E. Dahlhaus, J.A. Makowsky

Computable Directory Queries

The Choice of Programming Primitives for SETL-like Programming Languages

September 1985
Computable Directory Queries.

(August 1985)

E. Dahlhaus$^1$ and J.A. Makowsky$^2$

Department of Computer Science, Technion, 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).

---

$^1$ Visiting from Department of Mathematics, Technical University Berlin, West Berlin sponsored by Minerva Foundation.

$^2$ In the last stages of the paper 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, 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 will be shown in this paper that the introduc-
tion 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.

R. Gandy, in [Ga80], discusses some philosphical aspects of Church’s Thesis which are related to our
work. Gandy postulates four principles concerning frameworks of computability from which, in contrast to
Church’s Thesis, it is provable that functions in these frameworks 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, 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. In particular, Gandy tries to capture
mechanistic aspects of Computation machines, rather than to axiomatize the meaning of computability, as was
initiated in [CH80]. It might be stimulating to reconcile the two approaches to computability.

The main problem we address in this paper is that of defining precisely the semantic notion of a comput-
able 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, com-
plement, powerset, 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
directory $y_i$. 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. The exact relationship with SETL will be dis-
cussed in an appendix.
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 will lead us to an extended directory manipulation language EDL which is discussed 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 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 $i$, $R_i \subset D^k$. $B=(D, R_1, \ldots, R_k)$ 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.

\[
V_1(D) = \bigcup_{i \in \mathbb{N}} P_{f_{\infty}}(D^i)
\]

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

\( V_{f_{\infty}}(D) = V_j(D) \cup P_{f_{\infty}}(V_j(D)) \) 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 data base of higher order (dbho) is an ordered tuple \( B = (D, \Delta_1, \ldots, \Delta_k) \) where each \( \Delta_i \) is a directory in \( V(D) \).

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

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

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

(iii) Otherwise, there is a function \( f: \Delta \rightarrow \Delta^\delta \) 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 labeled directed acyclic graph 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 \( \varnothing_{dc} \). 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 \( \varnothing_k \) of rank \( k \). Two directories are similar if their labeled graphs are isomorphic.

Let \( B = (D, \Delta_1, \ldots, \Delta_k) \) and \( B^\delta = (D^\delta, \Delta_1^\delta, \ldots, \Delta_k^\delta) \) be two dbho's and let \( h: D \rightarrow D^\delta \) a function between the two domains. We define an extension \( \overline{h}: V(D) \rightarrow V(D^\delta) \) in the following way:

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

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

So \( \overline{h}(\delta) \) is a k-ary relation in \( V_1(D^\delta) \).

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

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

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

\( h \) is an isomorphisms from \( B \) into \( B^\delta \) iff \( h \) is one-one and onto and for \( 0 \leq k \leq k \) \( \overline{h}(\Delta_i) = \Delta_i^\delta \).

Two dbho's \( B = (D, \Delta_1, \ldots, \Delta_k) \) and \( B^\delta = (D, \Delta_1^\delta, \ldots, \Delta_k^\delta) \) are similar if each \( \Delta_i \) is similar to \( \Delta_i^\delta \).
Two dbho's $B=(D,\Delta_1,\ldots,\Delta_k)$ and $B'=(D,\Delta'_1,\ldots,\Delta'_k)$ are isomorphic if they are similar and there is an isomorphism $h:B\rightarrow B'$.

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.

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 $\delta_1, \ldots, \delta_k \in V(D)$ we have

$$T(h(\delta_1),\ldots,h(\delta_k)) = \overline{h}(T(\delta_1,\ldots,\delta_k))$$

If we replace $V(D)$ by $Rel(D)$ 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 $P(\delta)$ be the directory containing exactly each subset of subdirectories of $\delta$ as a subdirectory. The transformation $T_{\text{power}}$ which maps $\delta$ into $P(\delta)$ 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
δ as a subdirectory. The transformation $T_∪$ which maps δ into $U(δ)$ is clearly a computable directory transformation.

(vii) Let $R$ be an $n$-ary relation of power $p$. We associate with $R$ a directory $δ$ 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 $δ$. Clearly, this defines a computable directory transformation.

(viii) Let $δ$ be a directory and let $Files(δ)$ be the directory containing exactly the relations of $δ$ as its subdirectories. The transformation $T_{Files}$ which maps $δ$ into $Files(δ)$ is clearly a computable directory transformation.

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

(x) (Kuratowski pair) Set

$$Pair(δ_1, δ_2) = \{\{δ_1\}, \{δ_1, δ_2\}\}$$

Clearly $Pair$ is a computable directory transformation.

(xi) Let $δ$ be a directory and let $HTC(δ)$ be the 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, ... which are in $V_1(D)$ and are denoted by $∅_0, ∅_1, ∅_2, ...$ respectively, the empty directory in $V_2(D)$ which we denote by $∅_{dir}$ 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 $∅_0$ is defined to be the empty relation of arity 0.

(xiii) As in $QL$ we can use 1 as truth value $true$ and $∅_0$ 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 $V_δ(D)$. 

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 self-contained.

Syntax:

The set of terms of DL is inductively defined as follows:

(i) E is a term of QL; for i≥1 yi are terms of QL; if diri is a directory name then diri is a term of DL; if reli is a relation name then it is a term of QL.

(ii) For any terms t1,t2 of QL

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

are terms of QL.

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

(iv) For any terms t1,t2 of DL

\[
\{t_1\}, U(t_1), P(t_1), Singl(t_1), (t_1 \rightarrow t_2), (t_1 \cup t_2)
\]

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 \(\sum_{j} \).

Semantics:

Let \(B=(D,D_1,\ldots,D_k)\) be a dbho.

(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 $States(B)$.

(ii) The meaning of a 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:

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:

- $\mu_0(E)(x) = \{ (x, x) : x \in D \}$,
- $\mu_0(y)(x) = x(y)$,
- $\mu_0(r)(x) = R_{x}$,
- $\mu_0(t_1 \cap t_2)(x) = \mu_0(t_1)(x) \cap \mu_0(t_2)(x)$, if $\mu_0(t_1)(x)$ and $\mu_0(t_2)(x)$ have the same arity, otherwise $\mu_0(t_1 \cap t_2)(x) = \emptyset$.
- $\mu_0(^t(t_1))(x) = \{ (\mu_0(t_1)(x)) \}$, if $\mu_0(t_1)(x)$ is a relation, otherwise it is $\emptyset$. * stands here for $\neg$, $\downarrow$, $\uparrow$ or $\cdot$. The meaning *' of * is complement, projection of all components except of the first, extension of the relation by one last component, or cyclic permutation respectively.

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 $x \in States(B)$:

1. $\mu_0(\{t\})(x) = \{ \mu_0(t)(x) \}$
2. $\mu_0(P(t_1))(x) = \text{Powerset of } \mu_0(t_1)(x)$
3. $\mu_0(U(t_1))(x) = \bigcup \{ \mu_0(t_1)(x) \}$, if all subdirectories of $\mu_0(t_1)(x)$ are relations of the same arity or all subdirectories of it are not relations, otherwise it is set to be $\emptyset$.
4. $\mu_0(t_1 \cup t_2)(x) = \mu_0(t_1)(x) \cup \mu_0(t_2)(x)$, if $\mu_0(t_1)(x)$ and $\mu_0(t_2)(x)$ are both relations of the same arity or both not relations, otherwise it is set to be $\emptyset$.
5. $\mu_0(t_1 \rightarrow t_2)(x) = \mu_0(t_1)(x) - \mu_0(t_2)(x)$

Here $X \rightarrow Y$ is the set of all elements of $X$ not being in $Y$. Note that $X \rightarrow Y$ is a relation of arity $k$ resp. a nonrelational directory iff $X$ is a relation of arity $k$ resp. a nonrelational directory.

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

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

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

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

(c) If $P$ is while $y_j$ do $P_1$ then $\mu(P)(x)$ is defined in the usual way on a sequence of states $z_{t+1} = \mu(P_1)(z_t)$ with $z_0 = z$. $\mu(P)(x)$ is the first $z_t$ such that $z_t(y_j)$ 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

$$\mu(P)(x)(y_i) = \{ \mu(P_i)(x_j)(y_i) : z_i(y_i) \in z(x_j) \}$$

for $l \neq j$ and $z_i(y_i) \in z(x_j)$ if all $z_i(y_i) = z(x_j)$ for $l \neq j$ and $z_i(y_i) \in z(x_j) \mu(P_i)(x_j)(y_i)$ is defined, otherwise $\mu(P)(x)(y_i)$ is undefined.

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 $y_i$ the program $P_1$ and puts into $y_i$ all the results so obtained. If $j > m$ the new directory contains exactly one subdirectory of the form $(y_1, \ldots, y_m)$. Otherwise, the directories $t_1, \ldots, t_{j-1}, y_{j+1}, \ldots$ 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-Frankel set theory.

Queries: Let $B=(D,R_1, \ldots, R_k)$ be a dbho and $z_{\text{initial}}$ be the assignment with $z_{\text{initial}}(y_i) = R_i$ for all $i \leq k$ and $z_{\text{initial}}(y_i) = \emptyset$ for all $i > k$. Given a program $P(y_1, \ldots, y_m) \in DL$ and a variable $y_j$ we look at the function $T_{P,y}(V(D))^k \rightarrow V(D)$ $T_{P,y}(R_1, \ldots, R_k) = \mu(P)(z_{\text{initial}})(y_j)$.

**Theorem 4.1:** For every program $P \in DL$ and each variable $y_j$ the function $T_{P,y}(V(D))^k \rightarrow V(D)$ is a 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-constructor this follows from the following closure property of partial recursive functions:

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

Now let $P$ be of the form

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

To complete the proof we note that $f$ corresponds to the program $P_1$, $g(b)$ to $y_0$, $b$ 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 $T: V_1(D)^n \to V_1(D)$ there is a program $P \in QL$ 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.

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

$((T\downarrow), ((T\uparrow), ((T)\neg), E, U(T), Sing(T), (T_1\cdots T_2))$.

Remark:

(1) Generally, we can simulate the conditional statement by the while-construct. Consider

if $y_1 = \emptyset$ 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:

$y_j : = y_j \cup y_k : = \emptyset$ ;

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 $QL_0$. 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 $\delta_1$, s.t. $P(\delta_1)$ is expressible in $DL$:

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

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

Lemma 4.4: There is a program 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 \( F: \text{Singl}(D^3) \times V(D) \to V(D) \) defined as follows:

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

By the remark above \( F \) is computable in \( DL_0 \), because \( x=y \) means \( \{(x,y)\} \to E = \emptyset \). Using the \texttt{mkdir} construct the function which computes for each \( a \) and \( b \) the set \( \{F(u,a,b) : u \in \text{Singl}(D^3)\} = \{a,b\} \) is computable in \( DL_0 \).

Lemma 4.5: There is a program \texttt{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: \texttt{join}(\( \delta_1, \delta_2 \)) is the program \texttt{U(pair}(\( \delta_1, \delta_2 \))).

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

Proof: \texttt{join}(\( \delta, \text{pair}(\delta, \delta) \)) is an empty directory or relation iff \( \delta \) is a relation. Moreover we get following

Lemma 4.7: 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 \texttt{join}(\( R, Y_1 \)) is empty we set \( Y_1 := (Y_1)^\uparrow \). 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

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

in the case \( j > m \).

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

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

We conclude this section with some examples which will be used over and over again 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 Kuratowsky-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 $p_2(p)$ is $U(p \rightarrow \pi_1(p))$.

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

$P = z_i:=U(x \in z_* \in V_1(D)); y_i:=y \cup z; z:=z_i; y:=y_i.$

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

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

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 $DL_0 - \{\rightarrow\}$ and each surjective map $h:D \rightarrow D_1 : (1)$

$h(T(\delta_1, \ldots, \delta_m)) = T(h(\delta_1), \ldots, h(\delta_m))$.

Proof: This follows from the fact that each function of the base of $DL_0$ except $\rightarrow$ has this faithfulness property
(1) and for each directory δ we have δ is empty iff h(δ) 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_{\neg}(\neg).

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 \(DL_{\neg}(\downarrow)\) 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 \(DL)\). We get the following fact:

Lemma 5.4: All directories generated by a program of \(DL_{\neg}(\neg E)\) and an input with only empty leaves have only empty leaves.

Proof: All operations of \(DL_{\neg}(E)\) preserve emptyness of each leaf.

The cyclic right permutation:

Recall that \(\neg\) represents the right permutation in \(QL\) and is also a construct in \(DL_\neg\). We consider a program \(P\) in \(DL_{\neg}(\neg)\) 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

\[I_2(S) = \{(x_1,x_2,\ldots,x_k) : (x_1,\ldots) \in S\}\]

\(I_2\) is extended to directories in the canonical way. We apply the bijection here on all \(k\)-th components with \(k\geq 2\).

Then the following fact proves that \(DL_{\neg}(\neg)\) is incomplete:

Lemma 5.5: For each \(DL_{\neg}(\neg)\)-computable function \(T:V(D)^k \rightarrow V(D)\) and each bijection \(I:D \rightarrow D\) we have
\[ I_2(T(\delta_1, \cdots, \delta_k)) = T(I_2(\delta_1), \cdots, I_2(\delta_k)) \]

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

**The singleton 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

**Lemma 5.6:** Each program $P$ of $DL_0$-$\text{Singl}$ cannot generate non-relational directories if all inputs are relations.

*Proof:* Each directory generated from relations by an operation of $DL_0$-$\text{Singl}$ 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 $V_1(D)$. 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 non-relational 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 $\text{mkdir}$:

Recall that $\text{mkdir}$ 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. $\{x\}^0$ 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 $\langle D, \delta_1, \cdots, \delta_k \rangle$.

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

For example the powerset is not computable in $DL_0 \setminus \{\text{mkdir}\}$.

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 $\text{TAR}$ and $\text{TAR}^{-1}$. $\text{TAR}$ 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 $\text{TAR}^{-1}$. The difficulty in writing $\text{TAR}$ in $DL$ comes from the fact that we may not use names and other information of the directory structures. The programs $\text{TAR}$ and $\text{TAR}^{-1}$ allow us to reduce our completeness proof to the completeness proof for $QL$ in [CH80]. This is the middle step in our proof.

6.1. Construction of $\text{tar}$.

To construct $\text{TAR}$ and $\text{TAR}^{-1}$ we define at first a function $\text{tar}$, which maps directory of $V_2(D)$ to one 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 $V_2(D)$, let $n$ be the maximal arity and $m$ be the maximal product of arity+1 and power of the relations. Then $\text{tar}(X)$ is the following $m+n+3$-ary relation:

$$\text{tar}(X) = \{ (a^{k_1}, b, a^{m+n-k_1}, b, a_1, \cdots, a_i) : a \neq b, \{a_1, \cdots, a_i\} \in X \land \text{arity of arity } k \}$$

Each tuple in $\text{tar}(X)$ describes a relation in $X$. The number of equal components at the beginning describes the arity of the relation. After that appears a component $b$ 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. Using this definition of $\text{tar}$ we get the following.

Lemma 6.1.1: There is a computable directory query $\text{tar} \in DL$ such that

(i) The domain $\text{Dom}(\text{tar})$ of $\text{tar}$ consists of the directories having only relations as their subdirectories.

(ii) $\text{tar}(\delta) \in V_2(D)$ if it is defined.
Proof:

(i) We consider the mapping \( \text{tar} \) 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

\[
\{ (a^k, b, a^*_{m-k+1}, b, a_1, \ldots, a_l) : R(a_1, \ldots, a_l) \text{ and } a \neq b \}
\]

Using the \textit{mkdir}-construct and the union-operation we can compute \( \text{tar} \).

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

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

Proof: We construct \( \text{tar}^{-1}(R) \) 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 an \( S(v) \) is set \( \emptyset_{dir} \). Otherwise \( v=(c^{k+1}, v^*, b, c^*) \). 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 \( \emptyset_{dir} \). Otherwise if \( c=c_1, \ldots, c_l \) and each \( c_i \) is of arity \( k+1 \) then set \( S(v)=\{ x : \text{there is an } y \in D, (x, y)=c_i \text{ for some } i=1, \ldots, l \} \). 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 \( \text{tar}^{-1}(R) = \{ S(x) : x \in R \} \). Then \( \text{tar}^{-1} \) clearly is an inversion of \( \text{tar} \). We have now to prove that \( \text{tar}^{-1} \) is expressible in \( DL \).

Define for each relation \( X \)

\( S'(X)=\emptyset_{dir} \) if \( X \) is not a singleton, or if \( X=\{v\} \) and \( v \) does not code a relation, and \( S'(X)=S(v) \) if \( X=\{v\} \) and \( v \) does code a relation. 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 \( \text{tar}^{-1}(X) = \{ S'(y) : y \in \text{Sing}(X) \} \) and therefore \( \text{tar}^{-1} \) is expressible in \( DL \).

6.2. The construction of \( \text{TAR} \).

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

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

(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 \( \text{TAR}(\emptyset_{dir}) = \emptyset_{k+3} \). 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

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

(2)

Thus $TAR(\emptyset_{dir})$ 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 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

$$TAR(\delta) = \{(a,b,a,x) : a, b \in D, a \neq b \text{ and } x \in \text{ var}(\{TAR(\delta_i) : \delta_i \in \delta\})\}$$

(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 $TAR$.

**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 $\delta$.

**Proof:** We consider the function $TAR$ as described before the Lemma. 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 $TC(\delta) - Z$ 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 $TAR(\delta)$. 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 $\delta \in Dom(TAR)$ we have $TAR^{-1}(TAR(\delta)) = \delta$.

**Proof:** Let $P$ be same $DL$-program. Then generally it is possible to calculate the set $L_2(\delta)$ 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 $\{(a,b,a,x) : a \neq b \text{ and } x \in s\}$ then set $T(r) = TAR^{-1}(s)$. (2) If $r$ is of the form $\{(a,a,a,x) \in s\}$ 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 anymore. 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 [CL80] 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 is 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 $HF(\emptyset) = (HF(\emptyset), e >$ with universe the hereditary finite sets without urelements and membership as its only relation. In $HF(\emptyset)$ the $\Sigma_1$-definable sets are exactly the recursive enumerable sets.

The notion of $\Sigma_1$-definability has a natural meaning also in the structures $HF(D)$ where $A$ is a finite set of urelements. The structure $HF(D)$ is very similar to the structure $V(D) = (V(D), e >$ 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 $A \subset V(D)$. Then the following statements are equivalent:

(i) $A$ is recursive 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.

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

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 is just theorem 4.2.

We consider formulas using the function symbols $\uparrow$, $\downarrow$, $\neg$, $\cap$ 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 $V(D)$, 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 $\epsilon$ 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 $(\exists x \in y)P$.

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 $\psi$.

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 [DM85b] 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 \textit{DL} will be delayed to future research. It seems clear that various results of [CH82, Im82, Im83, HP84, DM85b] 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 \textit{DL} and the independence of the constructs of \textit{DL}_0) 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.

10. References.


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


The choice of programming primitives for SETL-like programming languages.

(Extended abstract, August 1985)

E. Dahlhaus\textsuperscript{3} and J.A. Makowsky
Department of Computer Science, Technion, Haifa, Israel

Abstract: We discuss the choice of programming primitives for set oriented programming languages such as SETL. For this purpose we introduce a mathematical model (hereditarily finite sets with urelements). In this model criteria for the choice of programming primitives are defined. The criteria are complexity, independence and computational completeness of the basic constructs. We propose primitives satisfying our criteria and also discuss briefly the possibility of defining abstract data types within our mathematical model. We give a characterization of the data types whose objects are recognizable in NP. Our work is a synthesis of several approaches previously introduced in other frameworks, such as query languages, generalized recursion theory and high level programming language design.

\textsuperscript{3} Visiting from Department of Mathematics, Technical University Berlin, West Berlin sponsored by Minerva Foundation.
1. Introduction

Programming languages which manipulate higher order objects have been considered for a long time, e.g. [Sch75] and [Ho83] for a historic survey. 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.

The question we want to discuss here is the choice of programming primitives for SETL-like programming languages. To make such a discussion reasonable we have to choose first a mathematical model for the meaning of our programming languages. Within such a model we then can address the following issues:

(1) Each program has to compute a "computable" function. In other words we have to relate our model to a traditional model of computability. There are various attempts in the literature to generalize the notion of computability (over natural numbers or strings) to arbitrary objects, see [Sh73, Mo74, Mo84, Ga80] among others, and especially [Fe80] for a survey.

(2) Once we have identified the computable functions for our objects we can define computational completeness of a programming language by requiring that all computable functions can be represented by a program, cf [CH80].

(3) The next criterion in the choice of programming primitives should be the low complexity of the basic operations and constructs. This will, to a certain extent depend on the exact model of computation (RAM, Turing Machines, etc) but it is safe to require that the basic programming primitives be polynomial in time or space. For iterative and parallel constructs the question of complexity is a bit more delicate. The best one can hope for naturally is some form of additivity: A while-construct has very little overhead and takes as much time is the sum of its iterations; for a parallel construct one would require the same for the parallelly executed subprograms. In other words, the overhead of such constructs should be uniformly small.
If one thinks of implementation of higher order languages one has to address the issue of redundancy in
the programming primitives. Though independence of the constructs is not a value in itself, it is still
significant to have an independent basic set of programming primitives available. Other primitives can
then be added depending on the particular programming applications or computer architecture features one
has in mind.

Since we are dealing with very high level languages one has also to address their inherent data abstraction
possibilities and their naturalness of expression. It should not be the case that the complement of a finite
set is computable only by a rather awkward program, as is the case in [Mo84] or that simple recursions
can only be expressed by invoking a universal function or machine as in [No78].

R. Gandy, in [Ga80], discusses some philosophical aspects of Church’s Thesis which are related to our
work. Gandy postulates four principles concerning frameworks of computability from which, in contrast to
Church’s Thesis, it is provable that functions in these frameworks 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, which also form the background of our work
here. The computable transformations, introduced in this paper, however, do not satisfy all of Gandy’s prin-
ciples. This shows, that not all computable functions satisfy Gandy’s principles. In particular, Gandy tries to cap-
ture mechanistic aspects of Computation machines, rather than to axiomatize the meaning of computability, as
was initiated in [CH80]. It might be stimulating to reconcile the two approaches to computability.

The main problem we address in this paper is that of defining precisely the semantic notion of a *comput-
able object transformation*. This is the content of section 2 and 3. Our mathematical model are the *heredi-
tarily finite sets with urelements*. Finite objects are usually built from some, not further specified, atoms such as ver-
tices in graphs or the components of tuples in relational data bases. These unspecified elements are suitably
modelled by the old (and in times unpopular) urelements of classical set theory (cf. [Ba75]). From these atoms
the objects then are built in a structured way with the only restriction that at every step of a construction only
finitely many objects are involved and that no objects can be infinitely decomposed. Infinite collections of
objects, therefore, are *not objects*. If they have to be considered they have to be treated like *classes* in set
theory. The computable classes of objects can be viewed as *abstract data types*. In this paper we shall discuss
abstract data types as *computable classes of objects*. Special attention should be given to classes whose objects
are recognizable (deterministically or non-deterministically) in polynomial time or space.
In this framework programs take as input some objects (from a given data type) and transform them into other objects (of some other data type). With such a definition one can now define the semantics of various object transformation languages. A object transformation language $L$ is complete if for every computable object transformation there is an expression (program) in $L$ corresponding to it.

In section 4 we define a object transformation language $OTL$ which is complete. $OTL$ has a very small number of basic object handling constructs. They correspond to the set theoretic operations union, complement, powerset, unordered pair and the replacement and induction principle. The induction principle occurs in the form of the while-construct. The replacement principle leads to a new programming construct

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

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 subobjects of $y_k$ simultaneously and puts them into the object $y_i$. The construct also allows parallel transformation processing to be expressible in DL. As mentioned before, the programming language $OTL$ turns out to be an abstract and well defined sublanguage of SETL which is equivalent to SETL both in computing power and flexibility. We also analyze the constructs of $OTL$ and exhibit an independent (non redundant) subset $OTL_0$ of $OTL$ which is of the same expressive power. At the end of section 4 we state our main mathematical results, which stand here to illustrate the suitability of our approach.

In section 5 we give a sketch of our proofs. For this we have to discuss the relationship between computable object transformations and various set theoretic definability concepts. This section is more of foundational interest than computational relevance. It relates computability in hereditarily finite sets over urelements to $\Sigma_i$-definability in the sense of A. Levy [Le65].

In section 6 we discuss the definition of abstract data types in our model from a complexity point of view. We introduce a programming language of while-free programs $TROT$, and of while-free programs with transitive closure $TROL(HTC)$, and associate with it a specification languages $\Sigma_i^p$ and $\Sigma_i^p(HTC)$ respectively. Our main results are that all programs in $TROT$ are polynomial-time computable and that the classes of objects which are in NP are exactly the $\Sigma_i^p(HTC)$-definable classes. Other characterizations are also given. This shows that our choice of programming and specification constructs satisfies also our complexity requirements.
2. The semantic model.

The purpose of this section is to define finite objects of higher order. The simplest objects are the urelements (or atoms), which are all isomorphic to each other and have no elements. On the next level we have finite sets of urelements, which are isomorphic if and only if they have the same cardinality. More complicated objects can be formed by allowing objects to contain finite sets of both urelements and objects of lower order.

More formally, we start our definition similar as in [DM85]. Let $U$ denote a fixed countable set, called the universal domain, which is an infinite set of urelements. Let $D \subseteq U$ be finite and nonempty.

We now define inductively our objects over $D$:

The objects of order 0 are exactly the elements of $D$ (the urelements).

The objects of order 1 are the finite sets of objects of order 0 (including the empty set $\emptyset$).

The objects of order $n+1$ are exactly the finite sets of objects of order smaller or equal to $n$.

We denote the collection of objects of order $n$ by $V_n(D)$. We also write $V(D) = \bigcup_{n \in \mathbb{N}} V_n(D)$ for the collection of all objects. Since $V_n(D)$ is an element of $V_{n+1}(D)$ the collection of objects of order $n$ is itself an object (i.e. a hereditarily finite set). $V(D)$ itself is not an object, since it is infinite. We shall call collections of objects which are not objects classes. (This follows the tradition of axiomatic set theory adapted to finite sets.)

Let $h: D \to D^\#$ a function between two domains. We define an extension $\overline{h}: V(D) \to V(D^\#)$ in the following way:

(i) For urelements $d \in V_0(D)$ we put $\overline{h}(d) = h(d)$;

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

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

Two objects $\Delta$ over a domain $D$ and $\Delta^\#$ over a domain $D^\#$ are isomorphic if there is a one-one and onto map $h: D \to D^\#$ such that $\overline{h}(\Delta) = \Delta^\#$.

In traditional set theory without urelements two sets are isomorphic iff they are equal (by their extensions). Once we introduce urelements, this is no longer true.

For the complexity considerations in section 6 we need still a stronger definition:

Definition:

(i) Let $Ob$ be an object. We define inductively the subobjects of depth $n$ for $n \geq 0$ a natural number. The subobjects of depth 1 are the subobjects $Ob_1 \in Ob$. The subobjects of depth $n+1$ are the subobjects $Ob_2$ which are
elements of the subobjects of depth $n$ but which are not subobjects of depth $n$ by themselves. 

(ii) Let $Ob$ be an object and $n \geq 0$ a natural number. We define a modified object $Ob \upharpoonright n$ by replacing distinct subobjects of depth $n$ of $Ob$ by distinct new urelements.

(iii) Let $n \geq 0$ be a natural number. We say that two objects $Ob_1, Ob_2$ are $n$-isomorphic if $Ob_1 \upharpoonright n = Ob_2 \upharpoonright n$.

3. Computable object transformations.

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

$$T(\overline{k}(\delta_1), \ldots, \overline{k}(\delta_k)) = \overline{k}(T(\delta_1, \ldots, \delta_k))$$

This definition is analogous to 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 object transformation is computable if it is computable using the standard coding.

Examples:

(i) Let $\delta$ be an object and let $\{\delta\}$ be the object containing $\delta$ as its only subobject. The transformation $T_{\text{singleton}}$ which maps $\delta$ into $\{\delta\}$ is clearly a computable object transformation.

(ii) Let $\delta_1$, $\delta_2$ be two objects and let $\delta_1 \cup \delta_2$ be the object which contains exactly the subobjects of $\delta_1$ and $\delta_2$ as its subobjects. The transformation $T_{\cup}$ which maps $\delta_1$ and $\delta_2$ into $\delta_1 \cup \delta_2$ is clearly a computable object transformation.

(iii) Let $\delta_1$, $\delta_2$ be two objects and let $\delta_1 \rightarrow \delta_2$ be the object which contains exactly the subobjects of $\delta_1$ which are not in $\delta_2$ as its subobjects. The transformation $T_{\text{difference}}$ which maps $\delta_1$ and $\delta_2$ into $\delta_1 \rightarrow \delta_2$ is clearly a computable object transformation.

(iv) Let $\delta$ be an object and let $P(\delta)$ be the object containing exactly each subset of subobjects of $\delta$ as a subobject. The transformation $T_{\text{power}}$ which maps $\delta$ into $P(\delta)$ is clearly a computable object transformation.

(v) Let $\delta$ be an object and let $U(\delta)$ be the object containing exactly each subobject of a subobject of $\delta$ as a subobject. The transformation $T_{\cup}$ which maps $\delta$ into $U(\delta)$ is clearly a computable object transformation.

(vi) Let $\delta$ be an object and let $Ur(\delta)$ be the object of order 1 containing exactly the urelements which are leaves of $\delta$ as its subobjects. The transformation $T_{\text{ur}}$ which maps $\delta$ into $Ur(\delta)$ is clearly a computable object transformation.
(vii) (Kuratowski pair) Set

\[ \text{Pair}(\delta_1, \delta_2) = \{ \{ \delta_1 \}, \{ \delta_1, \delta_2 \} \} \]

Clearly \textit{Pair} is a computable object transformation.

(viii) Let \( \delta \) be an object and let \( \text{HTC}_0(\delta) \) be the the set of all objects and relations, which are in its \textit{transitive closure under membership} (the hereditary transitive closure). Now we put \( \text{HTC}(\delta) \) to be \( \text{HTC}_0(\delta) \cup \{ \delta \} \). Then clearly \( \text{HTC} \) is a computable object transformation. Note that \( \text{HTC} \) preserves isomorphisms but, in contrast to the previous examples, for no \( n > 0 \) does it preserve \( n \)-isomorphisms.

(ix) We can use \( \{ \emptyset \} \) as truth value \textit{true} and \( \emptyset \) as truth value \textit{false}. This allows us to define \textit{computable predicates} as object transformations whose value are \textit{true} or \textit{false}. The collection of objects which satisfy a computable predicate generally is a class. Computable predicates can be used to define auxiliary data structures such as \textit{Natural numbers, lists, stacks} etc.

(x) To give one example of a auxiliary data structure we shall define the set of natural numbers with their successor, following the classical representation of (finite) ordinals attributed to J.v. Neumann. Zero is the empty set. If \( A \) is a natural number then \( s(A) = A \cup \{ A \} \) is the successor of \( A \). Clearly \( s(A) \) is a computable object transformation. The traditional definition of the natural numbers is the smallest class containing the empty set and being closed under the successor (Dedekind). This definition involves a fixed point. We now give a fixed-point-free definition of the natural numbers: An object \( Ob \) is \textit{transitive} if whenever \( x \in Ob \) then \( x \) is a subset of \( Ob \). \( Ob \) is \textit{connex} if whenever \( x, y \in Ob \) then either \( x \in y \), \( x = y \) or \( y \in x \). \( Ob \) is \textit{urelement-free} if no member of \( Ob \) is an urelement. It is folklore knowledge in set theory that the ordinals are exactly the class of transitive, connex and urelement-free sets. Here, the natural numbers are exactly the ordinals, since all our sets are hereditarily finite. It will follow from the Definability theorem in section 5 that the class of natural numbers is computable.

We shall discuss computable data structures in more detail in section 6.

The examples (i)-(v) (or slight variations thereof) will be among the basic constructs of our object transformation language \textit{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 an object is in \( V_k(D) \).
4. The object transformation language OTL.

The object transformation language OTL we define is essentially a programming language computing finite higher order objects over some finite domain.

4.1. Language definition: Syntax.

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

(i) $D$ is a term of OTL; for $i \geq 1$, $y_i$ are terms of OTL; if $Ob_i$ is an object name then $Ob_i$ is a term of OTL.

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

$$\{t_1, t_2\}, \ U(t_1), \ P(t_1), \ Singl(t_1), \ (t_1 - t_2), \ (t_1 \cup t_2)$$

are terms of OTL.

The set of programs of OTL is inductively defined as follows:

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

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

(iii) If $P$ is a program of OTL then

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

is a program of OTL. The variable $y_j$ occurs here as a bounded variable similar to $j$ in $\sum_{j} a_j$.

4.2. Language definition: Semantics.

Let $D$ be a finite set of urelements and $V(D)$ be the class of objects over $D$.

(i) Let $z$ be a function from the variables $y_1, y_2, \ldots$ into $V(D)$, the set of objects over $D$. We call such a function a object assignment over $D$ or assignment for short. We think of the set of all object assignments over $D$ as the set of states for our object transformation. We denote this set by $\text{States}(D)$.

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

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

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

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

Let $t_1$ and $t_2$ be terms in OTL. Then for each $z \in \text{States}(D)$:

(1) $\mu_0([t_1, t_2])(x) = [\mu_0(t_1)(x), \mu_0(t_2)(x)]$. 

(2) \( \mu_0(P(t_1))(x) = \text{Powerset of } \mu_0(t_1)(x) \)

(3) For \( \mu_0(U(t_1))(x) \) we distinguish three cases:

(i) If all subobjects of \( \mu_0(t_1)(x) \) are different from urelements then we set \( \mu_0(U(t_1))(x) = \bigcup(\mu_0(t_1))(x) \); (3i) If \( \mu_0(t_1)(x) \) has at least two subobjects we take the union as in (3i) treating its subobjects which are urelements as empty sets.

(iii) If \( \mu_0(t_1)(x) \) has exactly one subobject which is an urelement \( U_r \) then \( \mu_0(U(t_1))(x) = U_r \).

(4) \( \mu_0(t_1 \cup t_2)(x) = \mu_0(t_1)(x) \cup \mu_0(t_2)(x) \). If \( \mu_0(t_1)(x) \) or \( \mu_0(t_2)(x) \) is an urelement then \( \mu_0(t_1 \cup t_2)(x) \) is empty.

(5) \( \mu_0(t_1 \setminus t_2)(x) = \mu_0(t_1)(x) \setminus \mu_0(t_2)(x) \)

Here \( X \setminus Y \) is the set of all elements of \( X \) not being in \( Y \).

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

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

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

(b) If \( P = P_1; P_2 \) then \( \mu(P)(x) = \mu(P_2)(\mu(P_1)(x)) \). This is the usual composition of functions.

(c) If \( P \) is while \( y_j \) do \( P_1 \) then \( \mu(P)(x) \) is defined in the usual way on a sequence of states \( z_1 = \mu(P_1)(x_1) \) with \( z_0 = x \). \( \mu(P)(x) \) is the first \( z_i \) such that \( z_i(y_j) \) is not an empty relation or object.

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

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

In words this says, for the case \( m = j = 1 \), that the new object \( y_i \) is obtained in the following way: one applies in parallel to all the subobjects \( y_j \) of \( y_k \) the program \( P_1 \) and puts in to \( y_i \) all the results so obtained. If \( j = m \) the new object contains exactly one subobject of the form \( t(y_1, \ldots, y_m) \). Otherwise, the objects \( y_1, \ldots, y_{j-1}, y_{j+1}, \ldots \) 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-Frankel set theory.

4.3. Main results of constructs.

Let \( Ob_1, \ldots, Ob_k \) be \( k \) given objects and let \( z_{\text{initial}} \) be the assignment with \( z_{\text{initial}}(y_i) = Ob_i \) for all \( i \leq k \) and \( z_{\text{initial}}(y_i) = \emptyset \) for all \( i > k \). Given a program \( P(y_1, \ldots, y_k) \in OTL \) and a variable \( y_j \) we look at the function

\[ T_{P; V(D)^k} \rightarrow V(D) \ T_{P; (Ob_1, \ldots, Ob_k)}(y_j) = \mu(P)(z_{\text{initial}})(y_j). \]
Theorem (Computability of OTL-programs): For every program $P \in OTL$ and each variable $y_j$ the the function $T_{p, y_j}^k V(D)^k \to V(D)$ is a computable object transformation.

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

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

Now let $P$ be of the form $\text{mkob } y_j \text{ from } y_k \text{ by } P_1(y_1, \ldots, y_m)$.

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

Theorem (Completeness of OTL): The object transformation language OTL is complete, i.e. for every computable object transformation $T$ there is a program $P_T \in OTL$ 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 $OTL_0$ of OTL which has an independent set of constructs. The proof of the independence will be sketched in section 5.

Definition: Let $OTL_0$ be obtained from $OTL$ by restricting its definition to the constructs while and mkob together with

$$
D, \ U(t), \ \{t_1, t_2\}, \ (t_1 \leftarrow t_2).
$$

Theorem (Independent constructs for OTL):

(i) $OTL_0$ is complete (and therefore of equal computational power as $OTL$).

(ii) For each construct $c$ of $OTL_0$ there is a computable object transformation $T$ which is not expressible in $OTL - \{c\}$.

To evaluate the complexity of the basic constructs of $OTL_0$ we would have to define precisely the computational model in which we count operations. But it is easy to verify that, in any reasonable machine model (RAM or various notions of Turing machines etc), the basic operations are polynomial time computable and that the while-construct and the mkob-construct satisfy our additivity criterion.
5. Outline of proofs.

In this section we want to outline the proofs of our two main theorems, completeness and the independence of the constructs. For this we have 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, Mo80]. The pioneer paper for models of set theory is [Levy]. 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 $\mathbb{HF}(\emptyset) = \langle \mathbb{HF}(\emptyset), \in \rangle$ with universe the hereditary finite sets without urelements and membership as its only relation. In $\mathbb{HF}(\emptyset)$ the $\Sigma_1$-definable sets are exactly the recursive enumerable sets.

The notion of $\Sigma_1$-definability has a natural meaning also in the structures $\mathbb{HF}(D)$ where $D$ is a finite set of urelements. The structure $\mathbb{HF}(D)$ is the tuple $\langle V(D), \in \rangle$ in which our language $OTL$ operates. So, the question arises whether the computable object transformations 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 (Definability theorem for computable classes): Let $A$ be a class in $V(D)$. Then the following statements are equivalent:

(i) $A$ is recursive enumerable and isomorphism invariant (that means for each natural extension $h$ of a bijection from $D$ to $D$: $x \in A$ iff $\bar{h}(x) \in A$);

(ii) $A$ is recognizable by a $OTL$-program;

(iii) $A$ is $\Sigma_1$-definable, that means, there is a $\Sigma_1$-formula $\phi$, s.t.

$$A = \{ \delta: \mathbb{HF}(D) \models \phi(\delta) \}.$$ 

Note that (i) just states that the characteristic function of $A$ is a computable object transformation, and (ii), that the characteristic function of $A$ is the meaning of a $OTL$-program. Therefore, their equivalence is just the completeness theorem.

We consider first order formulas with equality using the binary membership relation symbol $\in$ and a constant symbol $D$ as its only nonlogical symbols.
We write $(\forall x \in y) P$ for $\forall x (x \in y \to P)$ 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 the definability theorem):** We will prove (i) $\to$ (iii) and (iii) $\to$ (ii). (ii) $\to$ (i) is trivial.

(i) $\to$ (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) $\to$ (ii): We want to prove that each $\text{SIMA}_1$-expressible subset of $V(D)$ is recognizable by a program in OTL. At first we can prove that $\epsilon$ is decidable by a OTL-program $(x \in y \iff \{x\} \cup y \neq \emptyset)$. Claim: if $P$ is a predicate, decidable by a OTL-program then also $(\exists x \in y) P$.

This claim can be proved by the following comprehension scheme in OTL:

Given a computable predicate $P$ in OTL, then the function $G$ which maps each object $\delta$ to the set of its subobjects $\delta_1$, s.t. $P(\delta_1)$ is expressible in OTL.

To see this we observe:

(i) Let $H$ be the function which maps each $\delta_1$ to its singleton if $P(\delta_1)$ and to the empty set otherwise. $H$ is obviously expressible in OTL if $P$ is in OTL.

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

From this comprehension scheme it follows easily that each $\Sigma_0$-predicate is decidable by a OTL-program.

Now we have to consider a $\Sigma_1$ formula $\exists x \psi$. We get a computable object transformation which computes for each for each natural number $k$ the set $V_k(D)$. We only have to write a OTL-program which computes the smallest $V_k(D)$, which has an $x$ satisfying $\psi$. The class of natural numbers is computable in OTL since it is $\Sigma_0$-definable as one can see by looking again at example (X) of section 3.

**Proof of the Independence Theorem:**

The proof of the independence of the constructs of OTL is always based on variations of the same idea: Omitting a construct adds to the resulting sublanguage a closure property which does not hold for OTL. For example, to show that the complement $\neg$ is needed we observe that complement-free programs commute with
homomorphic images of the domain. The least non-trivial result here is the necessity of the \texttt{mkob-construct}. Our proof uses the fact that we do not have the full power set operation among our basic constructs. The full power set, however, can be obtained using \texttt{mkob} and construct of unordered pairs \{,\}. To see that the construct of unordered pairs \{,\} cannot be omitted we observe that without it the class of constant terms has the following property: Given an interpretation of a constant term then all its elements are isomorphic.

6. Polynomially recognizable classes and abstract data types.

In this section we want to describe the mechanism in herent in SETL-like languages for the specification of abstract data types. Before we present the reader with a formal definition we want to discuss a motivating example.

Assume we want to introduce a new data type \texttt{GRAPH}(D) of finite undirected graphs with a bounded number of unspecified vertices. For this purpose we think of the vertices as urelements and the edges as unordered pairs of vertices. The class of graphs \texttt{GRAPH}(D) then consists of all objects of the form \langle V,E \rangle where \( V \) is a subset of \( D \) (the urelements) and \( E \) is a subset of the unordered pairs of \( V \) (denoted by \([V]^{2}\)).

If we want to look at the class of numbered graphs \texttt{NGRAPH} we would require additionally that \( V \) is an initial segment of the class \texttt{NAT} of natural numbers, as introduced in example (x) of section 2.

It is easy to verify that data types defined in this way have the property that their objects are polynomially recognizable. The same holds for all the other common data types such as \texttt{LISTS}, \texttt{STACKS}, \texttt{TREES}, etc. The purpose of this section is to define a specification language for data types recognizable non-deterministically in polynomial time. Let \( C \) be a class of objects. We shall write \( C \) is in \texttt{P} (\( C \) is in \texttt{NP}) if \( C \) is recognizable (non-deterministically) in polynomial time. Here the size of an object is given by the number of elements in its transitive closure \texttt{HTC}.

First we observe the following:

**Proposition:** Let \( C \) be a class of objects which is \( \Sigma_0 \)-definable, i.e. there is a \( \Sigma_0 \)-formula \( \phi(x) \) such that \( \phi \) is the collection of objects satisfying \( \phi \). Then \( C \) is in \texttt{P} and there is a natural number \( n>0 \) such that \( C \) is close under \( n \)-isomorphisms.

To characterize the classes of objects in \texttt{NP} we define the while-free terms of \texttt{OTL}_0 and denote their \texttt{TROTIL} (terms and replacement of \texttt{OTL}).
(1) The terms of OTL₀ are in TROTₐ.

(2) If t₁,t₂ are terms of TROTₐ and yᵢ is a variable then MKOB(t₁,t₂,yᵢ) is a term of TROTₐ.

The semantics of TROTₐ is defined as for OTL₀ with the difference that MKOB is a term construct replacing mkob. The interpretation of MKOB(t₁,t₂,yᵢ) is the object interpreting yᵢ in

\[ \text{mkob } yᵢ \text{ from } yⱼ \text{ in } t₂ \text{ by } t₁. \]

We also define the set of terms TROTₐ(HTC) which is obtained from TROTₐ by closing it under the operation HTC as defined in example (viii) of section 3.

Proposition: Every term in TROTₐ(HTC) represents a polynomial time computable object transformation.

We next define the specification language Σ₀ⁿ and Σ₀ⁿ(HTC).

(1) If t₁,t₂ are terms of TROTₐ (TROTₐ(HTC)) then t₁ ∈ t₂ and t₁=t₂ are formulas of Σ₀ⁿ. (Σ₀ⁿ(HTC)).

(2) If φ and ψ are formulas of Σ₀ⁿ (Σ₀ⁿ(HTC)), then also φ ∧ ψ, φ ∨ ψ, ¬φ are formulas of Σ₀ⁿ (Σ₀ⁿ(HTC)).

(3) If t₁ is a term of TROTₐ (TROTₐ(HTC)) and φ is a formula of Σ₀ⁿ (Σ₀ⁿ(HTC)), then \( ∃ yᵢ ∈ t₁φ \) is also a formula of Σ₀ⁿ (Σ₀ⁿ(HTC)).

The semantics of Σ₀ⁿ-formulas and Σ₀ⁿ(HTC)-formulas is obvious.

Proposition:

(i) Every class C definable in Σ₀ⁿ(HTC) is in P.

(ii) There are classes C in P which are not Σ₀ⁿ(HTC)-definable.

(iii) For every class C definable in Σ₀ⁿ there is a natural number n>0 such that C is closed under n-isomorphisms.

Proof: (i) is obvious and (ii) can be proved in a similar way as the statement in [CH82] that there are polynomial time queries which are not first order definable. To see (iii) we observe that the n is determined by the number of bounded quantifiers occurring in the formula defining C.

Next we define the set of Σ₁ⁿ-formulas and Σ₁ⁿ(HTC)-formulas:

(1) All Σ₀ⁿ-formulas (Σ₀ⁿ(HTC)-formulas) are Σ₁ⁿ-formulas (Σ₁ⁿ(HTC)-formulas).

(2) If t₁ is a term of TROTₐ (TROTₐ(HTC)) and φ is a formula of Σ₁ⁿ (Σ₁ⁿ(HTC)), then \( ∃ yᵢ ∈ t₁φ \) is also a formula of Σ₁ⁿ (Σ₁ⁿ(HTC)).
Theorem (Characterization of NP-recognizable classes):

Let $C$ be a class of objects which is closed under isomorphisms. (i) Then $C$ is in NP iff $C$ is $\Sigma^P_1\text{(HTC)}$-definable.

(ii) $C$ is $\Sigma^P_n$-definable iff there is a natural number $n>0$ such that $C$ is closed under $n$-isomorphisms and $C$ is in NP.

Proof: The proof is a simple modification of Fagin’s characterization of NP-recognizable classes of finite first order structures [Fa74]. Here it is essential that objects $Ob$ are always represented by their transitive closure $HTC(Ob)$.

This last theorem suggests that $\Sigma^P_1$ and $\Sigma^P_1\text{(HTC)}$ are very reasonable specification languages for abstract data types. The class NP arises naturally here since we have no order on the urelements. Characterizations of lower complexity classes as in [Im83] always require some auxiliary predicates.

Problem: Find a natural choice of basic constructs which allows to characterize the lower complexity classes such as logarithmic space and polynomial time.

7. References.

[CH80]

[CH82]

[DM85]

[Fa74]

[Fe80]
[Ga80]

[Ho83]

[HP84]

[Im82]

[Im83]

[Le65]

[Mo74]

[Mo80]

[Mo84]

[No78]
[SSS79]

[Sch75]

[Sh73]

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