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

(Extended abstract)

E. Dahlhaus¹

Department of Mathematics (FB 20)
Technical University Berlin (West Berlin)

and

J.A. Makowsky²

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.

¹ During the work on this paper visiting (Spring 1985) at the Department of Computer Science, Technion, Haifa, Israel, sponsored by Minerva Foundation.

² During work on this paper visiting at the Institut fuer Informatik (Winter 1984/85) and the Forschungsinstitut fuer Mathematik (Summer 1985), ETH-Zuerich, Switzerland.

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. 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.
- (4) 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.
- (5) 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 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 *computable object transformation*. This is the content of section 2 and 3. Our mathematical model are the *hereditarily finite sets with urelements*. Finite objects are usually built from some, not further specified, atoms such as vertices 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, power set, 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

mkob y_i **from** y_j **in** y_k **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 Σ_1 -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 *TROTL*, and of while-free programs with transitive closure *TROTL(HTC)*, and associate with it a specification languages Σ_1^P and $\Sigma_1^P(HTC)$ respectively. Our main results are that all programs in *TROTL* are polynomial-time computable and that the classes of objects which are in **NP** are exactly the $\Sigma_1^P(HTC)$ -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 $D \subset U$ 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 ϕ).

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 $V(D) = \bigcup_{n \in \mathbf{N}} V_n(D)$ 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 $h: D \rightarrow D^\#$ a function between two domains. We define an extension $\bar{h}: V(D) \rightarrow V(D^\#)$ in the following way:

- (i) For urelements $d \in V_0(D)$ we put $\bar{h}(d) = h(d)$;
- (ii) For $\delta \in V_m(D)$ we put

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

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

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 $n > 0$ 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 $n > 0$ 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 $n > 0$ be a natural number. We say that two objects Ob_1, Ob_2 are n -isomorphic if $Ob_1 \upharpoonright n \cong Ob_2 \upharpoonright n$.

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 $T: V(D)^n \rightarrow V(D)$ such that for every bijection $h: D \rightarrow D$ and every $\delta_1, \dots, \delta_k \in V(D)$ we have

$$T(\bar{h}(\delta_1), \dots, \bar{h}(\delta_k)) = \bar{h}(T(\delta_1, \dots, \delta_k))$$

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 $\mathbf{N}$. 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 δ be an object and let $\{\delta\}$ be the object containing δ as its only subobject. The transformation $T_{\text{singleton}}$ which maps δ into $\{\delta\}$ is clearly a computable object transformation.
- (ii) Let δ_1, δ_2 be two objects and let $\delta_1 \cup \delta_2$ be the object which contains exactly the subobjects of δ_1 and δ_2 as its subobjects. The transformation $T_{\cup}$ which maps δ_1 and δ_2 into $\delta_1 \cup \delta_2$ is clearly a computable object transformation.
- (iii) Let δ_1, δ_2 be two objects and let $\delta_1 - \delta_2$ be the object which contains exactly the subobjects of δ_1 which are not in δ_2 as its subobjects. The transformation $T_{\text{difference}}$ which maps δ_1 and δ_2 into $\delta_1 - \delta_2$ is clearly a computable object transformation.
- (iv) Let δ be an object and let $Pow(\delta)$ be the object containing exactly each subset of subobjects of δ as a subobject. The transformation T_{power} which maps δ into $Pow(\delta)$ is clearly a computable object transformation.
- (v) Let δ be an object and let $U(\delta)$ be the object containing exactly each subobject of a subobject of δ as a subobject. The transformation $T_{\cup}$ which maps δ into $U(\delta)$ is clearly a computable object transformation.
- (vi) Let δ be an object and let $Ur(\delta)$ be the object of order 1 containing exactly the urelements which are leaves of δ as its subobjects. The transformation T_{Ur} which maps δ into $Ur(\delta)$ is clearly a computable object transformation.
- (vii) (Kuratowski pair) Set

$$Pair(\delta_1, \delta_2) = \{\{\delta_1\}, \{\delta_1, \delta_2\}\}$$

Clearly *Pair* is a computable object transformation.

(viii) Let δ be an object and let $HTC_0(\delta)$ be the set of all objects and relations, which are in its *transitive closure under membership* (the hereditary transitive closure). Now we put $HTC(\delta)$ to be $HTC_0(\delta) \cup \{\delta\}$. Then clearly HTC is a computable object transformation. Note that HTC preserves isomorphisms but, in contrast to the previous examples, for no $n > 0$ does it preserve n -isomorphisms.

(ix) We can use $\{\phi\}$ as truth value *true* and ϕ as truth value *false*. This allows us to define *computable predicates* as object transformations whose value are *true* or *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 *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 *transitive* if whenever $x \in Ob$ then x is a subset of Ob . Ob is *connex* if whenever $x, y \in Ob$ then either $x \in y$, $x = y$ or $y \in x$. Ob is *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 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.

$y_1, y_2, \dots$ are *variables* of OTL . Let *Objects* be a set of object names (i.e. constant names for either urelements, sets, sets of sets, etc). The set of *terms (over Objects)* 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

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

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 $y_1, y_2, \dots$ 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 $States(D)$.

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

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

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

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

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

$$(1) \ \mu_0(\{t_1, t_2\})(z) = \{\mu_0(t_1)(z), \mu_0(t_2)(z)\},$$

$$(2) \ \mu_0(Pow(t_1))(z) = Powerset \ of \ \mu_0(t_1)(z)$$

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

(3i) If all subobjects of $\mu_0(t_1)(z)$ are different from urelements then we set $\mu_0(U(t_1)(z)) = \cup(\mu_0(t_1))(z)$;

(3ii) If $\mu_0(t_1)(z)$ has at least two subobjects we take the union as in (3i) treating its subobjects which are urelements as empty sets.

(3iii) If $\mu_0(t_1)(z)$ has exactly one subobject which is an urelement Ur then $\mu_0(U(t_1)(z)) = Ur$.

Remark: It does not really matter how we handle the "freak" cases, and any other "reasonable" way of doing it will not affect our results.

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

$$(5) \ \mu_0(t_1 - t_2)(z) = \mu_0(t_1)(z) - \mu_0(t_2)(z)$$

Here $X - Y$ is the set of all elements of X not being in Y .

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

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 $\mu(P)(z)(y_j) = z(y_j)$ if $j \neq i$ and $\mu(P)(z)(y_i) = \mu_0(t)(z)$ 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 object.

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

$\mu(P)(z)(y_i) = \{\mu(P_1)(z_1)(y_j); z_1(y_l) = z(y_l) \text{ for } l \neq j \text{ and } z_1(y_j) \in z(y_k)\}$, if for all z_1 , s.t. $z_1(y_l) = z(y_l)$ for $l \neq j$ and $z_1(y_j) \in z(y_k)$ $\mu(P_1)(z_1)(y_j)$ is defined, otherwise $\mu(P)(z)(y_i)$ is undefined.

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 $t(y_1, \dots, y_m)$. Otherwise, the objects $y_1, \dots, y_{j-1}, y_{j+1}, \dots$ 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.

Let $Ob_1, \dots, Ob_k$ be k given objects and let $z_{initial}$ be the assignment with $z_{initial}(y_i) = Ob_i$ for all $i \leq k$ and $z_{initial}(y_i) = \phi$ for all $i > k$. Given a program $P(y_1, \dots, y_n) \in OTL$ and a variable y_j we look at the function $T_{P,j}: V(D)^k \rightarrow V(D)$ $T_{P,j}(Ob_1, \dots, Ob_k) = \mu(P)(z_{initial})(y_j)$.

Theorem (Computability of *OTL*-programs): For every program $P \in OTL$ and each variable y_j the the function $T_{P,j}: V(D)^k \rightarrow V(D)$ is a computable object transformation.

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 **mkob**-construct this follows from the following closure property of partial recursive functions:

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

Now let P be of the form

$$\mathbf{mkob} \ y_i \ \mathbf{from} \ y_j \ \mathbf{in} \ y_k \ \mathbf{by} \ P_1(y_1, \dots, y_m).$$

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

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

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 Σ_1 -definability in set theory as a generalization of recursive enumerability in the infinite set theoretic context. The analogy of Σ_1 -definable and recursive enumerable sets is based on the following fact (which is folklore among set theorists):

Consider the structure $\mathbf{HF}(\phi) = \langle \mathbf{HF}(\phi), \in \rangle$ with universe the hereditary finite sets without urelements and membership as its only relation. In $\mathbf{HF}(\phi)$ the Σ_1 -definable sets are exactly the recursive enumerable sets.

The notion of Σ_1 -definability has a natural meaning also in the structures $\mathbf{HF}(D)$ where D is a finite set of urelements. The structure $\mathbf{HF}(D)$ 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 Σ_1 -definable sets. The purpose of this section is to define Σ_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 : $x \in A \text{ iff } h(x) \in A$);
- (ii) A is recognizable by a *OTL*-program;
- (iii) A is Σ_1 -definable, that means, there is a Σ_1 -formula φ , s.t.

$$A = \{\delta : \mathbf{HF}(D) \models \varphi(\delta)\}.$$

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 $\mathbf{D}$ as its only nonlogical symbols.

We write $(\forall x \in y)P$ for $\forall x(x \in y \rightarrow P)$ and $(\exists x \in y)P$ for $\exists x(x \in y \wedge P)$. $(\forall x \in y)$ and $(\exists x \in y)$ are called *bounded quantifiers*.

A formula φ is called Σ_0 iff all quantifiers in it are bounded and Σ_1 iff it is of the form $\exists x_1, \dots, \exists x_n \psi$ where ψ is Σ_0 .

Sketch of proof (of the definability theorem): 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 δ 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 Σ_1 -formula expressing A .

(iii) $\rightarrow$ (ii): We want to prove that each $SIMA_1$ -expressible subset of $V(D)$ is recognizable by a program in OTL . At first we can prove that $\in$ 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 δ to the set of its subobjects δ_1 , s.t. $P(\delta_1)$ is expressible in OTL .

To see this we observe:

(i) Let H be the function which maps each δ_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 Σ_0 -predicate is decidable by a OTL -program.

Now we have to consider a Σ_1 formula $\exists x \psi$. We get a computable object transformation which computes 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 ψ . The class of natural numbers is computable in OTL since it is Σ_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_0 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_0 . For example, to show that the complement $-$ 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 **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 **mkob** and construct of unordered pairs $\{\cdot, \cdot\}$. To see that the construct of unordered pairs $\{\cdot, \cdot\}$ 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 inherent 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 $GRAPH(D)$ 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 $GRAPH(D)$ 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 *NGRAPH* we would require additionally that V is an initial segment of the class *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 *LISTS*, *STACKS*, *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 **P** (C is in **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 *HTC*.

First we observe the following:

Proposition: Let C be a class of objects which is Σ_0 -definable, i.e. there is a Σ_0 -formula $\varphi(x)$ such that C is the collection of objects satisfying φ . Then C is in **P** and there is a natural number $n > 0$ such that C is closed under n -isomorphisms.

To characterize the classes of objects in **NP** we define the **while**-free terms of OTL_0 and denote them by *TROTL* (terms and replacement of *OTL*).

- (1) The terms of OTL_0 are in *TROTL*.
- (2) If t_1, t_2 are terms of *TROTL* and y_i is a variable then $Mkob(t_1, t_2, y_i)$ is a term of *TROTL*.

The semantics of *TROTL* is defined as for OTL_0 with the difference that *Mkob* is a term construct replacing **mkob**. The interpretation of $Mkob(t_1, t_2, y_j)$ is the object interpreting y_i in

$$\mathbf{mkob} \ y_i \ \text{from} \ y_j \ \text{in} \ t_2 \ \text{by} \ t_1.$$

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

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

We next define the specification language Σ_0^P and $\Sigma_0^P(HTC)$.

- (1) If t_1, t_2 are terms of *TROTL* (*TROTL*(*HTC*)) then $t_1 \in t_2$ and $t_1 = t_2$ are formulas of Σ_0^P . ($\Sigma_0^P(HTC)$).
- (2) If φ and ψ are formulas of Σ_0^P ($\Sigma_0^P(HTC)$), then also $\varphi \wedge \psi, \varphi \vee \psi, \neg \varphi$ are formulas of Σ_0^P ($\Sigma_0^P(HTC)$).
- (3) If t_1 is a term of *TROTL* (*TROTL*(*HTC*)) and φ is a formula of Σ_0^P ($\Sigma_0^P(HTC)$), then $\exists y_i \in t_1 \varphi$ is also a formula of Σ_0^P ($\Sigma_0^P(HTC)$).

The semantics of Σ_0^P -formulas and $\Sigma_0^P(HTC)$ -formulas is obvious.

Proposition:

- (i) Every class C definable in $\Sigma_0^P(HTC)$ is in **P**.
- (ii) There are classes C in **P** which are not $\Sigma_0^P(HTC)$ -definable.
- (iii) For every class C definable in Σ_0^P 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 Σ_1^P -formulas and $\Sigma_1^P(HTC)$ -formulas:

- (1) All Σ_0^P -formulas ($\Sigma_0^P(HTC)$ -formulas) are Σ_1^P -formulas ($\Sigma_1^P(HTC)$ -formulas).
- (2) If t_1 is a term of $TROTL$ ($TROTL(HTC)$) and φ is a formula of Σ_1^P ($\Sigma_1^P(HTC)$), then $\exists y_i \subset t_1 \varphi$ is also a formula of Σ_1^P ($\Sigma_1^P(HTC)$).

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 $\Sigma_1^P(HTC)$ -definable.

(ii) C is Σ_1^P -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 Σ_1^P and $\Sigma_1^P(HTC)$ 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.

- [Ba75] J.K.Barwise, Admissible sets and structures, Springer, Berlin 1975.
- [CH80] A.K.Chandra and D.Harel, Computable queries for relational data bases, JCSS vol. 21.2 (1980) pp. 156-178.
- [CH82] A.K.Chandra and D.Harel, Structure and complexity of relational queries, JCSS vol. 25, (1982) pp. 99-128.
- [DM85] E. Dahlhaus and J.A.Makowsky, Computable Directory Queries, Technical report of the Institut fuer Informatik, ETH-Zuerich, No. 65 (September 1985), extended abstract to appear in the Proceedings of CAAP '86, LNCS, Springer, Berlin 1986.
- [Fa74] R. Fagin, Generalized first order spectra and polynomial-time recognizable sets, in: Complexity of Computation, R.Karp ed., SIAM-AMS Proceedings no. 7 (1974) p.27-41.
- [Fe80] J.E. Fenstad, General Recursion Theory, Springer, Heidelberg 1980.
- [Ga80] R. Gandy, Church's Thesis and principles for mechanisms, in: The Kleene Symposium, J.Barwise et al. eds., Amsterdam 1980, pp.123-148.
- [Ho83] E. Horowitz, Fundamentals of programming languages, Computer Science Press, Rockville 1983.
- [HP84] D. Harel and D. Peleg, On static logics, dynamic logics and complexity classes, Information and Control vol. 60, (1984) p. 86-102.

- [Im82] N. Immerman, Relational queries computable in polynomial time, 14th ACM Symposium on Theory of Computing, (1982) p. 147-152.
- [Im83] N. Immerman, Languages which capture complexity classes, 15th ACM Symposium on Theory of Computing, (1983) p. 347-354.
- [Le65] A. Levy, A hierarchy of formulas in set theory, Memoirs of the American Mathematical Society 57, Providence 1965.
- [Mo74] Y.N. Moschovakis, Elementary induction on abstract structures, North Holland, Amsterdam 1974.
- [Mo80] Y.N. Moschovakis, Descriptive set theory, North Holland, Amsterdam 1980.
- [Mo84] Y.N. Moschovakis, Abstract recursion as a foundation for the theory of algorithms, in: Computation and Proof Theory, M.M.Richter et al. eds. LNM vol. 1104, Springer, Berlin-Heidelberg 1984, p. 289-362.
- [No78] D. Normann, Set recursion, in: Generalized recursion theory II, J.E. Fenstad et al. eds., North Holland, Amsterdam 1978, p. 303-320.
- [SSS79] E. Schonberg, J.T. Schwartz and M. Sharir, Automatic data structure selection in SETL, 6. Annual ACM Symposium on Principles of Programming Languages (1979) pp.197-210.
- [Sch75] J.T. Schwartz, On programming: An interim report on the SETL project, 2nd ed., Courant Institute of Mathematical Sciences, New York 1975.
- [Sh73] J.C. Shepherdson, Computations over abstract structures, in: Logic Colloquium '73, H.E. Rose et al. eds., North Holland, Amsterdam 1973, p.445-513.