COMPUTABLE DIRECTORY QUERIES

(Revised Version)

by

E. Dahlhaus and J.A. Makowsky

Technical Report #454

June 1987
Computable Directory Queries.

(Revised version April 1987)

E. Dahlhaus¹
Department of Computer Science,
University of Bonn,
Bonn, FRG
and
J.A. Makowsky²
Department of Computer Science,
Technion - Israel Institute of Technology,
Haifa, Israel

Abstract: We generalize relational data bases such as to include also directories of relations and directories of directories. In this framework we study computable directory transformations 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 parallelism (as in VAL).

¹ Till September 1986 Department of Mathematics, Technical University Berlin, West Berlin. During part of the preparation of this paper (Summer 1985) visiting the Technion sponsored by Minerva Foundation.
² During parts of the preparation of this paper (1985) visiting at the Forschungsinstitut für Mathematik, ETH-Zentrum, 8092 Zürich, Switzerland.
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. [U82, M83]). When dealing with a file/directory system as well as a data basis the question arises if one can describe the resulting user interface in a similar way. Such a description might be particularly interesting for the design and description of integrated systems such as "Office by example" [Zl82], mail handling software or any directory restructuring programs. It also can serve to model various approaches to hierarchic data bases, cf [U82], or for specifying file systems, cf. [MS84].

In the present paper we attempt to extend the relational model for data bases to allow directories. Directories are sets of (sets of sets of ...) relations, or, in the terminology of logic, higher order relations. 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, or 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 objects and operations between them, high level control structures 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 [Ho83]. 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 [CH82], 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 pro-
gramming 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 frame-
work. 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 com-
putable 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 [CH80]. In section 9 we out-
line 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 it to complexity
of parallel computations.

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 content 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 $QL$ of
[CH80] with various directory handling constructs. They correspond to the set theoretic operations union, comple-
ment, 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

\[
\text{mkdir } y_i \text{ from } y_j \text{ in } y_k \text{ by } P.
\]

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 $y_k$ simultaneously and puts them into the direc-
tory $y_i$. The construct also allows parallel query processing to be expressible in $DL$. As mentioned before, the pro-
gramming 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 $DL_0$ of $DL$ which is of the same expressive power.

In section 6 we prove the completeness of $DL$. The main idea 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 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.

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 [CH80, section 6]. 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 section 7.

In section 8 we discuss 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].

In section 9 we present conclusions and an outlook for further research.

2. The semantic model.

The purpose of this section is to define data bases of higher order. The traditional relational data bases are then 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.

More formally, we start our definition as in [CH80]. Let $U$ denote a fixed countable set, called the universal domain. Let $D \subset U$ be finite and nonempty, and let $R_1,\ldots,R_k$ for $k > 0$, be relations such that for, for all $i$, $R_i \subset D^n$. $B=(D,R_1,\ldots,R_k)$ 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.

Let $V_1(D)$ be the set of all directories of order 1.
\[ V_1(D) = \bigcup_{i \in \mathbb{N}} P_{fa}(D^i) \]

where \( P_{fa}(X) \) denotes the set of all finite subsets of \( X \).

Let \( V_{j+1}(D) = V_j(D) \cup P_{fa}(V_j(D)) \) 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 \( \delta \in V_j(D) \).

A higher order date base (hodb) is an ordered tuple \( B = (D, \Lambda_1, \Lambda_k) \) where each \( \Lambda_i \) is a directory in \( V(D) \).

Two directories \( \Delta \in V_m(D) \) and \( \Delta^g \in V_m(D^g) \) over domains \( D \) and \( D^g \) are similar if

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

(ii) If \( \Delta \in V_j(D) \) then \( \Delta \) and \( \Delta^g \) have the same rank.

(iii) Otherwise, there is a function \( f : \Delta \rightarrow \Delta^g \) 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 \( \emptyset_{dir} \). 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.

Let \( B = (D, \Lambda_1, \Lambda_k) \) and \( B^g = (D^g, \Lambda^g_1, \Lambda^g_k) \) be two hodbs and let \( h : D \rightarrow D^g \) a function between the two domains. We define an extension \( \overline{h} : V(D) \rightarrow V(D^g) \) in the following way:

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

\[ \overline{h}(\delta) = \{(h(d_1), \ldots, h(d_n)) : (d_1, \ldots, d_n) \in \delta \} \]

So \( \overline{h}(\emptyset) \) is a \( n \)-ary relation in \( V_1(D^g) \).

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

\[ \overline{h}(\delta) = \{h(\alpha) : \alpha \in \delta \} \]

If \( h \) is one-one then \( \overline{h}(\emptyset) \) is similar to \( \emptyset \). This is not true in general because we think of directories as sets, not as multisets. \( h \) is an isomorphisms from \( B \) into \( B^g \) iff \( h \) is one-one and onto and for \( 0 \leq i \leq k \) \( \overline{h}(\Lambda_i) = \Lambda^g_i \).

Two hodbs \( B = (D, \Lambda_1, \ldots, \Lambda_k) \) and \( B^g = (D^g, \Lambda^g_1, \ldots, \Lambda^g_k) \) are similar if each \( \Lambda_i \) is similar to \( \Lambda^g_i \).

Two hodbs \( B = (D, \Lambda_1, \ldots, \Lambda_k) \) and \( B^g = (D^g, \Lambda^g_1, \ldots, \Lambda^g_k) \) are isomorphic if they are similar and there is an isomorphism \( h : B \rightarrow B^g \).

In the case that each \( \Lambda_i \) 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.

Let $D$ be a finite set and $V(D)$ be the set of directories over $D$. An $k$-ary directory transformation is a function $T:V(D)^n \to V(D)$ such that for every bijection $h:D \to D$ and every $\delta_1, \ldots, \delta_k \in V(D)$ we have

$$T(h(\delta_1), \ldots, h(\delta_k)) = h(T(\delta_1, \ldots, \delta_k))$$

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 $V(D)$.

An $k$-ary directory transformation is computable if it is computable using the standard coding.

Examples:

(i) The computable queries are computable directory queries: If $B=(D, R_1, \ldots, R_k)$ 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. The transformation $T_{\text{singleton}}$ which maps $\delta$ into $\{\delta\}$ is clearly a computable directory transformation.

(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 $\delta_2$ as its subdirectories. The transformation $T \cup$ which maps $\delta_1$ and $\delta_2$ into $\delta_1 \cup \delta_2$ is clearly a computable directory transformation.

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

(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. The transformation $T_{\text{power}}$ which maps $\delta$ into $\text{Pow}(\delta)$ is clearly a computable directory transformation.

(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. The transformation $T_{\cup}$ which maps $\delta$ into $U(\delta)$ is clearly a computable directory transformation.

(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 $\delta$. Clearly, this defines a computable directory transformation.
(viii) Let $\delta$ be a directory and let $\text{Files}(\delta)$ be the directory containing exactly the relations of $\delta$ as its subdirectories. The transformation $T_{\text{Files}}$ which maps $\delta$ into $\text{Files}(\delta)$ is clearly a computable directory transformation.

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

(x) (Kuratowski pair) Set

$$K\text{-Pair}(\delta_1,\delta_2) = \{\delta_1, \delta_2\}$$

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

(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). Then clearly $HTC$ is a computable directory query.

(xii) Empty relations and directories: We distinguish between empty relations of arity $0, 1, 2, \ldots$ which are in $V_1(D)$ and are denoted by $\emptyset_0, \emptyset_1, \ldots$ respectively, the empty directory in $V_2(D)$ which we denote by $\emptyset_0$ and the projection of the unique non empty 0-ary relation has exactly one element, the empty sequence, and is denoted by $1$. The projection of 1 and $\emptyset_0$ is defined to be the empty relation of arity 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, arity of relations and testing whether a directory is in $V_k(D)$.

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 [CH80], 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 selfcontained.

Syntax:

$y_1, y_2, \ldots$ are variables of $DL$. The set of terms of $DL$ is inductively defined as follows:

(i) $E$ is a term of $QL$; for $i \geq 1$ $x_i$ are terms of $QL$; if $dir_i$ is a directory name then $dir_i$ is a term of $DL$; if $rel_i$ is a
relation name then it is a term of $QL$.

(ii) For any terms $t_1, t_2$ of $QL$

$$(t_1 \land t_2), \neg(t_1), (t_1)\downarrow, (t_1)\uparrow \text{ and } (t_1)^-$$

are terms of $QL$.

(iii) All terms of $QL$ are also terms of $DL$.

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

$$(t_1), U(t_1), \text{Pow}(t_1), \text{Singl}(t_1), (t_1 \neg t_2), (t_1 \cup t_2)$$

are terms of $DL$.

The set of programs of $DL$ is inductively defined as follows:

(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 $P_1$ are programs of $DL$ ($QL$).

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

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

$\text{mkdir } y_i \text{ from } y_j \text{ in } Y_k \text{ by } P(y_1, \ldots, y_m)$

is a program of $DL$. The variable $y_j$ occurs here as a bounded variable similar to $j$ in $\sum^2_j$.

Semantics:

Let $B = (D, A_1, \ldots, A_k)$ be a hodb.

(i) Let $z$ be a function from the variables $y_1, y_2, \ldots$ into $V(D)$, the set of directories over $D$. We call such a function a directory assignment over $B$ or assignment for short. 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 $\text{States}(B)$.

(ii) The meaning of a program $P$ acting on $B$ is a partial function $\mu(P): \text{States}(B) \rightarrow \text{States}(B)$.

First we define for every term $t$ of $DL$ inductively the meaning function $\mu_0(t): \text{States}(B) \rightarrow V(D)$ in the following way:

For terms $t$ in $QL$, $\mu_0(t)$ is defined as in [CH 80]. If $t_1$ and $t_2$ are terms in $QL$ then:

$\mu_0(E)(x) = \{ (x, x) : x \in D \}$,

$\mu_0(y_i)(x) = z (y_i)$,

$\mu_0(\neg) = \Delta_i$, if $\Delta_i$ is a relation,

$\mu_0(t_1 \land t_2)(x) = \mu_0(t_1)(x) \land \mu_0(t_2)(x)$, if $\mu_0(t_1)(x)$ and $\mu_0(t_2)(x)$ have the same arity, otherwise $\mu_0(t_1 \land t_2)(x) = \emptyset$.

$\mu_0(\neg t_1)(x) = \neg \mu_0(t_1)(x)$, if $\mu_0(t_1)(x)$ is a relation, otherwise it is $\emptyset$. $\neg$ stands here for $\neg$, $\downarrow$, $\uparrow$ or $\neg$. The meaning
of * is complement, projection of all components except of the first, extension of the relation by one last component, or cyclic right permutation of the tuples respectively. Note that in [CH80] cyclic right permutation is not used. Instead they have an operation interchanging the last two elements of a tuple, provided the tuple has length at least two.

For the other terms in DL, μ₀ is defined inductively in the following way:

Let t₁ and t₂ be terms in DL. Then for each z ∈ States(B):

(0) \( m_0(\text{dir}_1) = \Delta_z \)

(1) \( \mu_0(t_1)(z) = \{ \mu_0(t_1)(z) \} \)

(2) \( \mu_0(\text{Pow}(t_1))(z) = \text{Powerset of } \mu_0(t_1)(z) \)

(3) \( \mu_0(U(t_1))(z) = \bigcup \{ \mu_0(t_1)(z) \} \), if all subdirectories of \( \mu_0(t_1)(z) \) are relations of the same arity or all subdirectories of it are not relations, otherwise it is set to be \( \emptyset \).

(4) \( \mu_0(t_1 \cup t_2)(z) = \mu_0(t_1)(z) \cup \mu_0(t_2)(z) \), if both \( \mu_0(t_1)(z) \) and \( \mu_0(t_2)(z) \) are relations of the same arity or both not relations, otherwise it is set to be \( \emptyset \).

(5) \( \mu_0(t_1 \rightarrow t_2)(z) = \mu_0(t_1)(z) \setminus \mu_0(t_2)(z) \)

Here \( X \rightarrow Y \) is the set of all elements of \( X \) not being in \( Y \). Note the difference between \( \rightarrow \), which is an operation symbol, and \( \sim \), which is set theoretic difference. \( X \rightarrow Y \) is a relation of arity \( k \) (a nonrelational directory) iff \( X \) is a relation of arity \( k \) (a nonrelational directory).

(6) \( \mu_0(\text{Singl}(t_1))(z) = \{ \{ x \} : x \in \mu_0(t_1)(z) \} \)

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

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

(b) 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.

(c) 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.

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

\[
\mu(P)(z)(y_i) = \begin{cases} 
\{ \mu(P_1)(z_i)(y_j) : z_1(y_i) \in \mu(P_1)(z_i) \} & \text{for } l \neq j \text{ and } z_1(y_i) \in z(y_k) \\
\mu(P_1)(z_i)(y_i) & \text{if for all } z_1, \text{ s.t. } z_1(y_i) = z(y_i) \text{ for } l \neq j \text{ and } z_1(y_i) \in z(y_k) \text{ and } \mu(P_1)(z_i)(y_j) \text{ is defined, otherwise } \mu(P)(z)(y_i) \text{ is undefined.} 
\end{cases}
\]

In words this says, for the case \( m = j = 1 \), that 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 in to \( y_i \) all the results so obtained. If \( j > m \) the new directory contains exactly one subdirectory \( \mu(P_1)(z)(y_i) \). Otherwise, the directories \( y_{1 \ldots j-1}y_{j+1} \ldots \) are free parameters. Remember that \( y_j \) occurs here as a bounded variable. The reader acquainted with axiomatic set theory
DL

will easily recognize in this definition the replacement axiom of Zermelo-Fraenkel set theory.

Queries: Let \( B = (D; \Delta_1, \ldots, \Delta_k) \) be a hodb and \( z_{\text{initial}} \) be the assignment with \( z_{\text{initial}}(y_i) = \Delta_i \) for all \( i \leq k \) and \( z_{\text{initial}}(y_i) = \emptyset \) for all \( i > k \). Given a program \( P (y_1, \ldots, y_n) \in DL \) and a variable \( y_j \) we look at the partial function \( T_{P,j}: V(D)^k \rightarrow V(D) \quad T_{P,j}(\Delta_1, \ldots, \Delta_k) = (P)(z_{\text{initial}})(y_j) \).

**Theorem 4.1:** For every program \( P \in DL \) and each variable \( y_j \) the 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 \( \mathbb{N}^n \rightarrow \mathbb{N} \). We denote by \( \langle f(a_1, \ldots, a_{j-1}, a, a_{j+1}, \ldots, a_m); a < b \rangle \) the Gödel 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 \( \mathbb{N}^n \rightarrow \mathbb{N} \).

Now let \( P \) be of the form

\[
\text{mkdir } y_i \text{ from } y_j \text{ in } y_k \text{ by } P_1(y_1, \ldots, y_n).
\]

To complete the proof we note that \( f \) corresponds to the program \( P_1, g(b) \) to \( y_1, b \) to \( y_k \) and \( a \) to \( y_j \).

**Theorem 4.2:** 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 5. In the proof of 4.2 we shall use the main result of [CH80]:

**Theorem 4.3:** 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 sub-language \( DL_0 \) of \( DL \) which has an independent set of constructs. The proof of the independence will be presented in section 5.

**Definition:** Let \( DL_0 \) be obtained from \( DL \) by restricting its definition to the constructs while and mkdir together with

\[
((t)_\downarrow), ((t)_\uparrow), ((t)_\cdot), E, U(t), \text{Singl}(t), (t_1 \cdot t_2).
\]
Remark:

(1) Generally, as in [CH80], we can simulate the conditional statement by the while-construct. Consider

\[
\text{if } y_1 = \emptyset \text{ then } P \text{ else } Q.
\]

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:

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

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 \( DL \) resp. \( DL_0 \), because the negation can be expressed in \( DL_0 \) by the term \((E) \downarrow \downarrow \neg y\).

(2) Using the mkdir-construct we have a comprehension scheme in \( DL \):

Given a predicate \( P \) decidable in \( DL \). Then the function \( G \) which maps each directory \( \delta \) to the set of its subdirectories \( \delta_1, \ldots, \delta \), s.t. \( P(\delta_i) \) is expressible in \( DL \):

(i) Let \( H \) be the function which maps each \( \delta_i \) to its singleton \( \{\delta_i\} \) if \( P(\delta_i) \) and to the empty set otherwise. \( H \) is obviously expressible in \( DL \) if \( P \) is in \( DL \).

(ii) \( \cup \{H(\delta_i); \delta_i \in \delta\} \) is the set \( G(\delta) \).

**Lemma 4.4:** 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 \( F : Singl(D)^2 \times V(D) \rightarrow V(D) \) 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 \( x = y \) means \( (x,y) \rightarrow E = \emptyset \). Using the mkdir-construct the function which computes for each \( a \) and \( b \) the set \( \{F(u,a,b); u \in Singl(D)^2\} = \{a,b\} \) is computable in \( DL_0 \).

**Lemma 4.5:** 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 \( \delta_2 \).

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

**Lemma 4.6:** There is a program \( \text{Rel} \) in \( DL_0 \), which decides whether a directory is a relation or not.

**Proof:** \( \text{join}(\delta, u\text{-pair}(\delta, \delta)) \) is an empty directory or relation iff \( \delta \) is a relation. Moreover we get the following
Proposition 4.8: Every program in $DL_0$ is expressible by a program in $DL$.  

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 - 4.8.

Lemma 4.1: There is a function computable in $DL_0$ which maps each relation $R$ to $D^k$, where $k$ is the arity of $R$ and directories not being 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 $Y_1 := (Y_1)^\dagger$. After leaving this loop $Y_1$ is the wanted $D^k$.

Proposition 4.8: Every program in $DL$ is expressible by a program in $DL_0$.

Proof: We have to show that the missing term operations can expressed in $DL_0$. For intersection and complement for relations we use lemma 4.4 - 4.8.

For $\{t\}$ we use

\[ \text{mkdir } y_i \text{ from } y_j \text{ in } y_k \text{ by } P(y_1, \ldots, y_m) \]

in the case $j > m$.

To write a program for $P(t)$ 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, mkdir.

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

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

Proposition 4.9: 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 (x) 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 $\pi_1(\delta)$ is the intersection of the elements of $\delta$. That is expressible in $DL$.

The second projection $\pi_2(\delta)$ is $U(p \leftarrow \pi_1(\delta))$.

Proposition 4.10: The hereditary transitive closure $HTC(\delta)$ of a directory $\delta$ is computable by a program of $DL$.

Proof: As first step set $y := (\delta) \cup \delta$ and $z := \delta$.

As long the set of elements of $z$ not being in $V_1(D)$ is not empty do $P$, where
\[ P = z_1 := U \{ x \in z : x \not\in V_1(D) \}; y_1 := y \cup z; z := z_1; y := y_1. \]

The output \( y \) of this procedure is the hereditary transitive closure of \( \delta \).

5. Independence of Constructs.

In this section we will prove the independence of the constructs of \( DL_0 \), as announced in section 4. That means:

**Theorem 5.1:** 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 it one can easily check, that \( DL_0 - \{ c \} \) is not complete.

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

**Lemma 5.2:** For each directory query \( T \) of \( DL_0 - \{ \cdot \} \) and each surjective map \( h : D \rightarrow D_1 \):

\[ h(T(\delta_1, \ldots, \delta_m)) = T(h(\delta_1), \ldots, h(\delta_m)). \] (1)

**Proof:** This follows from the fact that each function of the base of \( DL_0 \) except \( \cdot \) 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 \( DL_0 - \{ \cdot \} \).

**The operators \( \uparrow \) and \( \downarrow \):**

**Lemma 5.3:**

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

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

**Lemma 5.4:** 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 cyclic right permutation:**

Recall that $\circ$ represents the cyclic right permutation in $DL$ and is also a construct in $DL_0$. We consider a program $P$ in $DL_0 - \{\circ\}$ 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 $DL_0 - \{\circ\}$ is incomplete:

**Lemma 5.5:** For each $DL_0 - \{\circ\}$-computable function $T:V(D)^k \rightarrow V(D)$ and each bijection $I:D \rightarrow D$ we have

$$I_2(T(\delta_1,\ldots,\delta_k)) = T(I_2(\delta_1),\ldots,I_2(\delta_k))$$

This lemma can be proved by induction on the length of the program. We consider here the fact that the only nonempty relational constant is the 2-ary diagonal and this constant has no influence on components $>2$.

**The singleton operation:**

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

**Lemma 5.6:** Each program $P$ of $DL_0 - Singl$ cannot generate nonrelational directories if all inputs are relations.

**Proof:** Each directory generated from relations by an operation of $DL_0 - Singl$ is again a relation.

**The union operation:**

Recall that $U$ represents the function mapping each set to the union of all its elements. Then the following fact is true:
Lemma 5.7: 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 $QL$.

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

Lemma 5.8: For each function $T$ computable in $DL_0\{-\text{while}\}$ there is a natural number $k$, s.t for each $n$ we have: if $x_1, \ldots, x_m \in V_n(D)$ then $T(x_1, \ldots, x_m) \in V_{n+k}(D)$.

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 $\{x\}$. $\{x\}^0$ we define to be $x$ itself. Let $I:D \rightarrow D$ a bijection. then a relation $r$ is preserved by $I$ iff $I(r) = r$, where $I$ applied to relations is defined canonically.

Lemma 5.9: 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 $DL_0\{-\text{mkdir}\}$.

6. Coding directories by files and the proof of theorem 4.2.

The proof of Theorem 4.2 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 [CH80]. This is the middle step in our proof.

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

At first we define tar:

Given a set directory \(X\) in \(V_2(D)\), let \(n\) be the maximal arity and \(m\) be the maximal product of arity+1 and cardinality of the relations. Then tar\((X)\) is the following \(m+n+3\)-ary relation:

\[
\text{tar}(X) = \{(a^{k+1}, b, a^{m-n+k-1}, b, b, a_1, b, a_2, b, \ldots, b, a_k): a \neq b, \{a_1, \ldots, a_k\} \in X \text{ and of arity } k\}
\]

where \(a^k\) denotes the \(k\)-tuple consisting of \(k\) many \(a\)'s.

Each tuple in tar\((X)\) describes a relation in \(X\). The number of equal components at the beginning describes the arity of the relation. After that a component \(b\) appears and its second appearance says that now the sequence of elements of the relation begins. The sequence of elements is empty iff this tuple codes an empty directory. The sequence of elements consists of one element iff the whole tuple codes 1, the nonempty 0-ary relation. Using this definition of tar we get the following

**Lemma 6.1.1:** There is a computable directory query in DL computing the directory transformation tar such that

(i) The domain Dom(tar) of tar consists of the directories having only relations as their subdirectories.

(ii) tar(Ø) \(\in V_1(D)\) if it is defined.

**Proof:**

(i) We consider the mapping tar as defined before the lemma and will prove that it is DL-computable. For each relation \(R \in X\) we can compute by the completeness of QL the relation

\[
\{(a^{k+1}, b, a^{m-n+k-1}, b, b, a_1, b, a_2, b, \ldots, b, a_k): R = \{a_1, \ldots, a_k\} \text{ and } a \neq b\}
\]

Using the mkdir-construct and the union-operation we can compute tar.

(ii) follows immediately from the definition of tar.

**Lemma 6.1.2:** There is a computable directory query tar\(^{-1}\) \(\in DL\) which is the inverse of tar, i.e. for every \(\delta \in Dom(tar)\) we have tar\(^{-1}\)(tar\((\delta)\)) = \(\delta\).

**Proof:** We construct tar\(^{-1}\)(\(R\)) for each relation \(R\) as follows:

Consider any \(v = (v_1, \ldots, v_m) \in R\). We want to construct the relation \(S(v)\) coded by \(v\). Let \(k+1\) be the number of equal components at the beginning of \(v\). Then \(v\) codes a \(k\)-ary relation if it codes a relation. \(v\) is now of the form \((a^k, b, v^*), a \neq b\). If \(b\) does not appear in \(v^*\) then \(v\) does not code a relation and \(S(v)\) is set \(\emptyset_{\text{dir}}\). Otherwise

...
Let $v = (a^{k+1}, b, v', b, c^*)$. Let $m$ be the length of $c^*$. For the case that $m$ is not divisible by $k+1$ it clearly does not code a relation and $S(v)$ is set again $\emptyset$. Otherwise if $c^* = c_1, \cdots, c_l$ and each $c_i$ is of arity $k+1$ then set $S(v) = \{x: \text{there is an } y \in D, (x, y) = c_i \text{ for some } i = 1, \cdots, l\}$. Note that $v$ codes an empty directory iff $c^*$ is an empty sequence and that $v$ codes 1 iff $k = 0$ and therefore each $c_i$ is of arity 1 and there exists at least one $c_i$. Set $tar^{-1}(R) = \{S(x): x \in R\}$. Then $tar^{-1}$ clearly is an inversion of $tar$. We have now to prove that $tar^{-1}$ is expressible in $DL$. Define for each relation $X$:

$$S'(X) = S(v) \text{ if } X = \{v\} \text{ and } v \text{ codes a relation, and } S'(X) = \emptyset \text{ otherwise.}$$

To be a singleton and to code a relation is decidable in $QL$ because it is decidable and therefore decidable in $DL$. Also $S'$ restricted to $\{v\}$, s.t. $v$ codes a relation is computable in $QL$ and therefore in $DL$. Hence $S'$ is computable in $DL$. Now $tar^{-1}(X) = \{S'(y): y \in Singl(X)\}$ and therefore $tar^{-1}$ is expressible in $DL$.

6.2. The construction of $TAR$.

Using $tar$ we now define $TAR$ recursively on the order of the directory. For a relation $\delta \in V_1(D)$

$$TAR(\delta) = \{(a, a, a, x): a \in D \text{ and } x \in \delta\} \quad (1)$$

In other words, if $\delta$ is a relation we add three arguments to it to make sure that in can be recognized as a single relation. Note that $TAR(\emptyset) = \emptyset$.

For $\delta = \emptyset_D$ we set

$$TAR(\delta) = \{(a, a, a, b): a, b \in D \text{ and } a \neq b\} \quad (2)$$

Thus $TAR(\emptyset_D)$ is coded by a relation in $D^4$ such that the first three arguments are equal and different from the forth argument. 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 arbitrary directories $\delta$ we set

$$TAR(\delta) = \{(a, b, a, x): a, b \in D, a \neq b \text{ and } x \in tar(TAR(\delta), \delta_1 \in \delta)\} \quad (3)$$

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 $TAR$.

Lemma 6.2.1: 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 $\delta$. 

Technion - Computer Science Department - Technical Report CS0454 - 1987
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(δ) of the given directory δ. This is expressible in DL. The leaves (elements without a subdirectory in TC(δ)) are relations or the empty directory. We can compute the set of leaves and call it Z₀. We compute \( P₀ = \{(x,TAR(x)) : x \in Z₀\} \). Here \((x,y)\) means the Kuratowski pair of \( x \) and \( y \) as defined before. We set now \( Z = Z₀ \) and \( P = P₀ \) and as long \( TC(δ) - Z \) is not empty we add to \( Z \) the set \( Y \) of all \( x \), 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(δ). The properties (i) and (ii) follow from the above definition of TAR.

Remark: The proof of lemma 6.2.1 gives us a general scheme, how to describe a recursive procedure in DL-constructs.

Lemma 6.2.2: There is a computable directory query TAR⁻¹ \( \in DL \) which is the inverse of TAR, i.e. for every \( δ \in Dom(TAR) \) we have TAR⁻¹(TAR(δ)) = δ.

Proof: Let \( P \) be same DL-program. Then generally it is possible to calculate the set \( Lₚ(x) \) 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(δ)). (1) If \( r \) is of the form \( \{(a,b,a,\bar{x}) : a \neq b \text{ and } \bar{x} \in s\} \) then set \( T(r) = tar⁻¹(s) \). (2) If \( r \) is of the form \( \{(a,a,a,\bar{x}) : \bar{x} \in s\} \) or \( \{(a,a,a,b) : a \neq b\} \) (\( r \) 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 there is not changed anything 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.

7. 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 this purposes. We take the extended semantic model of [CH80] as our starting point, i.e. \( V_1(D \cup F) \). 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 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 ls-command in UNIX) with all its variations, and the introduction of arithmetical and statistical functions. The set of urelements \( D \), 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.

8. Uniform \( \Sigma_1 \)-definability and \( DL \).

In this section we want to relate our results to set theoretic definability theory. 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 \( HF=\langle HF, \in \rangle \) 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 $\text{HF}(D)$ where $D$ is a finite set of urelements and $\text{HF}(D)=<\text{HF}(D), \in>$ 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 $\text{HF}(D)$ is very similar to the structure $\text{V}(D)=<\text{V}(D), \in>$ in which our language $DL$ operates. So, the question arises whether the computable directory 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:

**Theorem 8.1:** Let $A \subseteq \text{V}(D)$. 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 $D$: $x \in A$ 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.

$A=\{\delta: V(D) \mid \phi(\delta)\}$.

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 $\uparrow, \downarrow, -, \cap$ and $\cap$ of [CH80] and the 2-ary membership relation symbol $\in$ as its nonlogical symbols.

We write $(\forall x \in y)P$ for $\forall x(x \in y \rightarrow P)$ and $(\exists x \in y)P$ 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 8.1):** 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 $\text{V}(D)$, 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 $\text{V}(D)$ is recognizable by a program in $DL$.

First we can prove that $\in$ is decidable by a $DL$-program using the fact that $x \in y$ iff $\{x\} \cup y \neq \emptyset$.

Claim: if $P$ is a predicate, decidable by a $DL$-program then also $(\exists x \in y)P$ is decidable by a $DL$-program.
This claim can be proved by the comprehension scheme, presented in the remark of the chapter 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 arity 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 $M^k$ 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 an arity not greater than 1. Then $Z_{1,k} = \{\varnothing\} \cup \bigcup_{v=1}^{k} Pow(D^v)$ and $Z_{j+1,k} = Z_{j,k} \cup Pow(Z_{j,k})$. It is easily seen, that $Z_k = Z_{k,k}$ 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.

9. 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 [Fe78, Mo74, Mo80, M084, 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 [CH80] 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. In [DM86b] we show how to apply this approach for SETL-like programming languages, and how to draw from this approach also results on 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.

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 DL₀) 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. Projects in this direction are being pursued at the Computer Science Department of the TECHNION - Israel Institute of Technology.

Last but not least our framework allows us to address the fundamental question of computability by discrete mechanical devices as initiated by Gandy [Ga78]. 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 Gandy 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 \((S,F)\) \(S \subseteq V_k(D)\) 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 from 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 notion of efficient computability (in contrast 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 (VLSI). 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 \(\{\cdots\}\), which maps any directory \(\delta\) into the directory \(\{\delta\} = \delta\).
whose only subdirectory is \( \delta \). 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 \(TAR\) to obtain the Gandy machine \((TAR(V(D),TAR\circ \cdot \circ TAR^{-1}))\) 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 [B187] 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 \((TAR(S),TAR \circ F \circ TAR^{-1})\) satisfies II-IV.

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

10. References.


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