# Computational Inductive Definability

Dexter Kozen
Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA
kozen@cs.cornell.edu

April 3, 2002

**Abstract**

It is shown that over any countable first-order structure, IND programs with dictionaries accept exactly the $\Pi^1_1$ relations. This extends a result of Harel and Kozen (1984) relating IND and $\Pi^1_1$ over countable structures with some coding power, and provides a computational analog of a result of Barwise, Gandy, and Moschovakis (1971) relating the $\Pi^1_1$ relations on a countable structure to a certain family of inductively definable relations on the hereditarily finite sets over that structure.

## 1 Introduction

Perhaps the central result of the theory of inductive definability is *Kleene's theorem*, which states that over $\mathbb{N}$, a relation is $\Pi^1_1$ iff it is inductively definable. The coding power of $\mathbb{N}$ is essential in the proof, and considerable effort has been spent in trying to generalize the result to structures without a coding capability. One can do without coding in the presence of some set-theoretic apparatus over the structure, although the theory is somewhat less satisfactory. There are numerous results that approximate Kleene's theorem in general structures, but these results typically hold only under various special conditions which are often difficult to state (see [2, 10]).

One such result is the following. In [1] (see [2, Corollary VI.3.9(i), p. 214]) it is shown that over any countable structure $\mathfrak{A}$, the $\Pi^1_1$ relations are equivalent to a certain class of inductively definable relations over $\mathbb{HF}\,\mathfrak{A}$, where $\mathbb{HF}\,\mathfrak{A}$ refers to the structure $\mathfrak{A}$ augmented with its hereditarily finite sets. Not all inductively definable relations over $\mathbb{HF}\,\mathfrak{A}$ are allowed, but only a certain subclass defined in terms of a restricted form of quantification on the sets. A similar result can be found in [9]. If in addition the structure has a coding capability, then the hereditarily finite sets can be coded directly in the structure. This result allows Kleene's theorem to be broken into two parts, one using the auxiliary set-theoretic apparatus in the Kleene construction, and the second coding the set-theoretic apparatus into the structure itself.

Kleene's theorem has a more computational interpretation than is apparent from [10, 2]. In [5], a programming language $\mathsf{IND}$ was defined and shown to compute exactly the inductive relations over any structure. By Kleene's theorem, $\mathsf{IND}$ computes exactly the $\Pi^1_1$ sets over $\mathbb{N}$. In fact, the programming language $\mathsf{IND}$ can also be used to give a more computationally motivated proof of Kleene's theorem (see [6]).

In this paper we augment $\mathsf{IND}$ programs with *dictionaries*, a common abstract data structure allowing storage and retrieval of data indexed by keys. Operations of insertion, membership testing, and access of an element by its key are allowed. Deletion is often allowed as well, although we do not need it here. In real implementations, dictionaries can be built from any one of a number of concrete data structures: trees, hashtables, extensible arrays, linked lists, heaps, searchable queues, or cons structures as in the programming languages Lisp or Scheme.

We show that over any countable first-order structure, $\mathsf{IND}$ programs with dictionaries accept exactly the $\Pi^1_1$ relations. This is the computational analog of the result of [1] mentioned above. Here dictionaries play the same role as the hereditarily finite sets in [1]: they are a data structure, nothing more nor less.

The main contribution here is not so much the result itself, but rather a new perspective on the results of [1, 2]. The language of inductive definability and admissible set theory is largely static, whereas our approach is dynamic. The language $\mathsf{IND}$ is a true programming language (although it computes highly noncomputable things), and it is designed to be programmable. Similarly, dictionaries are a true data structure, also designed to be programmable, unlike the hereditarily finite sets. Thus this result may help to clarify the role of the hereditarily finite sets and the special conditions of [2, Corollary VI.3.9(i), p. 214].

Other results that study the power of auxiliary data structures and unbounded memory in programming languages and logics can be found in [4, 13, 12, 14, 7, 8] (see also [6]).

# 2 The Programming Language $\mathsf{IND}$

The programming language $\mathsf{IND}$ was introduced in [5] (see also [6]). At the most basic level, $\mathsf{IND}$ programs consist of finite sequences of labeled statements of three forms:

- assignment: $\ell :\ x := \exists \qquad \ell :\ y := \forall$
- conditional jump: $\ell :\ \mathsf{if}\ R(\overline{t})\ \mathsf{then\ goto}\ \ell'$
- halt statement: $\ell :\ \mathsf{accept} \qquad \ell :\ \mathsf{reject}.$

More complex programming constructs can be defined from these.

The semantics of the existential and universal assignment is very much like alternating Turing machines [3] (see also [6]), except that the branching is infinite. Intuitively, the execution of an existential or universal assignment to a variable $x$ causes infinitely many subprocesses to be spawned, one for each element of the domain. The subprocess corresponding to element $a$ continues with $a$ assigned to the variable $x$. If the statement is $x := \exists$, the branching is existential; if it is $x := \forall$, the branching is universal. The conditional jump tests the atomic formula $R(\overline{t})$, and if true, jumps to the indicated label in the program. The accept and reject commands halt and pass a Boolean value, true or false respectively, back up to the parent. A process waiting at an existential branch reports acceptance to its parent as soon as one of its children reports acceptance; a process waiting at a universal branch reports acceptance to its parent as soon as all of its children reports acceptance.

The input is an initial assignment to the program variables. Execution of statements causes an infinitely branching computation tree to be generated downward, and Boolean accept (true) or reject (false) values are passed back up the tree, a Boolean $\vee$ being computed at each existential node and a Boolean $\wedge$ being computed at each universal node. The program is said to *accept* the input if the root of the computation tree ever becomes labeled with the Boolean value true on that input; it is said to *reject* the input if the root ever becomes labeled with the Boolean value false on that input; and it is said to *halt* on an input if it either accepts or rejects that input. An IND program that halts on all inputs is said to be *total*.

Note that there is no explicit mechanism for spawning processes or for passing Boolean values back up the computation tree. These are just intuitive devices. The reader is referred to [6] for a more formal treatment of the semantics of IND programs.

In [5], it was shown that IND programs accept exactly the inductive relations on any first-order structure, and that total IND programs accept exactly the hyperelementary relations. The theorem that a relation is hyperelementary iff it is both inductive and coinductive is proved quite simply by running a program for the relation and its complement in parallel, as with the corresponding result for r.e. and co-r.e. sets. Over $\mathbb{N}$, IND programs can be used as a notation for recursive ordinals. In fact, the recursive ordinals are exactly the running times of IND programs over $\mathbb{N}$. This formalism turns out to be equivalent to more conventional approaches (see for example [10, 11]), but has a decidedly more computational flavor. However, note that the relations computed by IND programs are highly noncomputable in the usual sense of the word.

Other useful programming constructs can be defined in terms of those listed above. An unconditional jump is effected by a conditional jump with test true. More complicated forms of conditional branching, for and while loops, etc. can be effected by manipulation of control flow. For example, the statement

if $R(\overline{t})$ then reject else $\ell$

is simulated by the program segment

```
      if R(t̄) then goto ℓ';
      goto ℓ;
ℓ':   reject
```

A simple assignment is effected by guessing and verifying:

$x := t$

is simulated by

$x := \exists$;
if $x \neq t$ then reject

The process spawns infinitely many subprocesses, all but one of which immediately reject.

A relation is first-order iff it is definable by a loop-free program. However, IND can also accept inductively definable relations that are not first-order definable. For example, the reflexive transitive closure of a relation $R$ is definable by the following program, which takes its input in the variables $x, z$ and accepts if $(x, z) \in R^*$:

```
while x ≠ z {
   y := ∃;
   if ¬R(x, y) then reject;
   x := y;
}
accept;
```

Further examples involving two-person games of perfect information and well-founded binary relations can be found in [5, 6].

Any relation that is expressed as a least fixpoint of a monotone map defined by a positive first-order formula can be computed by an IND program. Essentially, the program deconstructs the formula in a top-down fashion, executing existential assignments at existential quantifiers, executing universal assignments at universal quantifiers, using control flow for the propositional connectives, using conditional tests for the atomic formulas, and looping back to the top of the program at (positive) occurrences of the inductive relation symbol.

Conversely, any relation computed by an IND program is inductive in the traditional sense, essentially because the formal semantics of acceptance involves the least fixpoint of an inductively defined set of labelings of the computation tree with Boolean values.

We refer the reader to [5, 6] for further details.

# 3 IND Programs with Dictionaries

A *dictionary* is an abstract data structure for storing data values indexed by keys. Operations supported are insertion, membership, and lookup of a data item by key. Deletion is also sometimes included, although we will not need it for our application. Dictionaries can be implemented in a variety of ways: hashtables, linked lists, extensible arrays, heaps, searchable queues, or cons structures as in Lisp and Scheme.

Formally, a *dictionary* is a partial function with finite domain from a set of *keys* to set of *data values*. In our application, the keys are $k$-tuples of elements of $A$ and the data values are elements of $A$, thus dictionaries are (extensional) partial functions $A^k \to A$. The following operations on dictionaries are supported:

| | |
|---|---|
| reset( ) | clear the dictionary |
| put($\overline{x}, y$) | insert data element $y$ with key $\overline{x} = x_1, \ldots, x_k$ |
| containsKey($\overline{x}$) | does there exist an entry with key $\overline{x}$? |
| get($\overline{x}$) | get the data element corresponding to key $\overline{x}$ |

There is a separate version for each arity $k$. For simplicity, we assume that there is a separate collection of program variables $d, e, \ldots$ ranging over dictionaries, and that they are initialized to the empty dictionary. Assignments and equality tests may not be applied to dictionaries; only the operations above are allowed.

**Lemma 3.1** *Let $\mathfrak{A}$ be a first-order structure with countable domain $A$. Any $\Pi_1^1$ relation on $\mathfrak{A}$ is accepted by an* IND *program with dictionaries.*

*Proof.* By transforming to prenex form and Skolemizing, every $\Pi_1^1$ formula can be written with a quantifier prefix of the form

$$\forall f_1 : A^{n_1} \to A \ \ldots \ \forall f_k : A^{n_k} \to A \quad \exists x_1 : A \ \ldots \ \exists x_m : A$$

followed by a quantifier-free part $\varphi(f_1, \ldots, f_k, x_1, \ldots, x_m)$. There may be other free variables besides those mentioned. In the presence of a definable pairing function (as is the case with $\mathbb{N}$), we could further reduce to the form

$$\forall f \ \exists x \ \varphi(f, x), \tag{1}$$

where $f$ is a unary, but in general we do not have this luxury. However, for simplicity of notation, we will give the construction only for the case (1), since all the main ideas are already contained here.

Amend the semantics of $\varphi$ to allow as first argument a partial function with finite domain as represented by a dictionary $d$. We think of $d$ as a finite approximation to a total function $f$. If $d$ does not have enough information to determine whether $\varphi(f, x)$, then the value of $\varphi(d, x)$ is defined to be false.

The value of $\varphi(d, x)$ can easily be determined by a loop-free IND program. For example, if $\varphi(f, x)$ is $x = f(f(x))$, then to simulate

if $x = f(f(x))$ then $\alpha$ else $\beta$,

we could write

```
if d.containsKey(x) {           //does d contain a value for f(x)?
   y := d.get(x);               //if so, get it
   if d.containsKey(y) {        //does d contain a value for f(f(x))?
      z := d.get(y);            //if so, get it
      if x = z then α else β;   //test whether x = f(f(x))
   }
}
goto β;
```

Note that if $d$ does not contain enough information to determine the value of $f(f(x))$, then control is transferred to $\beta$.

We write $d \sqsubseteq f$ and say that $f$ *extends* $d$ if the domain of $d$ is contained in the domain of $f$, and if $d$ and $f$ agree on the domain of $d$. Our analysis is based on the following two continuity properties:

**Proposition 3.2**

(i) *If $d \sqsubseteq f$ and $\varphi(d, x)$, then $\varphi(f, x)$.*

(ii) *If $\varphi(f,x)$, then there exists $d \sqsubseteq f$ with finite domain such that $\varphi(d,x)$.*

These properties hold because the truth value of $\varphi(f,x)$ is determined by finitely many values of $f$, since there are only finitely many occurrences of $f$ in $\varphi$. If all those values are represented by $d$, then $\varphi(f,x)$ and $\varphi(d,x)$ will have the same truth value.

Here is an IND program with a single unary dictionary $d$ that tests whether (1) holds. It uses $d$ to construct finite approximations of $f$. As mentioned, the test $\varphi(d,x)$ in the while statement can be computed by a loop-free IND program.

```
d.reset( );
x := ∃;
while ¬φ(d, x) {
   y := ∃;
   if d.containsKey(y) then reject;
   z := ∀;
   d.put(y, z);
   x := ∃;
}
accept;
```

The program iteratively extends $d$ in all possible ways, seeking a finite partial function $d$ and an $x$ for which $\varphi(d,x)$.

We wish to show that this program accepts iff (1) holds. Suppose first that (1) holds. To show that the program accepts, it suffices to exhibit an *accepting subtree* of the computation tree. This is a subtree obtained by determinizing every existential branch (that is, pruning all children except one), such that all paths in the resulting tree lead to an accept statement.

We determinize the existential branches as follows. Let $\leq$ be an arbitrary but fixed ordering of $A$ of order type $\omega$, which exists since $A$ is countable. Let next$(d)$ be the $\leq$-least element of $A$ not contained in the domain of $d$. Let witness$(d)$ be the $\leq$-least element $x$ for which there exists a total $f$ extending $d$ such that $\varphi(f,x)$. Such an $x$ exists by (1). To resolve the two assignments $x := \exists$, use the value witness$(d)$. To resolve the assignment $y := \exists$, use the value next$(d)$. If these values were expressible, the resulting subtree would be generated by the program

```
d.reset( );
x := witness(d);
while ¬φ(d, x) {
   y := next(d);
   z := ∀;
   d.put(y, z);
   x := witness(d);
}
accept;
```

Note that we were also able to omit the test

```
if d.containsKey(y) then reject
```

since $\mathsf{next}(d)$ is never in the domain of $d$.

To show that every path in the computation tree of this new program leads to acceptance, it suffices to show that the while loop terminates along every path. Suppose it did not. Any computation path for which the while loop does not terminate generates a total function $f : A \to A$, namely the limit of the values of $d$ along that path. By (1), there exists $x$ such that $\varphi(f, x)$. By Proposition 3.2(ii), there exists a finite approximation $e \sqsubseteq f$ such that $\varphi(e, x)$. By Proposition 3.2(i), there exists a value of $d$ along the computation path such that $\varphi(d, x)$, which would have caused the while loop to terminate. This is a contradiction.

Now we argue that if the original program accepts, then (1) holds. Consider any accepting subtree obtained by resolving the existential branches. For any total $f : A \to A$, resolve the universal branch $z := \forall$ by supplying the value of $f(y)$. This results in a single computation path of the accepting subtree, since there are no more branches. Since the subtree is accepting, the path must terminate. But by construction, $f$ extends all values of $d$ along that path, and the final values of $d$ and $x$ satisfy $\varphi(d, x)$, since the while loop terminated. By Proposition 3.2(i), $\varphi(f, x)$. Since $f$ was arbitrary, (1) holds.

This gives the construction for formulas of the simple form (1). More complicated formulas might require more dictionaries and dictionaries of higher arity, but the construction is no more difficult except notationally. □

**Lemma 3.3** *Let $\mathfrak{A}$ be a first-order structure with countable domain $A$. Any relation on $\mathfrak{A}$ accepted by an* IND *program with dictionaries is $\Pi^1_1$.*

*Proof.* First we pick an appropriate concrete representation of dictionaries. We will use an auxiliary data structure similar to cons structures of Lisp and Scheme. Let pair and nil be function symbols of arity 2 and 0, respectively. Let $\mathcal{C}(\mathfrak{A})$ be the free term algebra over pair and nil generated by $A$. The elements of $\mathcal{C}(\mathfrak{A})$ are finite labeled binary trees whose leaves are labeled with elements of $A$ or the empty tree nil and whose internal nodes are labeled with pair.

We endow $\mathcal{C}(\mathfrak{A})$ with distinguished operations $\mathsf{pair}^{\mathcal{C}(\mathfrak{A})}$, $\mathsf{nil}^{\mathcal{C}(\mathfrak{A})}$, $\mathsf{head}^{\mathcal{C}(\mathfrak{A})}$, and $\mathsf{tail}^{\mathcal{C}(\mathfrak{A})}$, where $\mathsf{pair}^{\mathcal{C}(\mathfrak{A})}$ and $\mathsf{nil}^{\mathcal{C}(\mathfrak{A})}$ are interpreted syntactically, and where $\mathsf{head}^{\mathcal{C}(\mathfrak{A})}$ and $\mathsf{tail}^{\mathcal{C}(\mathfrak{A})}$ are the left and right projections, respectively, corresponding to $\mathsf{pair}^{\mathcal{C}(\mathfrak{A})}$. In addition, we define the following unary predicates on $\mathcal{C}(\mathfrak{A})$:

$$\begin{aligned}
\mathsf{isPair}^{\mathcal{C}(\mathfrak{A})}(x) &\stackrel{\text{def}}{=} \exists y\, \exists z\; x = \mathsf{pair}^{\mathcal{C}(\mathfrak{A})}(y, z) \\
\mathsf{isNil}^{\mathcal{C}(\mathfrak{A})}(x) &\stackrel{\text{def}}{=} x = \mathsf{nil}^{\mathcal{C}(\mathfrak{A})} \\
\mathsf{isElement}^{\mathcal{C}(\mathfrak{A})}(x) &\stackrel{\text{def}}{=} \neg\mathsf{isPair}^{\mathcal{C}(\mathfrak{A})}(x) \wedge \neg\mathsf{isNil}^{\mathcal{C}(\mathfrak{A})}(x).
\end{aligned}$$

Since $A$ is countable, there exists an embedding of $\mathcal{C}(\mathfrak{A})$ into $A$, although the embedding is not necessarily explicitly definable in $\mathfrak{A}$. Specifically, there exist functions

$$\begin{array}{ll}
\mathsf{pair} : A^2 \to A & \quad \mathsf{encode} : A \to A \\
\mathsf{head} : A \to A & \quad \mathsf{decode} : A \to A \\
\mathsf{tail} : A \to A & \quad \mathsf{nil} \in A
\end{array}$$

and predicates

$$\begin{aligned}
\mathsf{isPair}(x) &\stackrel{\mathrm{def}}{=} \exists y\, \exists z\; x = \mathsf{pair}(y, z) \\
\mathsf{isNil}(x) &\stackrel{\mathrm{def}}{=} x = \mathsf{nil} \\
\mathsf{isElement}(x) &\stackrel{\mathrm{def}}{=} \neg\mathsf{isPair}(x) \wedge \neg\mathsf{isNil}(x)
\end{aligned}$$

satisfying the following *coherence conditions*:

$$\begin{aligned}
&\forall x\, \forall y\; \mathsf{head}(\mathsf{pair}(x, y)) = x \\
&\forall x\, \forall y\; \mathsf{tail}(\mathsf{pair}(x, y)) = y \\
&\forall x\; \mathsf{isPair}(x) \to \mathsf{pair}(\mathsf{head}(x), \mathsf{tail}(x)) = x \\
&\forall x\; \mathsf{decode}(\mathsf{encode}(x)) = x \\
&\forall x\; \mathsf{isElement}(x) \to \mathsf{encode}(\mathsf{decode}(x)) = x \\
&\forall x \text{ exactly one of } \mathsf{isPair}(x),\ \mathsf{isNil}(x),\ \mathsf{isElement}(x).
\end{aligned}$$

We abbreviate the conjunction of these conditions by

$$\mathsf{coherent}(\mathsf{pair}, \mathsf{head}, \mathsf{tail}, \mathsf{encode}, \mathsf{decode}, \mathsf{nil}).$$

Let $\Sigma$ be the signature of $\mathfrak{A}$. Let $f$ and $R$ stand for function and relation symbols, respectively, of $\Sigma$. For some choice of pair, head, tail, encode, decode, and nil satisfying the coherence conditions, consider the structure

$$\mathfrak{A}' \quad = \quad (A,\, \Sigma',\, \mathsf{pair},\, \mathsf{head},\, \mathsf{tail},\, \mathsf{nil},\, \mathsf{encode},\, \mathsf{decode}),$$

where

$$\begin{aligned}
\Sigma' &\stackrel{\mathrm{def}}{=} \{f' \mid f \in \Sigma\} \cup \{R' \mid R \in \Sigma\} \\
f' &\stackrel{\mathrm{def}}{=} \mathsf{encode} \circ f \circ \mathsf{decode}, \\
R' &\stackrel{\mathrm{def}}{=} R \circ \mathsf{decode};
\end{aligned}$$

that is,

$$\begin{aligned}
f'(a_1, \ldots, a_n) &\stackrel{\mathrm{def}}{=} \mathsf{encode}(f(\mathsf{decode}(a_1), \ldots, \mathsf{decode}(a_n))), \\
R'(a_1, \ldots, a_n) &\stackrel{\mathrm{def}}{=} R(\mathsf{decode}(a_1), \ldots, \mathsf{decode}(a_n)).
\end{aligned}$$

The map $\mathsf{encode} : A \to A$ extends uniquely to a homomorphism of the structure

$$\mathcal{C}(\mathfrak{A}) \quad = \quad (\mathcal{C}(\mathfrak{A}),\, \Sigma,\, \mathsf{pair}^{\mathcal{C}(\mathfrak{A})},\, \mathsf{head}^{\mathcal{C}(\mathfrak{A})},\, \mathsf{tail}^{\mathcal{C}(\mathfrak{A})},\, \mathsf{nil}^{\mathcal{C}(\mathfrak{A})})$$

into $\mathfrak{A}'$.

Now any program $P$ over $\mathcal{C}(\mathfrak{A})$ can be simulated by a program $P'$ over $\mathfrak{A}'$. The program $P'$ is obtained from $P$ by the following transformations:

(i) replace $f$ by $f'$ and $R$ by $R'$;

(ii) replace $x := \exists$ by the program $x := \exists\,;$ if $\neg\mathsf{isElement}(x)$ then reject;

(iii) replace $y := \forall$ by the program $y := \forall\,;$ if $\neg\mathsf{isElement}(x)$ then accept.

Then $P$ accepts $x_1, \ldots, x_n$ iff $P'$ accepts $\mathsf{encode}(x_1), \ldots, \mathsf{encode}(x_n)$. But by [5], the predicate "$P'$ accepts $\mathsf{encode}(x_1), \ldots, \mathsf{encode}(x_n)$" can be expressed by a $\Pi^1_1$ formula $\psi$ over $\mathfrak{A}'$, thus $P$ accepts $x_1, \ldots, x_n$ iff the following formula is true in $\mathfrak{A}$:

$$\begin{aligned}&\forall \mathsf{pair}\ \forall \mathsf{head}\ \forall \mathsf{tail}\ \forall \mathsf{encode}\ \forall \mathsf{decode}\ \forall \mathsf{nil}\\ &\quad \mathsf{coherent}(\mathsf{pair}, \mathsf{head}, \mathsf{tail}, \mathsf{encode}, \mathsf{decode}, \mathsf{nil}) \to \psi.\end{aligned}$$

This is a $\Pi^1_1$ formula.

It remains only to show how to use the pairing apparatus of $\mathcal{C}(\mathfrak{A})$ to implement dictionaries. This is quite standard. We represent a dictionary as a list of (key,value) pairs terminated by nil. For example, $a_1 \mapsto b_1, a_2 \mapsto b_2, a_3 \mapsto b_3$ would be represented as the list

$$\mathsf{pair}(\mathsf{pair}(a_1, b_1), \mathsf{pair}(\mathsf{pair}(a_2, b_2), \mathsf{pair}(\mathsf{pair}(a_3, b_3), \mathsf{nil}))).$$

The dictionary operations can be implemented as follows:

- $d$.reset( ):

```
d := nil;
```

- $d$.put$(x, y)$:

```
d := pair(pair(x, y), d);
```

- if $d$.containsKey$(x)$ then $\alpha$ else $\beta$:

```
for (e := d; e ≠ nil; e := tail e) {
   if (head(head e) = x) then α;
}
goto β;
```

- $y := d$.get$(x)$:

```
for (e := d; e ≠ nil; e := tail e) {
   if (head(head e) = x) {
      y := tail(head e);
      break;
   }
}
```

□

Combining Lemmas 3.1 and 3.3, we have shown

**Theorem 3.4** *Over any countable structure,* IND *programs with dictionaries accept exactly the* $\Pi^1_1$ *relations.*

## Acknowledgements

This work was supported in part by NSF grant CCR-0105586 and by ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

## References

[1] J. Barwise, R. Gandy, and Y. Moschovakis. The next admissible set. *J. Symbolic Logic*, 36:108–120, 1971.

[2] J. Barwise. *Admissible Sets and Structures*. North-Holland, 1975.

[3] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. Assoc. Comput. Mach.*, 28(1):114–133, 1981.

[4] P. Hajek and P. Kurka. A second-order dynamic logic with array assignments. *Fund. Informaticae*, IV:919–933, 1981.

[5] D. Harel and D. Kozen. A programming language for the inductive sets, and applications. *Information and Control*, 63(1–2):118–139, 1984.

[6] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, 2000.

[7] A. R. Meyer and J. Tiuryn. A note on equivalences among logics of programs. In D. Kozen, editor, *Proc. Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comput. Sci.*, pages 282–299. Springer-Verlag, 1981.

[8] A. R. Meyer and J. Tiuryn. Equivalences among logics of programs. *Journal of Computer and Systems Science*, 29:160–170, 1984.

[9] Y. N. Moschovakis. Abstract first order computability I. *Trans. Amer. Math. Soc.*, 138:427–464, 1969.

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

[11] Y. N. Moschovakis. *Descriptive Set Theory*. North-Holland, 1980.

[12] J. Tiuryn. Unbounded program memory adds to the expressive power of first-order programming logics. *Infor. and Control*, 60:12–35, 1984.

[13] J. Tiuryn. Higher-order arrays and stacks in programming: an application of complexity theory to logics of programs. In Gruska and Rovan, editors, *Proc. Math. Found. Comput. Sci.*, volume 233 of *Lect. Notes in Comput. Sci.*, pages 177–198. Springer-Verlag, 1986.

[14] P. Urzyczyn. “During” cannot be expressed by “after”. *Journal of Computer and System Sciences*, 32:97–104, 1986.