PB-143

Harel & Kozen: Inductive Sets

TRYK: DAIMI/RECAU

ISSN 0105-8517

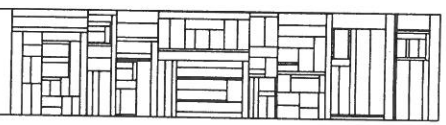
A PROGRAMMING LANGUAGE FOR THE INDUCTIVE SETS, AND APPLICATIONS

by

David Harel
Dexter C. Kozen

DAIMI PB-143
April 1982

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



A PROGRAMMING LANGUAGE FOR THE INDUCTIVE SETS, AND APPLICATIONS

David Harel[1,2]
The Weizmann Institute
Rehovot, Israel

Dexter Kozen[2,3]
Aarhus University
Aarhus, Denmark

Abstract

We introduce a programming language IND that generalizes alternating Turing machines to arbitrary first-order structures. We show that IND programs (respectively, everywhere-halting IND programs, loop-free IND programs) accept precisely the inductively definable (respectively, hyperelementary, elementary) relations. We give several examples showing how the language provides a robust and computational approach to the theory of first-order inductive definability. We then show: (1) on all acceptable structures (in the sense of Moschovakis [Mo]), r.e. Dynamic Logic is more expressive than finite-test Dynamic Logic. This refines a separation result of Meyer and Parikh [MP]; (2) IND provides a natural query language for the set of fixpoint queries over a relational database, answering a question of Chandra and Harel [CH2].

1. Introduction

In this paper we introduce a programming language IND. In its most basic form, the language consists of only 3 types of statements:

ℓ_1: $y \leftarrow \exists$ (or $y \leftarrow \forall$)
ℓ_2: accept (or reject)
ℓ_3: if $R(\bar{x})$ then goto ℓ_4

where $R(\bar{x})$ is an atomic first-order formula.

An IND program P can run in any first-order structure of the same similarity type. The input is an initial assignment to variables $\bar{x}=x_1,\ldots,x_n$, where $\bar{x}$ contains (at least) all the free variables of the program, and execution starts at the first statement. Statements of the form ℓ_1 assign an arbitrary element of the domain existentially (universally) to variable y, just as in alternating Turing machines [CKS]. A statement of the form ℓ_2 causes immediate acceptance or rejection, and ℓ_3 is an ordinary conditional branch. The definition of acceptance of the input $\bar{x}$ is the same as in alternating Turing machines [CKS] in-

1) Research supported in part by a Bath-Sheva fellowship.
2) Work done in part at IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, USA.
3) On leave from IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, USA.

volving an inductively-defined labeling of the computation tree with either 0 (reject), 1 (accept), or $\perp$ (undefined); see Section 3.

In Section 5 we show:

(1) IND programs accept precisely the relations definable by elementary (first-order) induction.

(2) IND programs which halt on all inputs (i.e., either accept or reject) accept precisely the hyperelementary (or inductive, co-inductive) relations.

(3) Loop-free IND programs accept precisely the elementary (first-order definable) relations.

In countable acceptable structures (see [Mo]) such as the natural numbers N, (1) and (2) become

(1) IND programs accept precisely the Π^1_1 relations.

(2) IND programs which halt on all inputs accept precisely the Δ^1_1 relations.

IND provides a computational intuition for the theory of inductive definability [Mo] which seems to be missing from the literature. For example, our proof of (2) involves showing that if both a relation and its complement are accepted by IND programs P_1 and P_2, then the two programs can be simulated by a third program that always halts, P_3. P_3 simulates steps of P_1 and P_2 alternately, halting whenever one or the other halts, just as in the usual Turing machine proof that an r.e., co-r.e. set is recursive. Many other elementary results, such as the Stage Comparison Theorem, Closure Theorem, and Separation Theorem [Mo], have machine-based proofs using IND that recall analogous proofs in recursion theory that use Turing machines. The availability of such a tool is especially important now that concepts central to inductive definability theory have resurfaced in computer science in recent work on program logics [MP, T], and program verification in the presence of fairness or unbounded nondeterminism [AP, LPS, GFMR].

In Section 6 we use IND to characterize the expressive power of Dynamic Logic. Meyer and Parikh [MP] have shown that Dynamic Logic with unrestricted recursively enumerable programs (DL_{re}) is strictly more expressive that many limited versions, such as DL with finite tests (DL_{ft}). The result is proved by transferring the problem to the problem of distinguishing ω^ω and $\omega^\omega \cdot 2$ in fragments of infinitary logic, and does not provide any insight into the inherent computational power of DL. In Section 6 we show that on any acceptable structure, DL_{re} is more expressive than DL_{ft}. Specifically, we show that in any acceptable struc-

ture, DL_{re} and DL_{ft} define exactly the IND complexity classes $\tau(\omega_1^{CK})$ and $\tau(\omega)$, respectively, where ω_1^{CK} is the first non-recursive ordinal. In any structure, $\tau(\omega)$={relations definable by first-order logic}, and on recursive acceptable structures (such as N), $\tau(\omega_1^{CK}) = \Delta_1^1$. The classes $\tau(\omega)$ and $\tau(\omega_1^{CK})$ can be separated on any acceptable structure by a simple diagonalization argument.

It should be emphasized that the expressiveness results of [MP] are schematic, in the sense that they consider $L_1 \leqq L_2$ if there is an interpretation of L_1 in L_2 which holds uniformly over all structures, whereas we will write $L_1 \leqq L_2$ if for each structure there is an interpretation of L_1 in L_2. The former gives stronger positive expressibility results, and the latter gives stronger negative expressibility results (such as $DL_{ft} \neq DL_{re}$). Our positive expressibility results (such as $DL_{re} \equiv \tau(\omega_1^{CK})$) are not to be interpreted schematically.

In Section 7 we show the connection between inductive definability and the fixpoint queries of [AU, CH] for relational data bases. Coupled with a recent result of Immerman [I], this easy observation shows that IND defines exactly the class of fixpoint queries FP, thus answering a question of Chandra and Harel [CH].

2. Programming examples

The language IND in its above form is very simple, and this makes formal semantics and formal proofs easier. However, the language is also quite robust, in the sense that more powerful programming constructs can be added without changing expressive power. For example, an unconditional jump can be obtained by using the test y=y in the conditional. Certain more complicated conditional forms, such as

```
if R(x̄) then goto ℓ1 else goto ℓ2
if R(x̄) then accept else reject
if ¬R(x̄) ∨ S(x̄) then goto ℓ
```

are obtained by manipulation of control flow. The assignment statement y←t is obtained by

```
y←∃
if y≠t then reject
```

or

```
x←∃
if x≠t then reject
y←∃
if y≠x then reject
```

where x is a new variable, if y occurs in t.

There is a loop-free program to compute any first-order formula. For example, an element x of a Boolean algebra is atomless if it satisfies the formula

$$\forall y \leq x \quad (y \neq 0 \supset \exists z \leq y \quad (0 \neq z \wedge z \neq y))$$

The set of such elements is accepted by the program

$y \leftarrow \forall$
<u>if</u> $\neg\, y \leq x \vee y=0$ <u>then</u> <u>accept</u>
$z \leftarrow \exists$
<u>if</u> $\neg\, z \leq y \vee 0=z \vee z=y$ <u>then</u> <u>reject</u>
<u>accept</u>.

It is clear, however, that IND programs can accept sets that are not first-order definable. For example, ℓ_1 below accepts all pairs (x,y) in the reflexive transitive closure of R, and ℓ_2 accepts y iff R is well-founded below y:

ℓ_1: <u>if</u> x=y <u>then</u> <u>accept</u>
$z \leftarrow \exists$
<u>if</u> $\neg R(x,z)$ <u>then</u> <u>reject</u>
$x \leftarrow z$
<u>goto</u> ℓ_1

ℓ_2: $x \leftarrow \forall$
<u>if</u> $\neg R(x,y)$ <u>then</u> <u>accept</u>
$y \leftarrow x$
<u>goto</u> ℓ_2

The statement $\ell_5 \vee \ell_6$ which accepts if either the program starting at ℓ_5 or that starting at ℓ_6 accepts is encoded by

$y \leftarrow \exists$
<u>if</u> $y=z$ <u>then</u> <u>goto</u> ℓ_5 <u>else</u> <u>goto</u> ℓ_6

where y is a new variable and z is any other variable. The statement $\ell_5 \wedge \ell_6$ is defined similarly, using $\forall$ instead of $\exists$. One can also encode the statement $\neg \ell_5$ which accepts (rejects) iff the computation starting at ℓ_5 rejects (accepts). This is done by taking the whole program P and constructing its dual $\bar{P}$ by interchanging $\forall/\exists$ and <u>accept</u>/<u>reject</u> statements. The program $\bar{P}$ then accepts (rejects) exactly when program P rejects (accepts), starting at any point.

The statement $\neg \ell_5$ is then given by <u>goto</u> $\bar{\ell}_5$ where $\bar{\ell}_5$ is the statement corresponding to ℓ_5 in the dual program. Of course, the statement $\neg \bar{\ell}_5$ in the dual program is replaced by <u>goto</u> ℓ_5.

These constructs allow us to encode the statement

<u>if</u> $\varphi(\bar{x})$ <u>then</u> <u>goto</u> ℓ_1 <u>else</u> <u>goto</u> ℓ_2,

where $\varphi(\bar{x})$ is any first-order formula, or for that matter any relation computed by a program that always halts:

$$(m_1 \wedge \ell_1) \vee (\neg m_1 \wedge \ell_2)$$

where m_1 is the first statement of a program computing $\varphi(\bar{x})$.

One of our main results is that there is an IND program that accepts any relation definable by first-order induction. For example, the subgroup H of G generated by a,b is the least subset of G such that

$$x \in H \leftrightarrow x=a \vee x=b \vee \exists y \in H \ \exists z \in H \ x=y \cdot z \vee x=y^{-1}$$

Membership in H is computed by the program

ℓ_1: if $x=a \vee x=b$ then accept
$y \leftarrow \exists$
if $x=y^{-1}$ then goto ℓ_3
$z \leftarrow \exists$
if $x \neq y \cdot z$ then reject
$\ell_2 \wedge \ell_3$
ℓ_2: $x \leftarrow z$
goto ℓ_1
ℓ_3: $x \leftarrow y$
goto ℓ_1

To give an example involving an unbounded alternation of quantifiers, consider a two-person game like chess or go. The set of board positions from which a given player has a forced win is defined inductively from the legal-move and (immediate) win predicates by

$$\text{force}(x) \leftrightarrow \text{win}(x) \vee \exists y(\text{legal-move}(x,y) \wedge \neg \text{win}(y) \wedge \forall z(\text{legal-move}(y,z) \rightarrow \text{force}(z)))$$

and is accepted by the program

ℓ_1: if win(x) then accept
$y \leftarrow \exists$
if $\neg$legal-move(x,y) $\vee$ win(y) then reject
$x \leftarrow \forall$
if $\neg$legal-move(y,x) then accept
goto ℓ_1

IND programs can run for more than a finite amount of time and still halt. An example of a program that runs in N for time $\omega+1$ is

$y \leftarrow \forall$
ℓ_1: if y=0 then accept
$y \leftarrow y-1$
goto ℓ_1

In N, the running times of IND programs are exactly the recursive ordinals. In fact, we can take the set of computation trees of IND programs as a set of notations for recursive ordinals.

3. Semantics of acceptance

The semantics of acceptance is formally almost identical to that of alternating Turing machines [CKS]. Intuitively, it consists of two stages: (1) generation of the computation tree downwards from the root, and (2) evaluation of the acceptance function upwards from the leaves to the root. Associated with each node of the computation tree is a unique configuration (ℓ,v) where ℓ is the label of one of the statements in the program and v is a valuation of program variables over the domain of computation A. If a node of the computation tree is labeled c, then its immediate descendants are labeled with all elements of $N(c)$, the next configurations of c, which are defined by cases, depending on the statement $stmt(c)$ labeled by c's first component:

$$N(\ell,v)=\begin{cases}\{(\ell',v[y\leftarrow a]) \mid a\in A\} & \text{if } stmt(\ell,v) \text{ is } y\leftarrow\exists \text{ or } y\leftarrow\forall\\ \emptyset & \text{if } stmt(\ell,v) \text{ is accept or reject}\\ \{(m,v)\} & \text{if } stmt(\ell,v) \text{ is if } R(\bar{x}) \text{ then goto } m \text{ and } A,v \models R(\bar{x})\\ \{(\ell',v)\} & \text{if } stmt(\ell,v) \text{ is if } R(\bar{x}) \text{ then goto } m \text{ and } A,v \models \neg R(\bar{x})\end{cases}$$

where ℓ' denotes the next statement after ℓ in the program (or the first statement, if ℓ was the last). The root of the tree is the start configuration. The acceptance function e^* can be regarded as a labeling of nodes of the computation tree with either 0 (reject), 1 (accept), or $\perp$ (undefined), but formally its domain is the set C of configurations. It is defined as the supremum of a chain of approximating labelings e^α. Intuitively, $e^\alpha(c)=1$ (respectively, 0) if c has been determined to be an accept (respectively, reject) configuration by time α; $e^\alpha(c)=\perp$ if neither has been determined by time α. At time 0, nothing is determined, thus $e^0(c)=\perp$ for all c. At time 1, the leaves of the tree, corresponding to accept and reject statements, are labeled 1 and 0, respectively, and everything else is labeled $\perp$. If $stmt(c)=y\leftarrow\exists$ and $e^\alpha(d)=1$ for some $d\in N(c)$, then $e^{\alpha+1}(d)=1$. If $stmt(c)=y\leftarrow\forall$ and all $d\in N(c)$ are eventually labeled 1, then c becomes labeled 1 upon completion of the labeling of $N(c)$. Note that, because of unbounded nondeterminism, it may take more than a finite amount of time for a configuration to become labeled 0 or 1.

Formally, let C be the set of configurations, let Ord be the class of ordinals, and let $*\notin Ord$ with $\alpha<*$ for all $\alpha\in Ord$. Define the sequence $e^\alpha: C\to\{0,1,\perp\}$ inductively by $e^\alpha=\bigsqcup_{\beta<\alpha}\tau(e^\beta)$, where τ is the $\sqsubseteq$-monotone map defined by

$$\tau(e)(c) = \begin{cases} 1 & \text{if stmt}(c) = \underline{\text{accept}} \\ 0 & \text{if stmt}(c) = \underline{\text{reject}} \\ \bigvee_{d\in N(c)} e(d) & \text{if stmt}(c) = y\leftarrow\exists \\ \bigwedge_{d\in N(c)} e(d) & \text{if stmt}(c) = y\leftarrow\forall \\ e(d) & \text{if stmt}(c) = \underline{\text{if}}\ R\ \underline{\text{then goto}}\ m \text{ and } N(c)=\{d\}. \end{cases}$$

The meet $\bigwedge$ and join $\bigvee$ are with respect to the ordering $0<\perp<1$, and should not be confused with the approximation ordering $\sqsubseteq$ with join $\sqcup$, defined by $\perp\sqsubseteq 0$, $\perp\sqsubseteq 1$ and extended pointwise to labelings. e^* is the $\sqsubseteq$-least fixpoint of τ.

We say <u>c becomes properly labeled at time α</u> if α is the least ordinal such that $e^{\alpha}(c)\neq\perp$, and write $o(c)=\alpha$. If no such α exists, we write $o(c)=*$. Thus $e^*(c)=e^{o(c)}(c)$. The <u>running time of P on input $\bar{x}$</u> is defined to be $o(c)$, where c is the start configuration $(\ell_1,\bar{x})$. We denote this by $\text{TIME}(P,\bar{x})$. The program P is said to <u>accept</u> $\bar{x}$ if $e^*(c)=1$ and <u>reject</u> $\bar{x}$ if $e^*(c)=0$, where c is the start configuration of P on $\bar{x}$; in either case $\text{TIME}(P,\bar{x})<*$ and P is said to <u>halt</u> on $\bar{x}$. If $\text{TIME}(P\bar{x})=*$, then $e^*(c)=\perp$ and P does not halt on $\bar{x}$. Call a program P <u>β-time bounded</u> if $\text{TIME}(P,\bar{x})\leq\beta$ for all inputs $\bar{x}$ accepted by P (P need not halt on other inputs). Define the complexity class

$$\tau(\alpha) = \bigcup_{\beta<\alpha}\{\text{relations accepted by } \beta\text{-time bounded IND programs}\}.$$

4. The shuffle construction

If A and $\bar{A}$ are both r.e, then A can be proved recursive by constructing a Turing machine T_3 which simulates steps of T_1 and T_2 alternately, where T_1 accepts A and T_2 accepts $\bar{A}$. In this section we give a similar construction for IND programs.

Suppose P and Q are programs with <u>disjoint</u> sets of variables $\bar{x}$ and $\bar{y}$, respectively, and statement labels $\ell_1,\ldots,\ell_p$ and $m_1,\ldots,m_q$, respectively. By adding dummy statements, we can assume without loss of generality that p and q are relatively prime. Now we shuffle the statements of P and Q to get the program PQ with 2pq statements labeled by all pairs of the form $(\underline{\ell_i},m_j)$ and $(\ell_i,\underline{m_j})$, $1\leq i\leq p$, $1\leq j\leq q$, arranged in the order

$$(\underline{\ell_1},m_1),(\ell_2,\underline{m_1}),(\underline{\ell_2},m_2),(\ell_3,\underline{m_2}),\ldots,(\underline{\ell_p},m_i),(\ell_1,\underline{m_i}),\ldots,(\underline{\ell_p},m_q),(\ell_1,\underline{m_q}).$$

The underline tells which statement of P or Q is the next to be simulated. The statement of PQ labeled by $(\underline{\ell_i},m_j)$ is the same as the statement of P labeled by ℓ_i, unless it is a conditional jump

$$\ell_i:\ \underline{\text{if}}\ R(\bar{x})\ \underline{\text{then goto}}\ \ell_k$$

in which case we take

$$(\underline{\ell_i},m_j)\colon \underline{\text{if}}\ R(\bar{x})\ \underline{\text{then goto}}\ (\ell_k,\underline{m_j}).$$

A symmetric remark holds for statements labeled $(\ell_i,\underline{m_j})$. Thus PQ simulates steps of P and Q alternately. Since the variables of P and Q are disjoint, these simulations do not interfere with each other. The formal statement of this property involves the relationship between the successor configuration maps N_{PQ} and N_P,N_Q. Observe that there is a natural one-to-one correspondence between the configurations C_{PQ} of PQ and pairs $(c,d) \in C_P \times C_Q \cup C_Q \times C_P$:

$$((\underline{\ell_i},m_j),\bar{a},\bar{b}) \longmapsto ((\ell_i,\bar{a}),(m_j,\bar{b}))$$
$$((\ell_i,\underline{m_j}),\bar{a},\bar{b}) \longmapsto ((m_j,\bar{b}),(\ell_i,\bar{a})).$$

The order of the components in $(c,d) \in C_P \times C_Q \cup C_Q \times C_P$ tells which of P,Q is next to be simulated. Hence we will identify elements of C_{PQ} with the corresponding elements of $C_P \times C_Q \cup C_Q \times C_P$. Then by construction of PQ,

$$N_{PQ}(c,d) = \{(d,c') \mid c' \in N(c)\}$$

where $N(c)=N_P(c)$ if $c \in C_P$, $N_Q(c)$ if $c \in C_Q$.

The following theorem says that the label assigned by e^* to a particular configuration $(c,d) \in C_P \times C_Q$ of PQ is either the one assigned to c by P or the one assigned to d by Q, depending on which is labeled sooner. Moreover (c,d) is labeled in PQ within at most double the time it takes to label either c or d in P or Q, respectively.

<u>Theorem 1</u> Let $(c,d) \in C_P \times C_Q \cup C_Q \times C_P$.

(i) $$e^*(c,d) = \begin{cases} e^*(c) & \text{if } o(c) \leq o(d) \\ e^*(d) & \text{if } o(c) > o(d) \end{cases}$$

(ii) $$o(c,d) = \min(2 \cdot o(c),\ 2 \cdot o(d)+1).$$

<u>Proof</u> Let $e\colon C_P \times C_Q \cup C_Q \times C_P \to \{0,1,\perp\}$ be the map

$$e(c,d) = \begin{cases} e^*(c) & \text{if } o(c) \leq o(d) \\ e^*(d) & \text{if } o(c) > o(d). \end{cases}$$

It is easily shown by cases that e is a fixpoint of τ, therefore $e^* \sqsubseteq e$. That e is no more defined than e^* follows from (ii), which can now be proved by transfinite induction on $\min(2 \cdot o(c),\ 2 \cdot o(d)+1)$. Curiously, the proof of (ii) depends on the fact that $e^* \sqsubseteq e$; this is because for statements $y \leftarrow \exists$ and $y \leftarrow \forall$, the time that the configuration becomes labeled depends on what the label is. □

If the variables of P and Q are not disjoint, define the shuffle PQ as follows: rename the variables of P to get a program P' having no variables in common with Q. Let $x_1,\ldots,x_k$ be the variables common to P and Q and let $y_1,\ldots,y_k$ be their replacements in P'. Define PQ to be the program which assigns $y_i \leftarrow x_i$, $1 \leq i \leq k$, then runs P'Q.

Corollary 1 Let P, Q be two programs with a common set $\bar{x}$ of input variables. Then PQ accepts (rejects) $\bar{x}$ iff either

(i) $\mathrm{TIME}(P,\bar{x}) \leq \mathrm{TIME}(Q,\bar{x})$ and P accepts (rejects) $\bar{x}$; or
(ii) $\mathrm{TIME}(P,\bar{x}) > \mathrm{TIME}(Q,\bar{x})$ and Q accepts (rejects) $\bar{x}$. □

5. Main results

Theorem 2 (i) IND programs accept precisely the relations definable by first-order induction;
(ii) IND programs which halt on all inputs accept precisely the hyper-elementary (or inductive, coinductive) relations;
(iii) loop-free IND programs accept precisely the first-order definable relations.

Proof (i) Every IND program accepts only inductively definable relations, because the definition of the acceptance function e* is a classical first-order inductive definition over the domain of computation (see [Mo]). Conversely, every inductively definable relation is given by a first-order formula $\varphi(S,\bar{x})$ with free variables $\bar{x}=x_1,\ldots,x_n$, an n-ary predicate symbol S occurring only positively in φ, and some constants $\bar{a}=a_1,\ldots,a_m$, $m<n$. The fixpoint defined by φ is the least S* such that $S^*=\varphi(S^*,\bar{x})$. The inductive relation defined by $\varphi,\bar{a}$ is the (n-m)-ary relation $S^*(a_1,\ldots,a_m,x_{m+1},\ldots,x_n)$. A program to accept all $(x_{m+1},\ldots,x_n)$ satisfying $S^*(a_1,\ldots,a_m,x_{m+1},\ldots,x_n)$ first assigns a_i to x_i, $1 \leq i \leq m$, and then enters a loop labeled ℓ_1 which determines whether $S^*(\bar{x})$. Within the loop it decomposes $\varphi(S,\bar{x})$, using $y \leftarrow \forall$ and $y \leftarrow \exists$ to eliminate quantifiers, $\ell_1 \wedge \ell_2$, $\ell_1 \vee \ell_2$, $\neg \ell_1$ to eliminate logical connectives, and conditionals for atomic formulas; this leaves only occurrences of $S(\bar{y})$, which are handled by assigning $\bar{y}$ to $\bar{x}$ followed by an unconditional jump back to ℓ_1.

(ii) Any program P accepting S which halts on all inputs has a dual $\bar{P}$ which also halts on all inputs, and accepts the complement of S. Thus by (i), S is both inductive and coinductive. Conversely, suppose the set S is both inductive and coinductive. By (i), there are programs P and Q accepting S and $\bar{S}$, respectively. Modify P and Q so that they never re-

ject, by replacing all statements ℓ: reject with ℓ: goto ℓ. By Corollary 1, the shuffle $P\bar{Q}$ accepts S and rejects $\bar{S}$.

(iii) It has already been argued in Section 2 that every first-order definable relation is computed by a loop-free program. The converse is obtained by observing that every loop-free program is equivalent to one with only forward jumps; a formula is now easy to construct. □

Observe from the proof of Theorem 2(i) that there is a strong connection between the running times of IND programs and the ordinals at which inductive definitions close (see [Mo]). The closure ordinal κ^A of a structure A is defined in [Mo] as the supremum of closure ordinals of all possible inductive definitions. By the proof of Theorem 2(i) we see that (for infinite structures) this is just the supremum of running times of IND programs in A. The following theorem relates these concepts to the complexity classes $\tau(\alpha)$ defined in Section 3.

Theorem 3

(i) $\tau(\omega) = \{\text{first-order definable relations}\}$

(ii) $\tau(\kappa^A) = \{\text{hyperelementary relations}\}$

Remark Part (ii) is exactly the closure theorem of Moschovakis [Mo, p. 33].

Proof (i) Clearly, any loop-free program can run for only finitely many steps, independent of the input. Conversely, any c-time bounded program, $c<\omega$, can be made to halt on all inputs by shuffling it with a "clock", i.e. a program that on all inputs runs for c+1 steps, then rejects. The resulting program now has a finite, uniform time bound d, independent of the input. But any such program is equivalent to a loop-free program obtained by unwinding the loops d+1 times. The result follows from Theorem 2(iii).

(ii) ($\supseteq$) Let P be a program that halts on all inputs. Let P_1 be P modified so as never to reject, as in the proof of Theorem 2(ii), and let P_2 be $\bar{P}$ modified so as never to reject, where $\bar{P}$ is the dual of P. Then P_1P_2 accepts all inputs and $\forall\bar{x}$, $\mathrm{TIME}(P,\bar{x}) \leq \mathrm{TIME}(P_1P_2,\bar{x})$. Let P_3 be the program which chooses the input universally by executing $y \leftarrow \forall$ for all input variables, then executes P_1P_2. Then $\mathrm{TIME}(P_3,\bar{x})$ is a constant β independent of the input, and $\forall\bar{x}\ \mathrm{TIME}(P,\bar{x}) \leq \beta < \kappa^A$.

($\subseteq$) Let P be β-time bounded, $\beta<\kappa^A$. If we can construct an α-clock, $\beta<\alpha$, then it can be shuffled with P to give a program accepting the same set as P, but always halting. The result then follows from Theorem 2(ii). Since $\beta<\kappa^A$, there exists a program Q which runs for time $\alpha>\beta$ on some input $\bar{x}$. Let Q_1 assign $\bar{x}$ to all input variables, then run Q. Q_1 halts on all inputs in time exactly $\alpha+c$ and either accepts all inputs or rejects all inputs, so either Q_1 or $\bar{Q}_1$ gives an appropriate clock. □

6. An application to Dynamic Logic

The programming language IND originally arose in our attempt to clarify a result of Meyer & Parikh [MP] on the relative expressibility of four variants of first-order Dynamic Logic (DL), namely DL_{reg}, DL_{cf}, DL_{ft}, and DL_{re}. Programs of DL_{re} are all r.e. sets of sequences of assignments $x:=t$ and tests $\varphi(\bar{x})?$, called seqs, where t is a term and φ a formula of DL_{re}. One obtains DL_{reg}, DL_{cf}, and DL_{ft} by allowing, respectively, only regular expressions or flowchart programs (so that the set of seqs is regular), recursion schemes (so that the set of seqs is context free), or r.e. sets of seqs, but each with at most finitely many distinct tests.

Meyer & Parikh prove that DL_{reg} is strictly less expressive than DL_{re} (in symbols, $DL_{reg}<DL_{re}$) by the following sequence:

$$(*) \qquad DL_{reg} \overset{\text{ⓐ}}{\leq} DL_{cf} \overset{\text{ⓑ}}{\leq} DL_{ft} \overset{\text{ⓒ}}{\leq} L_{ba} \overset{\text{ⓓ}}{<} L^{CK}_{\omega_1\omega} \overset{\text{ⓔ}}{\equiv} DL_{re}$$

where $L^{CK}_{\omega_1\omega}$ is infinitary first-order logic with r.e. disjunctions, and L_{ba} is the same language restricted to bounded quantifier alternation. The bulk of the proof is devoted to ⓓ, which uses an Ehrenfeucht-Frassé argument to show that L_{ba} cannot distinguish between the ordinals ω^ω and $\omega^\omega\cdot 2$, while $L^{CK}_{\omega_1\omega}$ can define any recursive ordinal up to isomorphism.

In this part of their paper, all resemblance to Dynamic Logic has been lost. This was taken as evidence in support of the stand that there is really nothing dynamic about Dynamic Logic, and one should do all one's work in infinitary logic [MT]. We disagree, and in fact find the main result of [MP] a bit misleading, for the simple reason that $L^{CK}_{\omega_1\omega}\equiv L_{ba}$ in virtually every structure arising in computer science (for example, the natural numbers, N, any recursively defined data type, or any structure whose elements are all named by closed terms). This is because every $L^{CK}_{\omega_1\omega}$ formula is equivalent to a quantifier-free formula, by replacing $\exists x\, \varphi(x)$ with $\bigvee_t \varphi(t)$, where the join is over the set of closed terms. One's intuition is still that $DL_{reg}<DL_{re}$, even restricted to such structures. Our results of this section show that $DL_{ft}<DL_{re}$ on any acceptable structure [Mo] (or arithmetic universe [H]). These structures contain a

first-order definable copy of the natural numbers and first-order predicates for coding and decoding sequences of elements into single elements. This allows assigning codes or Gödel numbers to programs and formulas so that they can be decoded and manipulated by other programs and formulas. The proof reveals the computational power of the various versions of DL in terms of the complexity classes $\tau(\alpha)$.

Theorem 4 On any acceptable structure,

(i) $DL_{reg} \equiv DL_{cf} \equiv DL_{ft} \equiv \tau(\omega)$

(ii) $DL_{re} \equiv \tau(\omega_1^{CK})$

(iii) $\tau(\omega_1^{CK}) - \tau(\omega) \neq \emptyset$

where ω_1^{CK} is the first nonrecursive ordinal.

For example, on $\mathbb{N}$, whose closure ordinal is ω_1^{CK}, $DL_{re} \equiv \Delta_1^1$ and $DL_{ft} \equiv$ {first-order definable relations}. This follows from Theorems 3, 4 and Kleene's Theorem (Δ_1^1 = hyperelementary on $\mathbb{N}$).

Proof (i), (ii), ($\supseteq$) This direction does not need the assumption of acceptability. The case (i) follows from Theorem 3(i) and the fact that DL contains first-order logic. Similarly, on any structure A, $\tau(\omega_1^{CK}) < DL_{re}$, since if P is any IND program and α any recursive ordinal, there is a DL_{re}-formula φ_α such that

$$A \models \varphi_\alpha(c) \text{ iff } e^\alpha(c) = 1$$

for any configuration c, defined recursively by

$$\begin{aligned}
\varphi_0 &\leftrightarrow \underline{\text{false}} \\
\varphi_{\alpha+1}(c) &\leftrightarrow \text{stmt}(c) = \underline{\text{accept}} \\
&\quad \vee\ \text{stmt}(c) = y \leftarrow \exists\ \wedge\ \exists d \in N(c)\ \varphi_\alpha(d) \\
&\quad \vee\ \text{stmt}(c) = y \leftarrow \forall\ \wedge\ \forall d \in N(c)\ \varphi_\alpha(d) \\
&\quad \vee\ \text{stmt}(c) = \underline{\text{if}} \ldots \underline{\text{then}} \ldots \wedge\ \exists d \in N(c)\ \varphi_\alpha(d) \\
\varphi_\lambda &\leftrightarrow \langle \{\varphi_\alpha? \mid \alpha < \lambda\} \rangle\ \underline{\text{true}},\ \lambda \text{ a limit ordinal.}
\end{aligned}$$

The crucial point of this definition is that it is effective, in the sense that there is a recursive function r such that $r('\alpha') = '\varphi_\alpha'$, where '$\alpha$' and '$\varphi$' denote codes for recursive ordinals and DL_{re} formulas. This fact is needed in the definition of φ to insure that the set $\{\varphi_\alpha? \mid \alpha < \lambda\}$ is r.e., so that φ_λ will be a DL_{re} formula. Now, if $A \in \tau(\omega_1^{CK})$, then A is accepted by an IND program P which is α-time bounded for some recursive ordinal α; thus for any input $\bar{a} \in A^k$,

$$P \text{ accepts } \bar{a} \leftrightarrow e^{\alpha}(\ell_1,\bar{a}) = 1 \leftrightarrow A \models \varphi_{\alpha}(\ell_1,\bar{a}).$$

Then $\varphi_{\alpha}(\ell_1,\bar{x})$ is a DL_{re} formula defining A.

(i), (ii), ($\subseteq$) We describe first an IND program to decide the satisfiability of DL_{re} formulas in A, consisting of a main program SATIS(p,x) and subroutine COMPUTE(q,x,y). SATIS('φ','$\bar{a}$') will determine if $A,\bar{a} \models \varphi$ and COMPUTE('π','$\bar{a}$','$\bar{b}$') will determine if state $\bar{a}$ goes to state $\bar{b}$ under program π, where '$\bar{a}$','$\bar{b}$' are codes of sequences $\bar{a},\bar{b}$ of elements of A, and 'φ' and 'π' are codes of a DL_{re} formula φ and a DL_{re} program π.

Initially, SATIS('φ','$\bar{a}$') assigns a code for the list of free variables of φ (available from 'φ') to a variable v, and assigns '$\bar{a}$' to w. This models the assignment of the values $\bar{a}$ to the list of variables in v, in the same order. It now proceeds by cases, depending on the form of φ. If $\varphi=\psi\wedge\sigma$, it uses the program construct $\wedge$ defined in Section 2 to check both ψ and σ, and similarly for $\vee,\neg$. If $\varphi=\exists y\psi$, it executes $z\leftarrow\exists$; then, if the DL_{re} variable y is in the list v, it modifies the corresponding value in the list w to the value of z; otherwise, it appends the name of y to the list v and the value of z to the list w. If $\varphi=\forall y\psi$ it does the same, using $z\leftarrow\forall$ instead of $z\leftarrow\exists$. If $\varphi=\langle\pi\rangle\psi$, since

$$A,\bar{a} \models \langle\pi\rangle\psi \text{ iff } \exists\bar{b}\ \bar{a} \text{ goes to } \bar{b} \text{ under } \pi \text{ and } A,\bar{b} \models \psi,$$

SATIS executes $z\leftarrow\exists$ and interprets the result as a code '$\bar{b}$'. It then calls SATIS('ψ','$\bar{b}$') and COMPUTE('π','$\bar{a}$','$\bar{b}$') in parallel, using $\wedge$.

Finally, if $\varphi=R(\bar{x})$ where $R(\bar{x})$ is atomic, it picks out the current values in the list w corresponding to DL_{re} variables $\bar{x}$ and assigns them to IND variables $\bar{y}$, then executes

$$\underline{\text{if}}\ R(\bar{y})\ \underline{\text{then}}\ \underline{\text{accept}}\ \underline{\text{else}}\ \underline{\text{reject}}.$$

COMPUTE('π','$\bar{a}$','$\bar{b}$') determines whether state $\bar{a}$ goes to state $\bar{b}$ under DL_{re} program π. Recall that a program π consists of an r.e. set of seqs; each seq is a finite sequence $s_0;\ldots;s_{k-1}$ for some k; and each s_i is either an assignment y:=t or a test φ?. The code 'π' gives a Gödel number for the set of seqs, and $\bar{a}$ goes to $\bar{b}$ under π iff $\bar{a}$ goes to $\bar{b}$ under some seq of π. COMPUTE chooses a seq existentially using $i\leftarrow\exists$, and then tries to determine if the i'th seq of π, say $seq_i=s_0;\ldots;s_{k-1}$, takes $\bar{a}$ to $\bar{b}$. It could do this by starting from $\bar{a}_0=\bar{a}$, deterministically applying $s_0,s_1,\ldots,s_{k-1}$ in succession to get a sequence $\bar{a}_1,\ldots,\bar{a}_k$ of intermediate states, and accepting if $\bar{a}_k=\bar{b}$. However, for a later application, it will be better to keep COMPUTE loop-free. Thus, the program instead guesses a code for the entire sequence $\bar{a}_0,\bar{a}_1,\ldots,\bar{a}_k$ with a single $z\leftarrow\exists$, and then determines whether s_j takes $\bar{a}_j$ to $\bar{a}_{j+1}$, $0\leq j<k$, by executing

$j \leftarrow \forall$; if j is not the code for a natural number $< k$ then accept; check if s_j takes $\bar{a}_j$ to $\bar{a}_{j+1}$. For s_j of the form $y:=t$, the check is straightforward. For s_j of the form $\psi?$, the program checks whether $\bar{a}_j = \bar{a}_{j+1}$, then whether $A, \bar{a}_j \models \psi$ by a recursive call to SATIS.

Holding 'φ' fixed, SATIS('φ','$\bar{a}$') accepts (the codes of) the set defined by φ. The theorem is now proved by analyzing the time complexity of SATIS and COMPUTE on fixed φ. All encoding and decoding operations can be done without loops, since they are first-order definable. The choice of seq_i in COMPUTE can be done without a loop since π is r.e. and thus first-order definable. We were careful to avoid loops in the processing of a seq in COMPUTE. Thus each iteration of SATIS and COMPUTE takes constant time before it recurs on a subformula; therefore there is a constant c such that

$$\forall \bar{a}\ \mathrm{TIME}(\mathrm{SATIS}('\varphi','\bar{a}')) \leq c \cdot h(\varphi)$$

where $h(\varphi)$ is the <u>height</u> of φ, defined by:

$$\begin{aligned}
h(\varphi) &= 1, \ \varphi \text{ atomic},\\
h(\varphi \vee \psi) &= h(\varphi \wedge \psi) = \max\{h(\varphi), h(\psi)\} + 1,\\
h(\forall x \varphi) &= h(\exists x \varphi) = h(\neg\varphi) = h(\varphi) + 1,\\
h(\langle\pi\rangle\varphi) &= \max\{h(\pi), h(\varphi)\} + 1,\\
h(\pi) &= \sup\{h(\sigma) \mid \sigma \text{ a seq of } \pi\} + 1,\\
h(\sigma) &= \sup\{h(\varphi) \mid \varphi? \text{ a test of } \sigma\} + 1, \ \sigma \text{ a seq}.
\end{aligned}$$

Thus it remains to show that

$$\begin{aligned}
h(\varphi) &< \omega_1^{CK}, \quad \varphi \text{ in } DL_{re},\\
h(\varphi) &< \omega, \quad \varphi \text{ in } DL_{ft}.
\end{aligned}$$

The former follows from the fact that there is a recursive code 'φ' for each φ in DL_{re}, and h is effective with respect to this code. The latter follows from the fact that the suprema in the definition of $h(\pi)$ and $h(\sigma)$ are finite, since there are only finitely many tests.

(iii) This is a straightforward diagonalization. Construct an IND program P which, on input '$\varphi(x)$', $\varphi(x)$ a first-order formula with one free variable x, accepts iff $\neg\varphi('\varphi')$. P runs for time $c \cdot h(\varphi) < \omega$, so the set it accepts is in $\tau(\omega+1) \subseteq \tau(\omega_1^{CK})$, and not in $\tau(\omega) = \{$first-order-definable sets$\}$ for obvious reasons. □

7. IND as a Data Base Query Language

There has been much recent work in the theory of relational data bases. In the relational model, a data base is a collection of finite tables [C1] and can be viewed simply as a finite first-order structure $B=(D,R_1,\ldots,R_k)$. Queries are (partial) functions from data bases to relations, and a query language is a set of formal expressions defining such functions; see [CH1].

In [C2] Codd introduced the languages of the relational algebra and calculus, which are equivalent in expressive power. The latter is essentially the first order language of similarity type $(=, R_1,\ldots,R_k)$. In [AU] it was pointed out that many useful queries definable naturally by least fixpoints of first-order formulas, such as the transitive closure of a binary relation, are not first-order-definable, and it was suggested therein that the first-order language of [C2] be augmented with an appropriate least fixpoint operator. In such a language the transitive closure of R would be the least fixpoint S of $S=R \cup R \circ S$, where $\circ$ is relational composition. A formal version of such an extension was subsequently supplied in [CH2], where fixpoint operators were allowed to alternate with any number of first-order constructs. A hierarchy of height ω^2 of sets of queries is defined in [CH2], in which those queries at level $\omega \cdot i$ are obtained by applying a least fixpoint operator to queries at lower levels. The set of queries constituting the entire hierarchy is termed FP, for fixpoint queries.

It is shown in [CH2] that FP is a very restricted subset of the set of all computable queries [CH1]; in particular, all queries in FP are polynomial-time computable. There is also a close correspondence with the queries definable by Kowalski's logic programs; see [K, GM, CH3]. However, it was left as an open problem in [CH2] whether there is a natural computational query language for defining the fixpoint queries.

At this point, one observes that the least fixpoint operator as defined in [AU, CH2] corresponds exactly to an inductive definition as defined in [Mo], so that a single fixpoint operator applied to a first-order formula corresponds to a first-order inductive definition. Recently, however, Immerman [I] has shown that the hierarchy of fixpoint queries in fact collapses down to level ω. In other words, all queries in FP are definable by a single application of a fixpoint operator to a first-order formula. Hence, we obtain:

Lemma A relational function on finite structures is in FP iff it is uniformly first-order inductively definable (i.e., there is a single first-order inductive definition which, given the input structure, defines the output relation).

Theorem IND defines precisely the fixpoint queries on relational data bases.

We might remark that the $x \leftarrow \forall$ statement of IND and the parallel method of execution implied by its semantics reminds one of the "for all tuples t in relation R" construct used in some real query language with parallel execution semantics; see [AU, section 7]. It remains to be seen whether a rigorous definition of the semantics of such a language, together with the dual "for some tuple t in R", yields a language equivalent to IND.

References

[AU] Aho, A.V. and J.D. Ullman, Universality of data retrieval languages. *Proc. 6th ACM Symp. on Principles of Programming Languages*, Jan. 1979, 110-117.

[AP] Apt, K. and G. Plotkin, A Cook's Tour of Countable Nondeterminism, ICALP '81.

[B] Barwise, J. *Admissible Sets & Structures*. Springer-Verlag 1975.

[CH1] Chandra, A.K. and D. Harel, Computable queries for relational data bases. *JCSS 21*; 2, Oct. 1980.

[CH2] Chandra, A.K. and D. Harel, Structure and Complexity of Relational Queries, *Proc. 21st IEEE Symp. on Foundations of Computer Science*, Oct. 1980, 333-347.

[CH3] Chandra, A.K. and D. Harel, Horn clauses and the fixpoint query hierarchy. *SIGACT-SIGMOD Symp. on Principles of Data Base Systems*, March, 1982.

[CKS] Chandra, A.K., D. Kozen and L. Stockmeyer, Alternation, *J. ACM*, Jan. 1981.

[C1] Codd, E.F., A relational model for large shared data bases. *CACM 13*; 6, June 1970.

[C2] Codd, E.F., Relational completeness of data base sublanguages. In *Data Base Systems* (Rustin, ed.), Prentice Hall, 1972.

[C] Cook, S.A. Soundness and Completeness of an Axiom System for Program Verification, *SIAM J. on Computing 7*; 1, (1978).

[GM] Gallaire, H. and J. Minker (eds.), *Logic and Data Bases*, Plenum, New York (1978).

[GFMR] Grumberg, Francez, Makowsky and de Roever, A Proof Rule for fair Termination of Guarded Commands, Technion Tech. Report #197, Feb. 1981.

[H] Harel, D. *First-Order Dynamic Logic*. Lecture Notes in Computer Science 68, Springer-Verlag 1979.

[I] Immerman, N. Relational queries computable in polynomial ti,e. *14th ACM Symp. on Theory of Computing*, May 1982.

[K] Kowalski, R.A. Predicate logic as a programming language. *Proc. IFIP74*, North-Holland 1974, 556-574.

[LPS] Lehman, D., A. Pnueli and J. Stavi, Impartiality, Justice and Fairness: The Ethics of Concurrent Termination, ICALP '81.

[MT] Meyer, A.R. and J. Tiuryn, A Note of Equivalences Among Logics of Programs. *Proc. Workshop on Logics of Programs 1981*, Lecture Notes in Computer Science 131, Springer-Verlag, 282-299.

[MP] Meyer, A.R. and R. Parikh, Definability in Dynamic Logic, *Proc. 12th ACM Symp. on Theory of Computing* (1980), 1-8.

[MW] Meyer, A.R. and K. Winkelmann, On the Expressive Power of Dynamic Logic, *Proc. 12th ACM Symp. on Theory of Computing* (1979), 167-175.

[Mi] Mirkowska, G. On Formalized Systems of Algorithmic Logic, Bull. Acad. Pol. Sci., Ser. Math. Astr. Phys. *22* (1974), 421-428.

[Mo] Moschovakis, Y.N. *Elementary Induction on Abstract Structures*, North-Holland, 1974.

[Pr] Pratt, V. Semantical Considerations on Floyd-Hoare Logic. *Proc. 17th IEEE Symp. on Found. of Comp. Science* (1976), 109-121.

[T] Tiuryn, J. Unbounded Program Memory adds to the Expressive Power of First-Order Dynamic Logic, *Proc. 22nd IEEE Symp. on Found. of Comp. Science* (1981).