concepts. Assume we had chosen a set $M = \{m_1, \ldots, m_k\}$ of basic concepts. Let $B_1$ and $B_2$ be two hodb's and $h$ be an isomorphism between $B_1$ and $B_2$. By the Universality Theorem, all the concepts of $M$ are definable in $DL$ and therefore $h$ also is an isomorphism for the operations and constructs of $M$. On the other hand, if the set $M$ gives rise to a corresponding Universality Theorem, then all the basic concepts of the hodb's are definable over $M$, and therefore, every $M$-isomorphism is also a hodb-isomorphism. The only case where there could be a $M$-isomorphism which is not a hodb-isomorphism occurs when $M$ does not satisfy a Universality Theorem. Such cases may be desirable in real implementations, and their study therefore has its own right. However, for our foundational discussion the Universality Theorem guarantees the flexibility needed to model arbitrary data structures and objects.

5.3 Axioms of set theory and programming constructs

Most programming constructs used in the definition of $DL$ correspond to some set theoretic axioms asserting the existence of further sets: Union, complement, cartesian product, etc. It is interesting to pursue this analogy.
of set theory and programming constructs further in the case of the while-loop and the mkdir-construct.

The while-loop asserts the existence of a smallest fixed point. In set theory the first assertion of the existence of a fixed occurs in the definition of the natural numbers, or for that matter, of some infinite set, hence it corresponds to the axiom of infinity.

The mkdir-construct asserts the existence of a new directory (set) built in parallel and iteratively from other directories. It corresponds in set theory to the axiom of substitution (or axiom of replacement, as it is sometimes called). The need for this axiom was overlooked in the early period of set theory, as it does only occur naturally when one starts to deal with more complex objects than just simple subsets of natural numbers.

It is no surprise, post festum speaking, that both constructs, the while-loop and the mkdir-construct, are needed to make DL complete. As a matter of fact, it is rather satisfactory, from a foundational point of view, that this is so, as it shows to what extent imperative and declarative thinking are interrelated. From this point of view the proof of the completeness theorem presented in section 7 is more natural, in as much as it is declarative (set theoretic), than the proof presented in section 6. The criticism expressed in [ABCG89], therefore was justified, though the argument put forward missed the point.

6 Coding directories by files and the proof of the completeness theorem

The proof of the Completeness Theorem (theorem 4.11) consists of three steps. In the first and third step we use a coding and decoding program TAR and TAR \(^{-1}\). TAR is, inspired by the UNIX program of the same name, a program that takes directories of arbitrary order and makes one file from which the original directory can be uniquely reconstructed by TAR \(^{-1}\). The difficulty in writing TAR in DL comes from the fact that we may not use names and other information of the directory structures. The programs TAR and TAR \(^{-1}\) allow us to reduce our completeness proof to the completeness proof for QL in [CH86]. This is the middle step in our proof.
6.1 Construction of \( \text{tar} \).

To construct \( \text{TAR} \) and \( \text{TA}^{-1} \) we define at first a function \( \text{tar} \), which maps every directory of \( V_2(D) \) to a single relation and is \( 1 - 1 \), and a function \( \text{tar}^{-1} \) which reconstructs a directory \( X \) of \( V_2(D) \) from \( \text{tar}(X) \).

At first we define \( \text{tar} \):

Given a set directory \( X \) in \( V_2(D) \). Note that all elements of \( X \) are relations. The idea of the construction of \( \text{tar} \) is that each tuple of \( \text{tar}(X) \) codes one relation.

Let \( r \) be a relation appearing in \( X \). Let \( (s_i)_{i=1}^\infty \) be an enumeration of \( r \). Let \( a_r \) be the rank of \( r \). Define the tuple \( s = (x^{a_r+1}, y, s_1, \ldots, s_p) \) to be a precode of \( r \), if \( x \neq y \).

Note that any relation \( r \in X \) is reconstructible from any of its precodes. Let \( s \) be of the form \( (x^q y^q) \). Then \( q \) is uniquely determined as the smallest \( i \) such that the \( i \)th component and the \((i + 1)\)th component are unequal. \( q - 1 \) assigns the rank of the relation, which is coded by \( s \). Let \( s' \) be the tuple \( (s_1, \ldots, s_k) \). Then \( s' \) is a precode of the relation \( ((s_{(q-1)+1}, \ldots, s_{(q+1)-1})) : 0 \leq i \leq k/(q - 1) \).

The remaining step is to take care that all tuples, which code a relation in \( X \), have the same rank.

We add additional components \( x \) and \( y \) at the beginning.

Let \( s = (x^{a_r+1}, y, f) \) be a precode of the relation \( r \). Then a code of the relation \( r \) has the form \( (x^{a_r}, y, s) \).

More precisely: Let \( m \) be the maximum cardinality of a relation in \( X \) and \( a \) be the maximum rank of a relation in \( X \). Then any concatenation \( s' \) of the elements of a relation in \( X \) has a length of at most \( am \). Therefore each precode of any relation has a length of at most \( a + 2 + am \).

A code of the relation \( r \) is a tuple \( (x^{a_r}, y, x^{a_r}, y, s) \) of length \( a + 4 + am \), where \( x \neq y \) and \( s \) is the concatenation of all elements of \( r \). Note that \( a_r > 0 \).

Note that each relation is reconstructible from any of its codes:

Given any code \( c = (c_1, \ldots, c_i) \) of the relation \( r \). Let \( u \) be the smallest \( i \) such that \( c_i \neq c_{i+1} \). Then \( (c_{i+2}, \ldots, c_t) \) is a precode of \( r \). From that \( r \) is uniquely constructible.

\( \text{tar}(X) \) consists of all codes of relations \( r \in X \).

6.1 Lemma: There is a computable directory query in DL computing the directory transformation \( \text{tar} \) such that
(i) The domain Dom(tar) of tar consists of the directories having only relations as their subdirectories.

(ii) \( \{ \delta \in V(D) \mid \text{if it is defined.} \) 

Proof: (i) At first we have to compute the maximum size and the maximum rank of any relation in \( X \). Using the replacement scheme we can compute for each \( r \in X \) the set \( \text{rank}(r, X) \) and \( \text{size}(r, X) \) of \( r \in X \) of larger rank and of larger size respectively. Using the replacement scheme we get the sets \( \text{Maxrank}(X) = \{ r \in X : \text{rank}(r, X) = 0 \} \) and \( \text{Maxsize}(X) = \{ r \in X : \text{size}(r, X) = 0 \} \) of relations of maximum rank and maximum size respectively. Since we can compute for each relation \( r \) the relation \( D_{\text{sizeof} r} \) and \( D_{\text{rankof} r} \) in \( QL \), we get \( D^* \) and \( D^m \) by the application of the mkdir-construct of \( \text{Maxrank}(X) \) and \( \text{Maxsize}(X) \) respectively.

Now we can compute the relation \( \text{code}(r, D^*, D^m) \) of codes of \( r \) by a Turing machine and by [CH 80] by a \( QL \)-program.

Using the mkdir-construct and the union construct we can compute \( \text{tar}(X) = \{ \text{code}(r, D^*, D^m) : r \in X \} = \{ s \in \text{code} : s \text{ is a code of some relation } r \in X \} \). 

(ii) follows from the definition of \( \text{tar} \).

\[ 6.2 \text{Lemma: There is a computable directory query } \forall \text{tar}^{-1} \in DL \text{ which is the inverse of } \text{tar}, i.e. for every } \delta \in \text{Dom(tar)} \text{ we have } \text{tar}^{-1}(\text{tar}(\delta)) = \delta. \]

Proof: As mentioned in the definition of \( \text{tar} \), each tuple is the code of at most one relation. The relation of which \( s \) is a code, is computable by a Turing machine. Therefore the function \( \text{Decode} \) which maps the singleton \{s\} of \( s \) to the relation it codes can be computed in \( QL \) and therefore is \( DL \).

\[ \text{tar}^{-1}(R) \text{ can be set as } \{ \text{Decode}(s) : s \in R \} = \{ \text{Decode}(s) : s \in \text{Singl}(R) \} \text{. } \text{tar}^{-1} \text{ is therefore computable in } DL 	ext{ using the Singl-construct and the mkdir-construct on the function } \text{Decode}. \]

\[ 6.2 \text{ The construction of } TAR, \]

Using \( \text{tar} \) we now define \( TAR \) recursively on the order of the directory. For a relation \( \delta \in V(D) \)

\[ TAR(\delta) = \{(a, a, x) : a \in D \text{ and } x \in \delta \} \quad (1) \]
In other words, if \( \delta \) is a relation we add three arguments to it to make sure that it can be recognized as a single relation. Note that \( TAR(\emptyset_6) = \emptyset_{6,4} \).

The program in \( DL \) expressing this is easily obtained once one has observed that "being a relation" is a computable directory query (see lemma 4.6).

For \( \delta = \emptyset_{dir} \) we set

\[
TAR(\delta) = \{(a, a, b, b) : a, b \in D \text{ and } a \neq b\}
\]  

(2)

Thus \( TAR(\emptyset_{dir}) \) is coded by a relation in \( D^4 \) of rank 4 such that the first two arguments are equal and different from the third and the fourth argument.

The program in \( DL \) expressing this is easily obtained once one has observed that "being an empty directory" is a computable directory query (see remark in section 4, number (1) and lemma 4.6).

For arbitrary directories \( \delta \) we set

\[
TAR(\delta) = \{(a, b, z) : a, b \in D, c \neq \delta \text{ and } z \in TAR((TAR(\delta), \delta_1) : \delta_1 \in \delta)\}
\]  

(3)

This is like a recursive procedure call where \( TAR \) is applied to the subdirectories of \( \delta \). Moreover, note that \( TAR(\delta) \) is not empty for each nonrelational directory \( \delta \). Therefore we can distinguish the empty directories also by \( TAR \).

6.3 Lemma: There is a computable directory query \( TAR \in DL \) such that

(i) The domain \( Dom(TAR) \) of \( TAR \) consists of all the directories of \( V(D) \).

(ii) \( TAR(\delta) \in V_1(D) \) for each directory \( \delta \).

Proof: We consider the function \( TAR \) as defined in (1), (2) and (3). We have to prove that this function is expressible in \( DL \). We compute at first the transitive closure \( TC(\delta) \) of the given directory \( \delta \). This is expressible in \( DL \). The leaves (elements without a subdirectory in \( TC(\delta) \)) are relations or the empty directory. We can compute the set of leaves and call it \( Z_0 \). We compute \( P_0 := \{(x, TAR(x)) : x \in Z_0\} \). Here \( (x, y) \) means the Kuratowski pair of \( x \) and \( y \) as defined before. We set now \( Z = Z_0 \) and \( P = P_0 \) and as long as \( TC(\delta) - Z \) is not empty, we add to \( Z \) the set \( Y \) of all \( z \), where all its subdirectories are in \( Z \) and add to \( P \) all \( (x, TAR(x)) \), \( s.t. \ x \in Y \). That procedure is expressible in \( DL \) and computes \( TAR(\delta) \). The properties (i) and (ii) follow from the above definition of \( TAR \).

6.4 Remark The proof of lemma 6.3 gives us a general scheme, how to describe a recursive procedure in \( DL \)-constructs.
6.5 Lemma: There is a computable directory query $TAR^{-1} \in DL$ which is the inverse of $TAR$, i.e. for every $\delta \in \text{Dom}(TAR)$ we have

$$TAR^{-1}(TAR(\delta)) = \delta.$$ 

Proof: Let $P$ be the same $DL$-program. Then generally it is possible to calculate the set $L_\alpha(x)$ which is obtained from $x$ by replacing every leaf $y$ of it by $P(y)$, because that can be expressed recursively. Given any relation $r$ (of the form $TAR(\delta)$), (1) if $r$ is of the form $\{(a,b,a,2): a \neq b$ and $z \in s\}$ then set $T(r) = tar^{-1}(s)$, (2) if $r$ is of the form $\{(a,a,a,2): z \in s\}$ or $\{(a,a,b,b): a \neq b\}$ ($r$ is the code of a leaf) then $T(r) = r$. To calculate $TAR^{-1}(r)$ of a relation $r$ we iteratively replace each leaf $u$ (at the beginning $r$ itself) by $T(u)$, until nothing is changed any more. After this iteration all leaves are codes of relations or the empty directory. They are then replaced by the empty directory or the relation it codes. That all can be expressed in $DL$. \qed

7 Uniform $\Sigma_1$-definability and $DL$

In this section we want to relate our results to set theoretic definability theory. This has two reasons: It puts our approach into a general context of computability of finite, set theoretically defined objects and it will allow us to give an alternative proof of the Completeness Theorem, which is not dependent on the special coding program $TAR$.

Definability theory studies the structure of first order definable sets in various structures such as arithmetic, the real numbers, models of set theory, etc. The purpose is to characterize definable sets in terms of recursion theory, topology or game theory. Classical monographs on the subject are [Ba75], [Mo74],[Mo84]. The pioneer paper for models of set theory is [Le65]. There he introduces the notion of $\Sigma_1$-definability in set theory as a generalization of recursive enumerability in the infinite set theoretic context. The analogy of $\Sigma_1$-definable and recursive enumerable sets is based on the following fact (which is folklore among set theorists):

Consider the structure $HF = (HF, \in)$ with the hereditary finite sets without urelements as its universe and membership as its only relation. In $HF$ the $\Sigma_1$-definable sets are exactly the recursively enumerable sets.

The notion of $\Sigma_1$-definability has a natural meaning also in the structures $HF(D)$ where $D$ is a finite set of urelements and $HF(D) = (HF(D), \in)$.
consists of the collection of the hereditary finite sets with urelements of
D as its universe and the membership as its only relation. The structure
\( \mathbb{H}(D) \) is very similar to the structure \( V(D) = \langle V(D), \in \rangle \) in which our
language \( DL \) operates. So, the question arises whether the computable di-
rectory queries are related to an appropriate version of \( \Sigma_1 \)-definable sets.
The purpose of this section is to define \( \Sigma_1 \)-definability appropriately and to
establish the following theorem:

7.1 Theorem (Completeness Theorem, new version)
Let \( A \subseteq V(D) \). Then the following statements are equivalent:

(i) \( A \) is recursively enumerable and isomorphism invariant (that means
for each natural extension \( h \) of a bijection from \( D \) to \( D \) such that
\( x \in A \) iff \( h(x) \in A \);

(ii) \( A \) is recognizable by a DL-program;

(iii) \( A \) is \( \Sigma_1 \)-definable, that means, there is a \( \Sigma_1 \)-formula \( \Phi \), s.t.
\[
A = \{ \delta : V(D)| = \Phi(\delta) \}.
\]

Note that (i) just states that the characteristic function of \( A \) is a computable
directory query, and (ii), that the characteristic function of \( A \) is the meaning
of a DL-program. Therefore, their equivalence are just theorems 4.1 and
4.2.

We consider formulas using the function symbols \( ^{-}, \cdot \), \( \lnot \), \( \land \) and \( \land \) of [CH80]
and the 2-ary membership relation symbol \( \epsilon \) as its nonlogical symbols.

We write \( (\forall x \in y)^P \) for \( \forall x(x \in y \rightarrow P) \) and, similarly, \( (\exists x \in y)^P \)
for \( \exists x(x \in y \land P) \). \( (\forall x \in y) \) and \( (\exists x \in y) \) are called bounded quantifiers.

A formula \( \Phi \) is called \( \Sigma_0 \) iff all quantifiers in it are bounded and \( \Sigma_1 \) iff
it is of the form \( \exists x_1, \ldots, \exists x_n \psi \) where \( \psi \) is \( \Sigma_0 \).

Sketch of proof of theorem 7.1: We will prove \( (i) \rightarrow (iii) \) and \( (iii) \rightarrow (ii) \).
\( (ii) \rightarrow (i) \) is trivial.
\( (i) \rightarrow (iii) \): Assume \( A \) is recognized by a Turing machine \( P \). Then \( \delta \in A \)
iff there is a correct coding of \( \delta \) and there is a \( P \)-computation on the coding
giving a positive answer. Codings on Turing machines and computations
can be coded as sets in $V(D)$, provided that $A$ is isomorphism invariant. Hence we get a $\Sigma_1$-formula expressing $A$.

(iii) $\rightarrow$ (ii): We want to prove that each $\Sigma_1$-expressible subset of $V(D)$ is recognizable by a program in $DL$. First we can prove that $\varepsilon$ is decidable by a $DL$-program using the fact that $x \in y$ iff $\{x\} \cap y \neq \emptyset$.

Claim: If $F$ is a predicate, decidable by a $DL$-program, then also $(\exists x \in y) P$ is decidable by a $DL$-program.

This claim follows from the Universality Theorem (Theorem 5.11. It can also be proved directly by the replacement scheme, presented in the remark 4.15 of section 4. From this follows that each $\Sigma_0$-predicate is decidable by a $DL$-program.

Now we have to consider a $\Sigma_1$-formula $\exists x \Psi$. Let $Z_k(D)$ be the (finite) set of all $x \in V_k(D)$, whose leaves have rank not greater than $k$. Clearly the union of all $Z_k(D)$ is $V(D)$. Moreover we get a computable directory query which computes for each $D^k$ the set $Z_k(D)$. We only have to write a $DL$-program which computes the smallest $Z_k(D)$, which has an $x$ satisfying $\Psi$. We give a short informal description of a $DL$-program. Let $Z_{k,1}$ be the finite set of all $x \in V_k(D)$, whose leaves have a rank not greater than 1. Then

$$Z_{i,k} = \emptyset \cup \bigcup_{v=1}^{i} \text{Pow}(D^v)$$

and $Z_{i+1,k} = Z_{i,k} \cup \text{Pow}(Z_{i,k})$. It is easily seen, that $Z_k = Z_{n,k}$ can be computed by a $DL$-program. Each $k$ is coded by a set in a standard way.

For the case that no $x$ exists, s.t. $\Psi$ is satisfied, the $DL$-program deciding the above $\Sigma_1$-formula does not terminate.

8 Modelling complex data and objects

In this section we discuss possible variations and extensions of the hodb's. We first discuss extended directory queries in the sense of [CK80]. In the following subsection we briefly discuss various papers which appeared after our original work was done and which address the issue of object-oriented data models.
6.1 Extended directory queries

When directory and data base systems are used in practice, several operations and predicates outside the formal relational and directory framework are useful, or even necessary, to turn the system into a practical and efficient model. Concerning the purely relational aspect of data bases, [CH80] addresses this issue and proposes the extended query language EQL. The main difference in [CH80] between computable and extended computable queries lies in the semantics. In the extended model they look at two sorted structures where an additional domain $F$ is added, whose elements may be numbers, or any other set of terms, whose interpretations are fixed.

If we want to adapt this approach to our framework, we should first examine what we really have in mind. The new objects to be introduced are really "names", i.e. interpretations of certain terms whose meaning is never changed and is part of the user interface. They can be words over some finite alphabet $A$ (including natural numbers in some $b$-ary notation). They usually have some standard operations and relations on them, such as concatenation, arithmetical operations and/or a linear order. This makes the new universe with its functions into a Herbrand universe. It is easy to modify our framework for these purposes. We take the extended semantic model of [CH80] as our starting point, i.e. $V_k(D \cup F)$. Here $D$ is a finite set of urelements, as before, and $F$ is a possibly infinite set disjoint from $D$. There must be enough functions to make sure that every element of $F$ is the interpretation of some term. Relations are always finite and their one-dimensional projections are always either in $D$ or in $F$. The restrictions of isomorphisms on $F$ are always the identity. The constructions of $V(D \cup F)$ is continued naturally. We leave it to the reader to formulate everything in detail.

In contrast to the case of [CH80], extending the directory model in this way does not give us increased expressive power. The universe of the natural numbers, e.g. does exist in $V(D)$, though it is not an element of any $V_k(D)$. Since we allow higher order relations, every finite set of natural numbers can be thought of as being in some $V_k(D)$, and therefore, relations involving natural numbers can be coded in $V(D)$. The advantage of the extended approach lies in its inherent economy, both conceptually and computationally. Conceptually, we can now formulate various aspects of directory systems, which were only expressible before in a rather cumbersome way. Among these are time stamp labels, listing the names of the subdirectories of a
directory (the `is`-command in UNIX) with all its variations, and the introduction of arithmetical and statistical functions. The set of urelements $\mathcal{D}$, however, is not assumed to be linearly ordered and cannot be linearly ordered within $DL$. In contrast to this, the directories and relations can be linearly ordered by the lexicographic order of the names. Theorem 5.11 puts all this into a precise mathematical context.

8.2 Object-oriented databases and the cumulative hierarchy

We now discuss how related approaches should be treated in our framework. In [AB88] and [ABGG89] there are several differences to our approach:

- They allow several domains rather than one. For this our model would have to be extended into a many-sorted set theory or sorts should be simulated by labels. To do this is straightforward, but notationally very taxing. We avoid it here for the same reasons as in [CH80].

- Our basic objects are relations and directories. They introduce a more refined distinction into basic objects, tuples, and sets. We can simulate this by defining labels and other features artificially. Again it would be notationally taxing to carry these features into our model approach, and it would not serve to clarify the role of programming primitives.

The model proposed in [SS86] is a special case of [AB88] and [ABGG89]. The model proposed in [AK89] allows cycles in the definition of both schemas and instances. Again this can be simulated in our model. However, this simulation would be rather complex and look unnatural. The situation of this coding is similar to the coding of ordered pairs in set theory. Once introduced, it is immediately treated as a new primitive. Clearly, if we incorporate such features into our model, this new primitive would have to be reflected also in the query language defined below.

In [ABDDMZ90] a list of mandatory, optional and unsettled features of database systems are discussed which would enable them to qualify as object-oriented. Among are the requirements that the associated query language should be complete. However, it lists object identity, classes and inheritance as mandatory requirements. The latter can again be simulated in our model with some notational effort.

Clearly, our approach is hiding features which may seem of great importance when discussing database systems with specific user interfaces.
However, it is not, as claimed in [ABGG89], a restricted system, as it is capable of coding many sorts, various typings of objects, objects with cycles, inheritance, classes, object identity.

9 Digression: Gandy Machines

Our framework allows us to address also the fundamental question of computability by discrete mechanical devices as initiated by Gandy [Ga80] and also discussed in [DM88]. There, Gandy addresses the question of mechanical realizability of processes in the framework of hereditarily finite sets. We call such processes Gandy Machines (GM). Gandy gives four principles which will guarantee that a Gandy Machine is mechanically realizable. The first principle states that every Machine can be represented by a pair \((S, F)\) where \(S \subseteq V(D)\) is closed under directory isomorphisms and \(F : S \rightarrow S\) is a directory transformation. As the principles II, III, and IV are stated in the language of \(V(D)\), it is straightforward to phrase them in our framework.

Principle II requires that for a Gandy Machine \((S, F)\) \(S \subseteq V_i(D)\) for some \(k \in \omega\). As we do not need a precise formulation of principles III and IV in the sequel, we leave it to the reader to translate them into our framework.

A Gandy Machine \((S, F)\) is computable if the characteristic functions of \(S\) and \((V(D) - S)\) and \(F\) are computable directory transformations. Gandy now proves that every Gandy Machine which satisfies I-IV, is computable in this sense. (This follows Gandy's theorem, principle II and our theorem 4.2). He also shows that there are Gandy Machines \((S, F)\) which are not computable but satisfy any two of II, III, and IV. He then formulates the Thesis P (Gandy's thesis) that every discrete mechanical device can be realized as a Gandy Machine satisfying II-IV.

In Complexity Theory, various complexity classes were proposed to capture the notions of efficient computability (in comparison to mechanical realizability). Lately, however, in the context of models of parallel processing, complexity theory was also linked to the issues of real time computability and realizability by physical networks (VSLI). One such prominent complexity class is the class NC, Nick's Class, introduced by N. Pippenger [Co85]. It is therefore challenging to test Gandy's thesis against the computable directory transformations which are in NC. It is now easy to exhibit Gandy Machines not satisfying II which are in NC, or even computable by a parallel network in constant time, for instance the directory transformation \(\ldots\), which maps any directory \(\delta\) into the directory \(\{\delta\} = \delta,\) whose only
subdirectory is \( \ell \). This might be construed as contradicting Gandy's thesis.

However, the proof of theorem 4.2 shows that, though \((V(D), \{\ldots\})\) is a Gandy Machine which does not satisfy principle II, we can use the directory transformation \( \text{TAR} \) to obtain the Gandy Machine

\[
(\text{TAR}(V(D)), \text{TAR} \circ \{\ldots\} \circ \text{TAR}^{-1})
\]

which does satisfy principle II (and also III and IV). This shows that some directory queries may violate the Gandy principles from a logical (representational) point of view, but not from the point of view of mechanical realization. This distinction has been recently discussed in [BHS7] from an experimental engineering point of view.

This suggests the following precise reformulation of Gandy's thesis:

Let \( XX \) be a complexity class which could be reasonably identified with some notion of realizability by discrete mechanical devices. Then for every Gandy Machine \((S, F)\) in \( XX \) the Gandy Machine \((\text{TAR}(S), \text{TAR} \circ F \circ \text{TAR}^{-1})\) satisfies II-IV.

A reasonable candidate for \( XX \) is the subclass of NC which describes constant parallel time.

10 Conclusions and further research

We see the main merits of this paper in the precise definition of the semantics of set-oriented programming languages and also as a contribution to generalized computation theory. In contrast to generalized recursion theory \([Fe74], [Mo74], [Mo80], [Mo84], [No78]\), which attempts to extend recursion theory to arbitrary infinite structures, we are more concerned here in computations using finite structures. One of the earliest papers in this direction which uses hereditary finite sets as its framework seems to be \([En78]\). But, as the reader must have realized, we were mostly influenced by the fundamental paper \([CH80]\). We tried to show, and we hope that we have succeeded, that the approach in \([CH86]\) does not only work for relational data bases, but also for more general situations. In this paper we have extended relational data bases by the directory concept. The relationship of our work with later independent developments was discussed in section 8. In \([DM86b]\) we show how to apply this approach for SETL-like programming languages, and how
to draw from this approach also results in languages capturing complexity classes similar to those obtained in [Fa74], [CH82], [HP84], [Im82], [Im83]. The study of the relationship between complexity classes and various sublanguages of DL will be delayed to future research. It seems clear that various results of [CH82], [Im82], [Im83], [HP84], [DM86b] have their analogues. A survey of more recent results on descriptive and computational complexity may be found in [Im89].

Traditionally, in set theory, all mathematical objects are built from the empty set alone, though the use of urelements (elements which are not sets, i.e. which do not have elements themselves) was never completely rejected. In [Ba75] it was actually argued that avoiding urelements results in a conceptual loss. Our semantics is based on a set theory of hereditarily finite sets with urelements, which allow us to make the concept of user interface invariance (isomorphism invariance) precise. Our two main theorems (the completeness of DL and the independence of the constructs of DLo) just illustrate that the chosen framework for our semantics is correct.

We also think that our paper may clarify what is really needed to build a satisfactory very high level language and may lead to a formal definition, and, ultimately, to more economical implementations of such languages.
References


[GR83] A. Goldberg and D. Robson, Smalltalk-80: The language and its implementation, Addison-Wesley, Reading MA, 1983.


Related later papers:


[ABGG89] S. Abiteboul, C. Beeri, M. Gyssens and D. Van Gucht, An introduction to the completeness of languages for complex objects, in: Nested Relations and Complex Objects in Databases,


