Chapter 1

Logic and the Challenge of Computer Science*

YURI GUREVICH†

Abstract—Nowadays computer science is surpassing mathematics as the primary field of logic applications, but logic is not tuned properly to the new role. In particular, classical logic is preoccupied mostly with infinite static structures whereas many objects of interest in computer science are dynamic objects with bounded resources. This chapter consists of two independent parts. The first part is devoted to finite model theory; it is mostly a survey of logics tailored for computational complexity. The second part is devoted to dynamic structures with bounded resources. In particular, we use dynamic structures with bounded resources to model Pascal.

INTRODUCTION

These days computer science is characterized by an explosive growth in activities intimately related to logic. Consider for example formal languages. For years formal languages were in the private domain of logicians. But what formal language is most popular today? Is it a Hilbert type predicate calculus or the Genzen sequent calculus? Neither. The most popular formal languages of today are programming languages. Another kind of popular formal languages are database query languages. Some other formal languages emerge in artificial intelligence like languages for knowledge representation. Old discussions on names, denotations, types, etc. are suddenly revitalized to unprecedented magnitude.

*Supported in part by NSF grants MCS 83-01022 and DCR 8503275.

†Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, MI 48109-2122.

2 Logic and the Challenge of Computer Science

The work on axiomatic semantics, logic programming and verification is related to classical proof theory; the work on computational complexity is related to the classical theory of algorithms. Even propositional logic is not left untouched by developments in computer science; for almost any number *k* between 3 and 20, there is a commercial logic circuit simulator based on *k*-valued logic (41).

This is altogether good news for logicians. Logic grows more relevant to computer science than any other part of mathematics. But the new applications call, we believe, for new developments in logic proper. First-order predicate calculus and its usual generalizations are not sufficient to support the new applications. On the other hand, the new developments will most probably build on existing achievements of logic. In this connection it is worth trying to understand what made classical mathematical logic so successful.

Even though logic is an ancient subject, the origins of modern mathematical logic are closely related to the discovery of paradoxes and the subsequent crisis in the foundations of mathematics (47). In 1930 came the triumph of Gödel's completeness theorem. The syntax of first-order predicate calculus and its semantics were proven to match perfectly. In addition first-order logic was restrictive enough to avoid paradoxes and expressive enough to provide a basis for Zermelo-Fraenkel set theory and resolve this way, to a large extent, the foundational crisis. This perfect match of syntax and semantics together with a reasonable expressive power made first-order logic an invaluable tool and a source of innumerable generalizations.

Extremely important features of first-order logic are a formal language and a clear notion of models. The models are so-called first-order structures or, simply, structures. (Some people object to the term "first-order structure" on the ground that logic is first-order rather than structures. This is a good point. But some structures are not first-order, topological spaces for example; and we all know exactly what first-order structures are.) This familiar pattern—a formal language with well-defined models—persists through familiar generalizations of first-order logic.

Let us mention another interesting feature of first-order logic. Even though a consistent first-order theory has usually a multitude of models, the theory itself does not refer directly to different models; it "speaks" about "the" model of discourse.

Classical logic facilitated numerous and impressive achievements. Let us mention only the Church-Turing thesis and the Gödel-Cohen resolution of the continuum hypothesis. It seems that we (the logicians) were somewhat hypnotized by the success of classical systems. We used first-order logic where it fits well and where it fits not so well. We went on working on computability without paying adequate attention to feasibility. One seemingly obvious but nevertheless important lesson is that different applications may require formalizations of different kinds. It is necessary to "listen" to the subject in order to come up with the right formalization. (We philosophized on this topic in [30].)

An important feature of many computer science objects is finiteness. Relational databases constitute an especially important example. Finiteness does not seem to be such a great novelty in classical logic. Nevertheless it poses a nontrivial challenge. Being so closely related to foundations of mathematics, classical logic is preoccupied with infinity. Many famous theorems collapse when only finite structures are allowed; among them are Gödel's Completeness Theorem, Craig's Interpolation Theorem, Bern's Definability Theorem and the Substructure Preservation Theorem (29).

Variants of first-order logic serve as standard relational query languages (15, 67), but the expressive power of first-order logic is not sufficient for many purposes (2). On the other hand, second-order logic is overly expressive. It expresses queries that are too hard to compute. Even existential monadic second-order formulas can express NP complete queries. Of course, the notion of what is hard may change from one application to another. One idea is to fix a reasonable complexity class, like polynomial time, and to devise an intermediate logic that "captures" this complexity class i.e. expresses exactly the queries of that complexity. The idea happens to be realizable to an extent. The pioneering papers include those of Aho and Ullman (2), Chandra and Harel (12b), Fagin (20), Immerman (44), and Vardi (68). In particular, Immerman and Vardi proved that, in the presence of linear order, the least-fixed-point extension of first-order logic captures polynomial time. The program of designing logics to capture complexity classes was clearly spelled out in (45) where Immerman captured log-space and a number of other natural complexity classes. We have written on finite model theory and logic tailored for complexity in different places; see in particular (29).

Part 1 (Sections 1-9) of this chapter is devoted to finite model theory; it is mostly a subjective survey of logics tailored for computational complexity. Section 1 contains provisos and definitions that are used throughout Part 1. In particular, the notions of global relations and global functions are introduced; these notions provide convenient semantics for complexity tailored logics. In Sections 2, 3, 4 and 6 we consider different extensions of first-order logic by additional constructs; in the presence of linear order the extended logics capture natural complexity classes. In Section 5 we consider two logics with an emphasis on functions rather than predicates; a linear order is built in, and the logics capture log-space and polynomial time respectively. Section 7 is devoted to those properties of structures which do not depend on presentation. In Section 8, some evidence is given that certain familiar complexity classes cannot be captured by any logic. Circuit definability and topology on finite sets are briefly discussed in Section 9.

Remark Several relevant issues are left out in this survey. In particular, we do not discuss derivability in first-order predicate calculus. The questions of expressibility and derivability are quite different. For example, no first-order

formula φ expresses on finite graphs that (x, y) belongs to the transitive closure of the edge relation E. This is well known (21, 23, 29) and remains true even if φ is allowed to use additional predicate symbols: just consider the case when all additional relations are trivial. On the other hand, the first-order formula

$$[\forall uv(\mathrm{E}uv \to \mathrm{T}uv) \,\&\, \forall uvw(\mathrm{E}uv \,\&\, \mathrm{T}vw \to \mathrm{T}uw)] \to \mathrm{T}(x,y)$$

is derivable from the diagram of an arbitrary finite graph if and only if (x, y) belongs to the transitive closure of the edge relation E.

Another important feature of many computer science structures, which is harder to swallow, is their dynamic character. Mathematical structures (graphs, groups, topological spaces, etc.) do not change in time whereas computer science objects (databases, machines) often do. Considering time as a new dimension, a mathematician turns a dynamic situation into a static one. Complexity considerations may make such a transformation inadvisable in computer science (see Section 10 in this connection).

In Part 2 of this chapter we generalize the static structures of mathematical logic to dynamic structures. We are especially interested in dynamic structures with bounded resources. In Section 10, among other things, we discuss the adaptation of Turing's thesis to the case of machines with bounded resources. In Section 11, on the example of Pascal, we demonstrate an approach to semantics based on dynamic structures.

There are still mathematicians that consider computer science a lower subject. There are former logicians that work now in computer science or computer applications and consider logic not very relevant to their new occupation. We happen to think that computer science badly needs what logicians are supposed to do best: logic. The situation seems to us reminiscent of that in the beginning of the century. Again we face most basic questions like what is the right logic and even what are the right structures.

Acknowledgements. This chapter grew out of my part in the Course on Computation Theory in the International Center for Mechanical Sciences, Udine, Italy in September-October 1984. I am happy to thank the organizer—Dr. Egon Börger—and the Center for the invitation, and the listeners for their attention, good will and hard work. Special thanks are due to Dr. Klaus Ambos-Spies who faithfully recorded my lectures. An edited version of the lectures was published as a technical report (32). I am thankful to John Holland for his comments on the report. In Summer and Fall of 1986, the report was updated; in particular, Section 10 was enhanced and Section 11 was added. These two sections carry their own acknowledgements, but I am only too glad to repeat here that I am thankful to Kit Fine, Bernie Galler, David Gries, Albert Meyer, and Jim Morris. Finally, it gives me special pleasure to thank Andreas Blass for his numerous comments and many clarifying enjoyable discussions.

PART 1. FINITE MODEL THEORY

SECTION 1. GLOBAL RELATIONS AND FUNCTIONS

This section is devoted primarily to the notions of global relations and global functions which will be used to provide semantics for numerous logics. The section contains a number of definitions, two principles and one proviso that will be widely used throughout Part 1.

The notions of global relations and global functions were introduced in (28). To motivate the definition of a global relation, let us consider a formula $\varphi(x,y)$, with two free individual variables, in the first-order language of graphs. What is the meaning of $\varphi(x,y)$? Is it a binary relation? Well, it is and it is not. Given a graph, one can interpret $\varphi(x,y)$ as a binary relation. In general, $\varphi(x,y)$ can be interpreted as a function that assigns a binary relation to each graph. Such functions will be called global relations.

Definition 1.1 Let K be a class of first-order structures of some signature (vocabulary) σ. An r-ary *K-global relation* ρ assigns to each structure S in K an r-ary relation ρ^S on S; the relation ρ^S is the *specialization* of ρ to S. The signature σ is the *signature* of ρ. If K is the class of all permissible σ-structures we say that p is *σ-global.*

Right now all structures are permissible. Later we will permit only finite structures satisfying some additional restrictions.

The notion of global relations generalizes Tarski's notion of sentential functions (62). Sentential functions are global relations of arity zero. In a sense, the notion of global relations reduces to the notion of sentential functions: an r-ary global relation of signature σ can be viewed as a sentential function whose signature is an extension of σ by r additional individual constants. But it is more convenient to work directly with global relations.

Tarski's semantics for first-order logic can be conveniently formulated in terms of global relations (disallow function symbols for a moment). The meaning of a first-order formula φ with r free individual variables is a σ-global r-ary relation where σ is the signature (vocabulary) of φ, i.e., the set of predicate symbols in φ. The meaning is defined by an obvious induction.

Remark There is one relatively minor issue that we are going to ignore. Different orderings of the free individual variables of a first-order formula give different global relations. One way to resolve this difficulty is to stick to the lexicographical ordering of individual variables. Another possibility is to use a more explicit notation like $\{(x_1 \ldots, x_n): \varphi\}$.

Examples of global relations:

1. Let GRAPH be the class of finite graphs seen as structures with exactly one relation which is binary, irreflexive and symmetric. The following GRAPH-global relations are of arities 0, 1, and 2, respectively:
   The graph is connected,
   Node x has at most log n neighbors where n is the number of nodes,
   There is a path from node x to node y.
2. Let GROUP be the class of finite groups. The following GROUP-global relations are of arities 0,1, and 3, respectively:
   The group is abelian,
   The index of the subgroup, generated by element $\boldsymbol{x}$, is at most log n where n is the number of elements,
   The subgroup generated by elements x and y contains element z.

Definition 1.2 Let K be a class of structures of some signature σ A *K-global function f of type* $(\text{Universe})^r \to \text{Universe}$ assigns to each structure S in K an r-ary function f^S that, given an r-tuple of elements of S, produces an element of S. The signature σ is the *signature* of f.

First-order terms denote global functions. In the obvious way, global relations and global functions of types $(\text{Universe})^r \to \text{Universe}$ provide semantics for first-order logic with function symbols. (View individual constants as zero-ary functions.)

We will keep the notion of global functions informal (and very general) and will deal only with global functions of specific types. In particular, an r-ary global relation is a global function of type $\text{Universe}^r \to \text{Bool}$ where Bool is the set of the two truth values. A K-global function f of type $(\text{Universe})^p \to (\text{Universe})^q$ assigns to each S in K a function f^S that, given a p-tuple of elements of S, produces a q-tuple of elements of S; we say that f, as well as each specialization f^S of f, is p-ary and q-coary. The notion of a K-global partial function f of type $(\text{Universe})^p \to (\text{Universe})^q$ is an obvious generalization; f itself is total (defined on the whole K) but its specializations may be partial. Other possible types of global function include

$$[(\text{Universe})^p \to \text{Bool}] \to [(\text{Universe})^q \to \text{Bool}], \quad \text{and}$$

$$[(\text{Universe})^p \times (\text{Power-Set}(\text{Universe}))^q] \to \text{Bool}.$$

The latter is the type of second-order formulas with p free individual variables and q free predicate variables that are all monadic. The meaning of any second-order formula is a global function of an appropriate type.

The Localization Principle. Think about global relations and global functions as relations and functions (of appropriate types) on the structure of discourse.

The localization principle allows us to speak about the negation of a given global relation, about the transitive closure of a given binary global relation, about composition of unary global functions, etc.

Proviso. In Part 1,

1. any structure is a finite first-order structure of finite signature,

2. the universe of any structure is an initial segment of natural numbers,

3. any class of structures consists of structures of the same signature, and

4. the domain of any global relation comprises all structures of some signature that are permitted in the context, unless the contrary is said explicitly.

The proviso allows us to associate a decision problem with each global relation.

Definition 1.3 Let p be an *r*-ary *K*-global relation. An *instance* of the decision problem for ρ is a pair $\langle S, x \rangle$ where *S* belongs to *K* and *x* is an *r*-tuple of elements of *S*; the corresponding *question* is whether $\rho(x)$ holds in *S* (in other words, whether *x* belongs to ρ^S).

However, we need to agree on a standard way to represent structures as inputs for computing devices. To simplify the exposition, we choose to represent structures by means of several input tapes. Suppose that *S* is a structure of cardinality *n*. One input tape, called the *universe tape,* represents the universe $\{0, 1, \ldots, n - 1\}$ of *S*; it is of length *n,* its end-cells are specially marked but the intermediate cells are all blank. (Ignore the case of $n = 1$.) If *R* is a basic *r*-ary relation of *S* or the graph of an $(r - 1)$-ary basic function of *S* then *R* is represented by a special tape of length n^r; for all elements $x_0, \ldots, x_{r-1}$, the cell number $\sum x_i \cdot n^i$ contains 1 if $R(x_{r-1}, \ldots, x_0)$ holds, and 0 otherwise.

The Globalization Principle. View relations and functions under discussion as the specializations of global relations and global functions to the structure of discourse.

The globalization principle can be applied only if the context uniquely defines appropriate relations or functions on all relevant structures. For example, suppose that a discussion involves the transitive closure *R* of a basic relation of the structure of discourse. Then the globalization principle allows us to speak about *R* being polynomial time recognizable.

Fagin proved (20) that existential second-order logic captures nondeterministic polynomial time.

Theorem 1.1 A global relation is definable by an existential second-order formula if and only if it is recognizable by a polynomial time bounded nondeterministic Turing machine. ■

The proof of Theorem 1.1 may be found in the third section of Börger's contribution to this volume (Chapter 2).

It is easy to see that every first-order definable global relation is log-space (and therefore polynomial time) recognizable. The converse is not true. For example, the global relation "The cardinality of the universe is even" is log-space recognizable but not first-order definable. As we will see below, some natural extensions of first-order logic express exactly log-space (respectively polynomial time) recognizable global relations.

Definition 1.4 Let L be first-order logic or an extension of first-order logic by additional logical operators. (A number of such extensions will be defined in subsequent sections.) $L + <$ is the extension of L by means of a logical constant $<$ (just as first-order logic with equality is the extension of first-order logic by means of a logical constant =). The logical constant $<$ is interpreted on each structure S as the restriction of the usual order of natural numbers to the universe of S.

SECTION 2. TRANSITIVE CLOSURES

This section is devoted to transitive closure logics. We start our treatment of different extensions of first-order logic with transitive closure logics because of their relative simplicity. The use of two-way multihead automata will allow us to simplify the proofs related to capturing complexity classes.

The localization principle implicitly introduces the transitive closure of a given binary global relation. The transitive closure of a first-order expressible binary global relation may be not first-order expressible; see (2, 23, 29). In this connection, Aho and Ullman (2) suggested extending the relational calculus, a standard relational query language and a variant of first-order logic, by a powerful least fixed point operator. Immerman (45) turned the transitive closure itself into a logical operator TC. He defined also a deterministic transitive closure operator DTC, and proved that the corresponding extensions FO + TC + < and FO + DTC + < of first-order logic capture natural complexity classes. We prove here some of Immerman's results.

Definition 1.5 If R is a relation of an even arity $2r$ over some universe U then the relation $\{(x,y)$: tuples x,y belong to U^r, and the concatenation $x*y$ belongs to $R\}$ is the *binary companion* BC(R) *of R.* The *transitive closure* of a $2r$-ary relation relation R is the $2r$-ary relation TC(R) (over the same universe) whose binary companion is the transitive closure of *BC(R).* With respect to the localization principle, the *transitive closure* of a $2r$-ary global relation ρ is the global relation TC(ρ) such that the domain of TC(ρ) equals that of ρ and for each structure S in the domain of ρ, the specialization of TC(ρ) to S is the transitive closure of ρ^S.

It will be convenient for us in this section to play down the distinction between relations of even arity and their binary companions.

Lemma 1.1 If a $2r$-ary global relation ρ is nondeterministically log-space recognizable (i.e. if the decision problem for ρ is solvable in nondeterministic log-space) then so is TC(ρ).

Proof Let S be a structure in the domain of ρ, R be the specialization of ρ to S, and a,b be r-tuples of elements of S. The desired algorithm is:

```
begin
  x := a;
  repeat
  guess y;
  if (x,y) ε R  then  x := y
  until x = b;
  halt with output YES
end.
```

■

Notice the use of the globalization principle in the exposition of the proof.

We define a logic FO + TC. The syntax of FO + TC is the extension of the syntax of first-order logic by:

Transitive Closure Formation Rule. Let r be a positive integer and $\varphi(x,y)$ be a well-formed formula where x and y are r-tuples of individual variables such that the $2r$ variables are distinct. Then $TC_{x,y}\varphi(x,y)$ is a well-formed predicate, and if s,t are r-tuples of well-formed terms then $[TC_{x,y}\varphi(x,y)](s,t)$ is a well-formed formula.

$TC_{x,y}$ binds the $2r$ individual variables in the new predicate (but the additional occurrences of these variables in the tail of a formula $[TC_{x,y}\varphi(x,y)](x,y)$ are free). $\varphi(x,y)$ may have additional free individual variables. A more explicit

notation for the new predicate is $TQ_{x,y}\ \varphi(w,x,y)$ where w is the list of those additional variables. The new formula $[TC_{x,y}\varphi(w,x,y)](s,t)$ means (on each relevant structure) that (s, t) belongs to the transitive closure of the relation $R_w = \{(x,y): \varphi(w,x,y)\}$. The global function semantics for first-order logic naturally extends to logic FO + TC; again the meaning of a formula with r free individual variables is a global r-ary relation.

The transitive closure formation rule introduces well-formed predicates in addition to well-formed formulas. The only well-formed predicates in first-order logic are predicate symbols. The transitive closure formation rule is essentially a predicate formation rule. The new predicate is then used to form new formulas. But it is possible to deal only with formulas of course.

Remark Immerman (45) seems to define directly a new formula $TC[\varphi(x,y)]$ which is a simpler notation for $[TC_{x,y}\varphi(x,y)](x,y)$. Unfortunately, the simpler notation is somewhat deficient. Try to express $[TC_{x,y}P(x,y,x)](x,x)$ or $[TC_{x,y}P(x,y,x)](fx,x)$ in the simplified notation.

Positive and negative occurrences of a predicate in a formula are defined by induction. In particular, every positive (respectively negative) occurrence of a predicate in a formula φ remains so in any formula $[TC... \varphi](. . .)$. Say that a formula φ is *positive with respect to TC* if every occurrence of every predicate $TC...\psi$ in φ is positive.

In Section 1, we spoke about extensions $L +<$ of logics L by means of the built-in linear order. In particular, we have an extension FO + TC + < of FO + TC. Viewing 0 and 1 as logical constants yields a further extension FO + TC + < + {0, 1}.

Theorem 1.2 Let ρ be a global relation. The following are equivalent:

1. ρ is nondeterministic log-space recognizable,

2. ρ is definable by an FO + TC + < formula φ which is positive with respect to TC.

3. ρ is definable by a FO + TC + < + {0,1} formula $[TC_{x,y}\ \psi(x,y)](s,t)$ where s,t are sequences of zeros and ones, and ψ is first-order.

Proof (3) → (2). The constants 0 and 1 are definable in FO + <.

(2) → (1). Without loss of generality, one may suppose that only first-order subformulas can be negated in the defining formula φ: use the usual duality laws for first-order logic. Then an easy induction shows that every subformula of the defining formula is nondeterministically log-space recognizable. The case of TC is taken care of in Lemma 1.1.

To prove the implication (1) → (3), suppose that ρ is recognizable in non-deterministic log-space. According to the Appendix, there is a nondeterministic two-way multihead automaton that recognizes ρ. Let formula Next(*w,x,y*) and tuples Initial, Final be as in the Appendix. Then the desired FO + TC + < + {0, 1} formula is

$$[TC_{x,y}\ \text{Next}(w,x,y)](\text{Initial},\text{Final})$$ ■

Definition 1.6 The *deterministic version* of a binary relation *R* is the relation $\{(x,y)$: $(x,y)\varepsilon R$ and there is no $z \neq y$ with $(x,z)\varepsilon R\}$. The *deterministic version* of a 2*r*-ary relation *R* is the 2*r*-ary relation whose binary companion is the deterministic version of BC(*R*). The *deterministic transitive closure* DTC(*R*) of a 2*r*-ary relation *R* is the transitive closure of the deterministic version of *R*. With respect to the localization principle, the *deterministic transitive closure* of a 2*r*-ary global relation ρ is the global relation DTC(ρ) such that the domain of DTC(ρ) equals that of ρ and for each structure *S* in the domain of ρ, the specialization of DTC(ρ) to *S* is the deterministic transitive closure of ρ^S.

Lemma 1.2 If a 2*r*-ary global relation ρ is log-space recognizable then so is DTC(ρ).

Proof Let *S* range over the domain of ρ, U be the universe of the structure *S*, $R = \rho^S$, and *a,b* be *r*-tuples of elements of U. We need an algorithm which, given *S* and (*a,b*), will decide whether (*a,b*) belongs to DTC(*R*).

The deterministic version *of R* is the graph of some partial function *f* on U^r. Given *x* in U^r, one can find in log-space whether there is some *y* with $(x,y)\varepsilon R$ and whether there are different *y,z* with $(x,y)\varepsilon R$ and $(x,z)\varepsilon R$ .This yields a log-space algorithm which, given *x*, computes *fx* or UNDEFINED. Given *a* and *b*, compute $f^k a$ for $k = 1, 2$, etc. and halt when *b* comes along or UNDEFINED is returned or *k* reaches *n*. If *b* has come along then return YES, otherwise return NO. ■

Again, the globalization principle was used to simplify the exposition of the proof.

The definition of an extension FO + DTC of first-order logic is similar to the definition of FO + TC. Just change 'TC" to "DTC," and "transitive closure" to "deterministic transitive closure."

Theorem 1.3 Let ρ be a global relation. The following are equivalent:

1. ρ is log-space recognizable,

2. ρ is definable in FO + DTC + <,

3. ρ is definable by a FO + DTC + < + {0,1} formula $[DTC_{x,y}\,\psi(x,y)](s,t)$ where s,t are sequences of zeros and ones, and ψ is first-order.

Proof Similar to that of Theorem 1.2. ■

Since the deterministic version of a given relation is first-order definable, FO + DTC can be seen as a sublogic of FO + TC.

SECTION 3. LEAST FIXED POINTS

In this section we define the extension FO + LFP of first-order logic by the least fixed point operator (2, 12) and prove Immerman and Vardi's theorem that FO + LFP + < captures polynomial time (44, 68). Again, the use of two-way multihead automata will allow certain simplification.

Definition 1.7 Let F be a unary operation on a partially ordered set. If $Fx = x$ then x is *a fixed point* of F. If $Fx = x$ and $\forall y(Fy = y \rightarrow x \leq y)$ then x is the *least fixed point* LFP(F) of F. If $Fx \leq Fy$ for all $x \leq y$ then F is *monotone*.

Definition 1.8 A partially ordered set is *complete* if every subset of it has a least upper bound and a greatest lower bound.

For example, the set of relations of a fixed arity on a fixed nonempty set is a complete partially ordered set with respect to inclusion. The following fact is well-known.

Fact Let D be a finite (or infinite complete) partially ordered set with a least element. A monotone unary operation F *on* D has a least fixed point.

Proof Let $g0 = \min(D)$ and each $g(\alpha+1) = F(g\alpha)$. (In the case of infinite D), let additionally $g\alpha = \sup\{g\beta: (\beta < \alpha\}$ for limit α.) By monotonicity, the function g is increasing (though not necessarily strictly increasing). Hence, there is α with $g\alpha. = g(\alpha + 1)$; let $\gamma = \min\{\alpha: g\alpha = g(\alpha + 1)\}$. Obviously, $g\gamma$ is a fixed point of F. Given a fixed point y of F, prove by induction that each $g\alpha \leq y$. Hence $g\gamma = \mathrm{LFP}(F)$. ■

The localization principle gives:

Definition 1.9 Let F be a σ-global function of type

$$[\text{Power-Set}(\text{Universe}^r)] \rightarrow [\text{Power-Set}(\text{Universe}^r)]$$

so that each specification of F takes an r-ary relation to an r-ary relation. F is *monotone* if every specialization of F is so. If F is monotone then the *least fixed*

point LFP(F) of F is the σ-global r-ary relation that assigns to each a-structure S the least fixed point of F^S. F is *polynomial time computable* if there is a polynomial time algorithm that, given a σ-structure S and an r-ary relation P on S, computes $F^S(P)$.

Lemma 1.3 Let F be a monotone σ-global function of type

$$[\text{Power-Set}(\text{Universe}^r)] \rightarrow [\text{Power-Set}(\text{Universe}^r)].$$

If F is polynomial time computable then LFP(F) is polynomial time recognizable.

Proof Given a structure S and an r-tuple x of elements of S, compute $P_0 = \emptyset$ $p_1 = F^S(P_0)$, $P_2 = F^S(P_1)$, etc. until you come across $P_{m+1} = P_m$. Then check whether x belongs to P_m. Since $m \leq |S|^r$, this algorithm works in polynomial time. ■

The syntax of logic FO + LFP is the result of augmenting the syntax of first-order logic by the following formation rule. (If all predicate symbols of first-order logic are treated as predicate constants then first-order logic should be augmented by predicate variables first.)

Least Fixed Point Formation Rule. Let r be a positive integer, x be an r-tuple $x_1, \ldots, x_r$ of individual variables, P be an r-ary predicate variable, and $\varphi(P, x)$ be a well-formed formula. If $\varphi(P,x)$ is positive in P (i.e. all free occurrences of P in $\varphi(P,x)$ are positive) then $\text{LFP}_{P;\,x}\,\varphi(P,x)$ is a well-formed predicate and, for every r-tuple t of well-formed terms, $[\text{LFP}_{P;\,x}\,\varphi(P,x)](t)$ is a well-formed formula.

$\text{LFP}_{P;\,x}$ binds the predicate variable P and the individual variables $x_1, \ldots, x_r$ (but of course additional occurrences of these individual variables in the tails of formulas $[\text{LFP}_{P;\,x}\,\varphi(P,x)](t)$ are free). If Q is a predicate variable different from P then every positive (respectively, negative) occurrence of Q in $\varphi(P,x)$ remains positive (respectively, negative) in the new predicate and the new formulas.

Remark A simplified notation $\text{LFP}_P\varphi(P,x)$ for $[\text{LFP}_{P;x}\varphi(P,x)](x)$ is deficient: just try to express $[\text{LFP}_{P;\,x}\varphi(P,x)](t)$ in the simplified notation.

To be on the safe side, let us emphasize that logic FO + LFP allows interleaving LFP with propositional connectives (including negation) and quantifiers; in particular, one can negate an LFP formula then use the LFP formation rule again, etc.

The formula $\varphi(P,x)$ may have additional free individual variables; let w be the list of the additional individual variables. The meaning of the predicate $\mathrm{LFP}_{P;x}\varphi(P,w,x)$ is the least fixed point of the operator $F_w(P) = \{x: \varphi(P,w,x)\}$ on the set of r-ary relations ordered by inclusion. Since the formula $\varphi(P,w,x)$ is positive in P, the operator F_w is monotone and therefore has a least fixed point. The global function semantics for first-order logic naturally extends to FO + LFP.

Theorem 1.4 Let ρ be a global relation. The following are equivalent:

1. ρ is polynomial time recognizable,

2. ρ is definable in logic FO + LFP + <,

3. ρ is definable by a FO + LFP + < + {0,1} formula $[\mathrm{LFP}_{P;x}\psi(P,x)](t)$ where ψ is first-order and t a sequence of zeros and ones.

Proof The implications (3) → (2) and (2) → (1) are obvious. To prove the implication (1) → (3), suppose that ρ is polynomial time recognizable. According to the Appendix, there is an alternating two-way multihead finite automaton that recognizes ρ. Let the formula Next(w,x,y) and the tuples Initial, Final be as in the Appendix. It is easy to write down first-order formulas Existential(x) and Universal (x) asserting that the internal state of the automaton in the given configuration x is respectively existential or universal. Let

$$\mathrm{Accepted}(w,\_) = \mathrm{LFP}_{P;x}[x = \mathrm{Final}, \quad \text{or}$$

$$\mathrm{Universal}(x) \,\&\, \forall y(\mathrm{Next}(w,x,y) \rightarrow P(y)), \quad \text{or}$$

$$\mathrm{Existential}(x) \,\&\, \exists y \mathrm{Next}(w,x,y) \,\&\, P(y))].$$

The desired FO + LFP formula is Accepted(w, Initial). ■

SECTION 4. BRANCHING QUANTIFIERS

We turn now to an extension of first-order logic by branching (or Henkin) quantifiers whose introduction was motivated by considerations quite distant from computer science (42).

Let us start with an example. The expression

$$\begin{bmatrix} \forall u \exists v \\ \forall x \exists y \end{bmatrix} \varphi(u,v,x,y) \qquad (1.1)$$

means that for all u and x there are v and y such that v depends only on u, y depends only on x, and $\varphi(u,,v,x,y)$ holds. In other words, there are functions $V(u)$ and $Y(x)$ such that $\varphi(u,V(u),x,Y(x))$ holds.

In general, a branching quantifier is a partially ordered set of expressions $\forall x$ and $\exists y$; an existentially quantified variable y depends upon the universally quantified variables x such that $\forall x$ precedes $\exists y$ in the partial order (69).

Theorem 1.5 For any global relation ρ the following are equivalent:

1. ρ is NP,

2. ρ is expressible by an existential second-order formula,

3. ρ is expressible by a formula $Q\varphi$ where Q is a branching quantifier and φ is a first-order formula.

Proof The equivalence (1) $\leftrightarrow$ (2) is Theorem 1.1 in §1, the implication (3) $\rightarrow$ (2) is obvious, the implication (2) $\rightarrow$ (3) is proved in (69). ■

In the rest of the section we describe a few results from (8). The only novelty is the direct proof of Theorem 1.8 below. A branching quantifier Q will be called *mighty* if there is a first-order formula φ such that the global relation $Q\varphi$ is NP-complete under polynomial time reductions.

Theorem 1.6 The quantifier (1.1) is mighty.

Proof The idea is to express 3-colorability of a graph with individual constants 0, 1, and 2. The desired φ is the conjunction of the formulas:

$$u = x \rightarrow v = y,$$

$$v = 0 \text{ or } v = 1 \text{ or } v = 2,$$

$$\text{Edge}(u, x) \rightarrow v \neq y.$$

■

Note that, in the proof of Theorem 1.6, the existentially quantified variables range, in effect, over $\{0,1,2\}$. Let α, β, γ range over $\{0,1\}$, and μ range over $\{0,1,2\}$.

Theorem 1.7 The quantifiers

$$\begin{bmatrix} \forall x\, \exists \alpha \\ \forall y\, \exists \beta \\ \forall z\, \exists \gamma \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \forall x\, \exists \alpha \\ \forall y\, \exists \mu \end{bmatrix} \quad \text{are mighty.}$$

We omit the proof of Theorem 1.7.

■

In the rest of this section, x and y are tuples of individual variables. The branching quantifier

$$\begin{bmatrix} \forall x\, \exists \alpha \\ \forall y\, \exists \beta \end{bmatrix}$$

will be called the *narrow Henkin quantifiers* and denoted NH(x,α; y,β). Without loss of generality, x and y always have the same length: just pad the shorter tuple. Let ENH(x,α; y,β) be the equality bound version of NH(x,α; y,β):

$$\mathrm{NH}(x,\alpha; y,\beta)[(x = y \rightarrow \alpha = \beta) \ \& \ \varphi(x, y, \alpha, \beta)].$$

ENH(x,α; y,β) asserts (in each relevant structure) the existence of a function f from the universe to {0, 1} such that for all x and y, φ(x,y,f(x),f(y)) holds. In the rest of this section, we assume that α and β range over the truth-values rather than over {0,1}. Then ENH(x,α; y,β)φ(x, y, α, β) is equivalent to the second-order formula

$$\exists R \forall xy \varphi(x,y,R(x),R(y)).$$

An arbitrary NH(x,α; y,β)φ(x, y, α, β) is equivalent to

$$\mathrm{ENH}(xu,\alpha; yv,\beta)[(u = 0 \ \& \ v = 1) \rightarrow \varphi(x, y, \alpha, \beta)].$$

Let FO + NH be the extension of first-order logic by narrow Henkin quantifiers. Positive and negative occurrences of a subformula ψ in a formula φ are defined by the obvious induction on φ; in particular, any positive (respectively negative) occurrence of ψ in φ(u, v, γ, δ) remains so in NH(x,α; y,β)φ(x, y, α, β). We will say that a formula φ is *positive with respect to* NH if every occurrence of every subformula of the form NH(x,α; y,β)ψ(x, y, α, β) in φ is positive. Abbreviate "nondeterministic log-space" as "Nlog-space."

Theorem 1.8 For a global relation ρ the following are equivalent:

1. ρ is co-Nlog-space recognizable,

2. ρ is expressible by an FO + NH + < formula which is positive with respect to NH,

3. ρ is expressible by an FO + NH + < formula ENH(x,α; y,β)φ(x, y, α, β) with a first-order φ.

Proof (1) → (3). Suppose that ρ is co-Nlog-space recognizable, and let ρ' be the complement of ρ (so that on each relevant structure, the specification of ρ' is the complement of the specification of ρ). According to the Appendix, there is a two-way multihead nondeterministic finite automaton that recognizes ρ '. Let formula Next(w,x,y) and tuples Initial and Final be as in the Appendix. The desired formula expresses the nonacceptance by the automaton:

$$\mathrm{ENH}(x,\alpha;\, y,\beta)[(x = \mathrm{Initial} \rightarrow \alpha = 1)\ \&$$
$$((\alpha = 1\ \&\ \mathrm{Next}(w,x,y)) \rightarrow \beta = 1)\ \&$$
$$(y = \mathrm{Final} \rightarrow \beta = 0)].$$

The implication (3) → (2) is trivial.

(2) → (l). Without loss of generality, we may suppose that only first-order subformulas of the defining formula can be negated: use the usual duality laws of first-order logic. By induction, we will prove that every subformula of the defining formula is co-Nlog-space recognizable. It suffices to prove that if $\varphi(x, y, \alpha, \beta)$ is co-Nlog-space then so is $\psi = \mathrm{ENH}(x,\alpha; y,\beta)\varphi(x, y, \alpha, \beta)$. Thus, suppose that *M′* is a log-space bounded nondeterministic Turing machine that recognizes the negation $\varphi'(x, y, \alpha, \beta)$ of $\varphi(x, y, \alpha, \beta)$. We have

$$\psi \leftrightarrow \exists R\ \forall x \forall y\ (x,y,Rx,Ry) \leftrightarrow \exists R\ \forall x \forall y \text{ not } \varphi'(x,y,Rx,Ry) \leftrightarrow$$

$$\exists R\ \Pi_{x,y} \text{ not } \sum_{\varphi'(x, y, \alpha, \beta)}\ (Rx = \alpha)\ \&\ Ry = \beta) \leftrightarrow$$

$$\exists R\ \Pi_{\varphi'(x, y, \alpha, \beta)}(Rx \neq \alpha)\ \&\ Ry \neq \beta).$$

Here $\Pi_{x,y}$ means (in each relevant structure) the conjunction over all values of x and y. For given values of x and y, $\sum_{\varphi'(x, y, \alpha, \beta)}$ means the disjunction over the values of α and β satisfying $\varphi'(x,y,Rx,Ry)$. And $\Pi_{\varphi'(x, y, \alpha, \beta)}$ means the conjunction over all values of x, y, α and β satisfying $\varphi'(x,y,Rx,Ry)$. For every value a of x, view Ra as a propositional variable. Then ψ asserts satisfiability of the propositional formula $\Pi_{\varphi'(x, y, \alpha, \beta)}(Rx \neq \alpha)$ or $Ry \neq \beta)$. Recall that a literal is a propositional variable or the negation of such.

Fact (49) A conjunction C of binary disjunctions of literals is unsatisfiable if and only if there are a propositional variable p and a sequence $l_1 \rightarrow l_2 \rightarrow \ \ldots \rightarrow l_m \rightarrow l_1$ of literals such that each implication $l_i \rightarrow l_{i+1}$ as well as the implication $l_m \rightarrow l_1$ is equivalent to a conjunct of C, and both p and the negation of p appear in the sequence.

Now we are ready to describe a log-space bounded nondeterministic Turing machine N that recognizes the negation of ψ. Let M be a log-space bounded nondeterministic Turing machine that recognizes φ'. Step-by-step N guesses a sequence $l_1 \rightarrow l_2 \rightarrow \ \ldots \rightarrow l_m \rightarrow\ l_1$ of literals that witnesses the unsatisfiability of $C = \Pi_{\varphi'(x, y, \alpha, \beta)}(Rx \neq \alpha)$ or $Ry \neq \beta)$. To check that an implication $l_i \rightarrow l_{i+1}$ (where $i+1 = 1$ if $i = m$) is equivalent to a conjunct of C, N presents l_i, in the form $Ra = \alpha$, presents l_{i+1} in the form $Rb \neq \beta$, and uses M to check $\varphi'(a, b, \alpha, \beta)$. ■

SECTION 5. FUNCTION LOGICS

First-order logic is essentially a logic of relations. It has one function construct: the composition, and a number of relation constructs: boolean connectives and the two quantifiers. It allows constructing formulas from terms but not the other way round. In Sections 2-4 we studied extensions of first-order logic by means of additional relation constructs. In this section, we turn to logic of functions in the case when only finite structures are permitted.

Consider the classical language of functions primitive recursive relative some given functions (47). Usually, primitive recursive terms are interpreted over the set of natural numbers. But here we interpret primitive recursive terms as functions over a nonempty finite initial segment of natural numbers. View the individual constant 0 as a name of the number zero, view the sign of successor function as a name of the partial successor function on the universe of discourse, and so on. Then every primitive recursive term means a global function. We take the liberty of extending the syntax by a new individual constant End for the last element of the universe of discourse. It turns out then that a global function is primitive recursive if and only if it is log-space computable (28). Similarly, a global function is recursive if and only if it is polynomial time computable (28, 57). This section recapitulates some of the paper (28). It has also a couple of new elements: a remark that primitive recursion can be replaced by a WHILE construct, and a simpler universal recursive schema.

We start with primitive recursive global functions. The language of primitive recursive functions will be reformulated in a form that is convenient for our purposes. In the same time we will extend the language by the individual constant End.

According to the proviso of Section 1, the universe of every structure is an initial segment of natural numbers. In this section, we have three additional provisos:

1. Every structure contains at least two elements. (Alternatively, one may assume existence of an extra universe Bool = {False, True}.)

2. Individual constants 0, End and a unary function symbol Successor are logical constants (as equality is a logical constant in first-order logic with equality). In every structure, 0 denotes the number zero, End denotes the maximal number in the universe, and Successor denotes the partial function $\lambda x(x+1)$. The three logical constants will not be counted as members of any signature.

3. A certain (possibly empty) signature σ is fixed. Every structure is a σ-structure. Every global function is σ-global.

In this section, a function (resp. global function) means a partial function (resp. partial global function) of type Universe^p -» Universe^q for some none-

gative integer p (the arity) and some positive integer q (the coarity). A function of coarity $q > 1$ can be seen as a sequence of q functions of coarity 1, but it will be convenient to deal directly with functions of higher coarity. If $t_1, \ldots, t_k$ are tuples of elements then $(t_1, \ldots, t_k)$ will denote the concatenation of tuples $t_1, \ldots, t_k$ rather than a k-tuple of tuples.

With respect to the localization principle, we view global functions as functions on the structure S of discourse. Let $U = \{0, \ldots, n - 1\}$ be the universe of S. The *value* of a nonempty tuple $(x_{k-1}, \ldots, x_1, x_0)$ of elements of U is the number $\sum_{i<k} x_i \bullet n^i$. If elements of U are seen as digits over the radix n then any nonempty tuple of elements of U is a positional notation over the radix n for its value.

Definition 1.10 The *initial functions* are:

1. For every nonnegative p, the constant p-ary functions with values 0 or End.

2. For every positive p, the p-ary p-coary successor function. Given a p-tuple of value $V < n^p$-1, the function produces the p-tuple of value $V + 1$; it is not defined on the p-tuple of value $n^p - 1$. We will denote the successor of a tuple t as $t + 1$.

3. For all $p \geq q \geq 1$ and every sequence $1 \leq i_1 \leq i_2 \leq \ldots \leq i_q \leq i_p$, the corresponding p-ary q-coary projection function. For example, if $p = 4$, $q = 2$ and $i_1 = 2$, $i_2 - 4$ then the projection of (0,1,2,3) is (1,3).

4. The basic σ-functions, and the characteristic functions of basic σ-predicates. (Individual constants are functions of arity 0 and coarity 1.)

The composition $g(h_1(x), \ldots, h_k(x))$ of functions g and $h_1, \ldots, h_k$ is defined in the obvious way. It is required that $\text{arity}(h_1) = \ldots = \text{arity}(h_k)$ and $\text{arity}(g) = \text{coarity}(h_1) + \ldots + \text{coarity}(h_k)$.

As usual, the *primitive recursion* schema is the schema

$$f(x, \text{Zero}) = g(x), \quad f(x,t+1) = h(x,t, f(x, t)) \qquad (1.2)$$

which defines a new function f by means of given functions g and h of the same coarity. Here Zero is the tuple of zeros of the appropriate length.

Definition 1.11 A global function is *primitive recursive* if it belongs to the closure of initial global functions under compositions and primitive recursions. A global relation is *primitive recursive* if its characteristic function is so.

Example Let us check that if a 2-coary function $f(x)$ and a 3-coary function $g(x)$ are primitive recursive then the 5-coary function $h(x) = (f(x), g(x))$ is primitive recursive. The 5-ary 5-coary identity function $I(y) = y$ is primitive recursive because it is an initial projection function. But $h(x) = I(f(x), g(x))$.

Theorem 1.9 A global function f is primitive recursive if and only if it is log-space computable.

We skip the proof of Theorem 1.9, see (28).

The language of primitive recursive functions can be viewed as a programming language such that exactly log-space computable global functions can be programmed. Programming languages of that sort may be useful in applications where the complexity of computations is bounded *a priori.* In this connection, let us mention that the primitive recursion schema can be replaced by more familiar programming constructs. Consider, for example, the construct

$$y := e_0;\ \text{FOR } s := e_1\ \text{TO } e_2\ \text{DO } y := e_3 \qquad (1.3)$$

where e_0, e_1, e_2, e_3 are expressions (terms) and e_0, e_1, e_2 contain neither s nor y. If the expressions e_i define primitive recursive global functions then (1.3) defines a primitive recursive global function $y = f(\ldots)$. The primitive recursion schema (1.2) is expressible by means of (1.3):

$$y := g(x);\ \text{FOR } s := \text{Zero TO } t - 1\ \text{DO } y := h(x, s, y)].$$

Another possible replacement for (1.2) is the construct

$$y := e_0;\ \text{WHILE } e_1 R e_2\ \text{DO } y := e_3 \quad (1.4)$$

where e_0, e_1, e_2, e_3 are expressions, and e_0, e_1, e_2 do not contain y, and R is a relation $=$, $<$ or $\leq$. (It would be desirable of course to introduce boolean expressions and to allow an arbitrary boolean expression b instead of $e_1 R e_2$.) If the expressions e_i define primitive recursive global functions then (1.4) defines a primitive recursive global function $y = f(\ldots)$. (1.2) is expressible by means of (1.4) and a projection:

$$(s, y) := (\text{Zero}, g(x));\ \text{WHILE } s < t\ \text{DO } (s, y) := (s + 1, h(x,s,y)).$$

Consider now the classical Herbrand-Gödel-Kleene equation language of recursive functions (47) extended by the individual constant End. The recursive definitions are naturally adaptable to global functions; it turns out that a global function is recursive iff it is polynomial time computable. Moreover, recursive functions form the closure of primitive recursive functions under a single additional recursion schema. Two schemas are specified for this purpose in (28). Here is a simpler recursion schema for the same purpose:

$$f(x, \text{Zero}) = gx,\ \ f(x,t+1) = h(x,f(\alpha x,t), f(\beta x,t)) \qquad (1.5)$$

which defines a new function f by means of given functions g, h, α and β.

Theorem 1.10 A global function is polynomial time computable if and only if it belongs to the closure of initial primitive recursive global functions by means of composition and recursion schemas (1.2), (1.5).

Proof The "if" implication is clear. To prove the "only if" implication, let RECFUN be the closure of initial primitive recursive global functions by means of composition and recursion schemas (1.2), (1.5), and let RECREL be the class of global relations with characteristic functions in RECFUN. The localization principle allows us to speak about the graph of a global function. It suffices to prove that an arbitrary polynomial time recognizable global relation ρ belongs to RECREL because a polynomial time computable global function can be recovered from its graph by primitive recursive means. According to the Appendix, there is an alternating two-way multihead automaton that recognizes p; it accepts a structure S with a tuple w of an appropriate length if and only if $\rho(w)$ holds in S. Let tuples Initial and Final be as in the Appendix.

Without loss of generality, every configuration of the automaton has at most two next configurations. There are primitive recursive functions α and β such that if y codes a configuration then $\alpha(w, y)$ and $\beta(w, y)$ code the next configurations; if there is only one next configuration then $\alpha(x, y) = \beta(w, y)$. Without loss of generality, every internal state of A is either existential or universal; the deterministic states (with only one next configuration) can be counted either way. We say that a configuration is existential (respectively universal) if the corresponding internal state is so. There is a primitive recursive function E such that if y codes an existential (respectively universal) configuration then Ey equals 0 (respectively 1). Schema (1.5) allows us to define an auxiliary function Accept(w, y, t):

Accept(w, y, Zero) = If y = Final then 1 else 0,

Accept(w, y, t+l) = If $Ey = 0$ then max{Accept($\alpha(w, y)$, t),
Accept($\beta(w, y)$, t} else min{Accept($\alpha(w, y)$, t),
Accept($\beta(w, y)$, t}.

Notice that $\rho(w) \leftrightarrow \exists\, t$ [Accept(w,Initial, t) = 1].

Here t is a tuple $(t_1, \ldots, t_r)$ of a fixed length r, and $\exists t$ means $\exists\, t_1 \ldots \exists\, t_r$. But RECREL is closed under the existential quantification over the elements. Hence ρ is in RECREL. ■

SECTION 6. INDUCTIVE FIXED POINTS

The LFP formation rule of Section 3 had one *ad hoc* feature. To ensure that the operator $F(P) = \{x: \varphi(P, x)\}$ is monotone, the formula $\varphi(P, x)$ was supposed to be positive in P. The positivity of φ is sufficient but not necessary for the monotonicity of F. Unfortunately, replacing the positivity condition by the monotonicity condition results in an extension FO + LFP' of first-order logic that we would not like to call a logic: the set of FO + LFP' formulas is undecidable (29). Fortunately, there is a better fixed-point extension of first-order logic,

called the inductive fixed-point extension FO + IFP, which is even more liberal than FO + LFP'. It was introduced in (29) as a development of an idea of Livchak (52). This section recapitulates the papers (37) and (38) where the inductive fixed-point logic was studied.

Definition 1.12 Let F be a unary operation on a (finite) complete partially ordered set D. Let $g0 = \min(D)$ and each $g(i + 1) = F(gi)$. F is *inductive* if $gi \leq g(i+1)$ for every i. It is easy to see that if F is inductive then it has a unique fixed point of the form gi; this fixed point will be called the *inductive fixed point* IFP(F) of F. F is *inflationary* if $X \leq F(X)$ for every $X \varepsilon D$.

Lemma 1.4 Let F be a unary operation on a complete partially ordered set.

(a) If F is inflationary then it is inductive.

(b) The operation $F'(X) = \sup\{X, F(X)\}$ is inflationary; if F is inductive then IFP(F') = IFP(F).

(c) If F is monotone then it is inductive and LFP(F) = IFP(F).

Proof is clear. ■

Examples Consider the power set of U = {0,1,2} ordered by inclusion.

1. Define $FX = X \cup \{\text{the cardinality of } X\}$ if $X \neq$ U, and FU = U. Then F is inflationary but not monotone. Moreover, F does not have a least fixed point: both {1} and {0, 2} are fixed points of f but $F\emptyset \neq \emptyset$.

2. Define $G = F$ except $G\{1\} = \emptyset$. Then G is inductive but neither inflationary nor monotone.

3. The constant operations $HX = \{0\}$ is monotone but not inflationary.

The syntax of logic FO + IFP is the extension of the syntax of first-order logic by:

The Inductive Fixed Point Formation Rule. Let r be a positive integer, x be an r-tuple $x_1, \ldots, x_r$ of individual variables, P be an r-ary predicate variable, $\varphi(P, x)$ be a well-formed formula, and $\varphi'(P,x) = [P(x) \text{ or } \varphi(P,x)]$. Then $\text{IFP}_{P;\mathbf{x}}\varphi'(P,x)$ is a well-formed predicate and $[\text{IFP}_{P;\mathbf{x}}\varphi'(P,x)](x)$ is a well-formed formula.

The meaning of the predicate $\text{IFP}_{P;\mathbf{x}}\varphi'(P,x)$ is the inductive fixed point of the inflationary operator $F(P) = \{x: \varphi'(P,x)\}$. The global function semantics for first-order logic naturally extends to FO + IFP.

The statement (c) of Lemma 1.4 implies that FO + IFP is at least as expressive as logic FO + LFP' mentioned above.

Theorem 1.11 The logics FO + LFP and FO + IFP have the same expressive power.

Corollary A global relation is expressible in FO + IFP + < if and only if it is polynomial time recognizable.

Proof Use Theorem 1.4 of Section 3. ■

Theorem 1.11 is a consequence of a stronger theorem. Let r be an arbitrary positive integer, and let Γ range over monotone global functions of the empty signature and of type

$$\text{Power-set}(\text{Universe}^r) \times \text{Power-set}(\text{Universe}^r) \times \text{Universe}^r \rightarrow \text{Bool}.$$

The monotonicity of Γ means that (on every finite structure) $\Gamma(P_1,P_2, x)$ implies $\Gamma(P_3,P_4, x)$ if $P_1 \subseteq P_3$ and $P_2 \subseteq P_4$. We are interested in the inflationary operator $G(P) = \{x: P(x) \text{ or } \Gamma(P, \text{not } P, x)\}$.

Define an extension FO + Γ of first-order logic by means of the following formation rule: if x is an r-tuple of individual variables and $\varphi(x)$, $\psi(x)$ are well-formed formulas then so is $\Gamma(\{x: \varphi(x)\}, \{x: \psi(x)\}, x)$. The global function semantics for the extended logic is clear. Treat Γ as a positive operator: every positive (respectively negative) occurrence of a predicate symbol in $\varphi(x)$or $\psi(x)$ remains so in $\Gamma(\{x: \varphi(x)\}, \{x: \psi(x)\}, x)$. If an FO + Γ formula $\psi(Q, y)$ is positive in a predicate symbol Q then the operator $\hat{\psi}(Q) = \{y: \psi(Q, y)\}$ is monotone; if

$\hat{\psi}$ is repetitive (i.e. , if the length of the sequence y of individual variables equals the arity of Q) then it has a least fixed point.

We say that a relation A is a *diagonal* of a relation B if A is obtained from B by identifying some arguments. For example, if B is given by some formula $\beta(v_1, v_2, v_3, v_4)$ and A is given by the formula $\alpha(v_1, v_2) = \beta(v_1, v_2, v_1, v_2)$ then A is a diagonal of B.

Theorem 1.12 There is an FO + Γ formula $\psi(Q, y)$ such that ψ is positive in Q, the operator $\hat{\psi}(Q) = \{y: \psi(Q, y)\}$ is repetitive, and the inductive fixed point of the operator $G(P) = \{x: P(x) \text{ or } \Gamma(P, \text{not } P, x)\}$ is a diagonal of the least fixed point of $\hat{\psi}$.

To deduce Theorem 1.11 from Theorem 1.12, prove by induction on FO + IFP formula φ that φ is equivalent to (i.e., defines the same global relation as) some FO + LFP formula. The only nontrivial case is when $\varphi = [\text{IFP}_{P, x}(P(x) \text{ or } \Phi(P(x))](x)$. Let $\Gamma(P,P', x)$ be the result of replacing all negative occurrences of P in Φ by a new predicate symbol P'. Then Γ is monotone in both relational variables, and $\Phi(P(x)$ is equivalent to $\Gamma(P, \text{not } P, x)$. Now use Theorem 1.12.

Theorem 1.13 For every FO + IFP definable global relation ρ there is a first-order formula $\varphi(P, x)$ such that the operator $\tilde{\varphi}(P) = \{x: P(x) \text{ or } \varphi(P, x)\}$ is repetitive and

(a) if the arity of ρ is positive then ρ is a diagonal of IFP($\tilde{\varphi}$),

(b) if ρ is 0-ary then the unary global relation ρ' such that $\forall v(\rho'(v) \leftrightarrow \rho)$ is a diagonal of IFP($\tilde{\varphi}$).

Theorems 1.12 and 1.13 imply the analog of Theorem 1.13 for FO + LFP announced in (44).

Jiazhen Cai, a student of Bob Paige in New York University, questioned the proof of Theorem 1.13 (more exactly, the proof of Lemma 2 in §4 of (38). The following claim removes the difficulty in the proof and is interesting all by itself.

Claim Let $\varphi(P, x, y) = [P(x) \text{ or } \varphi_0(P, x, y)]$ be an FO + IFP formula where x and y are tuples of individual variables such that the length of x equals the arity of P. Suppose that y-variables do not have bound occurrences in φ. Let Q be a new predicate variable whose arity allows to form a formula $Q(x,y)$, and let $\psi(Q, x, y)$ is the result of replacing each $P(u)$ by $Q(u,y)$ in $\varphi(P, x, y)$. Then

$$[\mathrm{IFP}_{P;x}\, \varphi(P, x, y)](x) \leftrightarrow [\mathrm{IFP}_{Q;\, x,y}\, \psi(Q, x, y)](x,y)$$

Proof For each y, let

$$P_0(y) = \emptyset,\ P_1(y) = \{x: \varphi(P_0, x, y)\},\ P_2(y) = \{x: \varphi(P_1, x, y)\}, \ldots$$

be the approximations to $\mathrm{IFP}_{P;x}\, \varphi(P, x, y)$, and let

$$Q_0 = \emptyset,\ Q_1 = \{(x,y): \varphi(Q_0, x, y)\},\ Q_2 = \{(x,y): \varphi(Q_1, x, y)\}, \ldots$$

be the approximations to $\mathrm{IFP}_{Q;\, x}\, \varphi(Q, x, y)$. (Here (x,y) is the concatenation of tuples rather than a pair of tuples.) It suffices to check that each $P_i(y) = \{x: Q_i(x, y)\}$. The case $i = 0$ is trivial. Further,

x belongs to $P_{i+1}(y) \leftrightarrow \varphi(P_i(y), x, y) \leftrightarrow$ (by the induction hypothesis) $\varphi(\{x: Q_i(x,y)\}, x,y) \leftrightarrow \psi(Q_i, x, y) \leftrightarrow (x,y)$ belongs to Q_{i+1}. ■

To formulate a similar claim for FO + LFP, replace the assumption that $\varphi(P, x, y) = [P(x) \text{ or } \varphi_0(P, x, y)]$ by the assumption that $\varphi(P, x, y)$ is positive in P.

Remark Theorem 1.13 can be strengthened further: φ can be taken to be a boolean combination of existential first-order formulas (11).

Remark The well-known zero-one law for first-order logic extends to inductive fixed-point logic (10).

SECTION 7. INVARIANT GLOBAL RELATIONS

We saw above that in the case of structures with built-in linear order there are nice logics which capture polynomial time. In this section we discuss the problem of capturing polynomial time in the general case. The problem was posed (in slightly different terms) in (12b) and discussed in (29).

Definition 1.13 An r-ary global relation ρ of some signature σ is *abstract* if for every isomorphism f from a σ-structure S onto a σ-structure T and all elements $x_1, \ldots, x_r$ of S,

$$\rho^S(x_1, \ldots, x_r) \leftrightarrow \rho^T(fx_1, \ldots, fx_r).$$

A logic capturing polynomial time is supposed to express exactly polynomial time computable abstract global relations.[1]

Remark Some polynomial time complete abstract properties are expressible in FO + LFP (45). But the abstract property that the universe is of even cardinality is not expressible in FO + LFP (12b). In virtue of Theorem 1.11 in Section 6, that abstract property is not expressible in FO + IFP.

We do not believe that there is a reasonable logic that captures polynomial time. To express our feeling in the form of a formal conjecture, we adapt the notion of logical systems (18) to our purpose.

Definition 1.14 A *logic* L is a pair (SEN, SAT) satisfying the following requirements. SEN is a function that associates with every finite signature σ a recursive set SEN(σ) whose elements are called *L-sentences* of signature σ. SAT is a function that associates with every finite signature σ a recursive subset SAT(σ) of $\{(S, \varphi)$: S is a finite first-order σ-structure and φ is an L-sentence of signature $\sigma\}$ such that if structures S and S' are isomorphic and (S, φ) belongs to SAT(σ) then (S', φ) belongs to SAT(σ) as well. If (S, φ) belongs to some SAT(σ), we say that *S satisfies* φ.

Definition 1.15 If L is a logic and φ is an L-sentence of some signature σ, then MOD(φ) be the set of σ-structures satisfying φ.

Definition 1.16 A logic L *captures* polynomial time if:

1. For every L-sentence φ, the class MOD(φ) is polynomial time recognizable; moreover, for every σ there is a Turing machine M that, given an L-sentence

---

[1] A logic capturing partial (not necessarily defined on all structures of the appropriate signature) recursive abstract global relations was designed in (12a).

φ of signature σ, produces a polynomial time bounded Turing machine $M(\varphi)$ that recognizes MOD(φ).

2. For every polynomial time recognizable class K of structures of some signature σ, if K is closed under isomorphisms then there is an L-sentence φ of signature σ such that MOD(φ) = K.

Remark In this section, a polynomial time bounded Turing machine can be viewed as a pair (T,p) where T is a Turing machine and p is a polynomial with integer coefficients; (T,p) accepts an input w of T if T accepts w within $p(|w|)$ steps.

Conjecture There is no logic that captures polynomial time.

Our adaptation of the notion of logics in (18) includes some alterations. In particular, we consider only finite signatures and finite structures, and require the recursivity of sets SEN(σ) and SAT(σ). Let $R1$ and $R2$ be the two recursivity requirements, respectively.

Claim 1 Waiving the recursivity requirements in Definition 1.14 falsifies the conjecture.

Proof Call a Turing machine M σ-appropriate if (a) it is able to take σ-structures as inputs, and (b) the class $\{S: S$ is a σ-structure and M accepts $S\}$ is closed under isomorphisms. Define L = (SEN, SAT) where each SEN(σ) consists of all σ-appropriate Turing machines, and each SAT(σ) consists of all pairs (S, M) such that S is a σ-structure and M is a σ-appropriate Turing machine that accepts S. It is easy to see that L is a logic in the liberalized sense and L captures polynomial time. ■

We could omit the second recursivity requirement $R2$ in Definition 1.14 because it follows from the condition of capturing polynomial time, but we consider it necessary in general and it complements $R1$ in the following sense (7). Suppose that L = (SEN, SAT) satisfies the requirements of Definition 1.14 except for $R1$ and $R2$, and suppose that the sets SEN(σ) are countable. Fix one-to-one mappings f_σ from SEN(σ) onto the set of natural numbers and rename every σ-sentence φ as the number $f_\sigma(\varphi)$. The resulting system is similar to L and satisfies $R1$.

The definition of logics may be justifiably tightened in many ways. One may require that every embedding $f: \sigma \rightarrow \sigma'$, taking any predicate (respectively function) symbol to a predicate (respectively function) symbol of the same or greater arity, gives rise to a recursive embedding of SEN(σ) to SEN(σ'); that the functions SEN and SAT themselves are recursive (when signatures are presented by

codes if necessary); that the signature of a sentence is computable from the sentence; that all recursivity conditions are replaced by corresponding polynomial time conditions; etc. Similarly, the notion of capturing polynomial time can be justifiably tightened in many ways. For example, one may require that the existence of a polynomial time bounded Turing machines M that, given (the code of) an arbitrary sentence φ, produces a Turing machine recognizing φ. We have chosen our definitions taking into account the negative character of the conjecture. Notice however the necessity of the requirement of the existence of machines M in clause (a) of Definition 1.16.

Claim 2 Waiving the requirement of the existence of machines M in Definition 1.16 falsifies the conjecture.

Proof (7) Let SEN(σ) comprise polynomial time bounded Turing machines able to take σ-structures as inputs. Call such a machine M symmetric with respect to n if for every pair ($S1$, $S2$) of isomorphic σ-structures of cardinality at most n, M accepts $S1$ if and only if it accepts $S2$. Given a σ-structure S of cardinality n and a machine M in SEN(σ), put (S, M) into SAT(σ) if M is symmetric with respect to n and M accepts S. Notice that each MOD(M) is closed under isomorphisms. The pair (SEN, SAT) is a logic capturing polynomial time in the liberalized sense. ■

Remark The conjecture is closely related to an open question of Chandra and Harel [(12b) Section 5]. They ask (in somewhat different words) whether there is a recursive set T of polynomial time bounded Turing machines such that for every σ and every polynomial time recognizable class K of σ-structures, K is closed under isomorphisms if and only if it is the collection of structures accepted by some machine in T. Also, see the paper (5) of Arvind and Biswas in connection with the conjecture.

The conjecture can be slightly simplified by restricting the attention to graphs.

Definition 1.17 A *graph logic* L is a pair (SEN, SAT) satisfying the following requirements. SEN is a recursive set whose elements are called *L-sentences*. SAT is a recursive subset of $\{(S, \varphi)$: S is a finite graph and φ is an L-sentence$\}$ such that if graphs S and S' are isomorphic and (S, φ) belongs to SAT then (S', φ) belongs to SAT as well. If (S, φ) belongs to some SAT, we say that S *satisfies* φ.

Definition 1.18 If L is a graph logic and φ is an L-sentence, then MOD(φ) is the set of σ-structures satisfying φ.

Definition 1.19 A graph logic *L captures* polynomial time if:

1. For every *L*-sentence φ, the class MOD(φ) is polynomial time recognizable; moreover, there is a Turing machine *M* that, given an *L*-sentence φ, produces a polynomial time bounded Turing machine M(φ) that recognizes MOD(φ).

2. For every polynomial time recognizable class *K* of graphs, if *K* is closed under isomorphisms then there is an *L*-sentence φ with MOD(φ) = *K*.

Theorem 1.14 The following statements are equivalent.

1. There is a logic that captures polynomial time.

2. There is a graph logic that captures polynomial time.

Proof The implication (1) → (2) is obvious. To prove the other implication, we use the well-known fact that an arbitrary structure *S* can be efficiently represented by a graph *G*(*S*) in such a way that two structures *S*1 and *S*2 of the same signature are isomorphic if and only if the graphs G(*S*1) and *G*(*S*2) are isomorphic. Moreover, there is a polynomial time Turing machine that, given the standard encoding of an arbitrary structure *S*, produces the desired graph *G*(*S*). If a graph logic *L* = (SEN, SAT) captures polynomial time, define *L'* = (SEN', SAT') where for each σ, SEN'(σ) - SEN and SAT'(σ) = {(S, φ): *S* is a σ-structure and *G*(*S*) satisfies φ}. Obviously, *L'* captures polynomial time. ■

Let us notice that both the special case, when the presence of linear order is assumed, and the general case, when the presence of linear order is not assumed, are important. As an input for a computing device, a structure should be represented in some way. A representation itself can be viewed as a structure, and in that richer structure a certain ordering of elements is usually definable. On the other hand, one is often interested in properties of structures that are independent of representation; let us call such properties invariant. To simplify somewhat the situation, let us view ordered versions of a given structure *S* as representations of *S*.

One way to ensure the invariance of a property of structures is to express the property in a logic that does not distinguish between different representations. For example, FO + LFP sentences express only invariant properties. There is another approach which is *a priori* more promising: allow linear order and concentrate on those properties that do not depend on order. In the rest of this section, interpretations of the binary predicate symbol $<$ are restricted to linear orders. If the signature of a structure *S* contains $<$ then *S* will be called *ordered*, otherwise it will be called *unordered.* If σ is a signature without $<$, *S* is a structure of signature $\sigma \cup \{<\}$ and S_0 is the reduct of *S* to σ, we will say that S_0 is the *unordered version* of *S*, and *S* is an *ordered version* of S_0, and any ordered version of S_0 is a *reordering* of *S*.

Definition 1.20 An r-ary global relation ρ of some signature $\sigma \cup \{<\}$ is *invariant on* a structure S of signature $\sigma \cup \{<\}$ if for every reordering T of S, $\rho^S(x) \leftrightarrow \rho^T(x)$. The global relation ρ is *invariant* if it is abstract and invariant on every structure of signature $\sigma \cup \{<\}$.

It is easy to see that ρ is invariant if and only if the boolean value of $\rho^S(x)$ depends only on the isomorphism type of $\langle S_0, x \rangle$ where S_0 is the unordered version of S.

The definition of invariant global relation ρ generalizes in a natural way to the case when ρ is K'-global where K is an arbitrary class of ordered structures of some signature $\sigma \cup \{<\}$ closed under isomorphisms and reorderings. Notice that an algorithm computing an invariant K'-global relation may use the given ordering.

Example Given a group with a linear order, the following algorithm computes the center of the group:

C : = Ø;

for x : = (the first element) to (the last element) do

begin

flag := l;

for y := (the first element) to (the last element) do

if $x \bullet y \neq y \bullet x$ then flag : = 0;

if flag = 1 then C : = $C \cup \{x\}$

end

Theorem 1.15 The decision problem whether a given first-order sentence with possible occurrences of $<$ yields an invariant global relation, is undecidable.

Proof Let α range over first-order sentences without occurrences of $<$. The validity of α on all finite structures is undecidable (64), hence the validity of α on all finite structures with at least two elements is undecidable. Let P be a unary predicate symbol that does not occur in α, and let β be a first-order sentence of signature $\{P,<\}$ asserting that $<$ is a linear order and that the first element in that order belongs to P whereas the last element does not. Then α is valid on all finite structures with at least two elements if and only if the disjunction (α or β) is invariant. ■

Remark Theorem 1.15 may be strengthened by means of different syntactic requirements on the given first-order formula: use numerous known strengthenings of Trakhtenbrot's theorem.

Theorem 1.16 There is a first-order sentence φ such that the decision problem whether φ is invariant on a given ordered structure, is coNP complete.

Proof We consider a restriction of the 3-colorability problem which is a known NP complete problem (25). Let H be the graph with vertices 0, 1, 2, 3 and the edges {0,1}, {1,2}, {2,3}, {3,0} and {0,2}; H is a cycle of length 4 plus one additional edge. H is 3-colorable, and every 3-coloring of H assigns the same color to vertices 1 and 3. Let Γ be the set of graphs that include H as a component. It is assumed—with respect to the proviso of Section 1—that the vertices of any member of Γ form an initial segment of natural numbers. It is easy to see that the restriction of 3-colorability problem to Γ is NP complete. For every graph G in Γ let G^* be the enrichment of G by means of the natural order of vertices.

The desired φ speaks about ordered graphs. It asserts that there are vertices $x < y$ such that the segments $\{v: v < x\}$, $\{v: x \leq v < y\}$ and $\{v: y < v\}$ constitute a 3-coloring. It suffices to prove that an arbitrary member G of Γ is 3-colorable if and only if φ is not invariant on G^*. If G is not 3-colorable then φ fails on any ordered version of G and, therefore, is invariant on G^*.

Suppose G is 3-colorable. Fix a 3-coloring of G. Let $G1$ be any ordered version of G where the vertices of color 1 form an initial segment and the vertices of color 3 form a final segment. Let $G2$ be an ordered version of G where vertex 1 is the first and vertex 3 is the last. It is easy to see that φ holds on $G1$ and fails on $G2$; hence G is not invariant on G^*. ■

SECTION 8. IS THERE A LOGIC FOR NP∩coNP or *R*?

We give some evidence that no logic captures NP∩coNP global relations or exactly R (random polynomial time recognizable) global relations. The argument is an elaboration of a remark in (28) and uses Sipser's result (59) that each of the two classes fails to have a complete problem (with respect to polynomial time reductions) under an appropriate oracle. The notion of logics was defined in the previous section. In connection with this section see a recent paper of Hartmanis and Immerman (40).

First we consider class NP∩coNP. Nondeterministic Turing machines M, N and a polynomial f will be said to *witness* that a class K of structures of some signature σ is NP∩coNP if for every n and every σ-structure S of cardinality n, (i) S belongs to K if and only if M accepts S within time $f(n)$, and (ii) S does not belong to K if and only *if* N accepts S within time $f(n)$.

Definition 1.21 A logic L *captures* NP∩coNP if:

1. For each L-sentence φ, the class MOD(φ) is NP∩coNP; moreover, for every signature a there is a Turing machine that, given an L-sentence φ of signature σ, produces a triple (M, N, f) witnessing that MOD(φ) is NP∩coNP; and

2. Every NP∩coNP class of structures of a fixed signature is definable by an L-sentence.

Theorem 1.17 If a logic L captures NP∩coNP then NP∩coNP has a complete problem with respect to polynomial time reducibility.

Proof Let σ be a signature comprising one unary predicate symbol. Fix a Turing machine A that generates all L-sentences of signature σ, and a Turing machine B that, given an L-sentence φ of signature σ, generates a triple witnessing that MOD(φ) is NP∩coNP. Let Q be the set of tuples (α, φ, β, M, N, f, S, $1^{f(n)}$) such that (i) α is a computation of A, φ is an L-sentence generated by α, β is the computation of B on φ, (M, N, f) is the output of β, S is a σ-structure, n is the cardinality of S, $1^{f(n)}$ is a string of 1's of length n, and (ii) S satisfies φ.

The condition (i) is polynomial time checkable. The condition (ii) is NP (respectively coNP): guess a computation of M (respectively N) on S of length $f(n)$ and verify that the computation is accepting. Thus the decision problem for Q is NP∩coNP. To show that this decision problem is NP∩coNP hard, we reduce to Q the decision problem for an arbitrary NP∩coNP class X of binary words. If w is a binary word $\alpha_1 \ldots \alpha_n$ let S_w be the σ-structure with universe $\{0,1, \ldots,n\}$ and relation $\{i: \alpha_i = 1\}$. (The universe contains $n + 1$ elements because it should be nonempty whereas n may be equal to 0.) Since L captures NP∩coNP, there is an L-sentence φ with MOD(φ) = $\{S_w: w \in X\}$. Let α be a computation of A that outputs φ, β be the computation of B on φ, and (M, N, f) be the output of β. Obviously, $w \in X$ iff $S_w \in$ MOD(φ) iff (α, φ, β, M, N, f, S_w, $1^{f(n+1)}$) belongs to Q. ∎

Theorem 1.17 contrasts with Sipser's result (59) that, relative to some oracle Δ, NP∩coNP does not possess a complete problem. (Certainly no logic captures NP∩coNP under the oracle Δ because the proof of Theorem 1.17 relativizes.) We conjecture that if some logic captures NP∩coNP then something drastic happens like NP∩coNP = P or NP = coNP. It may be desirable to restrict further the notion of a logic capturing NP∩coNP. For example, one may request that L-sentences are polynomial time recognizable.

Remark The converse of Theorem 1.17 is true to the extent that, given an NP∩coNP complete problem Q, one can construct a set of "sentences" and a satisfaction relation that capture NP∩coNP. Define sentences of signature a as triples (M, f, σ) where M is a deterministic Turing machine able to take σ-

structures as inputs, and f is a polynomial. Say that a σ-structure S of cardinality n satisfies $\varphi = (M, f, \sigma)$ if M halts on inputs S within time $f(n)$, and the result $M(S)$ belongs to Q.

Definition 1.21 and Theorem 1.17 generalize to some other classes with well defined witnesses. We turn now to random polynomial time. Recall that a set K of strings in an alphabet $\sum$ is R if and only if there are a deterministic Turing machine M and polynomials f, g such that for every n and every string $s \in \sum^*$ of length n the following are equivalent:

1. The string s belongs to K,

2. There is a string t in $\{0, 1\}^{g(n)}$ such that M accepts the pair (s,t) within time $f(n)$, and

3. For at least one half of strings t in $\{0, 1\}^{g(n)}$, M accepts the pair (s,t) within time $f(n)$.

We say that (M, f, g) *witnesses* that K belongs to R. Without loss of generality, we may suppose that $fn \geq gn$ for all n. The definition obviously generalizes to the case when K is a class of structures of a fixed signature.

Definition 1.22 A logic L *captures* R if:

1. For each L-sentence φ, MOD(φ) is R; moreover, for every signature σ there is a Turing machine that, given an L-sentence φ of signature σ, produces a triple (M, f, g) witnessing that $\{S: S \text{ satisfies } \varphi\}$ is R; and

2. Every R class of structures of a fixed signature is definable by an L-sentence.

Theorem 1.18 If a logic L captures R then R has a complete problem with respect to polynomial time reducibility.

Proof Similar to that of Theorem 1.17. ■

Theorem 1.18 contrasts with Sipser's result (59) that, relative to some oracle, there is no complete problem for R with respect to polynomial time reducibility.

SECTION 9. MISCELLANY

9.1 Sequences of Bounded-Depth Circuits

We suppose here that signatures comprise only predicate symbols, and boolean circuits have unique output gates. Recall that, according to the proviso of Section 1, the universes of structures are proper initial segments of natural numbers.

Definition 1.23 A boolean circuit C is *formatted* with respect to a signature σ and a positive integer n if input gates of C are labeled by sentences $Q(i_1, \ldots, i_r)$ where Q belongs to σ, r is the arity of Q, and every $i_p < n$. (Every input gate has exactly one label; hence, the number of input gates is bounded by the number of sentences $Q(i_1, \ldots, i_r)$.)

Definition 1.24 A circuit C, formatted with respect to σ and n, *accepts* a σ-structure S of cardinality n if C outputs 1 when the input gates of C are set with respect to S (an input gate labeled $Q(i_1, \ldots, i_r)$ gets value 1 if S satisfies $Q(i_1, \ldots, i_r)$, and value 0 otherwise).

Definition 1.25 A class K of σ-structures is *definable* by a sequence of circuits C_1, C_2, . . . if every C_n can be formatted with respect to σ and n in such a way that the formatted circuit accepts a σ-structure S of cardinality n if and only if S belongs to K.

Lemma 1.5 Let σ be a signature, φ be a first-order σ-sentence, and n be a positive natural number. There is a circuit C_n formatted with respect to σ and n in such a way that the depth of C_n is the logical depth of φ, and C_n accepts a σ-structure S of cardinality n if and only if S satisfies φ.

Proof Let σ_n be the extension of σ by individual constants 0, 1, . . . , n - 1. By induction, turn any sentence α, whose signature is included into σ_n, into a formatted circuit α_n. If α is atomic then α_n is the circuit comprising one gate labeled α. The cases of conjunction, disjunction and negation are obvious. If α is $\exists x\beta(x)$ (respectively $\forall x\beta(x)$) then join the circuits $\beta(0)_n$, $\beta(1)_n$, . . . , $\beta(n - 1)_n$ by an additional OR (respectively AND) gate. Finally, φ_n is the desired C_n. ■

The sequence of circuits, constructed in the previous paragraph, is very uniform. In particular, it is log-space constructible i.e. there is a log-space bounded Turing machine that, given the unary notation for n, produces (the standard code for) C_n.

Let L_0 be a logic that captures exactly log-space recognizable global relations of the empty vocabulary. L_0 can be the fragment of logic FO + DTC + < (see Section 3) whose formulas contain no individual constants, no function symbols and no predicate symbols except for <. L_0 can be the calculus of primitive recursive functions of the empty vocabulary (see Section 6); in this case the formulas are equations $t = 0$. Let FO + L_0 be the extension of first-order logic by L_0 whose formulas are built from first-order formulas and formulas in L_0 by first-order means (boolean connectives and quantifiers $\forall$,$\exists$); the global function semantics for FO + L_0 is obvious.

Theorem 1.19 (35) Let K be a class of structures of some signature σ. The following are equivalent:

1. K is definable by a log-space constructible sequence of circuits of bounded depth,

2. K is definable by a sentence in FO + L_0. ■

We skip the proof here. The theorem generalizes for many other complexity classes (35).

Logic FO + L_0 and many other extensions of first-order logic, considered above, were specially tailored to capture respective complexity classes. Logic FO + L_0 is the most modest of these extensions. An interesting question arises whether first-order logic itself captures any complexity class. Well, the answer to this question depends on the definition of complexity classes. The problem of the definition of complexity classes is a deep one, and we are not going to tackle it here. Let us only mention that Denenberg, Gurevich, and Shelah (17) have characterized first-order definable sequences of bounded-depth circuits by means of symmetry and uniformity conditions.

9.2 A Note on Topology on Finite Sets

There is a definite analogy between (i) classes of unary global relations definable by sequences of circuits of bounded depth and polynomially bounded size, and (ii) Borel subsets of the Cantor discontinuum. This analogy was exploited by Sipser in (60). Reading Ajtai's paper (3), we found it useful to think in terms of Borel subsets of finite topological spaces. The definition of Borel subsets of finite topological spaces is given in this subsection.

Recall that a topology is T_1 (50) if all one-point subsets are closed; we are not interested here in topologies that are not T_1. Every finite T_1 topological space is discrete, i.e., every subset is both closed and open. Thus, the theory of finite T_1 topological spaces seems to be quite trivial. However, one may ask how many intersections and unions does it take to express a given point-set in terms of sub-basic open sets. This leads to a generalization of the Borel hierarchy to finite topological spaces.

Definition 1.26 X_n is the topological space whose points are subsets of {0,1, ..., n - 1} and whose sub-basis comprises the n point-sets $\{P: i \in P\}$.

The analogous definition with the set ω of natural numbers instead of {0,1, . . . , n - 1} results in a topological space X_ω homeomorphic to the Cantor Discontinuum [(50), §3, IX]. Borel subsets of X_ω form the closure of the sub-basis

under complements, countable intersections and countable unions. This suggests the following:

Definition 1.27 A subset of X_n is Borel of *level* 0 if it is sub-basic. It is Borel of *level d,* where $d > 0$, if it is the intersection of at most n Borel sets of levels less than d, or the union of at most n Borel sets of levels less than d, or the complement of a Borel set of a level less than d.

There is an obvious connection between Borel point-sets and boolean circuits with unique output gates. Suppose that C is a circuit with n input gates labeled by integers $0, \ldots, n - 1$. In the obvious way, the input of C represents a point in X_n. C is said to *accept* a point P if the corresponding output is 1. C is said to *recognize* the set $\{P: C \text{ accepts } P\}$.

Claim 1 Let A be a subset of X_n, and d be a natural number. The following are equivalent:

1. F is Borel of level d,

2. There is a circuit C with n input gates labeled by integers $0, \ldots, n - 1$ such that the depth of C is at most d, the fan-in of C-gates is bounded by n, and C recognizes A.

Proof is clear. ∎

Definition 1.28 A *global point-set* π assigns a subset of X_n to each X_n. If there is a natural number d such that the specification of π on each X_n is Borel of level d then π is *Borel* (of level *d).*

Claim 2 The following are equivalent:

1. tt is Borel,

2. There is a bounded-depth polynomially-bounded-size sequence of circuits C_n such that each C_n recognizes the specialization of tt on X_n.

Proof is clear. ∎

Let P be a unary predicate symbol. A first-order sentence $\varphi(P)$ in signature $\{P\}$ defines a Borel global point-set $\{P: \varphi(P)\}$ of level d where d is the logical depth of $\varphi(P)$. First-order definable point-set π are symmetric in the following sense: if P_1 and P_2 are subsets of X_n of the same cardinality then P_1 belongs to is the specialization of π on X_n if and only if P_2 does. Some Borel global point-sets are symmetric in that sense but not first-order definable (17, 22).

PART 2. DYNAMIC STRUCTURES WITH BOUNDED RESOURCES

One finds a great many formal languages in computer science: programming languages, query languages, etc. It is natural for a logician to ask what structures are suited to model those formal languages. For example, what are models for Pascal? Of course, it is not necessary to start with formal languages. One can ask what structures are appropriate to formalize machines, databases and other objects of interest in computer science. We adopt the unspoken assumption of mathematicians that in principle there are appropriate structures; the problem is to find them.

SECTION 10. DYNAMIC STRUCTURES WITH BOUNDED RESOURCES, AND TURING'S THESIS

This section develops the ideas presented first in technical report (31) and is sort of an extended abstract for (34). I am thankful to Kit Fine for his comments on the report, and to Andreas Blass for many useful discussions.

10.1 Abstract Machines with Bounded Resources

The popular and useful abstraction of unbounded resources may be inappropriate under certain circumstances. For example, one may be reluctant to idealize his/her computer as a machine with unbounded memory if the computer keeps running out of memory all the time.

In (31) and (33) we discussed a new kind of abstract machines, called dynamic structures or dynamic algebras, whose resources may be bounded. Sometimes it is easier to express one's arguments in a discussion. Please allow me the liberty of introducing an opponent (a skeptical graduate student).

Objection 1 There is already a very well worked out formalization of machines with bounded resources. I mean finite state machines. Your dynamic structures with bounded resources are finite state machines too, aren't they?

Answer Yes, dynamic structures with bounded resources are finite state machines. But their theory does not reduce to the classical theory of finite state machines because the number of states may be overwhelming. Dynamic structure with bounded resources may capture the behavior of real computers (like PDP-11 or Macintosh) or model real programming languages (like Pascal or Smalltalk).

It is not feasible to draw the diagram or to write down the transition table (or a regular expression) for such a finite state machine.

Objection 2 I do not understand how a machine with bounded resources can model Pascal. Consider a Pascal program for computing factorials. A machine with unbounded resources is needed to provide an operational meaning to the program. Every machine with bounded resources fails to compute the factorial of some sufficiently large integer; it cannot give an adequate operational meaning to the program.

Answer A good point. The meaning will be given by a family of finite machines (in the same way as the meaning of a first-order formula is given by a family of first-order structures). A family of finite Pascal machines will be briefly sketched below. Here is a simpler example of a family: bounded-tape versions of a given Turing machine *T* with special end-of-tape marks.

Objection 3 Let me ignore the end-of-tape marks (I guess that bounded-tape versions of *T* without end-of-tape marks form a legitimate family too). Then the computation of any bounded-tape version of *T* is an initial segment of the computation of *T*, and the cut-off point is irrelevant to the meaning of the program. I would prefer to consider the computation of *T* itself rather than a pretty arbitrary collection of initial segments of the computation.

Answer Yes, in many cases, a machine with unbounded resources gives a cleaner operational semantics. But not always. The end-of-tape marks were there for a reason. Machines with bounded resources may know their resources and utilize this knowledge. A program for bounded-tape machines may use the end-of-tape mark for different purposes; for example, the end-of-tape mark may be used for dividing the tape equally into a left and a right part and executing two different processes in a time-sharing fashion. More convincing examples of how abstract machines with bounded resources may use their knowledge of bounded resources come from real life. Think about operating systems. In particular, think about an operating system which may run on many computers and which starts with an inventory of the available resources. Of course, this program (the operating system) can be modeled by a machine with unbounded resources, but this is not necessarily the best way to provide an operational semantics to the program.

10.2 Dynamic Structures

The usual mathematical structures, and in particular first-order structures, are static. They do not change in time. Mathematicians tend to formalize dynamic

situations in a static way. Given a dynamic process in an *n*-dimensional space, a mathematician introduces an additional dimension for time and studies the resulting *(n* + l)-dimensional body which represents all states of the original *n*-dimensional process at once.

Typical objects of computer science—machines, databases—are dynamic. They evolve in time. Of course, the same trick of representing all states at once can be used. This is exactly what you do when you draw the diagram of a finite automaton. Often, the trick does not work well. We believe that the difficulties are related to the overwhelming complexity of computing processes (versus, say, abstracted physical processes). There is no simple system of equations describing the behavior of a large time-sharing computer system. On the other hand, computing processes often have a certain simplicity in them: they evolve in discrete time and the states as well as atomic transitions are relatively simple.

The dynamic structure approach attempts to utilize the simple features of computing processes. A dynamic structure is a static structure (the initial configuration of the dynamic structure) evolving in discrete time with respect to specified transition rules. To specify a dynamic structure one needs to specify a static structure and transition rules.

The notion of dynamic structure may strike as an old news. There are similar notions in the literature; let me mention only transition systems in Gordon Plotkin's report (55). We see the main novelty in the intended use of dynamic structures. The configuration of discourse is going to play a greater role than the set of all configurations; from that point of view logic of dynamic structures is similar to temporal logic. Configurations are full-fledged static structures which have usually several universes. Here are some relevant questions. Do the universes change? Can new universes appear? Can old universes disappear? Does the signature change? What is the form of transition rules? Some other important notions are: bounded resources, families of dynamic structures, the dynamic structure of discourse.

Only special classes of static structures are defined formally in mathematical logic: usual first-order structures, many-sorted first-order structures, standard second-order structures, nonstandard second-order structures, etc. The general notion of static structures remains informal. Similarly, we leave the general notions of dynamic structures informal and define formally only special classes of dynamic structures. The notion of a family of dynamic structures is left informal too. We require however that all members of a family have the same transition rules.

To have a nontrivial example of a dynamic structure, one may formalize a modest computing device (on some level of abstraction). In connection with (33), my student, Bob Blakley, has worked out a formalization of PDP-11/04, the smallest machine using DEC's well-known PDP-11 architecture. The resulting dynamic structure is an evolving many-sorted first-order structure with static universes, static signature and transition rules of a very simple form.

Universes of the formalized PDF-11704 are Registers = {*R*0, ..., *R*7}, Addresses $\subseteq \{0,1\}^{16}$, Words = $\{0,1\}^{16}$, Opcodes, etc. Elements of Registers represent the 8 registers of the computer. PDP-11/04 uses register *R*7 as the instruction pointer. Elements of Opcodes represent legal assembly-language instructions in the PDP-11/04 instruction set. The set of addresses varies from one implementation of PDP-11 to another. To reflect this fact, the extent of Addresses is not fixed in Blakley's formalization.

Many basic functions of the dynamic structure are static: arithmetical operations, the eight zero-ary functions (i.e. distinguished elements) *R*l-*R*7 of type Registers, a function Addrtranslate from Words to (Addresses $\cup$ {Error}), a function Getop from Words to Opcodes, etc. Addrtranslate converts words (elements of Words) to addresses (elements of Addresses). If a word *w* is an address then Addrtranslate(*w*) = *w*, otherwise Addrtranslate(*w*) = Error. The analog of Addrtranslate for some other computers may be more complicated; one reason is that words may be longer than addresses.

Some dynamic basic functions of the formalized PDP-11 are

Regcontents: Registers → Words,

Contents: Addresses → Words,

Currentop, a distinguished element of Opcodes.

The following transition rule is self-explanatory:

Currentop ← Getop(Contents(Addrtranslate(Regcontents(*R*7)))).

Remark The idea of bounded resources and the idea of dynamic character are independent. One can study infinite static structures, finite static structures, dynamic structures with unbounded resources, and dynamic structures with bounded resources.

10.3 Turing's Thesis and Finite Dynamic Structures

A strong form of Turing's thesis states that every computing device can be simulated by an appropriate Turing machine (24). The thesis loses some appeal if one restricts attention to computing devices with bounded resources because the resources of Turing machines are unbounded. This brings us to the question of an analog of Turing's thesis for the case of machines with bounded resources.

New thesis problem (first draft formulation). Define a modest class U (for 'universal') of abstract machines with bounded resources such that every computing device with bounded resources can be closely simulated by a U-machine

of comparable specification and information sizes, and every family of computing devices with bounded resources can be appropriately simulated by a family of U-machines.

We speak about a modest class, close simulations and comparable sizes in order to exclude some unsatisfactory solutions. Let us explain that.

The computing devices in question are supposed to be real devices satisfying some minimal assumptions. (Actually, it is a little more complicated. We should be talking about a computing device and a fixed level of abstraction, of omitting details.) A part of the desired solution is that U-machines are dynamic structures. This is not a complete solution. First, the notion of dynamic structures was left informal. Second, dynamic structures may be used as virtual machines (for example, to model higher level programming languages) and therefore may be more complex than needed for the thesis. *The desired class U should be well-defined and as simple as possible.*

It is reasonable to assume that a computing device with bounded resources can be step-by-step simulated by a finite state transducer. (A finite state transducer is a finite automaton with output; a simulation is step-by-step if every step of the simulated machine corresponds to one step of the simulating machine.) Even if we ignore the question of families of finite state transducers, the reduction to finite state transducers should be treated with caution: the transition table of the simulating transducer may be much bigger than the description of the simulated device. *The specification size of the simulating machine should be severely bounded in terms of the specification size of the simulated machine.*

It would also be unsatisfactory if the number of states of the simulating machine too greatly exceeds the number of states of the simulated machine. The logarithm of number of states can be called the information size (9). *Thus, the information size of the simulating machine should be severely bounded in terms of the information size of the simulated machine.*

Bounded-tape versions of Turing machines with end-of-tape marks constitute one possible solution for the new thesis problem (31). An argument similar to Turing's informal proof of his thesis (66) establishes that for every computing device *D* with bounded resources there is an appropriate bounded-tape Turing machine that simulates *D*. This solution is unsatisfactory even if we put aside the question of the specification and information sizes: the simulations may be too complex and indirect. (Imagine, for example, that *D* is Apple's Macintosh in any of its incarnations.)

Remark One may argue that Turing's thesis itself has the same drawback. We agree; Turing machines are clumsy simulators. But who said that Turing's thesis cannot be improved? An early improvement of Turing's thesis was worked out by Kolmogoroff and Uspenski (48). So-called random access machines (1) are very popular. Very interesting machines were introduced by Schoenhage (58).

Which simulations should be permissible? Step-by-step simulations are ideal. Are they too restrictive? Recall that, in the case of machines with unbounded resources, a simulation is called real-time if one step of the simulated machine corresponds to at most *c* steps of the simulating machine where *c* is a constant. One can adapt this definition to the case of machines with bounded resources by imposing a restriction on the constant *c* in terms of the specification and information sizes of the simulated machine. Then one can require that *only real-time simulations are permissible.*

In (34) we intend to discuss more interesting solutions for the new thesis problem: Kolmogoroff-Uspenski machines with bounded resources, Schoenhage machines with bounded resources, random access machines with bounded resources and the solution proposed in (33). Andreas Blass and the author continue to work on the problem (9); the joint work has greatly influenced this section.

SECTION 11. MODELS FOR PASCAL

This section can be read independently.

11.1 Preliminaries

Imagine that you read a Pascal program and come across an assignment *x*:= *x*. What a silly thing to write, you may think. The assignment is obviously superfluous. Or is it there for a reason? Maybe it appears in the definition of a function procedure *x* in order to trigger the side effects of procedure *x*? You check and find out that *x* is a variable of type INTEGER. If there were several processes, then the purpose of the assignment could be related to synchronization or claiming a shared variable. But no, this is standard Pascal with only one process. You can think up some other possible effects of the assignment in some other languages, but all that seems to be irrelevant to Pascal. You become convinced that the assignment can be deleted. But the deletion changes the program and its execution somewhat. Maybe the right reason for the assignment just did not pop up in your mind. You would like to be able to *prove* that the deletion does not change your program in any essential way. Your semantics of Pascal should facilitate easy proofs of such simple facts.

Semantics of programming languages is a very rich field (some of our sources appear in the list of references). Still it seems to us that no known formal semantics is sufficiently convenient to deal with real-life imperative languages. "Unfortunately, all of the formal approaches to semantic definition require a great deal of sophisticated effort and produce a result which is impossible to read without extensive study," writes Ellis Horowitz in his *Fundamentals of Programming Languages* (43).

We would like to model programming languages, and in particular Pascal, by means of dynamic structures with bounded resources. Recall that dynamic structures are generalizations of the many-sorted static structures used in mathematical logic. They evolve in discrete time; every configuration of a dynamic structure is a static structure. To specify a dynamic structure one should specify its initial configuration and transition rules. Recall also that dynamic structures with finite resources are finite state machines (rather than potentially infinite machines), and that semantics of a programming language is supposed to be given by a family of resource-bounded dynamic structures with the same language of initial configurations and the same finite set of transition rules.

The desired Pascal models are ideal Pascal machines that directly execute Pascal programs. Many questions arise immediately. In what form should Pascal programs be given? What is the language of initial configurations? Should subsequent configurations have the same language or should the language of configurations evolve? How much can be executed in one step? How simple should the transition rules be? How should one impose a bound on the memory? And so on, and so on. This section was written with the active participation of my graduate student Jim Morris. To answer the above questions, we tried to use rich experience of real-world Pascal implementations, the insight and achievements of present-day semantics of programming languages, and analogies in classical logic.

A family of Pascal models is sketched in Section 11.2. Answering our solicitation of problems, Albert Meyer sent us a number of simple claims about Pascal including the claim about the superfluousness of the assignment $x := x$ where x is an integer variable. In Section 11.3 we sketch proofs of three of Meyer's claims in our semantics. In the final Section 11.4, we discuss in particular the question of when two Pascal programs have the same meaning.

We barely touch upon the issue of bounded resources in this section. That issue and others (possible applications of semantics of programming languages go far beyond proving simple claims about existing programming languages) will be addressed in (36) and elsewhere.

Acknowledgements I am thankful to Jim Morris for help, to Albert Meyer for sending the problems, to Albert Meyer, David Gries, and my Michigan colleagues Andreas Blass, Bernie Galler, and others for useful discussions.

11.2 Models for Pascal

We shall outline a finite dynamic structure M(Prog) where Prog is a Pascal program. Some parts of M(Prog) depend on Prog and some do not. One can abstract the underlying machine M which is a Pascal interpreter of a sort. For the sake of brevity, we will stress here the machine-like (rather than algebraic) aspects of M(Prog).

The initial configuration of *M*(Prog) is a finite many-sorted static structure. Some of the universes do not depend on Prog: an interval of integer numbers, a set of real numbers, the boolean universe {true, false}, a set of identifiers, and so on. The interval of integer numbers is equipped with the usual linear order, (the restrictions of) the usual arithmetical operations, distinguished elements MAXINT and MININT. We treat relations as boolean-valued functions. The only basic function defined on identifiers is the boolean function of equality. (Thus, identifiers are seen as mere tokens. In practice the token are represented by strings of letters and digits starting with a letter. Several strings may represent the same token. It may be, for example, that two strings of length 16 or more represent the same token if they have the same initial segment of length 16.)

The program Prog is given in the form of a decorated[1] parse tree which constitutes a universe in the initial configuration. Each type declared in Prog constitutes a universe of *M*(Prog). Many universes and many basic functions of *M*(Prog) are static; they are part of the initial configuration and do not change during the evolution of *M*(Prog).

One semantical complication is related to the fact that Pascal allows the programmer to reuse identifiers and labels. The name of a procedure, a variable, etc. may not identify the corresponding declaration uniquely. In order to provide unique names to different procedures, types and variables, we adopt a common convention. If a program Prog declares a procedure *P*1 which declares a procedure *P*2 which declares a procedure *P*3 (so that the procedures *P*1, *P*2 and *P*3 are of levels 1, 2 and 3, respectively, in Prog) then we will call the procedures (and the corresponding blocks) *P*1, *P*1.*P*2, *P*1.*P*2.*P*3 or Prog.*P*1, Prog.*P*1.*P*2, Prog.*P*1.*P*2.*P*3, respectively. If a Pascal variable *x* is declared in a block *B* (so that *B* is the smallest block containing the declaration), it will be called *B.x*. The denotation of *B.x* will be called a *raw variable* because in general the block *B* may be called recursively, and *B.x* may have several incarnations which are variables in their own right.

The complication arising from the reuse of identifiers and labels is not serious. Whenever one comes across a node of the parse tree decorated with an identifier, it is always clear which declaration of the identifier is relevant. To reveal this information, we use a special static function *Decl* which allows us to compute the *Signification* of the identifier in question. In the case of a variable name, Signification indicates the relevant raw variable. Signification is a dynamic function which solves, in particular, the aliasing problem (63).

Remark One may prefer to create the types and to compute the necessary values of the Decl function during the evolution of the machine. For some languages that may be the only alternative, but Pascal programs explicitly declare

---

[1]The decoration reflects the so-called static semantics of the program.

new types and use static binding of variables, which allows us to have all types and the Decl function from the beginning. Applying the principle of separation of concerns (26), we would like to get some relatively easy syntactical things out of the way by incorporating them into the initial configuration.

Transition rules of *M* specify the evolution from a given state to the next one. They do not depend on the given program. Let us perform an imaginary experiment. Imagine that you (rather than a computer) execute a Pascal program. What information should you keep in mind (or on paper)? You should know where you are currently in the program and where you should return after executing a procedure. You should know which variables exist and what their values are. You may need to remember the results of different subcomputations, in particular the values of different expressions and subexpressions; etc.

To record the current position in the program, *M*(Prog) has a 0-ary dynamic function (i.e. a dynamic distinguished element) called the *active node*, or *control*, whose possible values are nodes of the parse tree. Usually, one transition takes the control to a child or the parent of the currently active node. The exceptions are related to goto statements and procedure calls.

To record procedure calls and where to return from them, *M*(Prog) has a stack that will be called the *procedure stack*. The restriction allows us to get away with a simpler procedure stack. Formally speaking, the procedure stack is function from an initial segment of natural numbers to nodes of the parse tree. When a procedure is called at some node *N* of the parse tree, *M*(PROG) "pushes" *N* onto the stack and the control is transferred to Signification(*N*). When the execution of the procedure is finished, *N* is "popped off" the stack and control returns to *N*.

To record values of Pascal variables, *M* uses a dynamic function *V-val*. The domain of *V*-val contains all raw variables. To record the values of all incarnations of *B.x*, *V*-val maintains a special *B.x*-stack, a function from an initial segment of natural numbers to an appropriate type augmented with an additional value 'uninitialized.' Whenever control enters the block *B*, the value 'uninitialized' is pushed onto the *B.x*-stack; and when the execution of *B* is finished, the top of the *B.x*-stack is popped off.

A dynamic function *N-val* records (on appropriate nodes of the parse tree) the results of different subcomputations, in particular the values of different expressions and subexpressions. We use "OK" to indicate the successful execution of a command. (Another *a priori* possible result of the execution of a command is "Error.") Since procedures may be called recursively, *N*-val assigns a stack of values to a node. Consider for example the evaluation of $(a + b) + f(c)$ in the body of a function procedure *f*. The value of $a + b$ is "hanged" on some node *N* if "$a + b$" is evaluated during the first call on *f*, then a new value of

$a + b$ may be "hanged" on the same node N on top of the previous values each time the recursive call on f is executed, and so on.

There is also a universe called *Space* whose elements are called *units*. In implementations, a unit may correspond to one byte, to two bytes, or even to a single bit. A static function *Size* tells how many units of Space are needed for this or that purpose, and a dynamic function *Available* tells how many units of Space are available.

That ends our incomplete sketch of *M*(Prog). We did not specify any transition rules, did not discuss the parameter mechanism, did not discuss how the input is provided, etc. The details—for Modula-2 rather than Pascal—will appear in (36).

Notice that we are talking about a whole class *PM* of Pascal models. What may distinguish one member of *PM* from another? The interval of integers, the set of identifiers, etc. All members have literally the same transition rules. (Transition rules are given syntactically; they are written once for all members of *PM*.) The meaning of a Pascal program Prog is given by dynamic structures *M*(Prog) where *M* varies over those members of *PM* which contain all identifiers used in Prog. This is similar to the situation in mathematical logic where the global meaning of a first-order formula is given by the local meanings of the formula on those structures whose signatures include that of the formula.

11.3 Three Simple Problems of Meyer

For expository purposes, we allowed ourselves slight modifications of the original problems. In this subsection, a program means a Pascal program.

Claim 1 (The case of the superfluous statement.)

Let Prog2 be the result of deleting an assignment $x := x$, where x is an INTEGER variable, in a program Progl. Then Prog2 is equivalent to Progl.

Claim 2 (The case of the superfluous variable declaration.)

Suppose that a program Progl contains a procedure Progl.*P* with declaration

```
PROCEDURE P;
VAR x: INTEGER;
BEGIN
. . .
END;
```

where the body does not mention x. Let Prog2 be the result of deleting the declaration of $P.x$ in Progl. Then Prog2 is equivalent to Progl.

Claim 3 (The case of the superfluous procedure declaration.)

Suppose that a Pascal program Progl has parameter-free procedures Progl.*P*, Prog 1.*Q* and *Progl.Q:P* such that the declarations of *P* and *Q.P* are identical and the free identifiers of *Q.P* are not captured within *Q*. Let Prog2 be the result of deleting the declaration of *Q.P* in Progl. Then Prog2 is equivalent to Progl.

Clarification 1 What does it mean to delete an assignment, a variable declaration or a procedure declaration? To simplify the exposition, we suppose that we are allowed to use the empty statement, the empty variable declaration and the empty procedure declaration. Then the deletions can be interpreted as replacements with appropriate empty objects. Notice that if the assignment was labeled then the new empty statement is labeled.

Clarification 2 What does it mean that two Pascal programs are equivalent? This is a complicated question; it will be addressed in the next subsection. In this subsection two programs will be called equivalent if, provided the necessary resources, they exhibit the same input-output behavior. In other words, programs Progl and Prog2 are equivalent if they have the same input domain *D*, and for every Pascal model *M* and every input *X* in *D* the following condition is satisfied. If *M* contains all identifiers that occur in Progl or Prog2 and if *M*(Progl), *M*(Prog2) do not run out of memory on *X* then either both structures *M*(Progl), *M*(Prog2) converge on *X* or both structures diverge on *X*, but in either case the structures produce identical outputs on *X*.

Claim 1 was already discussed informally. Now let us discuss informally Claims 2 and 3.

The case of the superfluous variable declaration. Even though *x* does not occur in the body of *P*, some procedure *Q* with a free integer variable *x* may be called during the execution of *P*. The free variable *x* of the procedure *Q* will be interpreted as *B.x* where *B* is the least block that contains the appropriate declaration of *Q* and declares an integer variable *x*. Obviously, *B* is different from the block of *P*, and *B.x* is different from *P.x*. So the deletion of the declaration of *P.x* won't matter.

The case of the superfluous procedure declaration. Deleting *Q.P* means that calls on *Q.P* in Progl will be interpreted as calls on *P* in Prog2. Since *P* and *Q.P* are parameterless procedures with identical declarations, the execution of *P* can differ from the execution of *Q.P* only if the binding declaration of some free identifier *I* of *P* differs from the binding declaration of the free identifier *I* of *Q.P*, which means that *Q* contains a declaration of *I*, which means that *I* is captured within *Q*, which is impossible.

Proof Sketch for Claim 1 Given a machine *M*, let *Mi* = *M*(Prog***i***) and *Ti* be the parse tree of Prog*i*. *T*2 is obtained from *T*1 by replacing the subtree *t*1 of an assignment *x*:= *x* with a single-node tree *t*2 corresponding to the empty statement. Let *T* be the common part of *T*1and *T*2.

Call a state of *Mi* black if its active node is in *T*, otherwise call it red. (The terms "black" and "red" are related to the expressions "to be in the black" and "to be in the red.") Say that a state *S*1 of *M*l and a state *S*2 of *M*2 agree if

1. the active nodes of *S*1 and *S*2 coincide (which means in particular that both states are black),

2. the procedure stacks and the *V*-vals coincide, and

3. the *N*-vals coincide on *T*.

Say that a sequence of states of *M*l and a sequence of states of *M*2 agree if erasing all red states in both sequences results in two sequences of the same (finite or infinite) length where the corresponding members agree. It suffices to prove that the computations (viewed as sequences of states) of *M*l and *M*2 on the same input agree.

In the initial states, the active nodes are the roots of the respective parse trees, and the procedure stacks and the functions V-val, N-val are empty; thus the initial states agree. Suppose that *Si* is a black state of *Mi*, and the states *S*1, *S*2 agree. Then

1. *S*1 is final (halting) if and only if *S*2 is so,

2. the successor of *S*1 is black if and only if the successor of *S*2 is so,

3. if the successors are black then they agree, and

4. if the successors are red then for neither *i* is *Si* the last black state of *Mi*.

It remains to prove that if the successors of *S*1, *S*2 are red then the first black state after *S*1 agrees with the first black state after *S*2. Let us see what happens when each *Mi* goes through the red states following *Si*. The active node *X* of *Si* is either the parent of root(*ti*) or the root of the subtree of a goto statement. In either case the control leaves *X* without changing the *N*-val at *X*. No procedure is called or exited when *Mi* goes through the red states, therefore the procedure stack does not change. The *V*-val does not change because the only relevant raw variable is *B.x*, where *B* is the block of *ti*, and the *B.x*-stack does not change. The restriction of the *N*-val to *T* does not change because, in the absence of procedure calls, the *N*-val changes only at the active node. Eventually the control finds its way to the parent of root(*ti*), and *Mi* arrives to some black state *Si*'. Obviously, *S*1' and *S*2' agree.

Remark One may wonder whether there is any difference between *Si* and *Si'*. The answer is yes. In *Si*, the top value of the *N*-val at root(*ti*) is "undefined"; whereas in *Si'*, the top value of the *N*-val at root(*ti*) is "OK."

Proof Sketch for Claim 2 The proof is similar to that of Claim 1. There are only two important differences. One is related to the definition of agreeing states. The requirement that the two *V*-vals coincide is replaced by the requirement that the two *V*-vals coincide on the domain of *V*-vals of *M*2. The modification is necessary because the domain of *V*-vals of *M*l contains an additional raw variable *P.x*.

The second difference of importance is related to the verification that the black successors of agreeing states agree. The transition may deal with an integer variable *x*, but—as we explained above in the informal discussion—Progl will never interpret that *x* as *P.x*. Moreover, the two programs will interpret *x* in the same way. What we need is a separate simple lemma that the two Decl functions coincide on the common part of the parse trees. •

Proof Sketch of Claim 3 Again, the proof is similar to that of Claim 1. Let *Mi* = *M*(Prog*i*) and *Ti* be the parse tree of Prog*i*. *T*2 is obtained from *T*1 by replacing the subtree *t*(*Q.P*) of the declaration of *Q.P* with a single-node tree corresponding to the empty procedure declaration. Let *t*(*P*) be the subtree of the declaration of *P*, and let *T* be the common part of *T*1 and *T*2.

This time we do not need red states: when the active node of *M*l traverses *t*(*Q.P*), the active node of *M*2 traverses *t*(*P*). One little complication in the proof is that the corresponding states of *M*l and *M*2 may have somewhat different *V*-vals (even though the *V*-vals have the same domain) and somewhat different restrictions of the *N*-vals to *T*. To overcome this difficulty, we introduce extended stacks. The extended stacks are imaginary, they are not parts of our Pascal models.

Let *B. x* be an arbitrary raw variable of an arbitrary *M*(Prog). The *V*-val of *M*(Prog) maintains a *B.x*-stack. The *extended B .x-stack* is the *B.x*-stack possibly "diluted" with copies of a new item 'empty'. Whenever a block different from *B* is called, a copy of 'empty' is pushed on the extended *B.x*-stack, and whenever any block is exited, the top of the extended *B.x*-stack is popped off. The *N*-val assigns stacks of values to nodes; the corresponding extended stacks are defined in a similar way.

Consider the extended stacks of two raw variables *B*1.*x* and *B*2.*x*, where the blocks *B*1 and *B*2 are different. It is impossible that for some *i*, the extended stacks have *i*-th items that are both different from 'empty'. This allows us to merge the two extended stacks into a new stack whose height is the minimum of the heights of the two given stacks. Suppose that *u* is the *i*-th item of the extended *B*1.*x*-stack and v is the *i*-th item of the extended *B*2.*x*-stack. What is

the *i*-th item *w* in the new stack? If *u* differs from 'empty' then *w* = *u*; if *v* differs from 'empty' then *w* = *v*; otherwise w is 'empty.'

Now we are ready to define agreeing states. A state *S*1 of *M*l agrees with a state *S*2 of *M*2 if the following conditions (a)-(d) are satisfied.

(a) Either the active node of *S*1 coincides with the active node of *S*2, or the active node of *S*1 is in *t*(*Q.P*) and the corresponding node of *t*(*P*) is active in *S*2.

(b) The procedure stacks of *S*1 and *S*2 coincide.

(c) For every raw variable *B.x*, declared within the block of *P*, the extended *B.x*-stack of *S*2 is the merger of the exttended *B.x*-stack and the extended *Q.B.x*-stack of *S*1. For every other raw variable *C.y*, the *C.y*-stacks of *S*1 and *S*2 coincide.

(d) For every node *N* in *T*-*t*(*P*) the *N*-stacks of *S*1 and *S*2 coincide. If *N* belongs to *t*(*P*) and *N'* is the corresponding node in *t*(*Q.P*) then the extended *N*-stack of *S*2 is the merger of the extended *N*-stack of *S*1 and the extended *N*-stack of *S*1.

It is easy to check now that for every input *I*, the computation of *M*l on *I* and the computation of *M*2 on *I* have the same length, and the corresponding members agree. It follows that Progl and Prog2 are equivalent. ■

11.4 Final Remarks

11.4.1 Equivalent Programs

Let us address the question

(*) When do two Pascal programs have the same meaning?

The equivalence relation of Section 11.3 is one answer to (*). Under certain circumstances it may be unsatisfactory. Imagine that Progl and Prog2 solve the same NP problem and are equivalent in the sense of Section 11.3, but Progl works in linear time whereas Prog2 works in exponential time. It does not seem right to consider them as having the same meaning. One may refine the equivalence relation of Section 11.3 to give different answers to (*). One may require, for example, that the computations of Progl and Prog2 simulate each other in real time, or that the histories of global variables are identical. (The proof of each of the three claims above establishes both stronger equivalences.)

We think that there are many reasonable answers to (*) and that an appropriate answer depends on the circumstances.

It is interesting to compare question (*) with the similar question for first-order formulas. The standard answer to the latter question is that two first-order

formulas have the same meaning if and only if they are logically equivalent (i.e., define the same global relation). This ignores the computational aspect of formulas, in particular the cost of computing corresponding relations.

11.4.2 Distinguishing Features of the Proposed Semantics

Ellis Horowitz writes [(43), Section 2.1]: *"Interpretative* semantics begins by defining an abstract machine. This machine supports a simple set of operations and data structures. Then the semantics of the language being defined is given by a set of rules which show how programs will be translated onto the abstract machine. The Vienna Definition Language (70), which was developed as a means for formally defining PL/1, is the prime example of this approach." Our semantics is interpretative (or operational) because of the use of abstract machines, but it is somewhat different.

1. We define a family of abstract machines with bounded resources rather than one abstract machine with unbounded resources. Each machine gives a local meaning to a program, and the family gives the global meaning. (From that point of view, the semantics can be called global or multi-model.) Multi-model semantics is especially appropriate to handle implementation defined constants. (If your Pascal model has all integers then what is the meaning of MAXINT?) As a matter of fact, the presence of implementation defined constants makes modeling easier.

2. We tailor our machines to the given language (rather than translate the given language to the fixed language of a unique abstract machine). In this section, we described Pascal machines. We considered also models for different variations and extensions of Pascal (in particular, to check that passing a simple variable by name has the same effect as passing it by reference). In (36), we describe Modula-2 machines. We intend to model languages for parallel and distributing computing, and different other languages.

3. Our machines are algebraic structures of a sort, namely dynamic structures. (We would call the proposed semantics algebraic if the term were not taken (27).) To explain what we mean by the algebraic character of our models, let us point out the difference between usual algebraic structures, say graphs, and their representations. One does not care about the nature of the vertices, may not have unique names for the vertices, does not distinguish between isomorphic graphs. Similarly, we do not care about the nature of the elements of our models, may not have unique names for the elements, do not distinguish between isomorphic models. (For example, for any member M of the class PM of Pascal models, any permutation of identifiers gives rise to an automorphism of M.)

11.4.3 Solicitation

The examples above suggest using the proposed semantics for proving general properties of programs and correctness of different transformations (like source-to-source transformations employed by optimizing compilers). Dealing with two models of different levels of abstraction, we use the approach for proving correctness of source-to-target transformations (36). A related theoretical problem is to work out a useful notion of homomorphism of dynamic structures.

We are soliciting interesting and challenging problems about Pascal or other (real or imaginary) imperative languages. We are especially interested in problems related to limited resources.

APPENDIX. TWO-WAY MULTIHEAD AUTOMATA

A two-way multihead automaton can be described as a multihead Turing machine without any work tape. It is well-known that, as recognizing devices, deterministic (respectively, nondeterministic) two-way multihead automata are equivalent to deterministic (respectively nondeterministic) log-space bounded Turing machines. Hartmanis and Hunt say in their 1974 paper (39) that this is a well-known fact and refer for a more complete proof to a 1972 paper of Hartmanis. It is also well-known that the equivalence survives if alternation is allowed. For reader's convenience, we prove here these facts.

Remark To accommodate naturally standard representations of structures (see §1), we allow Turing machines and two-way multihead finite automata to have several input tapes. One of these input tapes is the universe tape that contains the unary notation for the cardinality of the given structure. We will ignore structures of cardinality 1, and will suppose that the end-cells of the universe tape are specially marked.

Theorem 1.20 A global relation is recognizable in log-space by a deterministic (respectively nondeterministic, alternating) Turing machine if and only if it is recognizable by a deterministic (respectively nondeterministic, alternating) two-way multihead finite automaton.

Proof The "if" implication is easy (and will not be used): record the current positions of the automaton heads on a work tape.

To prove the "only if" implication, suppose that a log-space bounded Turing machine M recognizes the global relation in question. Let n be the length of the universe tape. Without loss of generality, we can assume the following about

M: it has only one work tape, on each step the work tape head either writes or moves but not both, the work tape alphabet is {0,1} where 0 is also the blank, the end cells of the work tape never hold zeros, initially the head of the work tape is in the leftmost position, and a configuration of *M* is accepting if and only if the corresponding internal state is one of the specially designated accepting states.

Let *u* be the content of the initial segment of the current work tape up to and including the position of the head, *v** be the content of the corresponding final segment, and *v* be the reverse of *v**. The strings *u* and *v* are binary notations for some numbers that uniquely define the content of work tape. The symbol observed by the work tape head is exactly the parity of *u* (0 if *u* is even and 1 otherwise). If the work tape head changes 0 to 1 (respectively 1 to 0) then $u := u + 1$ (respectively $u := u - 1$) and *v* does not change. If the work tape head moves to the right then $u := 2u + \delta$ and $v := (v - \delta)/2$ where δ is the parity of *v*. If the work tape head moves to the left then $u := (u - \delta)/2$ and $v := 2v + \delta$ where δ is the parity of *u*.

Since the length of the work tape is bounded by a multiple of log *n*, the numbers *u* and *v* are bounded by some n^k. Thus,

$$u = \sum_{i<k} \alpha_i n^i \quad \text{and} \quad v = \sum_{i<k} \beta_i n^i$$

where $\alpha_i, \beta_i < n$ for each *i*. The desired two-way multihead automaton *A* represents *u* and *v* by 2*k* heads on the universe tape. Using a few auxiliary heads, *A* is able to compute the parities of *u*, *v* and to perform the operations $u := u + 1$, $u := u - 1$, $u := 2u + \text{parity}(v)$, $v := [v - \text{parity}(v)]/2$, etc. mentioned above.

Some internal states of *A* code the internal states of *M*, in addition *A* has auxiliary internal states. When *A* is in the internal state *q'* coding an internal state *q* of *M*, the configuration of *A* codes a configuration of *M*; if *q* is existential (respectively universal) then so is *q'*. The auxiliary internal states of *A* are deterministic. If *M* starts in the initial configuration C_0 and goes through subsequent configurations C_1, C_2, etc. then *A* starts in the initial configuration coding C_0, goes through a series of configurations with auxiliary internal states and arrives to the configuration coding C_1, goes through a series of configurations with auxiliary internal states and arrives to the configuration coding C_2, etc. A configuration of *A* is accepting if and only if it codes an accepting configuration of *M*. It is easy to see that *A* accepts a given structure if and only if *M* accepts it. ■

Corollary A global relation is polynomial time recognizable if and only if it is recognizable by an alternating two-way multihead finite automaton.

Proof Polynomial time equals alternating log-space (13). ■

Let us consider more closely the computation of a two-way multihead automaton A that recognizes a global relation ρ. A can be deterministic, nondeterministic or alternating. Represent the position of a head h on a tape of length n^p by p-tuple $x_{h0}, \ldots, x_{h(p-1)}$ with the intended interpretation $\sum x_{hi} \bullet n^i$. Here n is the length of the universe tape and each x_{hi} is a natural number $<n$. Further, represent the j-th internal state of the automaton by a q-tuple $y_1, \ldots, y_q$ where q is the number of internal states, $y_j = 1$ and $y_i = 0$ for $i \neq j$. Thus there is an r such that every configuration of A is represented by an r-tuple of natural numbers $<n$. Without loss of generality, we may assume that A has a unique accepting configuration and that both in the initial and in the accepting configuration of A all heads are in the leftmost positions. Then the r-tuples Initial and Final, representing the initial and the final configurations respectively, consist of zeros and ones.

Claim There is an FO + < formula Next satisfying the following. Let S be a structure in the domain of ρ, w be a tuple of elements of S whose length equals the arity of ρ, and x, y be r-tuples of elements of the universe of S. Then Next(w,x, y) holds in S if and only if x, y represent configurations of A on inputs (S, w) and A is able to go from configuration x to configuration y in one step.

Proof The desired formula Next is a conjunction where each conjunct describes (in the obvious way) one instruction of A. (The variables w appear since there are reading heads on the corresponding input tapes.) •

Remark The formula Next is especially simple if one uses the successor function (rather than order) and individual constants 0 and End. If the universe is $\{0, \ldots, n - 1\}$ then End is interpreted as $n - 1$. To make the successor function total, define Successor(End) = 0 or Successor(End) = End.

In the rest of Appendix, a global function is a partial a-global function of type $\text{Universe}^p \rightarrow \text{Universe}^q$ for some σ, q, p; such global function assigns to each σ-structure S a p-ary q-coary operation on the universe of S. Two-way multihead automata were defined as Turing machines without working tapes. They may have output tapes however.

Theorem 1.21 A global function is computable by a deterministic log-space bounded Turing machine if and only if it is computable by a deterministic two-way multihead automaton.

Proof Essentially the same proof as that of Theorem 1.20. If the simulated Turing machine M writes on an output tape in a configuration x then the simulating automaton A does the same in the configuration that codes x. ■

REFERENCES

1. Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

2. Aho, A. V. and J. D. Ullman, "Universality of data retrieval languages," *6thPOPL Symp.,* ACM, 1979, 110-117.

3. Ajtai, M., "$\sum^1_1$-formulae on finite structures," *Annals of Pure and Applied Logic 24* (1983), 1-48.

4. Ajtai, M. and Y. Gurevich, "Monotone versus positive," *Journal of ACM,* to appear.

5. Arvind, V. and S. Biswas, "Expressibility of first-order logic with a nondeterministic inductive operator," Manuscript, Indian Institute of Technology, Kanpur, India, May 1986.

6. de Bakker, J. W., *Mathematical Theory of Program Correctness.* Prentice-Hall, 1980.

7. Blass, A., *Private communication.*

8. Blass, A. and Y. Gurevich, "Henkin quantifiers and complete problems," *Annals of Pure and Applied Logic,* 32 (1986), 1-16.

9. Blass, A. and Y. Gurevich, "A new thesis" (tentative title), in preparation.

10. Blass, A., Y. Gurevich and D. C. Kozen, "A zero-one law for logic with a fixed-point operator," *Information and Control 67* (1985), 70-90.

11. Börger, E. and Y. Gurevich, "On fixed-point extensions of first-order logic"

(tentative title), in preparation.

12a. Chandra A. K. and D. Harel, "Computable queries for relational data bases," *J.*

Comput. and System Sciences 21 (1980), 156-178.

12b. Chandra, A. K. and D. Harel, "Structure and complexity of relational queries,"

J. Comput. and System Sciences 25 (1982), 99-128.

13. Chandra, A. K., D. C. Kozen and L. J. Stockmeyer, "Alternation," *J. of Association for Computing Machinery 28* (1981), 114-133.

14. Church, A., "An unsolvable problem of elementary number theory," *American Journal of Mathematics 58* (1936), 345-363.

15. Codd, E. F., "Relational completeness of database sublanguages," in *Database Systems* (ed. R. Rustin), Prentice-Hall, 1972, 65-98.

16. Compton, K., "The computational complexity of asymptotic problems I: partial orders," *Information and Control,* to appear.

17. Denenberg, L., Y. Gurevich and S. Shelah, "Cardinalities defined by constant depth polynomial size circuits," *Information and Control 70* (1986), 216-240.

18. Ebbinghaus, H.-D., J. Flum and W. Thomas, *Mathematical Logic,* Springer-Verlag, New York, 1984.

19. Ehrenfeucht, A., "An application of games to the completeness problem for formalized theories," *Fund. Math. 49* (1961), 129-141.

20. Fagin, R., "Generalized first-order spectra and polynomial time recognizable sets," *SIAM-AMS Proc. 7* (1974), 43-73.

21. Fagin, R., "Monadic generalized spectra," *Zeitschrift fur Math. Logik und Grund-lagen der Mathematik 21* (1975), 89-96.

22. Fagin, R., M. Klawe, N. J. Pippenger, and L. Stockmeyer, "Bounded depth polynomial size circuits for symmetric functions," *Theoretical Computer Science,* April 1985, 239-250.

23. Gaifman, H. and M. Vardi, "A simple proof that connectivity of finite graphs is not first-order definable," *Bulletin of European Assoc. for Theoretical Computer Science,* June 1985, 43-45.

24. Gandy, R., "Church's thesis and principles for mechanisms," in: *The Kleene Symposium* (ed. J. Barwise et al.), North-Holland, 1980, 123-148.

25. Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP Completeness,* Freeman, 1979.

26. Gries, D., *The Science of Programming,* Springer-Verlag, 1981.

27. Guessarian, L, "Algebraic semantics," *Lecture Notes in Computer Science 99,* Springer-Verlag, 1981.

28. Gurevich, Y., "Algebras of feasible functions," *24th Symposium on Foundations of Computer Science,* IEEE Computer Society Press, 1983, 210-214.

29. Gurevich, Y., "Toward logic tailored for computational complexity," in *Computation and Proof Theory* (Ed. M. M. Richter et al.), Springer Lecture Notes in Math. 1104(1984), 175-216.

30. Gurevich, Y., "Monadic second-order theories," in *Model-Theoretical Logics* (ed. J. Barwise and S. Feferman), Springer-Verlag, 1985, 479-506.

31. Gurevich, Y., "Reconsidering Turing's thesis (toward more realistic semantics of programs)," *Technical Report CRL-TR-36-84,* University of Michigan, Sep. 1984.

32. Gurevich, Y., "Logic and the challenge of computer science," *Technical Report CRL-TR-10-85,* University of Michigan, Sep. 1985.

33. Gurevich, Y., "A new thesis," *AMS Abstracts,* Aug. 1985, p. 317.

34. Gurevich, Y., "Algorithms in the world of bounded resources," in *The Universal Turing Machine—A Half-Century Survey* (ed. R. Herken), Oxford University Press, to appear.

35. Gurevich, Y. and H. R. Lewis, "A logic for constant-depth circuits," *Information and Control 61* (1984), 65-74.

36. Gurevich, Y. and J. M. Morris, *"Pragmatic semantic for Modula-2,"* (tentative title), in preparation.

37. Gurevich, Y. and S. Shelah, "Fixed-point extensions of first-order logic," *Annals of Pure and Applied Logic 32* (1986), 265-280.

38. Gurevich, Y. and S. Shelah, "Fixed-point extensions of first-order logic," *26th Annual Symp. on Foundation of Computer Science,* IEEE Computer Society Press, 1985, 346-353.

39. Hartmanis, J. and H. B. Hunt, "The LBA problem and its importance in the theory of computing," *SIAM-AMS Proc.,* vol. 7 (1974), 1-26.

40. Hartmanis, J. and N. Immerman, "On complete problems for NP∩coNP," *Lecture Notes in Computer Science 194* (1985), Springer-Verlag, 250-259.

41. Hayes, J. P., "Uncertainty, energy, and multiple-valued logics," *IEEE Trans. on Computers,* vol. C-35 (1986), 107-114.

42. Henkin, L., "Some remarks on infinitely long formulas," in *Infinitistic Methods,* Warsaw, 1961, 167-183.

43. Horowitz, E., *Fundamentals of Programming Languages,* Computer Science Press, 1983.

44. Immerman, N., "Relational queries computable in polynomial time," *14th Symposium on Theory of Computing,* Association for Computing Machinery, 1982, 147-152.

45. Immerman, N., "Languages which capture complexity classes," *15th Symposium on Theory of Computing,* Association for Computing Machinery, 1983, 347-354.

46. Jensen, K. and N. Wirth, *Pascal, User Manual and Report,* Springer-Verlag, 2nd edition, 1978.

47. Kleene, S. C., *Introduction to Metamathematics,* D. Van Nostrand, New York, Toronto, 1952.

48. Kolmogoroff, A. N. and V. A. Uspenski, "On the definition of an algorithm," *UspekhiMat. Nauk 13* (1958), 3-28 (Russian), AMS Transl. 29 (1963), 217-245.

49. Krom, M. R., "The decision problem for a class of first-order formulas in which all disjunctions are binary," *Zeitschrift fur math. Logik und Grundlagen der Math-ematik 13* (1967), 15-20.

50. Kuratowski, K., *Topology, volume 1,* Academic Press, 1966.

51. Lindstrom, P., "On extensions of elementary logic," *Theoria 35* (1969), 1-11.

52. Livchak, A., "The relational model for process control," *Automatic Documentation and Mathematical Linguistics 4* (1983), 27-29.

53. Lyndon, R., "An interpolation theorem in the predicate calculus," *Pacific J. Math. 9* (1959), 155-164.

54. Moschovakis, Y. N., *Foundations of the Theory of Algorithms, I,* Manuscript, University of California, Los Angeles, 1986.

55. Plotkin, G. D., "Structural approach to operational semantics," *Technical report DAIMIFN-19,* Computer Science Department, Aarhus University, Denmark, Sept. 1981.

56. Reynolds, J. C., *The Craft of Programming,* Prentice-Hall, 1981.

57. Sazonov, V. Y., "Polynomial computability and recursivity in finite domains," *Elektronische Informationverarbeitung und Kybernetik 16* (1980), 319-323.

58. Schoenhage, A., "Storage modification machines," *SIAMJ. on Computing 9* (1980), 490-508.

59. Sipser, M., "On relativization and the existence of complete sets," *ICALP* 1982, 523-531.

60. Sipser, M., "Borel sets and circuit complexity," *15th ACM Symposium on Theory of Computing* (1983), 61-69.

61. Stoy, I.E., *"Denotational semantics:* The *Scott-Strachey Approach* to Programming Language Theory," MIT Press, Cambridge, Mass., 1977.

62. Tarski, A., "Some notions and methods on the borderline of algebra and meta-mathematics," *Proc. 1950 International Congress of Mathematicians,* Cambridge, MA, AMS, 1952, 705-720.

63. Tennent, R. D., *Principles of Programming Languages,* Prentice-Hall International, 1981.

64. Trakhtenbrot, B. A., "Impossibility of an algorithm for the decision problem on finite classes," *Doklady 70* (1950), 569-572.

65. Trakhtenbrot, B., J. Y. Halpern and A. R. Meyer, ' 'From denotational to operational and axiomatic semantics for ALGOL-like languages: an overview," *Lecture Notes in Computer Science,* Volume 164, Springer-Verlag, 1983.

66. Turing, A. M., "On computable numbers, with an application to the Entscheidungs-problem," *Proc. of London Mathematical Society* 2, no. 42 (1936), 230-236, and no. 43 (1936), 544-546.

67. Ullman, J. D., *Principles of Database Systems,* Computer Science Press, 1982.

68. Vardi, M., "Complexity of relational query languages," *14th Symp. on Theory of Computing,* ACM, 1982, 137-146.

69. Walkoe, W., "Finite partially-ordered quantification," *Journal of Symbolic Logic 35* (1970), 535-555.

70. Wegner, P., ' The Vienna Definition Language," *ACM Computing Surveys 4* (1972), 5-63.

71. Wirth, N., *Programming in Modula-2, 3rd edition,* Springer-Verlag, 1985.