DEFINABILITY BY PROGRAMS IN FIRST-ORDER STRUCTURES

A.J. KFOURY

Mathematics Department, Boston University, Boston, MA 02215, U.S.A.

Communicated by A. Meyer
Received July 1981
Revised February 1982

Contents

Introduction ............................................................ 1
§1. Preliminary definitions and results .......................... 3
§2. Algebraic characterizations of the unwind property .... 8
§3. A structure with the unwind property for every iterative (but not every recursive) program 18
§4. The truth-table property ..................................... 22
§5. Structures with the truth-table property (but not the unwind property) for every iterative program .................. 31
Appendix 1: The pebble game on infinite dags ............. 47
Appendix 2: Remaining proofs ................................. 55
References .......................................................... 65

Introduction

Data types bear the same relationship to programming formalisms as relational structures bear to first-order languages. If we identify data types and relational structures, a comparative study emerges between programming formalisms and first-order languages.

Whereas definability by first-order formulas has been studied for a long time now, and extensively covered in first-order model theory, only in recent years have questions of definability by programs been considered. Interest in these questions is largely due to developments in the area of program semantics and verification. But this part of theoretical computer science is still in its infancy, and much mathematics remains to be done before some of the questions are fully understood.
The focus in this report is on definability by deterministic ‘recursive programs’ and ‘iterative (or flow-chart) programs’, which are undoubtedly the two most familiar programming formalisms. We only allow conventional instructions, i.e. simple assignments and atomic tests, in these programs – leaving for future research programs with more general instructions (such as random assignments, first-order tests, and others). The crucial difference between recursive and iterative programs, which will be exploited again and again throughout the paper, is that the first may use unbounded storage space while the second may not. As the theory develops, we shall also need to distinguish between programs that are equipped with ‘counters’ and those that are not. Restricted programming formalisms, such as ‘loop-free programs’ or ‘truth-table programs’ (which compute an output value by a finite table look-up), will be considered as the need arises.

The paper is organized as follows. In Section 1 we define the general setting for the study of program definability in first-order structures, used in this report and the forthcoming [13] and [14].

The remaining sections in this paper primarily deal with two kinds of program behavior over first-order structures:

1. every program has a uniform bound on the length of its terminating computations, i.e., every program has the ‘unwind property’, and

2. every program is equivalent to one which has a uniform bound on the length of its terminating computations, i.e., every program has the ‘truth-table property’.

We first develop tools for the examination of these two behaviors relative to the programming formalisms mentioned above. These are the characterizations given in Theorems 2.9 and 2.10 (supporting definitions are 2.1, 2.3 and 2.5). Corollary 2.13 is an interesting model-theoretic consequence. Also based on Theorems 2.9 and 2.10, Section 3 puts in sharper focus many of the differences between first-order quantification logic and programming formalisms.

The main results are established in Sections 4 and 5. Here we first prove that behaviors (1) and (2) coincide for recursive programs over all structures (Theorem 4.2). Theorems 4.4 and 5.2 together say that these two behaviors relative to iterative programs over computable structures also coincide, but even over $\Delta_2^0$ structures, they do not. What is more, Theorems 4.9 and 5.1 together say that if these two behaviors do not coincide relative to iterative programs over a structure with ‘infinite chains’, then this structure cannot be arithmetical.

Theorem 5.1 was first established to settle (in the affirmative) the following problem, “Does there exist a first-order structure over which (a) every iterative program is equivalent to a loop-free program, but (b) not every iterative program ‘unwinds’, i.e. runs in a bounded number of steps?’”

In the follow-up paper [13] we consider again some of these questions, but only when programs are restricted to compute total functions. The resulting theory is sufficiently different to justify a separate treatment.
The proof techniques we use are for the most part a mixture of recursion theory, model theory, a new version of the ‘pebble game’, and some careful programming. Interestingly, the techniques in [13] are less recursion-theoretic and considerably more model-theoretic, drawing heavily on questions of ‘type omission’ and ‘type realization’ relative to appropriately defined types (namely, r.e. sets of quantifier-free formulas).

Some of the results in this paper were presented in outline or without proof in [11] and [12]. Several earlier reports have also in one way or another contributed to the ideas developed here; most noteworthy are the papers by F. Abramson [1], E. Engeler [6–8, and others], and J.C. Shepherdson [20].

1. Preliminary definitions and results

The general setting of this paper is defined by the following concepts.

1.1 Data types. We take a data type $\mathfrak{A}$ to be an object of the form

$$\mathfrak{A} = (A; \sim, r^A_1, \ldots, r^A_m, f^A_1, \ldots, f^A_n)$$

where $A$ is a set of individual elements, $\sim$ is the equality relation (different for the metatheoretic $=$), and $r^A_1, \ldots, r^A_m; f^A_1, \ldots, f^A_n$ are primitive relations and functions on $A$, each with a fixed arity $\geq 0$. A data type $\mathfrak{A}$ as just defined may be viewed as a one-sorted (first-order) structure. The sequence $\tau$ of primitive relation and function symbols, namely $\tau = (r_1, \ldots, r_m; f_1, \ldots, f_n)$, is the similarity type of $\mathfrak{A}$.

1.2 Programs and program schemes. Given a similarity type $\tau$, associated with some structure $\mathfrak{A}$, we consider two classes of program schemes: iterative program schemes and recursive program schemes. An ‘iterative program scheme’ can be drawn as a flow-chart which only mentions relation and function names from $\tau$; whereas a ‘recursive program scheme’ can be drawn as a flow-chart which may also mention in its instructions names of program schemes (including itself).

Program schemes can be with or without counters. Unless we explicitly specify it not to be the case, we assume throughout this paper that all program schemes are equipped with counters, i.e. that elementary arithmetic is part of their control structure.

We can formally define an iterative (or flow-chart) program scheme $S$ – with input variables $\{x_1, x_2, \ldots, x_k\}$, $k \geq 0$, and over similarity type $\tau = (r_1, \ldots, r_m; f_1, \ldots, f_n)$ – to be a finite flow-diagram built up from two kinds of instructions: assignments and tests. In the following definitions we assume we have an infinite supply of program variables $\{y_i | i \in \omega\}$ and counters $\{c_i | i \in \omega\}$. Input and
program variables are assigned values from the universe $A$ of a structure $\mathcal{U}$, whereas counters are assigned values from $\mathbb{N}$ (the set of natural numbers).

(1) An assignment instruction can take one of the following forms:

$$v_1 := v_2$$

where $v_1 \in \{y_i \mid i \in \omega\}$ and $v_2 \in \{x_1, \ldots, x_k\} \cup \{y_i \mid i \in \omega\}$.

$$v := f(v_1, v_2, \ldots, v_j)$$

where $f \in \{f_1, \ldots, f_n\}$, $j \geq 0$ is the arity of $f$, and $v, v_1, \ldots, v_j \in \{y_i \mid i \in \omega\}$.

$$c_i := c_i$$

$$c_i := 0 \quad \text{or} \quad c_i := c_i + 1 \quad \text{or} \quad c_i := c_i \div 1$$

Note that counters do not 'communicate' with input and program variables. Also assignment instructions do not change the values of input variables, only the values of program variables and counters.

(2) A test instruction can take one of the following forms:

$$v_1 \neq v_2$$

where $v_1, v_2 \in \{y_i \mid i \in \omega\}$.

$$r(v_1, v_2, \ldots, v_t)$$

where $r \in \{r_1, \ldots, r_t\}$. 
where \( v_1, \ldots, v_l \in \{ y_i \mid i \in \omega \} \), \( r \in \{ r_1, \ldots, r_m \} \) and \( j \geq 1 \) is the arity of \( r \):

\[
\begin{align*}
  c_i &= c_i^j \\
  \text{yes} & \quad \text{no}
\end{align*}
\]

To complete the specification of program scheme \( S \) we require that it (as a flow-diagram) have exactly one entry point labelled with input variables \( x_1, \ldots, x_k \); and each of the exit points of \( S \) be labelled either with a variable in \( \{ y_i \mid i \in \omega \} \) from which an output value is to be read off, or with a special instruction DIVERGE which stands for any self-loop:

\[
\text{DIVERGE = } \begin{cases} 
\text{yes} \\ \text{no}
\end{cases}
\]

The presence of DIVERGE in our programming formalism will allow us to simplify several statements of later results.

If we give to each relation and function symbol appearing in \( S \) its proper meaning in structure \( \mathfrak{A} \), we obtain an iterative program over \( \mathfrak{A} \) (or more simply, an iterative \( \mathfrak{A} \)-program) denoted by \( S^\mathfrak{A} \). Clearly \( S^\mathfrak{A} \) defines a \( k \)-ary (partial) function on \( \mathbb{A} \). We denote the computation of \( S^\mathfrak{A} \) on input \( (a_1, \ldots, a_k) \in \mathbb{A}^k \) by \( S^\mathfrak{A}(a_1, \ldots, a_k) \). The computation \( S^\mathfrak{A}(a_1, \ldots, a_k) \) corresponds to a unique path, possibly infinite, through the flow-diagram of \( S \); and a step in the computation is any instruction along the path thus determined. (The results of this paper apply only to deterministic programs.)

The domain of a program \( S^\mathfrak{A} \) with input variables \( x_1, \ldots, x_k \), denoted \( \text{domain} (S^\mathfrak{A}) \), is the set of all input vectors \( a \in \mathbb{A}^k \) for which \( S^\mathfrak{A}(a) \) converges. We say that \( S^\mathfrak{A} \) is total if \( \text{domain} (S^\mathfrak{A}) = \mathbb{A}^k \).

Sometimes we want program \( S^\mathfrak{A} \) to define a \( k \)-ary predicate on \( \mathbb{A} \). In this case we do not need to label each of the exit points of \( S \) with an output variable, the predicate thus defined by \( S^\mathfrak{A} \) being \( \text{domain} (S^\mathfrak{A}) \). (Our convention here is different from the usual one, which defines a \( k \)-ary predicate by a total function \( : \mathbb{A}^k \to \{0, 1\} \).)

A recursive program scheme \( S \) – with input variables \( \{ x_1, x_2, \ldots, x_k \} \) and over similarity type \( \tau = (r_1, \ldots, r_m; f_1, \ldots, f_n) \) – is more general than an iterative program scheme in that it allows in an assignment instruction of the form (1.2) the function symbol \( f \) to be also the name of a program scheme (possibly \( S \) itself). An apparently more general notion of ‘recursive program scheme’ also allows in a test instruction of the form (2.2) the relation symbol \( r \) to be the name of a program scheme (possibly \( S \) itself) with each of its exit points having a label from \{yes, no\}; however, in the.
presence of the equality relation \( \sim \) and two distinguished elements (identified with 'yes' and 'no' respectively), there is no loss of generality in restricting recursive calls to appear in assignment instructions only.

More on the classical theory of program schemes can be found in [2, 4, 10, 15].

**Remarks on terminology.** (1) For clarity in the text we try to reserve the word 'recursive' to qualify program schemes and programs. A 'partial recursive function' on \( \mathbb{N} \) will be called a **computable function** on \( \mathbb{N} \), which may or may not be total.

(2) Also, somewhat unconventional, we take the word 'partial' to mean 'non-total'. So that when we talk about a 'function' without further qualification, the function is either 'total' or 'partial' but not both.

### 1.3 Programs with parameters.

The collection of all program schemes, whether iterative or recursive, in a fixed similarity type is recursively enumerable. When we talk about 'all iterative (recursive) \( \mathcal{A} \)-programs', we thus talk about the countably infinite set of all iterative (recursive) program schemes \( S \) interpreted in \( \mathcal{A} \). As in common programming practice however, we would like to allow finitely many parameters from the universe \( A \) of \( \mathcal{A} \) to appear in a program, in addition to those included as zeroary functions in the similarity type.

To write programs over \( \mathcal{A} \) with parameters from a finite subset \( X = \{a_1, \ldots, a_p\} \subseteq A \), we have to consider the similarity type of the following structure:

\[
(\mathcal{A}, X) = (A; a^A_1, \ldots, a^A_p; r^A_1, \ldots, r^A_m; f^A_1, \ldots, f^A_n, a_1, \ldots, a_p)
\]

where \( a_1, \ldots, a_p \) are viewed as constant primitive functions in \( (\mathcal{A}, X) \). The similarity type of \( (\mathcal{A}, X) \) is denoted by

\[
\tau_X = (r_1, \ldots, r_m; f_1, \ldots, f_n, a_1, \ldots, a_p)
\]

where \( a_i \) is a zeroary function symbol corresponding to constant function \( a_i \). If \( S \) is a program scheme in similarity type \( \tau_X \), then \( S^{(\mathcal{A}, X)} \) is a **program over \( \mathcal{A} \) with parameters from \( X \)**, or more simply a \( (\mathcal{A}, X) \)-program. The set of all iterative (recursive) programs over \( \mathcal{A} \) with finitely many parameters is then:

\[
\{ S^{(\mathcal{A}, X)} \mid S \text{ is an iterative (recursive) program scheme in similarity type } \tau_X \text{ of } (\mathcal{A}, X) \text{ for some finite } X \subseteq A \},
\]

which we shall also call the set of all iterative (recursive) \( (\mathcal{A}, A) \)-programs. A \( (\mathcal{A}, A) \)-program is thus a \( (\mathcal{A}, X) \)-program for some finite \( X \subseteq A \).

It is not only for the sake of greater generality that we allow parameters in programs. Some of the constructions are greatly simplified when we can freely introduce parameters in programs (as in Section 5 for example); also parts of the theory to be developed become smoother in the presence of parameters (especially in the case of definability by total programs [13]).
1.4 FED’s and RED’s. Sometimes program schemes are difficult to work with, especially in the presence of nested loops and nested recursive calls. It is often simpler to work with their translations into effective definitional schemes, which were defined by Friedman [9], and later examined by Shepherdson [2] in great detail. An effective definitional scheme can be either functional (abbreviated fed) or relational (abbreviated red).

Formally, a fed $F$ over similarity type $\tau = (r_1, \ldots, r_m; f_1, \ldots, f_n)$ and with input variables $\{x_1, \ldots, x_k\}$ is a recursively enumerable sequence of ordered pairs. Assuming $F$ infinite, we can write it as follows:

$$
F(x_1, \ldots, x_k) = (\langle \alpha_i, t_i \rangle | i \in \omega),
$$

where $\alpha_i$ is a finite conjunction of atomic and negated atomic formulas, and $t_i$ is a term—both $\alpha_i$ and $t_i$ being over similarity type $\tau$ and containing no free variables other than $\{x_1, \ldots, x_k\}$. When $F$ is finite, we write $F = (\langle \alpha_i, t_i \rangle | i \in I)$ for some initial finite segment $I \subset \omega$.

If we interpret fed $F$ in a structure $\mathfrak{A}$ we obtain a function $F^\mathfrak{A} : \mathbb{A}^k \to \mathbb{A}$ whose value at $(a_1, \ldots, a_k) \in \mathbb{A}^k$ is:

$$
F^\mathfrak{A}(a_1, \ldots, a_k) = \begin{cases} 
t_A(a_1, \ldots, a_k), & \text{where } i \in \omega \text{ is the smallest index for which } \mathfrak{A} \models \alpha_i[a_1, \ldots, a_k], \\
\text{undefined,} & \text{otherwise,}
\end{cases}
$$

where $t_A^i$ is the interpretation of term $t_i$ in $\mathfrak{A}$.

Friedman’s original definition of a red is identical to the definition of a fed, except that the second component $t_i$ of each ordered pair $\langle \alpha_i, t_i \rangle$ is now from the set $\{\text{yes, no}\}$. Following our treatment of program schemes in 1.2, our approach is to associate a red $R$ with every fed $F$, by omitting second components of ordered pairs; that is, corresponding to the fed $F$ above, we have the red $R$:

$$
R(x_1, \ldots, x_k) = (\langle \alpha_i \rangle | i \in \omega).
$$

The relation $R^\mathfrak{A}$ is clearly $\{a \in \mathbb{A}^k | F^\mathfrak{A}(a) \text{ defined}\}$.

We say that program scheme $S(x_1, \ldots, x_k)$ and fed $F(x_1, \ldots, x_k)$ are $\mathfrak{A}$-equivalent if program $S^\mathfrak{A}$ computes function $F^\mathfrak{A}$ on $\mathbb{A}$. And we say that $S$ and $F$ are equivalent if they are $\mathfrak{A}$-equivalent for every structure $\mathfrak{A}$. (We assume that $S$, $F$, and $\mathfrak{A}$ are all in the same similarity type.)

The important fact linking program schemes and fed’s is the following one. A proof can be found in [20].

1.5 Fact. (a) Given an arbitrary program scheme $S$ (iterative or recursive), we can effectively find a fed $F$ equivalent to $S$; (b) Given an arbitrary fed $F$, we can effectively find a recursive program scheme $S$ with counters which is equivalent to $F$.

In view of this basic result, if we are given a program scheme $S$, we shall write $\text{fed}(S)$ and $\text{red}(S)$ for the corresponding (effectively defined) fed and red. The
simplicity and convenience of working with \textit{fed}(S) or \textit{red}(S), instead of \textit{S} itself, is illustrated by the proofs of Lemmas 2.7 and 2.8, and Corollaries 2.11 and 2.12 below.

We can now consider any program, or program scheme, and view it as a definition by (possibly infinitely many) cases – this is what a \textit{fed} is. And if this definition comprises infinitely many cases, then these must somehow be effectively generated. This leads to another advantage of working with \textit{fed}’s and \textit{red}’s, instead of program schemes, as explained next.

Given a structure \( \mathfrak{A} \), an \textit{informal algorithm} over \( \mathfrak{A} \) is any well-defined, effectively generated, possibly infinite, sequence of instructions which:

\begin{enumerate}
  \item assign to variables (or memory locations) values from \( \mathbb{A} \),
  \item amend values stored in variables using the primitive functions of \( \mathfrak{A} \), and
  \item direct the flow of computation by testing values in variables using the primitive relations of \( \mathfrak{A} \).
\end{enumerate}

For book-keeping purposes, we allow an informal algorithm to also use counters, and the basic arithmetical operations and relations on counters; that is, elementary arithmetic is part of the control structure of an informal algorithm. (This explicit use of counters is not essential, because an informal algorithm is any effectively generated sequence of instructions, and any book-keeping purpose involving natural numbers can be absorbed into the effective generation of the instructions.) Finally we stipulate that an informal algorithm over \( \mathfrak{A} \) has a fixed number \( k \geq 1 \) of input variables, and one output variable; a computation by the informal algorithm can be carried out whenever \( k \) input values from \( \mathbb{A} \) are given, one for each of the input variables.

It is easily shown that any such informal algorithm (‘formalized algorithmic procedure’ in [9], ‘countable algorithmic procedure’ in [20]) can be translated into an effective definition by (possibly infinitely many) cases. And therefore any such informal algorithm over \( \mathfrak{A} \) can be also translated into a recursive program over \( \mathfrak{A} \) (with counters), by 1.5 above.

Further, if an informal algorithm as described above can be shown to use only finitely many variables or memory locations, then it can in fact be effectively translated into an iterative program. The convenience of working with informal algorithms is illustrated by the proofs of Lemmas 2.4, 2.6, and 4.7, and Theorem 4.9.

All the preceding facts about \textit{fed}’s and \textit{red}’s, as well as other related issues, are examined in Shepherdson’s comprehensive paper [20].

2. Algebraic characterizations of the unwind property

\textit{A loop} in an iterative program scheme \( S \) is a cycle (i.e. a simple closed path) in the finite flow-diagram of \( S \). As usual we defined a \textit{loop-free program scheme} to be an iterative program scheme with no cycles.
A loop-free program is not necessarily total, since some of its exit points may be labelled with the special instruction DIVERGE.

2.1 The unwind property. An iterative (or recursive) program $S^\mathcal{A}$ with $k \geq 1$ input variables unwinds – or equivalently, program scheme $S$ unwinds over structure $\mathcal{A}$ – if there is a $n \in \mathbb{N}$ such that for all input values $\mathbf{a} \in \mathcal{A}^k$, if $S^\mathcal{A}(\mathbf{a})$ converges then it converges in fewer than $n$ steps.

If $\mathcal{P}$ is a class of $\mathcal{A}$-programs and every program $S^\mathcal{A} \in \mathcal{P}$ unwinds, we say that $\mathcal{A}$ has the unwind property for $\mathcal{P}$. In this paper $\mathcal{P}$ will be typically the class of all iterative $\mathcal{A}$-programs or the class of all recursive $\mathcal{A}$-programs; in [13] we restrict these classes to programs that are total or whose domains are first-order definable in $\mathcal{A}$.

Two programs $S^\mathcal{A}_1$ and $S^\mathcal{A}_2$ are equivalent if $S^\mathcal{A}_1$ and $S^\mathcal{A}_2$ compute the same function over the universe $\mathcal{A}$ of $\mathcal{A}$. The proof of the next result is straightforward, and therefore omitted.

2.2 Proposition. If an iterative or recursive program $S^\mathcal{A}$ unwinds then $S^\mathcal{A}$ is equivalent to a loop-free program $S^\mathcal{A}$.

The converse of 2.2 is not generally true; that is, there is a structure $\mathcal{A}$ and a $\mathcal{A}$-program $S^\mathcal{A}$ such that $S^\mathcal{A}$ is equivalent to a loop-free $\mathcal{A}$-program, but $S^\mathcal{A}$ does not unwind. For example, it is not difficult to write a total iterative program over the structure $\mathcal{A}_0 = (\mathbb{N}; =; \text{succ}, 0)$, call it $P$, which computes the product of two natural numbers. Such a program $P$ cannot unwind over $\mathcal{A}_0$ (non-trivial verification left to the reader), nor can it therefore unwind over any expansion of $\mathcal{A}_0$. Now consider $P$ over the expansion $\mathcal{A}_1 = (\mathbb{N}; =; \times; +, \text{succ}, 0)$ of $\mathcal{A}_0$: $P$ does not unwind over $\mathcal{A}_1$, but is nonetheless equivalent to a loop-free program over $\mathcal{A}_1$, since $P$ computes one of the primitive functions of $\mathcal{A}_1$ (namely $\times$).

Forthcoming results are stated in terms of the following concepts.

2.3 Definition. (a) Structure $\mathcal{A}$ is locally finite if every non-empty finite subset of $A$ generates a finite substructure of $\mathcal{A}$.

(b) Structure $\mathcal{A}$ is uniformly locally finite if for every $k \geq 1$ there is an $n \in \mathbb{N}$ (possibly depending on $k$) such that every $k$-generated substructure of $\mathcal{A}$ has at most $n$ elements. ("$k$-generated" means finitely generated from $k$ distinct elements.)

Given similarity type $\tau = (r_1, \ldots, r_m; f_1, \ldots, f_n)$ and input variables $x = (x_1, \ldots, x_k)$, we define the set of all $x$-terms inductively as follows:

1. $x_1, x_2, \ldots, x_k$ are all $x$-terms;
2. if $f \in \{f_1, \ldots, f_n\}$ is a $j$-ary function symbol, $j \geq 0$, and $t_1, \ldots, t_j$ are $x$-terms, then so too is $f(t_1, \ldots, t_j)$.

A subterm of a term $t$ is one produced in the course of constructing $t$ inductively using rules (1) and (2) above.
We define in a natural way a function $\text{rank}$ from the set of $x$-terms to $\mathbb{N}$:

1. if $t$ is a variable or a 0-ary function symbol then $\text{rank}(t) = 0$;
2. if $t$ is of the form $f(t_1, \ldots, t_j)$, $j \geq 1$, then

$$\text{rank}(t) = \max\{\text{rank}(t_1), \ldots, \text{rank}(t_j)\} + 1.$$ 

If there is some $f$ of arity $> 0$, it is easy to check that $\text{rank}$ is onto $\mathbb{N}$.

Given an $x$-term $t$ we sometimes exhibit explicitly the variables by writing $t(x)$ or $t(x_1, \ldots, x_k)$. And if $\mathbb{A}$ is a structure with similarity type $\tau$, the interpretation of $t$ in $\mathbb{A}$ is denoted $t^\mathbb{A}$.

For convenience in the next and later proofs, we define the ‘distance’ from an arbitrary $a \in \mathbb{A}^k$ to an arbitrary $b \in \mathbb{A}$. If $b$ is not accessible from $a$, then distance $(a, b)$ is undefined, otherwise:

$$\text{distance}(a, b) = \inf\{\text{rank}(t) \mid t \text{ is a } x\text{-term and } t^\mathbb{A}(a) = b\}.$$ 

It is easy to check that if $\text{distance}(a, b) = n$ then for every $m < n$ there is an element $c \in \mathbb{A}$ such that $\text{distance}(a, c) = m$.

**2.4 Lemma.** If every recursive program over structure $\mathbb{A}$ unwinds, then $\mathbb{A}$ is uniformly locally finite.

**Proof.** Suppose $\mathbb{A}$ is not uniformly locally finite. This implies that

$$\exists k \forall n \exists a \in \mathbb{A}^k \text{ [the substructure generated by } a \text{ has more than } n \text{ elements].}$$

This in turn means there are elements $a \in \mathbb{A}^k$ and $b \in \mathbb{A}$ such that the distances from $a$ to $b$ are arbitrarily large; that is, we cannot put a uniform bound on the values of $\text{distance}(a, b)$. (Of course this is generally true only if $\mathbb{A}$ has finite similarity type, which is our standing assumption throughout the paper.)

We next observe that we can effectively enumerate all loop-free program schemes, with input variables $\{x_1, x_2, \ldots, x_k, y, \ldots, z\}$ and in the similarity type of $\mathbb{A}$—say this enumeration is: $Q_0, Q_1, \ldots, Q_i, \ldots, i \in \omega$. Further, given any $Q_i$ and any input $(a, b) \in \mathbb{A}^{k+1}$, it is decidable (relative to the ‘oracles’ we have for the primitive relations and operations of $\mathbb{A}$) which of the following two cases holds:

- $Q_i^\mathbb{A}(a, b)$ converges
- $Q_i^\mathbb{A}(a, b)$ diverges

This is not generally decidable in the presence of loops.

We now describe an informal algorithm $P$ over $\mathbb{A}$, which will turn out not to be equivalent to any loop-free program over $\mathbb{A}$. $P$ executes the following steps:

1. The input to $P$ is arbitrary $a \in \mathbb{A}^k$ and $b \in \mathbb{A}$.
2. $P$ effectively enumerates all $(x_1, \ldots, x_k)$-terms in order of non-decreasing ranks—say $t_0, t_1, \ldots, t_i, \ldots, i \in \omega$. 


All that is required in this step is a procedure (in fact an iterative program) which effectively lists all $x$-terms, using the counters of $P$ and based on some appropriate arithmetization of $x$-terms (syntactic objects).

(3) Every time a new $x$-term $t_i$ is generated in step (2), $P$ computes the value of $t_i^A(a)$ and compares it with $b$. If $t_i^A(a) \neq b$, $P$ proceeds to generate $x$-term $t_{i+1}$ in step (2); if $t_i^A(a) = b$, $P$ computes rank $(t_i)$, which is exactly $\text{distance}(a, b)$, and then goes to step (4).

(Note that all that is required to compute the rank of $t_i$ (a syntactic object) is some iterative programming on the counters of $P$. Further, if $b$ is not accessible from $a$, then $\text{distance}(a, b)$ is undefined and $P$ will never reach step (4), and therefore diverge; however, because $\mathcal{U}$ is not uniformly locally finite, for every $n \in \omega$ there are $a \in A^k$ and $b \in A$ such that $\text{distance}(a, b) = n$.)

(4) Let $\text{distance}(a, b) = n$. $P$ now generates the $n$th loop-free program scheme $Q_n^\mathcal{U}$ (a syntactic object) using its counters, and then runs $Q_n^\mathcal{U}$ on input $(a, b)$. If $Q_n^\mathcal{U}(a, b)$ converges, $P$ diverges; and if $Q_n^\mathcal{U}(a, b)$ diverges, $P$ converges – this is an effective step, since it is decidable whether $Q_n^\mathcal{U}(a, b)$ converges or not.

A simple diagonal argument shows that $P$ is not equivalent to any $Q^\mathcal{U}$, for $Q \in \{Q_i \mid i \in \omega\}$. □

To prove the counterpart of 2.4 relative to iterative programs, some additional machinery is required.

With every $x$-term $t$ we can associated a finite dag (directed acyclic graph) $G_n$ with as many input nodes as there are variables and 0-ary function symbols in $t$, and with exactly one output node labelled with the full expression for $t$. For example, if $c$ and $g$ are 0- and 2-ary function symbols, respectively, then we can represent the $(x_1, x_2)$-term $g(c, g(g(c, x_1), x_1))$ by the following dag:

Strictly speaking if nodes $u_1$ and $u_2$ are incident to node $v$ in this dag – and $u_1$, $u_2$, $v$ are associated with terms $t_1$, $t_2$, $g(t_1, t_2)$ respectively – we should label the edges $(u_1, v)$ and $(u_2, v)$ with 1 and 2, respectively, corresponding to the order of the two arguments in $g(t_1, t_2)$. For our present purposes however, we may ignore the ordering of edges incident to the same node in $G_n$.
We define the pebble complexity of $t$ as follows:

$$\text{pebble}(t) = \text{minimum number of pebbles required to pebble } G_n,$$

i.e. to reach the output node of $G_n$.

The rules of the 'pebble game' are briefly explained in Appendix 1; a more thorough exposition can be found in [18]. Note that the count for $\text{pebble}(t)$ corresponds to the number of program variables only (i.e. excluding input variables) used in the course of the computation.

We shall later consider subsets of the set of all $x$-terms according to the number of program variables used by an iterative program scheme. We thus define for each $l \geq 0$ the set of all $l$-$x$-terms as follows:

$$\{ t \mid t \text{ is a } x\text{-term and } \text{pebble}(t) \leq l \}.$$

The following are basic definitions for this paper, formulated in terms of the syntactic notions introduced above. We also repeat the definitions of 2.3 but differently.

**2.5 Definition.** (a) Structure $\mathcal{A}$ is locally finite if for all $k$ and all $a \in A^k$ the following set is finite: $\{ t^k(a) \mid t \text{ is a } x\text{-term} \}$.

(b) Structure $\mathcal{A}$ is locally finite w.r.t. (with respect to) bounded space if for all $k$, for all $l$ and for all $a \in A^k$ the following set is finite: $\{ t^k(a) \mid t \text{ is a } l-x\text{-term} \}$.

(c) Structure $\mathcal{A}$ is uniformly locally finite if for all $k$ there is a $n \in \mathbb{N}$, and for all $a \in A^k$ the following set has at most $n$ elements: $\{ t^k(a) \mid t \text{ is a } x\text{-term} \}$.

(d) Structure $\mathcal{A}$ is uniformly locally finite w.r.t. bounded space if for all $k$ and for all $l$ there is a $n \in \mathbb{N}$ and for all $a \in A^k$ the following set has at most $n$ elements: $\{ t^k(a) \mid t \text{ is a } l-x\text{-term} \}$.

In relation to an iterative program scheme $S$, we do not need to consider the set of all $x$-terms, and may restrict our attention to the subset of all $l$-$x$-terms (where $l$ is the number of program variables in $S$). Thus if $T$ is some subset of the $x$-terms, typically the set of all $l$-$x$-terms for a fixed $l$, many of our definitions will be relativized to $T$. For example, the distance from $a \in A^k$ to $b \in A$ relative to $T$, written $\text{distance}_T(a, b)$, is given by:

$$\text{distance}_T(a, b) = \inf \{ \text{rank}(t) \mid t \in T \text{ and } t^k(a) = b \},$$

and $\text{distance}_T(a, b)$ is undefined if for all $t \in T$, $t^k(a) \neq b$.

**2.6 Lemma.** If every iterative program over structure $\mathcal{A}$ unwinds, then $\mathcal{A}$ is uniformly locally finite w.r.t. bounded space.

**Proof.** If $\mathcal{A}$ is not uniformly locally finite w.r.t. bounded space then, by 2.5 part (d)

$$\exists k \exists l \forall n \exists a \in A^k \{ t^k(a) \mid t \text{ is an } l-\{x_1, \ldots, x_k\} \text{-term} \} = n.$$
Definability by programs

13

This means that there are elements $a \in A^k$ and $b \in A$ such that the distances from $a$ to $b$ (relative to $l$-$x$-terms) are arbitrarily large. If $T$ is the set of all $l$-$x$-terms, we cannot put a uniform bound on the values of $dist(T, a, b)$. As a consequence, for all $n \in \omega$ there are $a \in A^k$ and $b \in A$ such that $dist(T, a, b) = n$.

As in the proof of 2.4, we consider a fixed effective enumeration of all loop-free program schemes, with input variables $\{x_1, \ldots, x_k, x_{k+1}\}$, with no more than $l$ program variables, and in the similarity type of $\mathfrak{N}$—say $Q_0, Q_1, \ldots, Q_n, \ldots, i \in \omega$. Given any $Q_i$, any input $(a, b) \in A^{k+1}$, and exactly $l$ program variables, it is decidable (relative to the 'oracles' for the primitive relations and operations of $\mathfrak{M}$) whether $Q_i^\mathfrak{N}(a, b)$ converges or not.

The informal algorithm $P$ over $\mathfrak{N}$, to be described next, will not be equivalent to any loop-free $Q^\mathfrak{M}$ with $Q \in \{Q, Q_1, \ldots, Q_n, \ldots, i \in \omega \}$. If we can restrict $P$ to use no more than $l$ program variables, and this will require some extra care, then we conclude that there is an iterative program over $\mathfrak{N}$ with $l$ program variables which does not unwind.

$P$ carries out the following steps:

1. The input to $P$ is arbitrary $a \in A^k$ and $b \in A$.
2. $P$ effectively enumerates all $l$-$x$-terms in order of non-decreasing ranks—say $t_0, t_1, \ldots, t_i, \ldots, i \in \omega$. All that is required in this step is iterative programming on the counters of $P$, based on some appropriate arithmetization of the $l$-$x$-terms (syntactic objects).
3. Using $l$ program variables, and every time a new $l$-$x$-term $t_i$ is generated in step (2), $P$ computes the value of $t_i^\mathfrak{N}(a)$ and compares it with $b$. If $t_i^\mathfrak{N}(a) \neq b$, $P$ proceeds to generate $l$-$x$-term $t_{i+1}$ in step (2); if $t_i^\mathfrak{N}(a) = b$, $P$ computes $\text{rank}(t_i)$ which is exactly $dist(T, a, b)$ then goes to step (4).

(To compute $t_i^\mathfrak{N}(a)$, at most $l$ program variables are required. Once it is computed $t_i^\mathfrak{N}(a)$ is stored in one of the $l$ program variables, say $y_1$; then we store in one of the remaining $(l-1)$ program variables, say $y_2$, value $b$ (initially assigned to input variable $x_{k+1}$); and then we execute the test instruction '$y_1 \neq y_2$'. This shows that $P$ needs to use at least two program variables. We can assume that $l \geq 2$, because if $\mathfrak{N}$ is not uniformly locally finite w.r.t. space $= l$, then neither is it w.r.t. space $= l'$ for all $l' \geq l$.)
4. Let $dist(T, a, b) = n$. $P$ now generates the $n$th loop-free program scheme $Q_n$ (a syntactic object) using its counters. The $l$ program variables of $P$ are next used to simulate the computation of $Q_n^\mathfrak{N}$ on $(a, b)$. If $Q_n^\mathfrak{N}(a, b)$ converges, $P$ diverges; and if $Q_n^\mathfrak{N}(a, b)$ diverges, $P$ converges.

Remaining details are similar to those of 2.4. □
substructure generated by $a \in \mathbb{A}^k$ relative to $T$ is a partial substructure $\mathcal{B}$ of $\mathcal{A}$ defined as follows:

1) The universe $\mathbb{B}$ of $\mathcal{B}$ is \{$(a) \mid t(x) \in T$\}.

2) For every relation symbol $r \in \tau$, $r^\mathcal{B}$ is the restriction of $r^\mathcal{A}$ to $\mathbb{B} \subseteq \mathbb{A}$, i.e. $r^\mathcal{B} = r^\mathcal{A} \upharpoonright \mathbb{B}$.

3) For every function symbol $f \in \tau$ of arity $j \geq 0$, and every $t_1^\mathcal{A}(a), \ldots, t_j^\mathcal{A}(a) \in \mathbb{B}$:

$$f^\mathcal{B}(t_1^\mathcal{A}(a), \ldots, t_j^\mathcal{A}(a)) = \begin{cases} f^\mathcal{A}(t_1^\mathcal{A}(a), \ldots, t_j^\mathcal{A}(a)), & \text{if } f(t_1(x), \ldots, t_j(x)) \in T, \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

Let us denote the substructure $\mathcal{B}$ generated by $a$ relative to $T$ by $\text{sub}(a, T)$. Note that $\text{sub}(a, T)$ is a substructure of $\mathcal{A}$ in the usual sense, i.e., $\text{sub}(a, T)$ is not partial, if $T$ is the set of all $(x_1, \ldots, x_k)$-terms.

We now define an equivalence relation $\sim_T$ on $\mathbb{A}^k$ relative to $T$; namely for all $a, b \in \mathbb{A}^k$:

$$a \sim_T b \iff \text{sub}(a, T) \text{ isomorphic to } \text{sub}(b, T).$$

We denote the equivalence class of $a \in \mathbb{A}^k$ with respect to $\sim_T$ by $[a]_T$, so that $\mathbb{A}^k/\sim_T = \{[a]_T \mid a \in \mathbb{A}^k\}$. If $\mathcal{A}$ is locally finite and $T$ arbitrary, or if $\mathcal{A}$ is locally finite w.r.t. bounded space and $T$ a subset of $l$-$x$-terms for some $l$, then $\text{sub}(a, T)$ is finite for every $a \in \mathbb{A}^k$.

If $\mathcal{A}$ is uniformly locally finite or $T$ arbitrary, or if $\mathcal{A}$ is uniformly locally finite w.r.t. bounded space and $T$ is a subset of $l$-$x$-terms for some $l$, then there is an upper bound $n$ such that $|\text{sub}(a, T)| \leq n$ for every $a \in \mathbb{A}^k$. This means that the equivalence relation $\sim_T$ has finite index in this case – a fact we use in the next lemmas.

2.7 Lemma. If structure $\mathcal{A}$ is uniformly locally finite then every recursive program over $\mathcal{A}$ unwind.

Proof. Identical to the proof of 2.8 below, except that $S$ is a recursive program scheme and $T$ is the set of all $x$-terms.

2.8 Lemma. If structure $\mathcal{A}$ is uniformly locally finite w.r.t. bounded space then every iterative program over $\mathcal{A}$ unwind.

Proof. We consider an iterative $\mathcal{A}$-program $S^\mathcal{A}$ with input variables $x = (x_1, \ldots, x_k)$. The corresponding $\text{red}$ is:

$$\text{red}(S) = (\alpha_x(x) \mid i \in \omega),$$

where each $\alpha_x(x)$ is a finite conjunction of atomic and negated atomic formulas in variables $x = (x_1, \ldots, x_k)$. We also assume that $\text{red}(S)$ is infinite; when $\text{red}(S)$ is finite, the proof to follow becomes trivial.

* This is not a substructure in the usual sense.
The effective enumeration of \( \text{red}(S) \) corresponds to the effective enumeration of all finite paths (not necessarily consistent) in \( S \), whose endpoint is not \text{DIVERGE}, say: \( \pi_0, \pi_1, \ldots, \pi_n, \ldots, i \in \omega \). So that computation \( S^\omega(a) \) follows path \( \pi_i \) and converges iff \( \mathcal{A} \models \alpha_i[a] \). We can always assume that for \( i \neq j \), \( \alpha_i(x) \) is not consistent with \( \alpha_j(x) \), i.e. \( \exists x [\alpha_i(x) \land \alpha_j(x)] \) is a contradiction.

It is easily seen that \( S^\omega \) unwinds iff there is a \( n \in \omega \), such that for all \( a \in A^k \): if \( \mathcal{A} \models \alpha_i[a] \) then \( i < n \).

Since \( S \) is iterative, all terms appearing in \( \text{red}(S) \) are \( l \times \)-terms for some fixed \( l \); more specifically, the set

\[
U = \{ u \mid u \text{ is a } x\text{-term which appears as an argument of an atomic or negated atomic formula in } \text{red}(S) \}
\]

is a subset of the set \( T \) of all \( l \times \)-terms.

\( \mathcal{A} \) being uniformly locally finite w.r.t. bounded space, \( A^k/\sim_T \) has finitely many equivalence classes. To complete the proof it suffices to show that \( S^\omega \) unwinds when input vectors are restricted to any one of these equivalence classes \( X \in A^k/\sim_T \). This is easily verified since for all \( a, b \in X \), \( \text{sub}(a, T) \) and \( \text{sub}(b, T) \) are isomorphic finite substructures.

We can now state the main results of this section.

**2.9 Theorem.** Structure \( \mathcal{A} \) is uniformly locally finite \( \iff \) every recursive program over \( \mathcal{A} \) unwinds \( \iff \) every recursive program over \( \mathcal{A} \) with parameters unwinds.

**Proof.** The first double implication follows from 2.4 and 2.7. As for the second double implication, the right-to-left direction is obvious. For the left-to-right direction, suppose that there is a recursive program \( P \) over \( \mathcal{A} \), with input variables \( \{x_1, \ldots, x_k\} \) and parameters \( \{a_1, \ldots, a_n\} \), which does not unwind – call this program \( P(x_1, \ldots, x_k, a_1, \ldots, a_n) \). It is easy to check that program \( P(x_1, \ldots, x_k, x_{k+1}, \ldots, x_{k+n}) \) without parameters does not unwind either.

A familiar example of a uniformly locally finite structure is a distributive lattice \( \mathcal{M} = (A; \vdash; \cup, \cap) \). In particular, the structure

\[
\mathcal{M}^+ = (\mathbb{N}^+; \vdash; \text{gcd}, \text{lcm})
\]

where \( \mathbb{N}^+ = \mathbb{N} - \{0\} \), \( \text{gcd}(m, n) \) – greatest common divisor of \( m \) and \( n \), \( \text{lcm}(m, n) \) – least common multiple of \( m \) and \( n \), is a distributive lattice – and therefore has the unwind property for all recursive programs, with or without parameters. Any expansion of \( \mathcal{M}^+ \) has the same property as long as the uniform-local-finiteness of \( \mathcal{M}^+ \) is not violated; thus for instance, the following expansion of \( \mathcal{M}^+ \):

\[
(\mathbb{N}^+; \vdash; |; \text{gcd}, \text{lcm}, \text{max}, \text{min})
\]

where \( x \mid y \iff x \text{ divides } y \), has the unwind property for all recursive programs with or without parameters.
2.10 Theorem. Structure $\mathcal{A}$ is uniformly locally finite w.r.t. bounded space $\iff$ every iterative program over $\mathcal{A}$ unwinds $\iff$ every iterative program over $\mathcal{A}$ with parameters unwinds.

Proof. The first double implication is a consequence of 2.6 and 2.8. The right-to-left direction of the second double implication is obvious; as for the left-to-right direction it is proved as in 2.9. $\square$

Any structure which is uniformly locally finite is of course uniformly locally finite w.r.t. bounded space too. In the next section we give a natural example of a structure which is uniformly locally finite but only w.r.t. bounded space.

We now draw some conclusions concerning definability by programs versus definability by first-order formulas.

As usual we say that a $k$-ary predicate $X$ on $\mathcal{A}$ is parametrically definable in $\mathcal{A}$ if there is a first-order formula $\psi(x_1, \ldots, x_k, y_1, \ldots, y_n)$ and parameters $b = (b_1, \ldots, b_n) \in \mathcal{A}^n$ such that:

$$X = \{ a \mid a \in \mathcal{A}^k \text{ and } \mathcal{A} \models \psi[a, b] \}.$$

If $n = 0$ we say that $X$ is definable in $\mathcal{A}$, i.e., without parameters. A $k$-ary function $Y$ from $\mathcal{A}^k$ to $\mathcal{A}$ is (parametrically) definable in $\mathcal{A}$ if $Y$, as a $(k + 1)$-ary predicate, is (parametrically) definable in $\mathcal{A}$.

2.11 Corollary. Structure $\mathcal{A}$ is uniformly locally finite. 

(a) A predicate $X$ on $\mathcal{A}$ is computed by a recursive program over $\mathcal{A}$ $\iff$ $X$ is definable in $\mathcal{A}$ by a quantifier-free formula.

(b) A predicate $X$ on $\mathcal{A}$ is computed by a recursive program over $\mathcal{A}$ with parameters $\iff$ $X$ is parametrically definable in $\mathcal{A}$ by a quantifier-free formula.

Proof. The right-to-left direction in both (a) and (b) is true whether or not $\mathcal{A}$ is uniformly locally finite. If $\psi$ is the quantifier-free formula defining $X$, we first transform $\psi$ into a disjunction of conjuncts; i.e., $\psi \equiv \psi_1 \vee \psi_2 \vee \cdots \vee \psi_n$ where each $\psi_i$ is a conjunction of finitely many atomic and negated atomic formulas. Then we write a finite red, namely:

$$(\psi_1, \psi_2, \ldots, \psi_n)$$

which defines the same predicate over $\mathcal{A}$ as the original formula $\psi$. By 1.5 and the ensuing discussion, we can find a recursive program scheme (in fact, it will be a loop-free program scheme) equivalent to $\text{red} (\psi_1, \ldots, \psi_n)$.

For the left-to-right direction in both (a) and (b), we first unwind the recursive program scheme that computes $X$ over $\mathcal{A}$, then find the corresponding $\text{red}$ using 1.5. This will be a finite $\text{red}$, say

$$(\alpha_1, \alpha_2, \ldots, \alpha_n).$$

The desired quantifier-free formula is $\alpha = \alpha_1 \vee \cdots \vee \alpha_n$. $\square$
2.12 Corollary. Structure \( \mathcal{A} \) is uniformly locally finite w.r.t. bounded space.

(a) A predicate \( X \) on \( \mathcal{A} \) is computed by an iterative program over \( \mathcal{A} \) \( \Leftrightarrow X \) is definable in \( \mathcal{A} \) by a quantifier-free formula.

(b) A predicate \( X \) on \( \mathcal{A} \) is computed by an iterative program over \( \mathcal{A} \) with parameters \( \Leftrightarrow X \) is parametrically definable in \( \mathcal{A} \) by a quantifier-free formula.

Proof. Similar to the proof of 2.11. Details omitted. □

In the preceding two corollaries the left-to-right implications are all true if \( X \) is a function (instead of a predicate) on \( \mathcal{A} \), because any finite \( fed \) with input variables \( x = (x_1, \ldots, x_k) \), say:

\[
((\alpha_1, t_1), \ldots, (\alpha_m, t_m))
\]

computes over structure \( \mathcal{A} \) the function defined by the quantifier-free formula:

\[
(\alpha_1 \land t_1 = x_{k+1}) \lor \cdots \lor (\alpha_m \land t_m = x_{k+1}).
\]

We may assume with no loss of generality that \( \alpha_1 \) is inconsistent with \( \alpha_i \) for \( i \neq j \).

On the other hand, if \( X \) is a function, the right-to-left implications in the preceding two corollaries may or may not be true. A trivial counterexample is given by the structure \( (\mathbb{N}; =, \sigma) \), where \( \sigma \) is a binary relation on \( \mathbb{N} \) such that \( \sigma(m, n) \Leftrightarrow m + 1 = n \); then the quantifier-free formula \( \sigma(x_1, x_2) \) defines the successor function, while there is no program over \( (\mathbb{N}; =, \sigma) \) with one input variable which computes the successor function.

In Section 4 we show that the converse of 2.11 is also true; namely, if the set of predicates computable by recursive programs over structure \( \mathcal{A} \) is identical to the set of predicates definable by quantifier-free formulas in \( \mathcal{A} \), then \( \mathcal{A} \) is uniformly locally finite. Interestingly however, the examples of Section 5 will show that the converse of 2.12 is not true in general.

We close this section with a connection between the unwind property and \( \aleph_0 \)-categoricity. A first-order theory \( T \) is \( \aleph_0 \)-categorical if all models of \( T \) of cardinality \( \aleph_0 \) are isomorphic.

We denote the theory of structure \( \mathcal{A} \) by \( \text{Th}(\mathcal{A}) \), which is the set of all first-order sentences true in \( \mathcal{A} \).

2.13 Corollary. \( \mathcal{A} \) is an arbitrary structure. If \( \text{Th}(\mathcal{A}) \) is \( \aleph_0 \)-categorical, \( \mathcal{A} \) has the unwind property for all recursive programs, with or without parameters.

Proof. By 2.9 it suffices to show that \( \mathcal{A} \) is uniformly locally finite. The present proof is strictly model-theoretic, and based on the characterization of \( \aleph_0 \)-categorical theories (Theorem 2.3.13 in [5]).

Suppose there is a \( k \)-tuple \( \mathbf{a} \in \mathcal{A}^k \) such that \( \mathbf{a} \) generates an infinite substructure of \( \mathcal{A} \). There is then an infinite set of \( x \)-terms, say \( \{ t_i \mid i \in \omega \} \), such that \( \{ t_i^A(\mathbf{a}) \mid i \in \omega \} \) is also infinite. We can choose the terms in \( \{ t_i \mid i \in \omega \} \) so that \( t_i^A(\mathbf{a}) \neq t_j^A(\mathbf{a}) \) for \( i \neq j \).

Let the value of \( t_i^A(\mathbf{a}) \) be \( b_i \in \mathcal{A} \).
Now for every \( i \neq j \), the type of \((a, b_i)\) is different from the type of \((a, b_j)\), since we have:

\[
(\mathcal{A}, a, b_i) \models t_i(x_1, \ldots, x_k) \Rightarrow x_{k+1} \quad \text{and} \quad (\mathcal{A}, a, b_j) \models t_j(x_1, \ldots, x_k) \not\Rightarrow x_{k+1}
\]

while on the other hand:

\[
(\mathcal{A}, a, b_i) \models t_i(x_1, \ldots, x_k) \not\Rightarrow x_{k+1} \quad \text{and} \quad (\mathcal{A}, a, b_j) \models t_j(x_1, \ldots, x_k) \Rightarrow x_{k+1}.
\]

Hence \( \mathcal{A} \) has infinitely many types in variables \( \{x_1, \ldots, x_k, x_{k+1}\} \) and its theory cannot therefore be \( \mathcal{K}_0 \)-categorical.

Suppose now there are \( k \)-tuples \( a_0, a_1, \ldots, a_n, \ldots \in \mathbb{A}^k \) such that the substructure generated by \( a_i \) has finite cardinality \( n_i \) and that \( n_0 < n_1 < n_2 < \cdots < n_i < \cdots \). This means that the type of \( a_i \) is different from the type of \( a_j \), for \( i \neq j \). Hence \( \mathcal{A} \) has again infinitely many types in \( \{x_1, \ldots, x_k\} \), and its theory cannot be \( \mathcal{K}_0 \)-categorical.

We conclude that if \( \text{Th}(\mathcal{A}) \) is \( \mathcal{K}_0 \)-categorical, \( \mathcal{A} \) is uniformly locally finite. \( \Box \)

Many of the familiar \( \mathcal{K}_0 \)-categorical first-order theories do not include function symbols in their language (see for example [5]), and therefore models for such theories do not have primitive functions -- only primitive relations and constants. Over models of this kind we cannot 'compute' new values from given input values, and therefore all programs trivially unwind.

A familiar \( \mathcal{K}_0 \)-categorical theory, which includes a function symbol in its language, is the theory of Abelian groups with all elements of order \( p \). Such groups have therefore the unwind property for all recursive programs, with or without parameters.

Questions of 'saturation' and '\( \mathcal{K}_1 \)-categoricity' will play a much more interesting role when we study structures \( \mathcal{A} \) with the unwind property for total programs and/or for programs whose domains are first-order definable in \( \mathcal{A} \) [13].

3. A structure with the unwind property for every iterative (but not every recursive) program

The structure \( \mathcal{N} = (\mathbb{N}; =, g, 0) \) we now define is not locally finite, but is nonetheless uniformly locally finite w.r.t. bounded space. Hence this is an example of a structure for which every iterative program unwinds, but not every recursive program does, by Theorems 2.9 and 2.10. This, together with the fact that the theory \( \text{Th}(\mathcal{N}) \) of \( \mathcal{N} \) turns out to be decidable, will allow us to put in sharper focus some of the differences between definability by programs and definability by first-order formulas.

The universe \( \mathbb{N} \) of \( \mathcal{N} \) is the set of all natural numbers. The function \( g : \mathbb{N} + \mathbb{N} \to \mathbb{N} \) is defined by:

\[
g(i, j) = \begin{cases} j + 1, & \text{if } i = \lfloor j/2 \rfloor, \\ 0, & \text{otherwise.} \end{cases}
\]
The structure $\mathcal{N}$ is essentially the ‘$\omega$-chained complete binary tree’ $T$ defined in Appendix 1. The function $g$ relative to $T$ is the following map: Given two nodes $x$ and $y$, if there is a node $z$ with a plain edge from $x$ to $z$, and with a dashed edge from $y$ to $z$, then $g(x, y) = z$; otherwise $g(x, y) = 0$.

Proposition 3 in Appendix 1 shows that $\mathcal{N}$ is uniformly locally finite w.r.t. bounded space.

It is easy to see, however, that $\mathcal{N}$ is not locally finite (w.r.t. unbounded space); in fact the entire universe of $\mathcal{N}$ is generated by element 0:

$$\mathcal{N} = \{t^N(0) \mid t(x) \text{ is a } x\text{-term in similarity type of } \mathcal{N}\}.$$  

We can say more about structure $\mathcal{N}$.

3.1. Lemma. There is a recursive program over $\mathcal{N}$, call it SUCC, which computes the successor function on $\mathbb{N}$.

**Proof.** The recursive $\mathcal{N}$-program SUCC is given in Fig. 3.1. It is not difficult to verify that the recursive $\mathcal{N}$-program of Fig. 3.1 defines the successor function $\text{SUCC}: \mathbb{N} \to \mathbb{N}$. □

Since iterative programs using ‘successor’ and ‘constant 0’ (in assignments) and ‘equality’ (in tests) suffice to define all the computable functions on $\mathbb{N}$, we now have the following result.

3.2 Proposition. Every computable function on $\mathbb{N}$ is computed by some iterative program over the structure $(\mathbb{N}; \leq, \text{SUCC}, g, 0)$; put differently, every computable function on $\mathbb{N}$ is computed by some $\mathcal{N}$-program with at most one recursive subroutine.

The preceding result is not true in general; that is, there are structures $\mathcal{A}$ with universe $A = \mathbb{N}$ such that not every computable function on $\mathbb{N}$ is computed by some recursive $\mathcal{A}$-program. However, if we know that every computable function $\theta$ on $A = \mathbb{N}$ is computed by a recursive $\mathcal{A}$-program then $\theta$ is in fact computed by some $\mathcal{A}$-program with at most one recursive subroutine.

We now turn to the definability in structure $\mathcal{N}$ of functions computed by iterative and recursive $\mathcal{N}$-programs. First, we prove a lemma, which is also of independent interest.

3.3 Lemma. The theory $\text{Th}(\mathcal{N})$ of structure $\mathcal{N}$ is decidable.

**Proof.** This consists in interpreting $\text{Th}(\mathcal{N})$ into the theory of Presburger arithmetic, $\text{Th}(\mathbb{N}; \leq; +, \text{succ}, 0)$. That is, given any sentence $\sigma$ (in the first-order language of $\mathcal{N}$) we effectively find another sentence $\hat{\sigma}$ (in the first-order language of Presburger arithmetic) such that:

$$\mathcal{N} \vdash \sigma \text{ iff } (\mathbb{N}; \leq; +, \text{succ}, 0) \vdash \hat{\sigma},$$  

Since the latter is decidable, we conclude that $\text{Th}(\mathcal{N})$ is too.
To establish the interpretation of $\text{Th}(\mathbb{N})$ into $\text{Th}(\mathbb{N}; \leadsto, +, \text{succ}, 0)$, it suffices to construct a formula $\sigma_\mathcal{E}(x, y, z)$ in the language of the latter such that for all $a, b, c \in \mathbb{N}$

$$(\mathbb{N}, \leadsto, +, \text{succ}, 0) \equiv \sigma_\mathcal{E}[a, b, c] \iff g(a, b) = c.$$ 

We can rewrite $g(x, y) = z$ as follows:

$$g(x, y) = z \iff [(x + x = y) \land (z = \text{succ}(y))]$$
$$\lor [(x + x \neq \text{pred}(y)) \land (z = \text{succ}(y))]$$
$$\lor [(x + x \neq y) \land (x + x \neq \text{pred}(y)) \land (z = 0)].$$
where \( \text{pred} \) is the predecessor function, which is definable in \((\mathbb{N}; \preceq; +, \text{succ}, 0)\). Remaining details are left to the reader. \(\Box\)

It is worth noting that \(\text{Th}(\mathbb{N})\) does not admit elimination of quantifiers. Indeed, the set of even numbers is definable in \(\mathbb{N}\) by the following formula \(\varphi(y)\):

\[
\exists x [g(x, y) \neq 0 \land g(x, g(x, y)) \neq 0].
\]

\(\varphi(y)\) is not equivalent to any quantifier-free formula (proof left to the reader).

**3.4 Proposition.** (a) Every function on \(\mathbb{N}\) computed by an iterative \(\mathbb{N}\)-program is definable in \(\mathbb{N}\) by a quantifier-free formula.

(b) There are functions on \(\mathbb{N}\) computed by recursive \(\mathbb{N}\)-programs which are not definable in \(\mathbb{N}\) by any first-order formulas.

**Proof.** (a) This follows from the fact that \(\mathbb{N}\) is uniformly locally finite w.r.t. bounded space and 2.12.

(b) Suppose that every predicate computed by a recursive \(\mathbb{N}\)-program is definable in \(\mathbb{N}\) by a first-order formula – and we shall get a contradiction.

Let \(X\) be a r.e. but not decidable subset of \(\mathbb{N}\). By 3.2, \(X = \text{domain}(S^{31})\) for some recursive \(\mathbb{N}\)-program \(S^{31}\) with one input variable. Let \(\psi(x)\) be the formula which defines \(X\) in \(\mathbb{N}\), so that for all \(n \in \mathbb{N}\):

\[
\mathbb{N}, \psi[n] \text{ iff } S^{31}(n) \text{ converges.}
\]

Since \(n \in \mathbb{N}\) is accessible from 0 by finitely many applications of the function \(g\), \(\psi[n]\) can be replaced by a closed formula \(\psi_n\) in the language of \(\mathbb{N}\). Hence \(\psi_n \in \text{Th}(\mathbb{N})\) iff \(S^{31}(n)\) converges, but the former is decidable while the latter is not - a contradiction. \(\Box\)

Let us point out that not every function on \(\mathbb{N}\) definable in \(\mathbb{N}\) by a quantifier-free formula is computed by an iterative \(\mathbb{N}\)-program. For example, it is not too difficult to check that the predicate defined in \(\mathbb{N}\) by the quantifier-free formula \(\psi(x_1, x_2, y)\):

\[
\psi(x_1, x_2, y) \equiv [x_2 \neq 0 \land g(y, x_1) \equiv x_2] \lor [y_2 \equiv 0 \land y \equiv 0]
\]

is a single-valued predicate, and its domain is all of \(\mathbb{N} \times \mathbb{N}\) (we use here Rogers' terminology [19]). That is, \(\psi\) defines a total function \(f\) such that for all \(a_1, a_2, b \in \mathbb{N}\), \(\psi[a_1, a_2, b] \iff f(a_1, a_2) \sim b\). Function \(f\) is not computed by any loop-free program over \(\mathbb{N}\), nor is it computed by any iterative program over \(\mathbb{N}\). On the other hand, since the predicate defined by \(\psi\) is decidable, \(f\) is computed by some recursive program over \(\mathbb{N}\), by 3.2.

In view of the preceding results, it remains to investigate the class of functions defined in \(\mathbb{N}\) by quantifier-free formulas, as well as the class of functions defined in \(\mathbb{N}\) by first-order formulas in general (not done in this paper) – all of these functions being computed by recursive programs over \(\mathbb{N}\).
What we have now established is the following proper inclusions:

\{\text{functions computed by iterative programs over } \mathcal{N}\}\subseteq \{\text{functions definable in } \mathcal{N} \text{ by quantifier-free formulas}\}

\subseteq \{\text{functions first-order definable in } \mathcal{N}\}

\subseteq \{\text{functions computed by recursive programs over } \mathcal{N}\}.

If we denoted by \( \mathcal{N}_0 \) the structure \((\mathbb{N}; \equiv; \text{succ}, 0)\) and by \( \mathcal{N}_1 \) the standard model of arithmetic \((\mathbb{N}; \equiv; \times, +, 0, 1)\) we then have by contrast the following results:

\{\text{functions first-order definable in } \mathcal{N}_0\}\subseteq \{\text{functions computed by iterative programs over } \mathcal{N}_0\}

\subseteq \{\text{functions computed by recursive programs over } \mathcal{N}_0\}.

The above proper inclusion follows from the fact that the first-order theory of \( \mathcal{N}_0 \) is decidable, while there is an iterative program over \( \mathcal{N}_0 \) whose domain is not decidable, i.e. which computes a r.e. but not recursive subset of \( \mathbb{N} \). (Details are omitted.) On the other hand, since iterative programs over \( \mathcal{N}_0 \) suffice to define all the computable functions on \( \mathbb{N} \), iterative programs over \( \mathcal{N}_0 \) 'can compute as much as' recursive programs over \( \mathcal{N}_0 \) – thus establishing the equality above. We also have the following (known) results:

\{\text{functions definable by first-order existential formulas in } \mathcal{N}_1\}\subseteq \{\text{functions computed by iterative programs over } \mathcal{N}_1\}

\subseteq \{\text{functions computed by recursive programs over } \mathcal{N}_1\}.

4. The truth-table property

As pointed out in Section 1, programs can be viewed as effective definitions by (possibly infinitely many) cases. One natural restriction on such effective definitions is to bound memory, corresponding to the class of iterative programs. Another natural restriction is to limit such definitions to finitely many cases, corresponding to the class of loop-free programs.

Since an effective definition by finitely many cases is none other than the usual notion of a truth-table, we introduce the following definition.

4.1 Definition. Let \( \mathcal{P} \) be a class of programs over structure \( \mathcal{N} \). We say that \( \mathcal{N} \) has the \textit{truth-table property} for \( \mathcal{P} \) if every program in \( \mathcal{P} \) is equivalent to a loop-free program over \( \mathcal{N} \).
Clearly if $\mathcal{A}$ has the unwind property for a class $\mathcal{P}$ of programs then $\mathcal{A}$ has the truth-table property for $\mathcal{P}$, but not necessarily the other way around.

**4.2 Theorem.** Structure $\mathcal{A}$ has the truth-table property for all recursive programs $\iff$ $\mathcal{A}$ has the unwind property for all recursive programs.

**Proof.** The right-to-left implication is obvious. For the converse, we assume that there is a recursive $\mathcal{A}$-program which does not unwind. By 2.9 $\mathcal{A}$ is not uniformly locally finite. We can now adapt the proof of 2.4 where we showed that if $\mathcal{A}$ is not uniformly locally finite, we can construct a recursive $\mathcal{A}$-program which is not equivalent to any loop-free $\mathcal{A}$-program. □

A slightly different and somewhat longer proof of 4.2 was given in [11].

The counterpart of 4.2 relative to iterative programs is not true in general, unless the primitive relations and operations of structure $\mathcal{A}$ are assumed to be effective (Theorem 4.4).

For the next definition we identify the universe $A$ of structure $\mathcal{A}$ with the natural numbers $\mathbb{N}$, or with a recursive subset of $\mathbb{N}$.

**4.3 Definition.** Structure $\mathcal{A}$ is $\Delta_i^n$ for some $i \in \{0, 1\}$ and $n \in \omega$, if the primitive relations and operations of $\mathcal{A}$ are $\Delta_i^n$. If $\mathcal{A}$ is $\Delta_0^n = \Delta_0$, we also say that $\mathcal{A}$ is computable. If $\mathcal{A}$ is $\Delta_n^n$ for some $n \in \omega$, $\mathcal{A}$ is arithmetical. If $\mathcal{A}$ is $\Delta_1^1$, $\mathcal{A}$ is hyperarithmetical.

Since our standing assumption is that the primitive operations of a structure are always total, there is no loss of generality in ignoring the case when these operations are $\Sigma_n^n$ or $\Pi_n^n$. Indeed if $f$ is a $k$-ary operation of $\mathcal{A}$ such that the predicate $'f(x_1, \ldots, x_k) = x_{k+1}'$ is $\Sigma_n^n$, then its complement $'f(x_1, \ldots, x_k) \neq x_{k+1}' \iff (\exists y) [f(x_1, \ldots, x_k) = y \land y \neq x_{k+1}]$ is also $\Sigma_n^n$, so that it is itself $\Delta_n^n$.

**4.4 Theorem.** Let $\mathcal{A}$ be a computable structure. Then $\mathcal{A}$ has the truth-table property for all iterative programs $\iff \mathcal{A}$ has the unwind property for all iterative programs.

**Proof.** The right-to-left implication is obvious. For the converse, we prove the contrapositive statement: If there is an iterative $\mathcal{A}$-program which does not unwind, there is an iterative $\mathcal{A}$-program $P$ which is not equivalent to any loop-free $\mathcal{A}$-program.

By 2.8 $\mathcal{A}$ is not uniformly locally finite w.r.t. bounded space, and therefore

$$\exists k \forall l \forall n \exists a \in A^k |[l^A(a)]_l \text{ is a } l-(x_1, \ldots, x_k)-\text{term}] > n.$$  

If $T$ is the set of all $l-(x_1, \ldots, x_k)$-terms, this means that for all $n \in \omega$ there are $a \in A^k$ and $b \in A$ such that distance$_T(a, b) = n$ (see the proof of 2.6).
Iterative program $P$ performs the following steps (omitted details in (1) are identical to those in steps (1), (2) and (3) of 2.6):

1. Given arbitrary inputs $a \in \mathbb{A}^k$ and $b \in \mathbb{A}$, $P$ effectively enumerates all $l$-x-terms $T = \{t_i \mid i \in \omega\}$ in order of non-decreasing ranks. If and when $P$ finds a term $t_i$ such that $t_i^\wedge(a) = b$, it computes $\text{rank}(t_i) = \text{distance}_T(a, b)$ and stores this value in counter $c$.

2. $P$ enters a subroutine $\overline{P}$ which uses only counters, with $c$ being its input counter. $\overline{P}$ is chosen so as to compute some r.e. but not decidable predicate. If and when $\overline{P}$ converges, $P$ stops its operations.

As in the proof of 2.6, $P$ is an informal algorithm over $\mathbb{A}$ that operates within space = $l$. It corresponds therefore to an iterative $\mathbb{A}$-program. Further, it is easily seen that the $(k + 1)$-ary predicate on $\mathbb{A}$ computed by $P$ is $\Sigma^0_1$ but not $\Delta^0_1$. On the other hand, any loop-free $\mathbb{A}$-program computes a $\Delta^0_1$ predicate on $\mathbb{A}$, since $\mathbb{A}$ is $\Delta^0_1$. Hence, $P$ cannot be equivalent to a loop-free $\mathbb{A}$-program. □

The preceding result is best possible, in that there is a $\Delta^0_2$ structure $\mathcal{C}$ (i.e. $\mathcal{C}$ is computable relative to the diagonal set $K$) which has the truth-table property, but not the unwind property, for all iterative programs. The construction of $\mathcal{C}$ is given in Section 5.

4.5 Corollary. The following are equivalent statements:

(a) Structure $\mathbb{A}$ is uniformly locally finite;

(b) A predicate $X$ on $\mathbb{A}$ is computed by a recursive program over $\mathbb{A} \iff X$ is definable in $\mathbb{A}$ by a quantifier-free formula;

(c) A predicate $X$ on $\mathbb{A}$ is computed by a recursive program over $\mathbb{A}$ with parameters $\iff X$ is parametrically definable in $\mathbb{A}$ by a quantifier-free formula.

Proof. The equivalence of (b) and (c) is easily established and left to the reader. That (a) implies (b) follows from 2.11. It remains to show that (not a) implies (not b). But this is a straightforward consequence of 2.9. □

4.6 Corollary. Let $\mathbb{A}$ be a computable structure. Then the following are equivalent statements:

(a) Structure $\mathbb{A}$ is uniformly locally finite w.r.t. bounded space;

(b) A predicate $X$ on $\mathbb{A}$ is computed by an iterative program over $\mathbb{A} \iff X$ is definable in $\mathbb{A}$ by a quantifier-free formula;

(c) A predicate $X$ on $\mathbb{A}$ is computed by an iterative program over $\mathbb{A}$ with parameters $\iff X$ is parametrically definable in $\mathbb{A}$ by a quantifier-free formula.

Proof. This follows from 4.4 The details are similar to the proof of 4.5. □

For the reasons mentioned after 2.12 (see also the counter-example given after 3.4), the two preceding corollaries are not generally true if we substitute 'function'
for 'predicate' in (b) and (c) of each of 4.5 and 4.6. However, they are true if we also omit the right-to-left implication in (b) and (c) of each of 4.5 and 4.6 – as one may easily verify.

In our study of the unwind and truth-table properties, structures that are locally finite (but not uniformly so) have not played any other than an incidental role so far. Forthcoming results will show that the distinction between structures that are locally finite and structures that are not, is fundamental for questions of program definability.

4.7 Lemma (a) If structure \( A \) is locally finite, then it is \('decidable' \) whether or not a computation by a recursive \( A \)-program without counters converges.

(b) If structure \( A \) is locally finite \( \text{w.r.t. bounded space} \), then it is \('decidable' \) whether or not a computation by an iterative \( A \)-program without counters converges.

('Decidable' means 'decidable relative to oracles for the primitive relations and operations of \( A \)).

Proof. In spite of their similarity, (a) and (b) require different proof techniques. The proof for (a) is based on a characterization of the \( n \)-types of locally finite structures as appropriately restricted deterministic context-free sets. Because no later result depends on (a), except for 4.8 part (a), we delay its lengthy proof to a subsequent report [14].

The proof for (b), on the other hand, uses ideas introduced in this paper. We can prove in fact a stronger statement: Let structure \( A \) be locally finite \( \text{w.r.t. space} \leq l \), and \( Q \) an arbitrary iterative program scheme without counters, with input variables \( \{x_1, \ldots, x_k\} \) and \( l \) program variables. We show the existence of an iterative \( A \)-program \( P \) with input variables \( \{x_1, \ldots, x_k, 1\} \), \( l + 1 \) program variables, and a special counter \( c \) such that:

If we run \( P \) by assigning arbitrary values \( a \in \mathbb{A}^k \) to \( x \), and by storing the Gödel number of \( Q \) in counter \( c \), the corresponding computation by \( P \) will always terminate – and the final value in \( c \) will be 0 if \( Q^A(a) \) converges, 1 if \( Q^A(a) \) diverges.

Clearly, part (b) of the lemma will follow from the existence of such a program \( P \). But note in addition, that \( P \) is iterative and uses \( l + 1 \) program variables. The special counter \( c \) may be viewed as an 'input counter', where we store the Gödel number of \( Q \) (assuming a fixed arithmetization of program schemes).

Informally, \( P \) carries out the following steps:

1. The inputs to \( P \) are arbitrary \( a \in \mathbb{A}^k \) and (the Gödel number of) an arbitrary program scheme \( Q \).

2. If \( Q \) is iterative and without counters, with \( k \) input variables and at most \( l \) program variables, \( P \) computes the number \(|Q| \) of instructions in \( Q \); otherwise, \( P \) stops. All that is needed in this step is iterative programming on the counters of \( P \).
(3) $P$ starts generating all $l$-x-terms in order of non-decreasing ranks—say $t_0, t_1, \ldots, t_n, \ldots, t_i \in \omega$. Again here, all that is required is iterative programming on the counters of $P$, based on some fixed arithmetization of the set $T = \{t_i | i \in \omega\}$ of all $l$-x-terms.

(4) Simultaneously with step (3), as $P$ effectively enumerates $T$, $P$ constructs a finite subset $U \subset T$ such that $\text{sub}(a, U) = \text{sub}(a, T)$. We know such a finite subset $U$ exists since $\mathcal{A}$ is locally finite w.r.t. bounded space. When the construction of $U$ is completed $P$ goes to step (5).

The construction of $U = \{u_1, u_2, \ldots, u_n\}$ is carried out inductively and requires no more than $(l + 1)$ program variables. We first set $u_1 = t_0$. Then if we have so far selected $u_1, u_2, \ldots, u_m$ with $u_m = t_i$, we continue the generation of the $l$-x-terms $t_{i+1}, t_{i+2}, \ldots$ until we find $t_j$, with $j > i$, such that $t_i^A(a) \in \{u_1^A(a), \ldots, u_m^A(a)\}$—and we set $u_{m+1} = t_j$. To test whether $t_i^A(a) \in \{u_1^A(a), \ldots, u_m^A(a)\}$, we first compute $t_i^A(a)$ using $l$ program variables and save the resulting value, call it $b \in \mathcal{A}$, in some special program variable $y$; we then compute in turn each of $t_j^A(a), t_2^A(a), \ldots, u_m^A(a)$ using the remaining $l$ program variables and compare each in turn with the value stored in $y$.

But this is not enough. We still need a 'stopping test', which will also guarantee that the finite set $U$ constructed so far indeed satisfies $\text{sub}(a, U) = \text{sub}(a, T)$. First note that, since $T$ is generated in order of non-decreasing ranks, $\text{distance}(u, \mathcal{A}) = \text{rank}(u)$ for all $u \in U$ and $\text{rank}(u_1) \leq \text{rank}(u_2) \leq \cdots \leq \text{rank}(u_m)$. Second, if there is an element $b \in \text{sub}(a, T) - \text{sub}(a, U)$, then there is a term $t \in T - U$ of minimum rank such that $b < r^A(t)$. If in addition $\text{rank}(t) > \text{rank}(u_m)$ this means there is a subterm $\overline{t}$ of $t$ (perhaps $t$ itself) such that $\text{rank}(\overline{t}) = \text{rank}(u_m) + 1$ and $\overline{t}^A(a) \in \text{sub}(a, T) - \text{sub}(a, U)$. The required 'stopping test' is therefore the following: If we have so far selected $u_1, u_2, \ldots, u_m$ with $u_m = t_i$, we generate all $l$-x-terms $t$ after $t_i$ such that $\text{rank}(t) < \text{rank}(u_m) + 1$ (there are finitely many of them); we then select the first such $t$ for which $t_i^A(a) \notin \{u_1^A(a), \ldots, u_m^A(a)\}$—if no such $t$ satisfies this condition the construction of $U$ is completed.

(5) We now know the number $|Q|$ of instructions in $Q$, as well as the size of $\text{sub}(a, T)$ which is exactly $|U| = |\{u_1, \ldots, u_n\}| = n$. Define a state of $Q^m$ to be an instruction of $Q^m$ together with an entry from the set $\mathcal{A} \cup \{\Omega\}$ for each of its $l$ program variables, where the symbol $\Omega$ stands for 'not yet assigned a value from $\mathcal{A}$'. Clearly $Q^m$ has at most $|Q| \cdot (n + 1)^l$ states. If, during the computation $Q^m(a)$ a state is repeated, then $Q^m(a)$ is doomed to loop: i.e., if $Q^m(a)$ converges then it must be in $\leq |Q| \cdot (n + 1)^l$ steps. If $Q^m(a)$ converges $P$ stores value $0$ in counter $c$ and stops; if $Q^m(a)$ diverges $P$ stores value $1$ in $c$ and stops.

In order to simulate the computation of $Q^m$ on $a$, it is clear that $P$ needs at most $l$ program variables. □

The next result shows that if there are no elements in the universe $\mathcal{A}$ that can generate over $\mathcal{A}$ an infinite chain (this is what it means for $\mathcal{A}$ to be locally finite
but not uniformly), the presence of counters does add to the power of a programming language. A special case of (b) below is proved in [9, Lemma 1.5.4].

4.8 Proposition. (a) If structure $\mathcal{A}$ is locally finite but not uniformly, there is a recursive $\mathcal{A}$-program with counters which is not equivalent to any recursive $\mathcal{A}$-program without counters.

(b) Let structure $\mathcal{B}$ be computable. If $\mathcal{B}$ is locally finite w.r.t. bounded space but not uniformly, there is an iterative $\mathcal{B}$-program with counters which is not equivalent to any iterative $\mathcal{B}$-program without counters.

Proof. For part (a) we construct a recursive $\mathcal{A}$-program $P$ with counters which is not equivalent to any recursive $\mathcal{A}$-program without counters. The construction of $P$ here is identical to that in the proof of 2.4 – with the following difference: $\{Q_i | i \in \omega\}$ is not the set of all loop-free program schemes, but rather the set of all recursive program schemes without counters, each with $k + 1$ input variables. By 4.7 part (a), it is decidable whether or not $Q^\mathcal{A}(a, b)$ converges for any $(a, b) \in \mathcal{A}^{k+1}$.

For part (b) we construct an iterative $\mathcal{B}$-program $P$ with counters just as in the proof of 4.4. At the end we note that the predicate on $\mathcal{B}$ computed by $P$ is in $\Sigma_1^0 - \Delta_1^0$. On the other hand since the 'halting problem' of iterative $\mathcal{B}$-programs without counters is decidable, by 4.7 part (b), any iterative $\mathcal{B}$-program without counters computes a $\Delta_1^0$ predicate. The result follows.

In part (b) of 4.8 we cannot eliminate the requirement that 'structure $\mathcal{B}$ be computable'. Indeed structure $\mathcal{C}$ in Section 5 is $\Delta_1^0$, locally finite w.r.t. bounded space but not uniformly, and has the truth-table property for all iterative programs – and therefore any function computed by an iterative $\mathcal{C}$-program with counters is already computed by one without counters.

It was pointed out after 2.9 that structures such as

$$(\mathbb{N}^+; \leq, \gcd, \text{lcm})$$

or

$$(\mathbb{N}^+; \cong, |; \gcd, \text{lcm}, \text{max}, \text{min})$$

are uniformly locally finite. We can re-introduce 0 in the universe without violating the uniform-local-finiteness of these structures, provided we also extend gcd and lcm to 0 (by setting for example $\gcd(x, 0) = \gcd(0, x) = 1$ and $\text{lcm}(x, 0) = \text{lcm}(0, x) = 0$). The following expansion of the preceding structures is still locally finite, but no longer uniformly:

$$(\mathbb{N}; \cong, |; \gcd, \text{lcm}, \text{max}, \text{min}, \text{div}, \text{mod}, \text{pred})$$

where we set $(x \text{ div } 0) = 0$ and $(x \text{ mod } 0) = 0$. Over this structure, there are functions computed by programs with counters which are not computed by programs without.
We close this section by showing that if an arithmetical structure $\mathfrak{A}$ has the truth-table property for all iterative programs, then $\mathfrak{A}$ is locally finite w.r.t. bounded space. By contrast we note that we can restate 4.4 as follows: If a computable structure $\mathfrak{A}$ has the truth-table property for all iterative programs, then $\mathfrak{A}$ is uniformly locally finite w.r.t. bounded space.

This next result is best possible, in that there is a hyperarithmetical structure $\mathfrak{T}$ (defined in Section 5) which has the truth-table property for all iterative programs but is nonetheless not locally finite w.r.t. bounded space.

**4.9 Theorem.** Let structure $\mathfrak{A}$ be arithmetical. If $\mathfrak{A}$ has truth-table property for all iterative programs, $\mathfrak{A}$ is locally finite w.r.t. bounded space.

**Proof.** We prove that if $\mathfrak{A}$ is an arithmetical structure which is not locally finite w.r.t. bounded space, then there is an iterative $\mathfrak{A}$-program which is not equivalent to any loop-free $\mathfrak{A}$-program.

If $\mathfrak{A}$ is not locally finite w.r.t. bounded space, this means there exist a $k \geq 1$, a $l_0 > 1$, and elements $a = (a_1, a_2, \ldots, a_k) \in \mathbb{A}^k$ such that:

$$\forall t \exists l_0 \{ t^\wedge(a) | t \text{ is a } l\text{-x-term} \} \text{ is infinite.}$$

With no loss of generality we assume that $a_1, \ldots, a_k$ are zero-ary (constant) functions appearing among the primitive functions of $\mathfrak{A}$. Without this assumption, we would prove that there is an iterative $\mathfrak{A}$-program with parameters $a$ which is not equivalent to any loop-free $\mathfrak{A}$-program with parameters $a$ - and this result would again imply the proposition.

Since the structure $\mathfrak{A}$ is arithmetical, any function or predicate on $\mathbb{A}$ computed by some $\mathfrak{A}$-program is also arithmetical, i.e. in $\Delta^0_n$ for some $n \in \omega$. For the rest of this proof, if the predicate computed by an $\mathfrak{A}$-program $P$ is in $\Delta^0_n - \Delta^0_{n-1}$, we shall say that the degree of $P$ is $n$; that is, if $\text{domain}(P) \in \Delta^0_n - \Delta^0_{n-1}$ then $\text{degree}(P) = n$. More generally, if a predicate or a function $X$ is in $\Delta^0_n - \Delta^0_{n-1}$, we shall also say that the degree of $X$ is $n$.

Our goal is to construct a sequence of iterative $\mathfrak{A}$-programs: $P_0, P_1, \ldots, P_n, \ldots$ such that $\text{degree}(P_0) \leq \text{degree}(P_1) \leq \cdots \leq \text{degree}(P_n) \leq \cdots$. Since $\mathfrak{A}$ is arithmetical this increasing sequence must eventually stabilize. More specifically we shall show that if $P_i$ is equivalent to a loop-free $\mathfrak{A}$-program, then we can construct program $P_{i+1}$ such that $\text{degree}(P_{i+1}) \geq \text{degree}(P_i) + 1$. From this we conclude that there is a program $P_n$ in this sequence which is not equivalent to any loop-free $\mathfrak{A}$-program.

Each program $P_n$ in this sequence has one input variable, and uses $l_n + 1$ program variables. The construction to follow will be such that $l_0 \leq l_1 \leq \cdots \leq l_n \leq \cdots$, where $l_0$ is the number mentioned in the opening paragraph of the proof.

For some $l \geq l_1$, consider the infinite set

$$\{ t^\wedge(a) | t \text{ is a } l\text{-x-term} \}.$$ 

We first show that this set can be ordered, say as $h_0, h_1, \ldots, h_i, \ldots, i \in \omega$. Relative to this ordering we then prove the existence of an iterative $\mathfrak{A}$-program $\Sigma$ with one
input variable and \((l + 1)\) program variables, which defines the following function 
\(\sigma\) on \(A\):

\[
\sigma(x) = \begin{cases} 
  b_{i+1}, & \text{if } x = b_i \in \{b_i \mid i \in \omega\}, \\
  \text{undefined,} & \text{if } x \notin \{b_i \mid i \in \omega\}.
\end{cases}
\]

In other words, \(\sigma\) generates an \(\omega\)-chain whose initial point is \(b_0\).

Suppose we have an enumeration of all \(l\)-\(x\)-terms \(T = \{t_i \mid i \in \omega\}\) in some arbitrary but fixed order. We order the set \(\{b_i \mid i \in \omega\}\) according to the following inductive definition:

\[
b_i = \begin{cases} 
  t_0^A(a), & \text{if } i = 0, \\
  t_i^A(a), & \text{if } i \neq 0 \text{ and } j \text{ is the smallest index such that } t_j^A(a) \in \{b_0, b_1, \ldots, b_{i-1}\}.
\end{cases}
\]

We can also define a subenumeration \(U = \{u_i \mid i \in \omega\} \subseteq T\) such that \(u_i^A(a) \neq u_j^A(a)\) for \(i \neq j\), and \(u_i^A(a) = b_i\).

The informal algorithm corresponding to iterative program \(\Sigma\) carries out the following steps:

1. The input to \(\Sigma\) is an arbitrary \(b \in A\).
2. \(\Sigma\) starts enumerating the set \(T\) of \(l\)-\(x\)-terms in some fixed order – say \(t_0, t_1, \ldots, t_i, \ldots\), \(i \in \omega\). This step requires only iterative programming on the counters of \(\Sigma\).
3. Simultaneously with step (2), \(\Sigma\) enumerates \(U = \{u_i \mid i \in \omega\} \subseteq T\), according to the following scheme which requires \(l + 1\) program variables. First \(\Sigma\) sets \(u_0 = t_0\). Then if \(\Sigma\) has so far enumerated \(u_0, u_1, \ldots, u_m\) with \(u_m = t_i\), \(\Sigma\) continues the enumeration of the \(l\)-\(x\)-terms \(t_{i+1}, t_{i+2}, \ldots\) until it finds \(t_j\) with \(j > i\) such that \(t_j^A(a) \in \{u_0^A(a), \ldots, u_m^A(a)\}\), and sets \(u_{m+1} = t_j\). (Note that \(a \in A^k\) are parameters in \(\Sigma\).) To test whether \(t_i^A(a) \in \{u_0^A(a), \ldots, u_m^A(a)\}\), \(\Sigma\) computes \(t_i^A(a)\) using \(l\) program variables, and saves the resulting value in some variable \(y\); then \(\Sigma\) computes in turn each of \(u_0^A(a), u_1^A(a), \ldots, u_m^A(a)\) using the remaining \(l\) program variables, and compares each with the value stored in \(y\).
4. Every time a new term \(u_m \in U\) is generated in step (3), \(\Sigma\) tests whether or not \(u_m^A(a) = b\). If and when such a term \(u_m\) is found \(\Sigma\) generates \(u_{m+1}\), returns output value \(u_{m+1}^A(a)\) and stops.

This completes the description of \(\Sigma\). With function \(\sigma\) (playing the role of \(\text{successor}\)) and \(b_0 = t_0^A(a)\) (playing the role of \(\text{zero}\)), we can simulate all of \(\text{recursion theory}\) on \(\{b_i \mid i \in \omega\} \subseteq A\) using iterative programs only.

We now describe a related iterative \(\mathcal{M}\)-program \(\overline{\Sigma}\), which also uses parameters \(a \in A^k\) and has \(l + 1\) program variables. \(\overline{\Sigma}\) is essentially \(\Sigma\), except that the inputs and outputs of \(\overline{\Sigma}\) are from the set \(T\) of \(l\)-\(x\)-terms. (An element \(t\) of \(T\), a syntactic object, can indeed be used as an input or output value once we choose an appropriate arithmetization of \(l\)-\(x\)-terms; so that what is manipulated by \(\overline{\Sigma}\) is not \(t\) itself but rather its Gödel number, which is stored in a counter and not in a variable.) \(\overline{\Sigma}\) carries out the following steps:
(1) The input to $\Sigma$ is an arbitrary $l$-x-term $t$, which is stored in an ‘input’ counter.

(2) Using parameters $a \in A^k$ and $l+1$ program variables, $\Sigma$ effectively enumerates the set $U = \{u_i | i \in \omega\} \subseteq T$ (just as in step (3) of $\Sigma$).

(3) Every time a new term $u_m \in U$ is generated, $\Sigma$ tests whether or not $t = u_m$. If
and when such a term $u_m$ is found, $\Sigma$ returns output value $u_{m+1}$ (stored in some ‘output’ counter), and stops.

The function $\tilde{\sigma}$ computed by $\Sigma$ is the ‘successor’ function on the set $U = \{u_i | i \in \omega\} \subseteq T$; that is, for all $u_m \in U$, $\tilde{\sigma}(u_m) = u_{m+1}$.

We have defined programs $\Sigma$ and $\tilde{\Sigma}$, and functions $\sigma$ and $\tilde{\sigma}$, relative to any $l \geq l_0$. When we consider a specific $l_0$, we shall call the corresponding programs $\Sigma_n$ and $\tilde{\Sigma}_n$, and the functions they compute $\sigma_n$ and $\tilde{\sigma}_n$. Similarly the sets $U$ and $T$ relative to $l_0$ will be called $U_n$ and $T_n$.

It is clear that $T_0 \subseteq T_1 \subseteq \cdots \subseteq T_n \subseteq \cdots$ since $l_0 \leq l_1 \leq \cdots \leq l_n \leq \cdots$. Hence we also have $\text{sub}(a, T_0) \subseteq \text{sub}(a, T_1) \subseteq \cdots \subseteq \text{sub}(a, T_n) \subseteq \cdots$. \footnote{It can be shown that if for some $n$, $\text{sub}(a, T_n)$ is exactly the substructure of $A$ generated by $a$ relative to all $x$-terms (so that also $\text{sub}(a, T_n) = \text{sub}(a, T_{n+1}) = \cdots$) then 4.9 is true even if $A$ is not arithmetical. Hence the interesting case occurs when the nested sequence: $\text{sub}(a, T_0) \subset \text{sub}(a, T_1) \subset \cdots \subset \text{sub}(a, T_n) \subset \cdots$ does not stabilize.}

We now define iterative program $P_0$, which uses exactly $l_0 + 1$ program variables. $P_0$ requires an arithmetization of all iterative program schemes with one input variable in the similarity type of $l_0 = (U_0; =; \tilde{\sigma}_0, u_0)$ where $u_0$ is the first element in the enumeration of $U_0$ by $\tilde{\sigma}_0$. Call the set of all such iterative program schemes $\{Q_i | i \in \omega\}$. $P_0$ carries out the following steps:

(1) The input to $P_0$ is an arbitrary $b \in A$.

(2) $P_0$ effectively generates the set $U_0$. If and when $P_0$ finds an element $u_i \in U_0$ such that $b = u_i^A(a)$, $P_0$ saves index $i$ and $l_0$-x-term $u_n$, and then goes to step (3). This step requires no more than $l_0 + 1$ program variables (review steps (1), (2), and (3) in $\Sigma$ and note how the set $U$ is generated).

(3) $P_0$ simulates on its counters the computation of $Q_i^{l_0}$ on input $u_i$. In this simulation, whenever $\tilde{\sigma}$ is applied to some $u \in U_0$, $P_0$ calls iterative $A$-program $\Sigma_0$ which uses $l_0 + 1$ program variables. If and when $Q_i^{l_0}(u_i)$ converges, $P_0$ stops.

This completes the definition of $P_0$. Program $P_0$ essentially computes the diagonal set $\{u_i | Q_i^{l_0}(u_i) \text{ converges}\}$. It is easy to see that the predicate computed by $P_0$, $\text{domain}(P_0)$, is r.e. but not recursive in $U_0$ and $\tilde{\sigma}_0$. Since the function $\tilde{\sigma}_0$ generates $U_0$, it is readily verified that $\text{degree}(\tilde{\sigma}_0) = \text{degree}(U_0)$. We can now conclude that $\text{degree}(P_0) = \text{degree}(\tilde{\sigma}_0) + 1$.

If $P_0$ is not equivalent to any loop-free $A$-program, we set $P_1 = P_0$.

If $P_0$ is equivalent to a loop-free $A$-program, call it $H$, we next define $P_1$ such that $\text{degree}(P_1) \geq \text{degree}(P_0) + 1$. We first modify $H$ by introducing a special counter $c$; then by adding instruction $[c := 0]$ to every exit point, and replacing every instruction $\text{DIVERGE}$ by $[c := 1]$; and finally by adding $\text{OUTPUT } c$ after every $[c := 0]$ and $[c := 1]$. The resulting loop-free program $H$ is now total and computes a
function $\eta: \mathbb{A} \to \{0, 1\}$, such that for all $b \in \mathbb{A}$:

- $P_0(b)$ converges $\iff \eta(b)^t = 0$,
- $P_0(b)$ diverges $\iff \eta(b) = 1$.

$\eta$ is clearly decidable relative to (i.e. recursive in) the diagonal set $\{u_i | Q_{11}^\text{lo}(u_i) \text{ converges}\}$, or equivalently the set domain$(P_0)$. Clearly also $\text{degree}(\eta) = \text{degree}(P_0)$.

Let $l_H$ be the number of program variables in $\mathbb{H}$, and define $l_1 = \max\{l_H, l_0\}$. Relative to $l_1$ we have the set $T_1$ of all $l_1$-x-terms, the subset $U_1 \subseteq T_1$, and the function $\bar{\sigma}_1$ which generates $U_1$ starting from $u_0 \in U_1$. We define a total function $\bar{\eta}: U_1 \to \{0, 1\}$ as follows:

$$\bar{\eta}(u) = \begin{cases} 0, & \text{if } \eta(u^\wedge(a)) = 0, \\ 1, & \text{if } \eta(u^\wedge(a)) = 1 \end{cases}$$

for all $u \in U_1$. It is easy to check that $\text{degree}(\bar{\eta}) = \text{degree}(\eta) = \text{degree}(P_0)$.

We now define iterative program $P_1$, which uses exactly $l_1 + 1$ program variables. $P_1$ uses an arithmetization of all iterative program schemes with one input variable in the similarity type of $l_1 = (U_1; \approx; \bar{\sigma}_1, u_0, \bar{\eta})$. Call the set of all such iterative program schemes $\{Q_i | i \in \omega\}$. $P_1$ carries out the following steps:

1. The input to $P_1$ is an arbitrary $b \in \mathbb{A}$.
2. $P_1$ effectively generates the set $U_1$. If and when $P_1$ finds an element $u_i \in U_1$ such that $b \approx u_i^\wedge(a)$, $P_1$ saves index $i$ and $l_1$-x-term $u_i$, and then goes to step (3). This step requires no more than $l_1 + 1$ program variables.
3. $P_1$ simulates on its counters the computation of $Q_i^\text{H1}$ on input $u_i$. In this simulation, whenever $\bar{\sigma}_1$ is applied to some $u \in U_1$, $P_1$ calls iterative $\mathbb{H}$-program $\bar{\Sigma}_1$ which uses $l_1 + 1$ program variables; and whenever $\bar{\eta}$ is applied to $u \in U_1$, $P_1$ first evaluates $u^\wedge(a)$ using $l_1$ program variables, then saves $u^\wedge(a)$ in program variable $y$, and finally calls loop-free $\mathbb{H}$-program $\bar{\hat{H}}$ which uses $y$ as input variable and $l_H \leq l_1$ program variables. If and when $Q_i^\text{H1}(u_i)$ converges, $P_1$ stops.

It is easy to see that domain$(P_1)$ and the diagonal set $\{u_i | Q_i^\text{H1}(u_i) \text{ converges}\}$ are of the same degree. Further, since $\bar{\sigma}_1$ effectively generates $U_1$ (relative to the oracles for the primitive operations of $\mathbb{H}$), $\text{degree}(U_1) = \text{degree}(\bar{\sigma}_1)$. Because domain$(P_1)$ is r.e. but not recursive in $U_1$, $\bar{\sigma}_1$ and $\bar{\eta}$, we have that $\text{degree}(P_1) = \max\{\text{degree}(\bar{\sigma}_1), \text{degree}(\bar{\eta})\} + 1$. Hence $\text{degree}(P_1) > \text{degree}(P_0)$, as desired.

The construction of $P_{i+1}$ from $P_i$ is identical to the construction of $P_1$ from $P_0$. \qed

5. Structures with the truth-table property (but not the unwind property) for every iterative program

In this section we prove the following results.

5.1 Theorem. There exists a hyperarithmetical structure $\mathfrak{D}$ such that

(a) Every iterative program over $\mathfrak{D}$ is equivalent to a loop-free program over $\mathfrak{D}$;
(b) \( \mathcal{D} \) is not locally finite w.r.t. bounded space, and therefore there are iterative programs over \( \mathcal{D} \) which do not unwind (by 2.10).

We can strengthen (b) by replacing ‘iterative programs’ by ‘iterative programs without counters’.

5.2 Theorem. There exists a \( \Delta^0_2 \) structure \( \mathcal{E} \) such that:

(a) Every iterative program over \( \mathcal{E} \) is equivalent to a loop-free program over \( \mathcal{E} \);
(b) \( \mathcal{E} \) is locally finite w.r.t. bounded space, but not uniformly, and therefore there are iterative programs over \( \mathcal{E} \) which do not unwind (by 2.10).

The initial motivation to construct structures \( \mathcal{D} \) and \( \mathcal{E} \) was to settle a problem raised by J. Tiuryn [21] and P. Urzyczyn [23]. It was known that if a structure \( \mathcal{A} \) has the truth-table property for all recursive programs, then \( \mathcal{A} \) also has the unwind property for all recursive programs (Theorem 4.2). What are other classes of programs for which the truth-table property implies the unwind property? In particular, does the implication hold for the class of all iterative programs? The existence of structures \( \mathcal{D} \) and \( \mathcal{E} \) shows that the implication fails for the class of iterative programs. In [13] we show that the implication also fails for the class of all total iterative programs.

We start with the construction of \( \mathcal{D} \) because it is easier than that of \( \mathcal{E} \). The construction of \( \mathcal{D} \) involves an encoding of the full arithmetical hierarchy, such that no iterative program over \( \mathcal{D} \) can access more than a bounded initial segment of the hierarchy, the size of the segment being determined by the storage available to the program. Furthermore, using a little more storage than any particular iterative program has, the entire computation of that program can be reduced to a single question of an ‘oracle’ which the given program is unable to consult.

We start with a structure \( \mathcal{D}_0 \) of the form:

\[
\mathcal{D}_0 = (\mathcal{D}; \prec; \text{succ}, g, 0)
\]

where \( \mathcal{D} = \{n_i | i, n \in \omega\} \cup \{\Omega\} \), \( \Omega \) being a special element to be identified with ‘undefined’; \( 0 \) is \( \Omega_0 \); \( \text{succ} \) is the successor function on the subset \( \{0, 1, \ldots, n, \ldots\} \subset \mathcal{D} \); and \( g \) is the binary function defined in Section 3 on the subset \( \{n, n_1, \ldots, n_i, \ldots\} \subset \mathcal{D} \) for each \( n \in \omega \). More precisely, \( \text{succ}(\Omega) = \Omega \) and for all \( n \):

\[
\text{succ}(n_i) = \begin{cases} 
(n + 1)_0, & \text{if } i = 0, \\
\Omega, & \text{if } i > 1.
\end{cases}
\]

For all \( a \in \mathcal{D} \), \( g(a, \Omega) = g(\Omega, a) = \Omega \), and for all \( m, n \) :

\[
g(m, n_i) = \begin{cases} 
n_{i+1}, & \text{if } m = n \text{ and } i = \lfloor j/2 \rfloor, \\
n_0, & \text{if } m = n \text{ and } i - j \neq 0, \\
\Omega, & \text{otherwise}.
\end{cases}
\]
Note that the second line in this definition of $g$ is slightly different from that in Section 3; now the value of $g(n_i, n_j)$ is $n_0$ only if $i \neq j \neq 0$.

Structure $\mathfrak{D}_0$ may be represented by the diagram shown in Fig. 5.1. It is useful to think of $\mathfrak{D}_0$ as consisting of one copy of the natural numbers (namely $\{0_0, 1_0, \ldots, n_0, \ldots\}$ which, for convenience, we shall also denote $\{0, 1, \ldots, n, \ldots\}$ without subscript 0) equipped with the successor function, together with a separate copy of the ‘$\omega$-chained complete binary tree’ attached to each element in $\{0, 1, \ldots, n, \ldots\}$. In Fig. 5.1 all edges describing mappings to $\Omega$ are omitted.

By the results of Section 3 and Proposition 3 in Appendix 1, the reduct $(\mathfrak{D}; \simeq; g, 0)$ of $\mathfrak{D}_0$ is uniformly locally finite w.r.t. bounded space. Hence every iterative program over $(\mathfrak{D}; \simeq; g, 0)$ unwinds, by 2.10. On the other hand, iterative programs over $(\mathfrak{D}; \simeq; \text{succ}, 0)$ are powerful enough to define all the computable number-theoretic functions.

![Fig. 5.1. Structure $\mathfrak{D}_0$.](image)

But structure $\mathfrak{D}_0$ itself is not locally finite w.r.t. bounded space. Hence $\mathfrak{D}_0$, or any expansion of $\mathfrak{D}_0$, does not have the unwind property for every iterative program.

We now introduce an intermediary structure, denoted $(\mathfrak{D}_0, F)$, which is an expansion of $\mathfrak{D}_0$ defined above:

$$(\mathfrak{D}_0, F) = (\mathfrak{D}; \simeq; \text{succ}, g, 0, F).$$

$F$ is a unary function on $\mathfrak{D}$ which we define by assuming a fixed enumeration of all unary arithmetical functions: $f_0, f_1, \ldots, f_n, \ldots$. In the presence of the special element $\Omega$ ('undefined') we can view each $f_i$ as a total unary function on $\{0, 1, \ldots, n, \ldots, \Omega\}$. We set $F(\Omega) = \Omega$, and for all $n_i \in \mathfrak{D}$:

$$F(n_i) = \begin{cases} f_i(n), & \text{if } f_i(n) \text{ is defined,} \\ \Omega, & \text{otherwise.} \end{cases}$$

$F$ enumerates all the arithmetical functions, and is therefore hyperarithmetical. This also makes the structure $(\mathfrak{D}_0, F)$, and its expansion $\mathfrak{D}$ below, hyperarithmetical.
5.3 Lemma. Let \( P \) be an iterative program over \((\Sigma_0, F)\), with \( k \geq 1 \) input variables and \( l \geq 1 \) program variables. If the inputs to \( P \) are restricted to \( \mathbb{N} = \{0, 1, \ldots, n, \ldots\} \subseteq \mathbb{D} \), \( P \) computes a function which is algorithmic relative to the first \( q \) arithmetical functions \( f_0, f_1, \ldots, f_{q-1} \), where \( q = 3 \cdot 2^{(l-1)} - 1 \).

**Proof.** Using Proposition 3 part (1) of Appendix 1, it is easy to see that \( P \) with only \( l \) program variables has access to only the ‘oracles’ for \( f_0, f_1, \ldots, f_{q-1} \) where \( q = 3 \cdot 2^{(l-1)} - 1 \). \( \square \)

Let \( \tau : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) be some fixed pairing function, as defined in Section 5.3 of [19]. We extend \( \tau \) to \( \mathbb{D} \) by letting \( \tau(x, y) = \Omega \) for all \( x \in \mathbb{D} - \mathbb{N} \) and/or \( y \in \mathbb{D} - \mathbb{N} \).

Let \( \pi_1 \) and \( \pi_2 \) be fixed projection functions from \( \mathbb{N} \) to \( \mathbb{N} \) such that for all \( m, n \in \mathbb{N} : \pi_1(\tau(m, n)) = m \) and \( \pi_2(\tau(m, n)) = n \). We extend \( \pi_1 \) and \( \pi_2 \) to \( \mathbb{D} \) by letting \( \pi_1(x) = \Omega \) and \( \pi_2(x) = \Omega \) for all \( x \in \mathbb{D} - \mathbb{N} \).

Since \( \tau, \pi_1, \) and \( \pi_2 \) can be computed by iterative programs over \((\mathbb{N}; =; \text{succ}, 0)\), it is an easy exercise to check that they can also be computed by iterative programs over \( \Sigma_0 \). This also means that any function on \( \mathbb{D} \) computed by an iterative program over

\[ \Sigma = (\Sigma_0, F, \tau, \pi_1, \pi_2) = (\mathbb{D}; =; \text{succ}, g, 0, F, \tau, \pi_1, \pi_2) \]

is already computed by an iterative program over \((\Sigma_0, F)\).

5.4 Lemma. Any iterative program \( P \) over \( \Sigma \) is equivalent to a loop-free program over \( \Sigma \) – when inputs are restricted to \( \mathbb{N} \subseteq \mathbb{D} \).

**Proof.** By the remark preceding the lemma, we may assume that \( P \) is a program over \((\Sigma_0, F)\). Let \( P \) have \( k \geq 1 \) input variables and \( l \geq 1 \) program variables. We assume throughout this proof that all inputs are from the subset \( \mathbb{N} \subseteq \mathbb{D} \), so that with \( l \) program variables only no value \( n_i \in \mathbb{D} \) for \( i \geq q \) is ever computed, where \( q = 3 \cdot 2^{(l-1)} - 1 \).

We first transform \( P \) into an equivalent iterative program \( P' \) over \( \Sigma \). To obtain \( P' \), every exit point of \( P \) labelled with ‘output \( z \)’ is extended by two loop-free programs \( Q_1 \) and \( Q_2 \) – as shown in Fig. 5.2. With no loss of generality we assume that \( P \) has only one exit point, and we write \( P' \) as \([P; Q_1; Q_2]\).

We next define each of \( Q_1 \) and \( Q_2 \). By Proposition 3 in Appendix 1, there are \( l \times \)-terms \( t_0(x) \), \ldots, \( t_{q-1}(x) \) involving function symbol \( g \) only such that for all \( n \in \mathbb{N} \), \( t_0(n) = n, t_0'(n) = n_1, \ldots, t_{q-1}'(n) = n_{q-1} \). Program \( Q_1 \) has the form as shown in Fig. 5.3. The output of \([P; Q_1]\) is always in \( \mathbb{N} \subseteq \mathbb{D} \). What \( Q_1 \) does is to code the output \( \tau P \) as a value in \( \mathbb{N} \subseteq \mathbb{D} \). The task of \( Q_2 \) is to decode this value and restore the original output of \( P \), see Fig. 5.4.

Now consider the program \([P; Q_1]\). It computes a ‘partial \( t_f(f_0, f_1, \ldots, f_{q-1}) \)-recursive’ function \( \theta \) from \( \mathbb{N}^k \) to \( \mathbb{N} \cup \{\Omega\} \), by the preceding lemma. This function is arithmetical, say \( \theta = \lambda x_1 \cdots x_k [f_{i_p}(\tau(x_1, \ldots, x_k))] \) for some \( p \in \mathbb{N} \).
Definability by programs

(For convenience we write \( \tau(x_1, \ldots, x_k) \) instead of \( \tau(\tau(\cdots(\tau(x_1, x_2), x_3)\cdots), x_k) \).)

Note that \( \theta \) can be undefined in two different ways. \( \theta \) may be undefined at a certain input because the corresponding computation by \([P; Q_1]\) does not halt; \( \theta \) may also be undefined because one of the primitive functions in \([P; Q_1]\) has returned value \( \Omega \) in the course of the computation, and the final value of \( \theta \) must also be \( \Omega \). In order to find a loop-free program equivalent to \([P; Q_1]\), we must distinguish between these two different ways of being 'undefined'—in the first the program does not half, in the second it does.

It is worth stressing that '\( \theta \) is partial (\( f_0, f_1, \ldots, f_{q-1} \))-recursive' here is taken in the sense '\( \theta \) as a relation is recursively enumerable in \( f_0, f_1, \ldots, f_{q-1} \)', and not in the sense '\( \theta \) is partial recursive in \( f_0, f_1, \ldots, f_{q-1} \)' (see discussion in Sections 9.2 and 9.7 of [19]).

There is an arithmetical two-valued function, say \( f_r \), which solves the halting problem for \([P; Q_1]\); that is, for all \( x_1, x_2, \ldots, x_k, f_r(\tau(x_1, \ldots, x_k)) = 0 \) if \([P; Q_1]\) on input \((x_1, \ldots, x_k)\) halts, and \( = 1 \) otherwise. This is so because \([P; Q_1]\) is an algorithm relative to \( \{f_0, \ldots, f_{q-1}\} \), and the diagonal set \( K^{f_0, f_1, \ldots, f_{q-1}} \) of all such algorithms is recursively enumerable in \( \{f_0, f_1, \ldots, f_{q-1}\} \).

Let \( l \) be the smallest integer such that \( 3 \cdot 2^{(l-1)} - 1 > \max(p, r) \). By Proposition 3 in Appendix 1, there are \( \ell \)-x-terms \( t_p(x) \) and \( t_r(x) \) such that \( t_p^n(n) \equiv n_p \) and
$t^D_r(n) \approx n$, for all $n \in \mathbb{N}$. We can now write a loop-free program $\hat{P}$ over $(\Sigma_0, F, \tau)$ which is equivalent to $[P; Q_1]$ namely:

$\hat{P}$: input $x_1, x_2, \ldots, x_k$

\begin{align*}
y &:= \tau(x_1, \ldots, x_k) \\
F(t^D_r(y)) &\approx 0 \\
\hat{\xi} &:= F(t^D_r(y))
\end{align*}
It is easily checked that program $[\hat{P} ; Q_2]$ over $(\mathfrak{D}_0, F, \tau, \pi_1, \pi_2)$ which is loop-free, is equivalent to the original program $P$. □

5.5 Lemma. Any iterative program $P$ over $\mathfrak{D}$ is equivalent to a loop-free program over $\mathfrak{D}$ – when inputs are restricted to $\{n_i | n \in \omega, i < q \} \subseteq \mathfrak{D}$ for some $q \in \omega$.

Proof (outlined). The technique of coding and decoding outputs, in the preceding proof, can be applied to inputs. So that given any iterative program $P$ over $(\mathfrak{D}_0, F)$, we can replace it by an equivalent iterative program $[Q_1 ; Q_2 ; P]$ over $(\mathfrak{D}_0, F, \tau, \pi_1, \pi_2)$, where $Q_1$ is a loop-free program that codes arbitrary inputs from $\{n_i | n \in \omega, i < q \}$ into inputs (for $[Q_2 ; P]$) from $\{n | n \in \omega \}$. By 5.4, there is a loop-free program $\hat{P}$ over $(\mathfrak{D}_0, F, \tau, \pi_1, \pi_2)$ equivalent to $[Q_2 ; P]$. The desired loop-free program equivalent to the original $P$ is $[Q_1 ; \hat{P}]$. □
The proof of part (a) in Theorem 5.1 consists in showing that 5.4 and 5.5 still hold when no restriction is placed on inputs. This requires a certain amount of technical manipulation and is delayed to Appendix 2.

We now turn to the construction of structure \( \mathcal{C} \). In what follows we use the notation \( \mathcal{A} \subseteq \mathcal{B} \) to mean that \( \mathcal{A} \) is a reduct of \( \mathcal{B} \) or \( \mathcal{B} \) is an expansion of \( \mathcal{A} \).

We define a countably infinite sequence of structures \( \mathcal{N}_n, n \in \omega \), such that \( \mathcal{N}_n \subseteq \mathcal{N}_{n+1} \). Structure \( \mathcal{N}_0 \) will be the smallest expansion containing each of the structures in \( \{ \mathcal{N}_n | n \in \omega \} \).

We set \( \mathcal{N}_0 = (\mathbb{N} \cup \{\Omega\}; \Rightarrow; \text{succ}, 0, \Omega) \), where \( \text{succ} \) is the successor function on \( \mathbb{N} \) and \( \text{succ}(\Omega) = \Omega \). Let \( \{P_i | i \in \omega \} \) be the set of all iterative programs over \( \mathcal{N}_0 \), with one input variable, and denote by \( \{\psi_i | i \in \omega \} \) the set of functions they compute. The restriction of the \( \psi_i \)'s to \( \mathbb{N} \) is exactly the set of all (unary) computable functions on \( \mathbb{N} \). Again here, the element \( \Omega \) is to be identified with 'undefined'.

We define the completion \( \bar{\psi} \) of a function \( \psi \) as follows:

\[
\bar{\psi}(x) = \begin{cases} 
\psi(x), & \text{if } \psi(x) \text{ is defined}, \\
\Omega, & \text{if } \psi(x) \text{ is undefined}.
\end{cases}
\]

While the halting function \( \psi^h \) of \( \psi \) is defined as:

\[
\psi^h(x) = \begin{cases} 
0, & \text{if } \psi(x) = \Omega, \\
1, & \text{if } \psi(x) \text{ is defined and } \psi(x) \neq \Omega, \\
\Omega, & \text{if } \psi(x) \text{ is undefined}.
\end{cases}
\]

Note that \( \psi^h \) is the completion of some function in \( \{\psi_i | i \in \omega \} \).

The expansion \( \mathcal{N}_1 \) is obtained from \( \mathcal{N}_0 \) by adding all the completions \( \bar{\psi} \)'s to the primitive functions:

\[
\mathcal{N}_1 = (\mathcal{N}_0, \bar{\psi}_0, \bar{\psi}_1, \bar{\psi}_2, \ldots).
\]

It is easy to see that every iterative program \( P \) over \( \mathcal{N}_0 \) is equivalent to a loop-free program \( Q \) over \( \mathcal{N}_1 \): namely, if \( P \) computes \( \psi : \mathbb{N} \cup \{\Omega\} \rightarrow \mathbb{N} \cup \{\Omega\} \), and \( \bar{\psi} \) and \( \psi^h \) are the completion and halting function of \( \psi \), program \( Q \) is:

![Diagram](image-url)
Assuming that $M_n$ has been defined, we construct the expansion $M_{n+1} = M_n$ as follows. $\{P_i | i \in \omega\}$ is the set of all iterative programs over $M_n$, with exactly one input variable $x$, each $P_i$ satisfying the following condition:

If $\theta$ is one of the primitive functions of $M_n$ other than $\text{suc}$, $P_i$ contains no assignment instruction of the form $[y := \theta(y')]$ unless $y'$ is in the finite set $\Sigma(x) \subseteq \mathbb{N}$,  

where $\Sigma(\Omega) = \emptyset$ and for all $x \in \mathbb{N}$:

$$\Sigma(x) = \{\tau(a_0, \ldots, a_k) | k \geq 0, a_0, \ldots, a_k \leq x, \& a_i \neq a_j \forall 0 \leq i < j \leq k\}$$

and $\tau: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is a fixed pairing function. (We write $\tau(a_0, \ldots, a_k)$ for $\tau (\cdot \cdot (\tau(a_0, a_1), a_2) \cdot \cdot, a_k)$, and let $\tau(a_0) = a_0$.)

Let $\{\psi_i | i \in \omega\}$ be the functions computed by the iterative programs over $M_n$ satisfying condition $(*)$, and $\{\tilde{\psi}_i | i \in \omega\}$ be their completions. Structure $M_{n+1}$ is:

$$M_{n+1} = (M_n, \tilde{\psi}_0, \tilde{\psi}_1, \tilde{\psi}_2, \ldots)$$

5.6 Lemma. (1) Given an iterative program $P$ with one input variable over $M_n$, there is a loop-free program $Q$ over $M_{n+1}$ which is equivalent to $P$.

(2) Every structure in $\{M_n | n \in \omega\}$ is $\Delta^0_2$.

Proof. (1) This was already proved above for the case $n = 0$. For the case $n \geq 1$, the proof is similar.

(2) Structure $M_0$ is clearly $\Delta^0_2$, i.e. it is a computable structure. It is easy to see that the primitive functions of $M_1$ are all computable relative to the ‘halting problem’ (the diagonal set $K$), and therefore $M_1$ is $\Delta^0_2$.

Assume now that all the primitive functions of $M_n$ are computable relative to $K$. We want to show that the same is true of $M_{n+1}$. First, it is easy to check that the function $\Sigma$ is total computable on $\mathbb{N} \cup \{\Omega\}$, since it is defined in terms of the pairing function $\tau$. Second, a primitive function of $M_{n+1}$ is the completion $\tilde{\psi}$ of a function $\psi$ computed by an iterative program $P$ over $M_n$; such an iterative program $P$ uses finitely many primitive functions $\theta$ of $M_n$ other than $\text{suc}$. Suppose for simplicity that $P$ uses only one such function $\theta$, which is assumed to be $\Delta^0_2$ by the induction hypothesis. The following procedure shows that $\tilde{\psi}$ is a (total) computable function on $\mathbb{N} \cup \{\Omega\}$ provided we have an oracle for $K$:

(a) Given input $x$, compute the finite set $\Sigma(x)$ — say $\Sigma(x) = \{x_1, x_2, \ldots, x_i\}$:

(b) Using oracle for $K$, set variable $y_1$ to $\theta(x_1)$, $y_2$ to $\theta(x_2)$, \ldots, $y_i$ to $\theta(x_i)$;

(c) Run program $\tilde{P}$ on input $x$, where $\tilde{P}$ is obtained from $P$ by replacing every instruction of the form $[y := \theta(y')]$ by the sequence:
Note that program $\hat{P}$ in part (c) of the procedure is a program over $\mathcal{R}_0$, and its halting problem is decidable relative to $K$. \qed

**Remark.** What we have shown above is that $\psi$ is 'weak truth-table reducible' to $\theta$ via the function $\Sigma$ (see [19, p. 158]), and since $\theta$ is computable relative to $K$, so is $\psi$ relative to $K$.

The smallest expansion containing every structure in the chain:

$$\mathcal{R}_0 \subseteq \mathcal{R}_1 \subseteq \cdots \subseteq \mathcal{R}_n \subseteq \cdots$$

is $\mathcal{R}_\omega$. Let $\theta_0, \theta_1, \ldots$ be an effective enumeration of all the primitive functions in $\mathcal{R}_\omega$ and $\Theta: \mathbb{N} \times (\mathbb{N} \cup \{\Omega\}) \rightarrow \mathbb{N} \cup \{\Omega\}$ be a universal function for the $\theta$'s; that is, for all $i \in \mathbb{N}$ and $x \in \mathbb{N} \cup \{\Omega\}$:

$$\Theta(i, x) = \theta_i(x).$$

To make $\Theta$ total on $\mathbb{N} \cup \{\Omega\}$, we set $\Theta(\Omega, x) = \Omega$ for all $x$. The following is an immediate result.

**5.7 Lemma.** The universal function $\theta$ is $\Delta^0_1$.

It is worth noting that the functions on $\mathbb{N} \cup \{\Omega\}$ computable by iterative programs over $(\mathbb{N} \cup \{\Omega\}; \vdash; \text{succ}, 0, \Omega, \theta)$ strictly include those computed by iterative programs over $\mathcal{R}_\omega$.

A first attempt to define the structure $C$ is to consider $(\mathbb{Q}; \vdash; \text{pred}, g, 0, \theta)$, where $\theta$ can be redefined appropriately to be a unary function (as we did with the function $F$ in the case of structure $\Sigma$). The resulting structure $C$ would not however have the truth-table property for iterative programs, because $C$ would lack a coding and decoding capability which is necessary in order to prove the counterparts of 5.4 and 5.5. We need therefore to build into $C$ this coding and decoding capability, without violating the local finiteness requirement.
Structure \( \mathcal{E} \) will be of the form \( \mathcal{E} = (\mathcal{E}_0, \Theta) \), which is locally finite w.r.t. bounded space and has the truth-table property (but not the unwind property) for all iterative programs. The reduct \( \mathcal{E}_0 \) of \( (\mathcal{E}_0, \Theta) \) is of the form:

\[
\mathcal{E}_0 = (\mathcal{E}; \Rightarrow; \text{pred}, \text{pair}, \text{proj}, g)
\]

Before defining the universe \( \mathcal{E} \), and then the behavior of the primitive functions on \( \mathcal{E} \), we take a look at a subset \( \bar{\mathcal{E}} \subset \mathcal{E} \). \( \bar{\mathcal{E}} \) is the set of all non-empty finite sequences of natural numbers, all numbers in any such sequence being distinct:

\[
\bar{\mathcal{E}} = \{(a_0, \ldots, a_k) \mid k \geq 0; a_0, \ldots, a_k \in \mathbb{N} \& a_i \neq a_j \text{ for } 0 \leq i < j \leq k\}.
\]

We can partition \( \bar{\mathcal{E}} \) into \( \bigcup \{ \bar{\mathcal{E}}_k \mid k \geq 0 \} \) where

\[
\bar{\mathcal{E}}_k = \bigcap_{k+1} \mathbb{N} \times \cdots \times \mathbb{N}
\]

is the set of all sequences of length \( k + 1 \). We identify \( \mathbb{N} \) with the subset \( \bar{\mathcal{E}}_0 \).

The primitive function \( \text{pred} \) is only defined on the subset \( \mathbb{N} = \bar{\mathcal{E}}_0 \subset \mathcal{E} \); that is, for all \( x \in \mathbb{E} \), \( \text{pred}(x) = \Omega \) unless \( x \in \mathbb{N} \), in which case

\[
\text{pred}(x) = \begin{cases} 
0, & \text{if } x = 0, \\
1, & \text{if } x \geq 1,
\end{cases}
\]

where \( \Omega \) ('undefined') is again a special element in \( \mathcal{E} \).

The primitive function \( \text{pair} \), which is only defined on the subset \( \bar{\mathcal{E}} \subset \mathcal{E} \), is a special 'pairing function'. It is restricted in a way that makes \( \mathcal{E}_0 \) locally finite. For all \( x, y \in \bar{\mathcal{E}} \), \( \text{pair}(x, y) = \Omega \) unless: \( x, y \in \bar{\mathcal{E}} \) and there is a \( k \geq 0 \) such that \( x(a_0, a_1, \ldots, a_k) \), \( y = (a_1, a_2, \ldots, a_{k+1}) \), and \( a_0 \neq a_{k+1} \), in which case

\[
\text{pair}(x, y) = \text{pair}(y, x) = (a_0, a_1, \ldots, a_k, a_{k+1}).
\]

Also, for all \( x \in \bar{\mathcal{E}} \) we set \( \text{pair}(x, \Omega) = \text{pair}(\Omega, x) = x \), which simplifies somewhat the constructions in the proofs of 5.11 and 5.12 below.

To define the behavior of primitive functions \( \text{proj} \) and \( g \), we need to explain how the rest of \( \mathcal{E} \) is constructed. The universe \( \mathcal{E} \) is obtained from \( \bar{\mathcal{E}} \) by attaching a copy of the '\( \omega \)-chained complete binary tree' to each element in \( \bar{\mathcal{E}} \). More specifically,

\[
\mathcal{E} = \{ x_i \mid x \in \bar{\mathcal{E}} \text{ and } i \in \omega \} \cup \{ \Omega \},
\]

where we identify \( \bar{\mathcal{E}} \) with all the elements of \( \mathcal{E} \) indexed with 0. For each \( x \in \bar{\mathcal{E}} \), the behavior of function \( g \) on \( \{ x, x_1, x_2, \ldots, x_n, \ldots \} \) is identical to that of function \( g \) on \( \mathbb{N} \) as defined in Section 3, otherwise \( g \) is undefined; that is, for all \( x, y_i \in \mathcal{E} - \{ \Omega \} \):

\[
g(x, y_i) = \begin{cases} 
y_{i+1}, & \text{if } x = y \text{ and } i = \lfloor j/2 \rfloor, \\
y_0, & \text{if } x = y \text{ and } i = j \neq 0, \\
\Omega, & \text{otherwise.}
\end{cases}
\]

And for all \( a \in \mathcal{E} \), \( g(a, \Omega) = g(\Omega, a) = \Omega \).
The primitive \( \text{proj} \) is a special 'projection function' which operates on two arguments. Given a natural number \( i \) and a \( k \)-tuple \( y = (a_0, a_1, \ldots, a_{k-1}) \in \mathbb{E} \) we want \( \text{proj}(i, y) \) to be the \( (i + 1) \)st component of \( y \). But we also want to restrict \( \text{proj} \) in such a way that, if space is bounded, then \( \text{proj} \) can only be applied to tuples of bounded length. To this end, we distinguish a copy of the \( \omega \)-chained complete binary tree in \( \mathbb{E} \), namely that copy whose root is 0, i.e. the subset \( \{0, 0_1, 0_2, \ldots, 0_n, \ldots\} \). For all \( x, y \in \mathbb{E} \) we now define \( \text{proj}(x, y) = \Omega \) unless: \( x = 0 \) and \( y = (a_0, a_1, \ldots, a_{k-1}) \in \mathbb{E} \) for some \( 0 \leq i < k \), in which case

\[
\text{proj}(0, (a_0, a_1, \ldots, a_{k-1})) = a_i.
\]

Note that an iterative program \( P \) over \( \mathcal{E}_0 \) cannot generate more than a finite subset of elements in \( \{0, 0_1, 0_2, \ldots, 0_n, \ldots\} \), and the number of such elements which \( P \) can generate depends only on the number \( l \) of program variables used by \( P \), by Proposition 3 in Appendix 1. This means that within space \( \leq l \) and by means of the function \( \text{proj} \), we can only 'decompose' tuples in \( \mathbb{E} \) whose length is uniformly bounded by a function of \( l \).

5.8 Lemma. The structure \((\mathcal{E}; =; \text{pred}, \text{pair}, \text{proj}, g)\) is locally finite w.r.t. bounded space.

Proof. Consider the natural numbers in their usual order: 0, 1, \ldots, \( n \), \ldots. Now apply \( \text{pair} \) to every pair of consecutive numbers, to obtain elements: \( (0, 1), (1, 2), (2, 3), \ldots, (n, n + 1), \ldots \). Then apply \( \text{pair} \) again to every pair of consecutive elements in the preceding sequence, to obtain elements: \( (0, 1, 2), (1, 2, 3), (2, 3, 4), \ldots, (n, n + 1, n + 2), \ldots \). And so on. What we generate in this process is essentially the 'infinite rectangular grid' \( G \) described in Appendix 1. Upward-going edges in \( G \) describe the behavior of \( \text{pair} \), whereas leftward-going edges along level 0 of \( G \) describe the behavior of \( \text{pred} \) (see Fig. 2 of Appendix 1).

Consider again the natural numbers, in some arbitrary order: \( a_0, a_1, a_2, \ldots, a_n, \ldots \). Applying \( \text{pair} \) to every pair of consecutive numbers in this sequence, we obtain: \( (a_0, a_1), (a_1, a_2), (a_2, a_3), \ldots, (a_n, a_{n+1}), \ldots \). Applying again \( \text{pair} \) to every pair of consecutive elements in the preceding sequence, we obtain next: \( (a_0, a_1, a_2), (a_1, a_2, a_3), \ldots, (a_n, a_{n+1}, a_{n+2}), \ldots \). Continuing this process indefinitely, we generate another copy of the infinite rectangular grid \( G \), except that now the elements of level 0 (namely \( a_0, a_1, \ldots, a_n, \ldots \)) are not linearly stringed from right-to-left.

Using (1) of Proposition 2 in Appendix 1, it is easily seen that \((\mathcal{E}; =; \text{pred}, \text{pair})\) is locally finite. Introducing primitive functions \( \text{proj} \) and \( g \), the resulting structure \( \mathcal{E} \) is locally finite w.r.t. bounded space, by Proposition 3 in Appendix 1. \( \square \)

The proof of the following result is straightforward.

5.9 Lemma. The reduct \((\mathcal{E}; =; \text{pair}, \text{proj}, g)\) obtained from \( \mathcal{E}_0 \) by omitting \( \text{pred} \) is uniformly locally finite w.r.t. bounded space.
We are now ready to define structure $\mathcal{C} = (\mathcal{E}, \hat{\Theta})$. We assume we have a fixed enumeration of the primitive functions of $\mathcal{Y}$: $\theta_0, \theta_1, \theta_2, \ldots, \theta_0, \ldots$, as defined before Lemma 5.7. The function $\hat{\Theta}$ here is obtained from the function $\Theta$ of 5.7 – by turning the latter into a unary function, and restricting it in a way that preserves the 'local finiteness' of $\mathcal{E}_0$. We set $\hat{\Theta}(\Omega) = \Omega$, and for all $\langle a_0, \ldots, a_k \rangle_i \in \mathcal{E}$:

$$\hat{\Theta}(\langle a_0, \ldots, a_k \rangle_i) = \begin{cases} 
\theta_i(\tau(a_0, \ldots, a_k)), & \text{if } \theta_i(\tau(a_0, \ldots, a_k)) \neq \Omega \& \theta_i(\tau(a_0, \ldots, a_k)) \leq \max\{a_0, \ldots, a_k\}, \\
\Omega, & \text{otherwise}.
\end{cases}$$

This definition makes sense because $\mathbb{N} = \mathbb{E}_0$ is a subset of $\mathcal{E}$.

There is a subtle point worth stressing about the definition of $\hat{\Theta}$. Not only does it preserve the local finiteness of $\mathcal{E}_0$, it also forces an iterative program $P$ over $\mathcal{C}$ to (essentially) compute a function in $\{\theta_i | i \in \omega\}$. Had we simply defined $\hat{\Theta}(\langle a_0, \ldots, a_k \rangle_i) = \theta_i(\tau(a_0, \ldots, a_k))$, such a program $P$ would not necessarily satisfy condition (1) given before 5.6; that is, $P$ could possibly apply a function $\theta \in \{\theta_i | i \in \omega\}$ to values outside the set $\Sigma(x)$ – assuming for a moment that the input $x$ in $P$ is from $\mathbb{N} = \mathbb{E}_0$ – in which case the function computed by $P$ would not be in $\{\theta_i | i \in \omega\}$.

Let us also point out that the presence of $\text{pred}$ among the primitive functions of $\mathcal{C}$ is not essential, because $\text{pred}$ is already in the set $\{\theta_i | i \in \omega\}$. We have included $\text{pred}$ for reasons of clarity and to make explicit the non-uniform local finiteness of $\mathcal{C}$. (The same remark applies to the presence of $\text{succ}$ among the primitives of $\mathcal{C}$, because $\text{succ}$ is one of the arithmetical functions $\{f_i | i \in \omega\}$.)

Our definition of $\hat{\Theta}$ makes $\mathcal{C}$ locally finite w.r.t. bounded space; in fact the reduct of $\mathcal{C}$ obtained by omitting the primitive function $g$ is locally finite. Further since $\hat{\Theta}$ is $\Delta^0_2$ and $\mathcal{E}_0$ is computable, $\mathcal{C} = (\mathcal{E}_0, \hat{\Theta})$ is itself $\Delta^0_2$.

It remains to show that every iterative program over $\mathcal{C}$ is equivalent to a loop-free program over $\mathcal{C}$.

The three next lemmata are the counterparts of 5.3, 5.4, and 5.5 relative to structure $\mathcal{C}$.

**5.10 Lemma.** Let $P$ be an iterative program over $\mathcal{C}$ with $k \geq 1$ input variables and $l \geq 1$ program variables. If the inputs to $P$ are restricted to $\mathbb{N} = \mathbb{E}_0 \subseteq \mathcal{E}$, $P$ defines a function on $\mathbb{N}$ algorithmic relative to $\theta_0, \theta_1, \ldots, \theta_{q-1}$, where $q = 3 \cdot 2^{(l-1)} - 1$.

**5.11 Lemma.** An iterative program $P$ over $\mathcal{C}$ is equivalent to a loop-free program over $\mathcal{C}$, when inputs are restricted to: (a) $\mathbb{N} = \mathbb{E}_0 \subseteq \mathcal{E}$ or (b) $\mathbb{E}_0 \cup \cdots \cup \mathbb{E}_{r-1} \subseteq \mathcal{E}$ for some $r \in \omega$.

**Proof** (outlined). Let $P$ be an iterative program over $\mathcal{C}$ with $k \geq 1$ input variables and $l \geq 1$ program variables.

(a) Inputs of $P$ are from $\mathbb{N} = \mathbb{E}_0$. The crucial fact is that, with this restriction, $P$ cannot compute values outside the following set:

$$\{n_i | n \in \mathbb{E}_0, i < q\} \cup \{n_i | n \in \mathbb{E}_1, i < q\} \cup \cdots \cup \{n_i | n \in \mathbb{E}_{t-1}, i < q\},$$
where \( q = 3 \cdot 2^{l-1} - 1 \). This fact is an immediate consequence of Proposition 2 part (3), and Proposition 3 part (1), in Appendix 1.

Hence if the inputs of \( P \) are from \( \mathcal{E}_0 \), the output of \( P \) is a value \( n_i \in \mathcal{E} \) such that \( n \in \mathcal{E}_0 \cup \cdots \cup \mathcal{E}_{l-1} \) and \( i < q \). Considering the behavior of \( g \) on the \( \omega \)-chained complete binary tree (see Section 3), there are \( l \)-x-terms \( t_0(x), \ldots, t_{q-1}(x) \) involving function symbol \( g \) only, such that

\[
t_0^E(n) \approx n, \quad t_1^E(n) \approx n_1, \ldots, \quad t_{q-1}^E(n) \approx n_{q-1}.
\]

Further if

\[
m_0 := \text{proj}(0, n) \\
m_1 := \text{proj}(0_1, n) \\
\vdots \\
m_{l-1} := \text{proj}(0_{l-1}, n)
\]

then there is a \( l-(y_1, \ldots, y_l) \)-term \( t(y_1, \ldots, y_l) \), namely:

![Diagram](image)

such that \( l^t(m_o, \ldots, m_{l-1}) \approx n \). That \( \text{pebble}(t) \) is indeed \( l \) is a consequence of Lemma 1 in Appendix 1. Note that \( m_0, m_1, \ldots, m_{l-1} \in \mathcal{E}_0 \cup \{\Omega\} \), and if there is some \( m_i = \Omega \) then \( m_i \approx \Omega \) for all \( i < j < l \).

Hence no matter what the output value \( n_i \) of \( P \) is, there is a \( l \)-x-term \( t_i \in \{t_0, \ldots, t_{l-1}\} \) and a \( l-(y_1, \ldots, y_l) \)-term \( t \) such that:

\[
t^t(t^E(m_0, m_1, \ldots, m_{l-1})) \approx n_i,
\]

where \( m_0, \ldots, m_{l-1} \in \mathcal{E}_0 \cup \{\Omega\} \).

Next we construct iterative programs over \( \mathcal{E} \): \( P_0, P_1, \ldots, P_{l-1} \), and \( P_0 \) each with \( k \) input variables, such that if \( P(a_1, \ldots, a_k) \) converges and returns output \( n_i \), for
some \( a = (a_1, \ldots, a_k) \in \overline{\mathbb{E}}_0 \), then:

\[
P_0(a) \text{ converges and returns output } m_0 \text{ (in variable } z_0, \text{ say)},
\]
\[
\ldots
\]
\[
P_{i-1}(a) \text{ converges and returns output } m_{i-1} \text{ (in variable } z_{i-1}),
\]
\[
P_i(a) \text{ converges and returns output } i \text{ (in counter } c).
\]

The construction of \( P_0, \ldots, P_{i-1}, \) and \( P_i \) is similar to the construction of program \([P; Q_1]\) in the proof of 5.4 (details omitted). Hence, when inputs are restricted to \( \overline{\mathbb{E}}_0, \) \( P \) is equivalent to – schematically:

When inputs are restricted to \( \mathbb{N} = \overline{\mathbb{E}}_0, \) each of the programs in \([P_0, \ldots, P_{i-1}]\) computes a function from \( \mathbb{N}^k \) to \( \mathbb{N} \cup \{0\}, \) and \( P_i \) a function from \( \mathbb{N}^k \) to \( \{0, 1, \ldots, q-1\} \) – and all these functions are ‘partial \((\theta_0, \theta_1, \ldots, \theta_{q-1})\)-recursive’ by 5.10.

It remains to replace each program \( P_i \in [P_0, \ldots, P_{i-1}] \) by an equivalent loop-free program \( \hat{P}_i \) when inputs are restricted to \( \mathbb{N} = \overline{\mathbb{E}}_0. \) We distinguish several cases, the first being when all \( k \) inputs to \( P_i \) are distinct. As in the proof of 5.4 where we replaced \([P; Q_1]\) by an equivalent loop-free program, \( P_i \) computes function \( \lambda x_1 \cdots x_k [\theta_p(\tau(x_1, \ldots, x_k))] \) for some \( p \in \omega; \) and the ‘halting problem’ for \( P_i \) is solved by function \( \lambda x_1 \cdots x_k [\theta_r(\tau(x_1, \ldots, x_k))] \) for some \( r \in \omega. \) Given \( k \) distinct inputs from \( \mathbb{N}, \) say \( a_0, a_1, \ldots, a_{k-1}, \) loop-free program \( \hat{P}_i \) encodes them into a single value \( \langle a_0, a_1, \ldots, a_{k-1} \rangle \) using the pairing function pair; then \( \hat{P}_i \) tests whether \( \theta_r(\langle a_0, \ldots, a_{k-1} \rangle) = 0 \) or 1 – if 0, \( \hat{P}_i \) computes \( \theta_0(\langle a_0, \ldots, a_{k-1} \rangle) \), and if 1, \( \hat{P}_i \) diverges.

The other cases to be considered are when two or more of the inputs to \( P_i \) are equal. For example, if all inputs stored in \( x_1, \ldots, x_{k-1} \) are distinct, but the inputs in \( x_{k-1} \) and \( x_k \) are equal, then \( P_i \) computes function \( \lambda x_1 \cdots x_{k-1} [\theta(s(\tau(x_1, \ldots, x_{k-1}))) \text{ for some } s \in \omega; \) and the ‘halting problem’ for \( P_i \) is solved by function \( \lambda x_1 \cdots x_{k-1} [\theta_i(\tau(x_1, \ldots, x_{k-1}))] \) for some \( i \in \omega. \)
We have to consider each of these finitely many cases separately, because over \( \mathcal{C} \) we cannot encode \( k \)-tuples in \( \mathbb{N}^k \) which have two or more equal components.

(b) Inputs of \( P \) are from \( \mathbb{E}_0 \cup \cdots \cup \mathbb{E}_{r-1} \). With no loss of generality consider the case when \( P \) has one input variable \( x \) and \( k = 1 \). We replace \( P \) by a program of the form \([Q_1; Q_2; P]\) where \( Q_1 \) is:

\[
\text{input } x
\]

\[
y_1 := \text{proj}(0, x); y_2 := \text{proj}(0, x); \ldots ; y_r := \text{proj}(0_{r-1}, x)
\]

and \( Q_2 \) is:

\[
\text{input } y_1, \ldots , y_r
\]

\[
\hat{x} := t^F(y_1, \ldots , y_r)
\]

and every occurrence of \( x \) in \( P \) is replaced by \( \hat{x} \). The term \( t(y_1, \ldots , y_r) \) in \( Q_2 \) is identical to the term \( t(y_1, \ldots , y_l) \) described in part (a), when \( r = l \). It is easy to see that when inputs are restricted to \( \mathbb{E}_0 \cup \cdots \cup \mathbb{E}_{r-1} \), the inputs of \([Q_2; P]\) are all in \( \mathbb{E}_0 \cup \{\Omega\} \). Using the technique of part (a), we can replace \([Q_2; P]\) by an equivalent loop-free program - when inputs are all in \( \mathbb{E}_0 \cup \{\Omega\} \).

5.12 Lemma. An iterative program \( P \) over \( \mathcal{C} \) is equivalent to a loop-free program over \( \mathcal{C} \) - when inputs are restricted to:

\[
\{n_i | n \in \mathbb{E}_0 \cup \cdots \cup \mathbb{E}_{r-1} \text{ and } i < q\}
\]

for some \( q, r \in \omega \).

Proof. The technique of coding and decoding values not in \( \mathbb{E}_0 \cup \{\Omega\} \), outlined in the preceding proof, can be used here again. Tedious details are omitted. See also how the proof technique of 5.4 is adapted in 5.5.

To complete the proof of 5.2, we have to show that 5.12 is true even when no restriction is placed on inputs. This requires further technical machinery and is delayed to Appendix 2.
Appendix 1: The pebble game on infinite dags

Some of our results depend on certain properties of the pebble game on infinite dags (directed acyclic graphs).

We consider first one of Cook's results concerning the pebbling of particular finite dags, called pyramids. Pyramids are finite fragments of rectangular grids having \( k \) input nodes, 1 output node, and a total of \( k(k + 1)/2 \) nodes. A pyramid of size 15 is shown in Fig. 1.

![Fig. 1. The pyramid of size 15, with 5 input nodes.](image)

The pebble game on a finite dag \( \mathcal{D} \) is defined as follows. At any point in the game, some nodes of \( \mathcal{D} \) will have pebbles on them (one pebble per node), while the remaining nodes will not. A configuration is a subset of the nodes, comprising just those nodes that have pebbles on them. As usual a legal move consists of placing a pebble \( \pi \) on a node \( v \) such that all the nodes incident to \( v \) have each a pebble already on it; pebble \( \pi \) is not necessarily a fresh pebble, i.e., it may be a pebble we remove from some node, including possibly from one of the nodes incident to \( v \). Note that since an input node \( u \) has no nodes incident to it, a pebble may be placed on \( u \) at any time.

If the result of such a move is to go from a configuration \( C \) to a configuration \( C' \), we say that \( C \) moves to \( C' \) — and we write \( C \leadsto C' \). A calculation is a sequence of configurations, each successive pair of which forms a legal move.

The pebble game on a finite dag has usually been considered to study questions of time–space trade-offs. (‘Time’ corresponds to the number of moves in a calculation, and ‘space’ to the maximum number of nodes in any configuration in this calculation.) For our purposes we may define the aim of the pebble game on a finite dag \( \mathcal{D} \) as follows: If \( \mathcal{D} \) has one output node \( w \), we want to determine a lower
bound on the number of pebbles required by a calculation that begins with the empty configuration and ends with a configuration containing $w$.

1 Lemma. Starting with the empty configuration, $k$ pebbles are required to reach the output node of a pyramid with $k(k+1)/2$ nodes.

Proof. Restating the lemma, we can say that if $w$ is the output node of the pyramid of size $k(k+1)/2$ and $\mathcal{C} = (\emptyset, C_1, C_2, \ldots, C_n)$ is a calculation such that $w \in C_n$, then:

$$\sup \{|C| | C \in \mathcal{C}\} \geq k,$$

where $|C|$ is the number of nodes in configuration $C$. This is proved by Cook in [3], in a different context with a slightly different terminology. □

Before going on to the pebble game on infinite dags, recall that placing a pebble on a node corresponds to executing an assignment instruction in a program and storing the result in a variable (or memory location); while removing a pebble from a node corresponds to freeing a variable (or memory location) so that it can be re-used to store another value.

When $\mathcal{G}$ is an infinite dag, which may or may not have input and/or output nodes, we modify the pebble game as follows. We are first given a selection of $k \geq 0$ nodes in $\mathcal{G}$, called initial nodes, and we want to determine the nodes of $\mathcal{G}$ accessible from the initial nodes using no more than a fixed supply of $l \geq 0$ pebbles. Although an initial node may have in-degree $\neq 0$, we can place a pebble on it at any time in the course of the game. Further, we cannot place a pebble on a node with in-degree $= 0$ unless it is also an initial node (note the difference with the pebble game on a finite dag). The rules for placing a pebble on a node which is not initial are identical to the rules of the game on a finite dag.

Reminder. Whenever we say that node $v$ is 'accessible' from a set of $k$ initial nodes, we mean it in the sense of the pebble game, and not in the graph-theoretic sense that there is a path from one of the initial nodes to $v$.

We consider two infinite dags, $\mathcal{G}$ and $\mathcal{T}$. We shall call $\mathcal{G}$ the 'infinite rectangular grid', and $\mathcal{T}$ the 'ω-chained complete binary tree'. $\mathcal{G}$ and $\mathcal{T}$ are best described by the diagrams of Figs. 2 and 3.

If $v$ is a node on level $n \geq 1$ in $\mathcal{G}$, $v$ has exactly two nodes incident to it, $u_1$ and $u_2$ on level $n-1$. If $v$ is labelled with the sequence of non-negative integers $(m, m+1, \ldots, m+n)$, then $u_1$ and $u_2$ are labelled with the sequences $(m, m+1, \ldots, m+n-1)$ and $(m+1, m+2, \ldots, m+n)$ respectively.

If $v$ is a node on level $0$ in $\mathcal{G}$, $v$ has exactly one node $u$ incident to it, also on level $0$. In this case the labels of $v$ and $u$ are two consecutive non-negative integers. Note that all nodes in $\mathcal{G}$ have in-degree $= 1$ (on level $0$) or in-degree $= 2$ (on level $n - 1$).
The infinite dag $\mathcal{T}$ has two distinct intertwined 'structures': the structure of an infinite complete binary tree (the plain edges in Fig. 3), and the structure of a $\omega$-chain (the dashed edges). All the nodes of $\mathcal{T}$, except for 0, have in-degree $= 2$.

**2 Proposition.** Let $X = \{x_1, \ldots, x_k\}$ be a selection of $k$ initial nodes in the infinite rectangular grid, $G$, $k \geq 0$.

1. If the $k$ nodes in $X$ are: $(m_1, \ldots, m_1 + n_1), \ldots, (m_k, \ldots, m_k + n_k)$, the number of nodes accessible from $X$ does not exceed $K(K+1)/2$ where $K = \max\{m_1 + n_1, \ldots, m_k + n_k\}$.

2. If none of the $k$ initial nodes is on level 0 on $G$, then the number of nodes accessible from $X$ does not exceed $k(k + 1)/2$.

3. If all of the $k$ initial nodes are on level 0 of $G$, and the number of available pebbles is $l \geq 1$, then all nodes accessible from $X$ are on \{level 0, level 1, \ldots, level $(l - 1)$\}.
Note that the upper bounds in (1) and (2) above do not depend on the number \( l \) of available pebbles.

**Proof.** For ease of exposition we adopt the following terminology. The line joining all nodes on level 0 of \( G \) will be called the ‘base’ of \( G \); the line joining all the nodes in the set \( \{ (i, i + 1, \ldots, i + j) | j \in \mathbb{N} \} \) will be called the \( i \)th ‘right diagonal’ of \( G \); and the line joining all the nodes in \( \{ (i), (i - 1, i), \ldots, (0, 1, \ldots, i - 1, i) \} \) will be called the \( i \)th ‘left diagonal’ of \( G \). A right diagonal is infinite, while a left diagonal is finite.

If \( v \) is a node in \( G \), let \( P(v) \) denote the (infinite) set of ancestors of \( v \):

\[
P(v) = \{ u \in G | \text{there is a path from } u \text{ to } v \}.
\]

\( P(v) \) is a pyramid whose top node is \( v \), together with that portion of the base of \( G \) which stops at the right diagonal containing \( v \), as shown in Fig. 4. An easy (but important) fact about \( P(v) \) is this: If \( v \) is accessible from a finite selection \( X \) of initial nodes, all infinite paths containing \( v \) must intersect \( X \).

It is now easy to check that part (1) of the proposition is a consequence of the following consideration: No node to the right of the rightmost left-diagonal inter-
setting $X$ is accessible from $X$. ($X$ being finite there are finitely many left-diagonals intersecting $X$.) If $K = \max\{m_1 + n_1, \ldots, m_k + n_k\}$, then the rightmost left-diagonal intersecting $X$ is none other than the $K$th left-diagonal.

To prove part (2) of the proposition, a more subtle argument is required. Since we now assume that none of the $k$-initial nodes is on the base of $G$ – and therefore no pebble may be placed on the base either in the course of a calculation – the argument to follow does not depend on the presence of a base in $G$ (and in fact applies to a more general rectangular grid extending to infinity in all directions). Let us then assume, in the rest of the present proof, that $G$ extends downward to infinity.

Let $v$ be a node in $G$, and $P(v)$ the (now infinite) pyramid whose top node is $v$. We call a connected line across $P(v)$ – made up of segments of vertical and horizontal lines, left and right diagonals – a ‘cross-section’ of $P(v)$. Every rectilinear segment of such a cross-section joins two nodes of $P(v)$; further, a cross-section does not contain cycles (i.e., it does not cross itself). To avoid undue notational details, a few examples will make precise the notion of a cross-section – see Fig. 5. With no fear of confusion, we identify a cross-section with all the nodes it contains.

It is not difficult to see that if $X$ is a finite set of initial nodes, then $v$ is accessible from $X$ if and only if $P(v) \cap X$ contains a cross-section of $P(v)$.

More generally now, call a finite set of nodes of $G$ a ‘cross-section’ if it is a cross-section of some pyramid $P(v)$ for some node $v$. Given an arbitrary cross-section $\gamma$, denote by $S(\gamma)$ the set of nodes accessible from $\gamma$. If $P(v)$ is the pyramid of which $\gamma$ is a cross-section, it is easy to see that $S(\gamma)$ consists of all the nodes of $P(v)$ that are on or ‘above’ $\gamma$. Using elementary geometry, if $\gamma$ has $k$ nodes then the size $|S(\gamma)|$ of $S(\gamma)$ does not exceed $k(k+1)/2$ (assuming no upper bound on the number of pebbles available).
Let \( X \) be an arbitrary set of \( k \) initial nodes in \( G \). Note that a node \( v \) is accessible from \( X \) iff \( v \) is accessible from some cross-section contained in \( X \). We call a cross-section \( \gamma \subseteq X \) maximal in \( X \) if, given any other cross-section \( \bar{\gamma} \subseteq X \), \( S(\gamma) \not\subseteq S(\bar{\gamma}) \); that is, there is no other cross-section \( \bar{\gamma} \) in \( X \) from which all the nodes of \( S(\gamma) \) are accessible.

We now have two easily established facts. First, if a node \( v \) is accessible from \( X \) then \( v \) is in fact accessible from some cross-section which is maximal in \( X \). Second, if \( \gamma \) and \( \bar{\gamma} \) are two distinct cross-sections which are maximal in \( X \), then \( \gamma \cap \bar{\gamma} = \emptyset \) (see Fig. 6) – note however that this does not necessarily mean that \( S(\gamma) \cap S(\bar{\gamma}) = \emptyset \) (Fig. 7).

Let \( \{\gamma_1, \ldots, \gamma_n\} \) be the set of all cross-sections maximal in \( X = \{x_1, \ldots, x_k\} \). Let \( k_i \) be the number of nodes in \( \gamma_i \), \( 1 \leq i \leq n \). Clearly \( \gamma_1 \cup \cdots \cup \gamma_n \subseteq X \) and \( k_1 + \cdots + \)

---

\[ P(v) \]

\[ \bar{\gamma} \]

---

**Fig. 5.** Three different cross-sections of \( P(v) \). A cross-section \( \gamma \) divides \( P(v) \) into two disjoint parts – the part 'above' \( \gamma \) is finite, the part 'under' \( \gamma \) is infinite.

**Fig. 6.** If two cross-sections \( \gamma \) and \( \bar{\gamma} \) overlap, then the following set of nodes form a cross-section:

\[ \gamma \cap P(v) \cup \{x : x \text{ is on } \gamma \text{ or on } \bar{\gamma} \text{ and under } \gamma \} \]
Definability by programs

Fig. 7. Two maximal cross-sections \( \gamma \) and \( \tilde{\gamma} \) such that \( S(\gamma) \cap S(\tilde{\gamma}) \neq \emptyset \).

\( k_n \approx k \). Moreover the set of nodes accessible from \( X \) is precisely \( S(\gamma_1) \cup \cdots \cup S(\gamma_n) \), and its size does not exceed:

\[ [k_1(k_1+1)/2] + \cdots + [k_n(k_n+1)/2] \leq k(k+1)/2. \]

This upper bound is tight in that there is a set \( X = \{x_1, \ldots, x_k\} \) (namely when \( X \) is a straight horizontal cross-section) from which exactly \( k(k+1)/2 \) nodes are accessible.

Part (3) of the proposition follows from Lemma 1. We omit the straightforward details.

In the \( \omega \)-chained complete binary tree \( \mathbb{T} \), node 0 is the only one with in-degree = 0. A moment of thought will show that any node in \( \mathbb{T} \) is accessible from node 0 - if sufficiently many pebbles are available. By contrast with the preceding proposition, the upper bounds established below will therefore depend on the number \( l \) of pebbles used.

3 Proposition. Consider the \( \omega \)-chained complete binary tree \( \mathbb{T} \), and let \( l \geq 1 \) be the number of available pebbles.

(1) The nodes accessible from \( X = \{0\} \) are exactly

\[ \{0, 1, 2, \ldots, [3 \cdot 2^{l-1} - 2]\}. \]

(2) The number of nodes accessible from an arbitrary selection of initial nodes \( X \) does not exceed

\[ [7 \cdot 2^{l-1} - (l+3)]. \]

Proof. Let \( \mathbb{U} \) be the subtree of \( \mathbb{T} \) whose root is some node \( u \); i.e. the nodes of \( \mathbb{U} \) are defined by:

(i) \( u \in \mathbb{U} \);

(ii) if \( v \in \mathbb{U} \) then \((2v+1) \in \mathbb{U}\) and \((2v+2) \in \mathbb{U}\).

Suppose \( X \cap \mathbb{U} = \{u\} \). We claim that if the rightmost node \( w \) on level \( (n-1) \) of \( \mathbb{U} \) is accessible from \( X \), then the number of available pebbles is \( \leq n \). To prove this claim we embed a pyramid with \( l \) input nodes into \( \mathbb{U} \), as shown in Fig. 8.
Fig. 8. Embedding a pyramid with $n = 5$ input nodes, and a total of $n(n+1)/2 = 15$ nodes, into the subtree $U$ of $T$.

In Fig. 8 the boldface edges are those of the pyramid. Some edges of the pyramid (the straight ones) are joining nodes which are adjacent in $U$; others (the curved ones) are joining nodes which are not adjacent in $U$. However, in all cases, if node $v$ is incident to node $v'$ in the pyramid, there is a path from $v$ to $v'$ in $U$. This kind of embedding shows that the pebble game on $U$ (to reach node $w$ on level $n - 1$) cannot be 'easier' than the pebble game on the pyramid with $n$ input nodes; so that the number of pebbles required to reach $w$ cannot be smaller than the number of pebbles required to reach the output node (also $w$) of the embedded pyramid. This number is $n$, by Lemma 1, proving the claim.

If we now let $u = 0$, so that $U = T$, and $X = \{0\}$, we conclude that with $l$ pebbles we cannot reach the rightmost node on level $l$ (i.e. node $2^{l+1} - 2$), and therefore none of the nodes above level $l$ either. Hence, with $l$ pebbles, the nodes accessible from $X = \{0\}$ are all included in $\{0, 1, 2, \ldots, 2^{l+1} - 2\}$.

Are there in fact nodes on level $l$ which can be reached using only $l$ pebbles? A more careful analysis shows that the farthest node accessible from $X = \{0\}$ using $l$ pebbles is node $3 \cdot 2^{l-1} - 2$. We do not include this analysis here because the
results in the main part of the paper do not depend on it. This concludes the proof of part (1).

Turning to part (2) of the proposition, we assume that $X = \{u_1, \ldots, u_k\} \subseteq T$. We have shown above that if $U$ is the tree whose root is $u \in X$ and $X \cap U = \{u\}$, then less than the first $\mu(l)$ nodes of $U$ are accessible from $X$, for some upper bound function $\mu$. (As with the nodes of $T$, we assume an ordering on the nodes of $U$ according to a bottom-up, left-to-right traversal.)

Suppose now only one other node $\bar{u}$ of $X$ occurs in $U$. If $\bar{u}$ is sufficiently far from $u$, say $\bar{u}$ is not among the first $\mu(l)$ nodes of $U$, $u$ and $\bar{u}$ do not 'interact' – and the number of nodes in $U$ accessible from $X$ does not exceed $2 \cdot \mu(l)$, i.e. less than the first $\mu(l)$ nodes of $U$ as well as less than the first $\mu(l)$ nodes of $\bar{U} \subseteq \bar{U}$.

Suppose instead that $\bar{u}$ is among the first $\mu(l)$ nodes of $U$, and still no node of $X$ besides $u$ and $\bar{u}$ occurs in $U$. It may now be the case that the nodes accessible from $X$ includes of $U$ more than the first $\mu(l)$ nodes (but still less than the first $\mu(l)$ nodes of $\bar{U} \subseteq \bar{U}$), what we can say, however, is that all of the nodes in $U$ accessible from $X$ using $l$ pebbles are included in all the nodes in $U$ accessible from $X - \{\bar{u}\}$ using $(l + 1)$ pebbles – which are less than the first $\mu(l + 1)$ nodes of $U$.

By a similar argument we show that if all of the $(k - 1)$ nodes of $X$ occur in $U$ sufficiently close to $u$, then the nodes in $U$ accessible from $X$ are less than the first $\mu(l + k - 1)$. Considering each of the other nodes of $X = \{u_1, \ldots, u_k\}$ in turn, we conclude that the number of nodes in $U \cup \cdots \cup U_k$ accessible from $X$ does not exceed:

$$\mu(l + k - 1) + \mu(l + k - 2) + \cdots + \mu(l) < k \cdot \mu(k + l).$$

A more careful analysis (not needed for the main part of the paper) shows that a tight upper bound on the number of nodes accessible from $X = \{u_1, \ldots, u_k\}$ when we have $l \geq k \geq 1$ pebbles at our disposal is: $7 \cdot 2^{l-1} - (l + 3)$.  \[\Box\]

All the results in this Appendix can be established without mention of the pebble game on finite dags. The use of the finite dags called ‘pyramids’, the embedding described in the preceding proof, and Cook’s result (Lemma 1 here), were suggested by Michael Paterson [17], which all led to a considerable saving of technical drudgery.

Appendix 2: Remaining proofs

We complete here the proofs of Theorems 5.1 and 5.2. More specifically, we want to show that Lemmas 5.5 and 5.12 are true in general, i.e. even when no restriction is placed on inputs of iterative programs over $\mathcal{D}$ and $\mathcal{E}$, respectively. We have delayed the general cases of Lemmas 5.5 and 5.12 to this Appendix because they do not provide further insight into the construction of $\mathcal{D}$ and $\mathcal{E}$. We include them here for completeness.
Let us start with $\Sigma$ and the general case of Lemma 5.5. We have an iterative program $P$ over $\Sigma$, and we want to show that $P$ is equivalent to a loop-free program over $\Sigma$. As in 5.4 and 5.5, we may assume that $P$ is in the similarity type of $(\Sigma_0, F) = (D; =; \text{succ}, g, 0, F)$.

Let $T$ be the set of all $l$-$(x_1, \ldots, x_k)$-terms in the similarity type of $(\Sigma_0, F)$, for some fixed $k \geq 1$ and $l \geq 1$.

Let $\bar{T}$ be the set of all $m$-y-terms in the similarity type of the reduct $(D; =; g)$ for some fixed $m \geq k + l$.

Every $k$-tuple $a \in \mathbb{D}^k$ induces a partition on $T$, namely $T = U_a \cup V_a$ where:

- $U_a = \{ t \in T \mid$ there is a term $\bar{t} \in \bar{T}$ and a value $b \in \mathbb{N} \cup \{ \Omega \}$ such that $t^D(a) = \bar{t}^D(b) \}$.
- $V_a = \{ t \in T \mid$ there is no term $\bar{t} \in \bar{T}$ and no value $b \in \mathbb{D} \cup \{ \Omega \}$ such that $t^D(a) = \bar{t}^D(b) \}$.

Reviewing Proposition 3 in Appendix 1, we may think of the set $\{ t^D(a) \mid t \in V_a \}$ as the set of all values that are 'sufficiently high up' in the $\omega$-chained complete-binary tree components of $D$, that is, the set of values that are not accessible from $\mathbb{N} \cup \{ \Omega \}$ using only $m$ program variables.

By Proposition 3 in Appendix 1, if we set $q = 3 \cdot 2^{(m-1)} - 1$, then it is easily verified that for all $a \in \mathbb{D}^k$:

- $\{ t^D(a) \mid t \in U_a \} \subset \{ n \mid n \in \mathbb{N}, i < q \} \cup \{ \Omega \}$,
- $\{ t^D(a) \mid t \in V_a \} \subset \{ n \mid n \in \mathbb{N}, i \geq q \}$.

If a $l$-$(x_1, \ldots, x_k)$-term $t$ contains parameters in $\mathbb{N} \cup \{ \Omega \}$ then for all $a = (u_1, \ldots, u_k) \in \mathbb{D}^k$ we have that $t \in U_a$. To see this, suppose that $t^D(a) \neq \Omega$ and, with no loss of generality, that $t$ does not mention function symbols $\text{succ}$ and $F$. (Any term $u$ of the form $\text{succ}(\cdots x \cdots)$ or $F(\cdots x \cdots)$ is such that $u^D(a) \in \mathbb{N} \cup \{ \Omega \}$.) Hence $t$ is a term involving function symbol $g$, variables $x_1, \ldots, x_k$, and one or more parameters in $\mathbb{N} \cup \{ \Omega \}$. In fact since $t^D(a) \neq \Omega$, it is easy to see that $t$ can mention only one parameter, call it $b$, in $\mathbb{N} \cup \{ \Omega \}$ and that $t^D(a)$ and $b$ must belong to the same $\omega$-chained complete-binary-tree component of $D$. By Proposition 3 in Appendix 1 (see the proof of its part (2) also), we can conclude that $t^D(a) = n$, for some $i \cdot 3 \cdot 2^{(m-1)} - 1 \leq n$, so that $t \in U_a$.

Hence if $l$-$x$-term $t$ is in $V_a$, for some $a \in \mathbb{D}^k$, then $t$ cannot mention parameters in $\mathbb{N} \cup \{ \Omega \}$, nor can $t$ mention subterms that map $a$ to a value in $\mathbb{N} \cup \{ \Omega \}$. Hence such a $t$ cannot involve $\text{succ}$ or $F$ either.

The preceding argument also explains our choice of $\bar{T}$ to be the set of all $m$-$y$-terms in the similarity type of $(D; =; g)$, with $m \geq k + l$. Had we chosen $m = k + l$, we would have for some $a \in \mathbb{D}^k$, $l$-$x$-terms in $V_a$ that mention $0$, $\text{succ}$, and $F$.

We have thus shown that for all $a \in \mathbb{D}^k$, a $l$-$x$-term $t \in V_a$ is an expression involving only function symbol $g$ and variables $\{x_1, \ldots, x_k\}$, and such that no subterm of $t$ maps $a$ to a value in $\mathbb{N} \cup \{ \Omega \}$. 

A.J. Kfoury
1 Lemma. There is a finite collection \( \mathcal{V} = \{ V_1, \ldots, V_p \} \) where each \( V_i \) is a finite set of \( l \)-\( x \)-terms in the similarity type of \( (\mathcal{D}; \leq; g) \), satisfying the following conditions:

1. For all \( k \)-tuple \( a \in \mathcal{D}^k \), \( V_a = V_i \) for some \( V_i \in \mathcal{V} \).
2. There is a loop-free program over \( (\mathcal{D}; \leq; g) \) which on all input \( a \in \mathcal{D}^k \) determines the index \( i \) such that \( V_a = V_i \), \( 1 \leq i \leq p \).

Proof. (1) Recall that \( (\mathcal{D}; \leq; g) \) is uniformly locally finite w.r.t. bounded space. Hence there is an upper bound \( N \), such that for all sets \( W \) of \( l \)-\( (x_1, \ldots, x_k) \)-terms in the similarity type of \( (\mathcal{D}; \leq; g) \) and for all \( a \in \mathcal{D}^k \):

\[
|\text{sub}(a, W)| \leq N
\]

(see the discussion preceding Lemma 2.7). Consider now a \( k \)-tuple \( (a_1, \ldots, a_k) \in \mathcal{D}^k \) all of whose \( k \) components are distinct, and let \( t \) be an arbitrary term in \( V_a \). Let \( \{t_1, \ldots, t_n\} \) be the set of all (distinct) subterms of \( t \). Since \( g \) is one-to-one whenever its value is not in \( \mathbb{N} \cup \{ \Omega \} \), and since no subterm of \( t \) maps \( a \) to a value in \( \mathbb{N} \cup \{ \Omega \} \), it is easy to verify (by induction on the rank of the terms in \( \{t_1, \ldots, t_n\} \) that:

\[
\{t^D_1(a), t^D_2(a), \ldots, t^D_n(a)\}
\]

is a set of \( n \) distinct elements. Hence we have that \( n \leq N \). Hence, if the \( k \) components of \( a \) are distinct, no term \( t \in V_a \) has more than \( N \) distinct subterms. This also means that \( \text{rank}(t) \leq N \).

The same conclusion holds when two or more of the \( k \) components of \( a \in \mathcal{D}^k \) are equal. For example, suppose all the elements in \( \{a_1, \ldots, a_{k-1}\} \) are distinct, but \( a_{k-1} = a_k \). We can repeat the preceding argument to conclude that no \( l \)-\( (x_1, \ldots, x_{k-1}) \)-term \( t' \in V_{(a_1, \ldots, a_{k-1})} \) has more than \( N' \) distinct subterms, so that \( \text{rank}(t') \leq N' \), where \( N' \) is an upper bound on the size of substructures of \( (\mathcal{D}; \leq; g) \) generated from \( (k-1) \) elements within space \( l \). But \( N' \leq N \), so that \( \text{rank}(t') \leq N \). Since a \( l \)-\( (x_1, \ldots, x_k) \)-term \( t \in V_{(a_1, \ldots, a_{k-1}, a_k)} \) is obtained from a \( l \)-\( (x_1, \ldots, x_{k-1}) \)-term \( t' \in V_{(a_1, \ldots, a_{k-1})} \) by replacing zero or more occurrences of \( x_{k-1} \) by \( x_k \), we conclude that \( \text{rank}(t) \leq N \) also.

If we denote by \( T_N \) the following set of \( l \)-\( (x_1, \ldots, x_k) \)-terms:

\[
T_N = \{ t : \text{rank}(t) \leq N, \ t \text{ is in the similarity type of } (\mathcal{D}; \leq; g) \}
\]

then \( V_a \subset T_N \) for all \( a \in \mathcal{D}^k \), by the preceding argument. Since \( T_N \) is a finite set, the desired result follows.

(2) On the basis of the preceding discussion, the construction of an algorithm over \( (\mathcal{D}; \leq; g) \) which determines the index \( i \) such that \( V_a = V_i \) is straightforward.

If \( \alpha = 3 \cdot 2^{(m-1)} - 1 \), there are terms \( t_0(y), t_1(y), \ldots, t_{\alpha-1}(y) \in T \) (as defined before \( (*) \) and \( (**) \)) such that for all \( n \in \mathbb{N} \): \( t_0^D(n) = n, t_1^D(n) = n_1, \ldots, t_{\alpha-1}^D(n) = n_{\alpha-1} \). The desired algorithm over \( (\mathcal{D}; \leq; g) \) with input variables \( \{x_1, \ldots, x_k\} \), carries out the
following steps:

\[
\begin{align*}
&\text{if } x_1 \equiv g(g(x_1, x_1), g(x_1, x_1)) \text{ then } y_1 := x_1 \quad \text{else } y_1 := g(x_1, x_1); \\
&\vdots \\
&\text{if } x_k \equiv g(g(x_k, x_k), g(x_k, x_k)) \text{ then } y_k := x_k \quad \text{else } y_k := g(x_k, x_k); \\
&V := \emptyset; \\
&\text{for every } t \in T_N \text{ (as defined in (1)) do} \\
&\text{if } t^D(x) \notin \{t^D_0(y_1), \ldots, t^D_{q-1}(y_1)\} \cup \cdots \cup \{t^D_0(y_k), \ldots, t^D_{q-1}(y_k)\} \\
&\text{then } V := V \cup \{t\}
\end{align*}
\]

This procedure is clearly loop-free. On all input \(a \in \mathbb{D}^k\), it is easily verified that this procedure returns in variable \(V\) the value of \(V_a\). Recall that only for elements \(a \in \mathbb{N} \cup \{\Omega\} \subseteq \mathbb{D}\) is it the case that \(g(g(a, a), g(a, a)) = a\). \(\square\)

Let us call a \(k\)-tuple \(a \in \mathbb{D}^k\) good if all of its \(k\) components are distinct.

For the next proof, it is easy to check that for all good \(a, b \in \mathbb{D}^k\), \(V_a = V_b\) if and only if:

1. \(\text{sub}(a, V_a)\) is isomorphic to \(\text{sub}(b, V_b)\), and
2. the isomorphism in (1) carries \(a_1\) to \(b_1, a_2\) to \(b_2, \ldots, a_k\) to \(b_k\).

With no loss of generality, we have assumed for (2) that all the variables in \(\{x_1, \ldots, x_k\}\) are mentioned in \(V_a = V_b\). The implication \([(1) \& (2) \Rightarrow V_a = V_b]\) is obvious, while the opposite implication is easily established by induction on the rank of terms in \(V_a\) and \(V_b\).

If \(a\) and \(b\) are not good, then condition (2) must be slightly amended. For example, if \(a_1 \equiv a_k\) and \(b_1 \equiv b_k\) while all the elements in \(\{a_1, \ldots, a_{k-1}\}\) are distinct and as are all those in \(\{b_1, \ldots, b_{k-1}\}\), then (2) becomes:

1. the isomorphism in (1) carries \(a_1\) to \(b_1, a_2\) to \(b_2, \ldots, a_{k-1}\) to \(b_{k-1}\).

Hence \(V_a\) determines the substructure \(\text{sub}(a, V_a)\) up to isomorphism. We can therefore qualify \(V_a\) as being good or not, according to whether the \(k\)-tuple \(a\) which led to it is good or not. And if \(V_a\) is not good, then we shall say that two distinct variable \(x_i\), and \(x_j\) mentioned by terms in \(V_a\) are equivalent if \(a_i \equiv a_j\). It is worth observing that \(x_i\) and \(x_j\), as terms, may or may not belong to \(V_a\).

**Proof of Theorem 5.1.** Consider an arbitrary iterative program \(P\) over \((\Sigma, F)\) with \(k\) input variables and \(l\) program variables. Let \(\mathcal{V} = \{V_1, \ldots, V_p\}\) be the collection determined in the preceding lemma, where each \(V_i\) is a finite set of \((l+1, \ldots, x_k)\)-terms. Note that, by the lemma:

\[
\mathbb{D}^k = \{a \in \mathbb{D}^k \mid V_a = V_1\} \cup \cdots \cup \{a \in \mathbb{D}^k \mid V_a = V_p\}
\]

We first show that for every \(V \in \mathcal{V}\), \(P\) is equivalent to a loop-free program over \(\Sigma\) when inputs are restricted to \(\{a \in \mathbb{D}^k \mid V_a = V\}\). Let \(V = \{v_1(x), \ldots, v_n(x)\} \in \mathcal{V}\) and \(l' = T \cdot V\), where \(T\) is the set of all \((l+1, \ldots, x_k)\)-terms. Let \(S\) be the program scheme
corresponding to program $P$, and let

$$\text{fed}(S) = (\langle \alpha_i, t_i \rangle \mid i \in \omega),$$

where each $\alpha_i$ is a finite conjunction of atomic and negated atomic formulas and $t_i$ is an $l$-x-term, $\alpha_i$ and $t_i$ being in the similarity type of $(\Sigma_0, F)$. Also, all terms appearing in $\alpha_i$ are $l$-x-terms. With no loss of generality, we assume that $S$ has infinitely many finite paths, so that $\text{fed}(S)$ is written as an infinite r.e. sequence.

We now give another $\text{fed}$, called $B$, based on the definition of $\text{fed}(S)$. When $B$ (or more properly, the program scheme corresponding to $B$) is introduced by an appropriately defined sequence of assignment instructions, the resulting program scheme will be equivalent to $S$ over $\Sigma$ when inputs are restricted to $A \in D^k \mid V_a = V$. The appropriate sequence of assignments introducing $B$ is:

Note that the values stored in $\{y_1, \ldots, y_{2n}\}$ are always in $\mathbb{N} \cup \{0\}$. The input variables of $B$ will be $\{y_1, \ldots, y_{2n}\} \cup \{x_i \mid 1 \leq i \leq k \text{ and } x_i \in U\}$.

We next define $\text{fed} B = (\langle \beta, u \rangle \mid i \in \omega)$. Each pair $\langle \beta, u \rangle$ in $B$ is obtained from the corresponding pair $\langle \alpha, u \rangle$ in $\text{fed}(S)$ by the following steps:

(a) If $V$ is not good, replace every variable $x_i$ in $\alpha$ by an equivalent variable $x_i$ whose index $i$ is the smallest - if such $x_i$ exists.

This preliminary step (a) guarantees the correctness of steps (c) and (d) below.

(b) Replace every atomic (respectively, negated atomic) formula of the form $v \equiv w$ (respectively, $v \neq w$) with $v \in V$ and $w \in U$, by $\text{false}$ (respectively, $\text{true}$).

(c) Replace every atomic (respectively, negated atomic) formula of the form $v \equiv v'$ (respectively, $v \neq v'$) with $v, v' \in V$ and $v \neq v'$, by $\text{false}$ (respectively, $\text{true}$).

(d) Replace every atomic (respectively, negated atomic) formula of the form $v \Rightarrow v$ (respectively, $v \Rightarrow v$) with $v \in V$, by $\text{true}$ (respectively, $\text{false}$).
After steps (b), (c), and (d), if \( \beta \) still contains a term from \( V \) then this term must appear in \( \beta \) as a subterm of another term from \( U \). We next proceed to eliminate from \( \beta \) all subterms that are terms in \( V \).

(e) For every \( v_i \in V \), replace in \( \beta \) every subterm of the form \( \text{succ}(v_i) \) by \( \Omega \); every subterm of the form \( g(v_i, *) \) of \( g(\ast, v_i) \) not in \( V \) by \( \ast \) if \( \ast \neq v_i \), and by \( y_i \) if \( \ast = v_i \); and every subterm of the form \( F(v_i) \) by \( y_{n+i} \).

As a result of step (e) all terms belonging to \( V \) have been eliminated from the expression of \( \beta \), in particular also all variables which (as terms) belong to \( V \). So far, the \( l \)-x-term \( t \), in the pair \( \langle a, t \rangle \), has not been changed. We next transform \( t \) to obtain \( u \), in the pair \( \langle \beta, u \rangle \).

\( \text{If } t = r_i \in V \), let \( u = t \); and \( \text{if } t \notin V \), then repeat the substitutions of step (e) in \( t \). \)

It is readily seen that all terms appearing in \( B = \langle \beta_i, u_i \rangle \) have pebble-complexity \( \leq l \). Hence \( B \) is indeed the fed of an iterative program scheme.

Strictly speaking, the input variables of \( B \) also contain \( \{x_i | 1 \leq i \leq k \text{ and } x_i \in V \} \), since the latter appear in the terms in \( V = \{v_1, \ldots, v_n \} \) which were left untouched in step (f) above. However, \( \{v_1, \ldots, v_n \} \) are only mentioned as outputs of \( B \) and do not interfere with the satisfaction and non-satisfaction of the \( \beta \)'s, and therefore can be treated as (finitely many) formal parameters not affecting the flow of computation in \( B \).

Observing that the values assigned to the input variables of \( B \), \( \{y_1, \ldots, y_{2n}\} \setminus \{x_i | 1 \leq i \leq k \text{ and } x_i \in U \} \), are restricted to the set:

\[ \{n_i | n \in \mathbb{N}, i < q \} \cup \{\Omega\}, \]

where \( q = 3 \cdot 2^{m^l} - 1 \), we conclude that \( B \) can be replaced by a loop-free program scheme (equivalent to \( B \) over \( \Xi \) when inputs are thus restricted) by Lemma 5.5.

Reviewing the definitions of \( B \) (together with the sequence of assignments introducing \( B \)), of \( \text{fed}(S) \) and \( S \), we also conclude that they are all equivalent to the same loop-free program scheme over \( \Xi \) when inputs are restricted to \( \{a \in \mathbb{D}^k | V_a = V \} \). An immediate consequence of the preceding is that, for all \( V_i \in \mathcal{V} \), there is a loop-free program over \( \Xi \) denoted \( Q_i \) which is equivalent to \( P \) when inputs are restricted to \( \{a \in \mathbb{D}^k | V_a = V_i \} \). We now conclude that \( P \) itself is equivalent to a loop-free program over \( \Xi \) that carries out the following steps:

1. Given arbitrary \( a \in \mathbb{D}^k \), determine the set \( V_a \) and the index \( i \) such that \( V_a = V_i \); using the loop-free program of part (2) of the lemma;

\( \text{(2) Run loop-free program } Q_i \text{ on input } a ; \)

3. If and when \( Q_i(a) \) converges, return the resulting output of \( Q_i \).

The verification of this procedure is straightforward. \( \square \)
The preceding development can be directly adapted to structure $\mathcal{E}$. Now we have an iterative program $P$ over structure $\mathcal{E}$, and we want to show that $P$ is equivalent to a loop-free program over $\mathcal{E}$ – which is the general case of Lemma 5.12.

We indicate how each concept defined in relation to $\Sigma$ must be amended in relation to $\mathcal{E}$. The set $T$ is now set of all $l$-($x_1, \ldots, x_k$)-terms in the similarity type of $\mathcal{E}$. 

The set $\widetilde{T}$ is now the set of all $m$-($y_1, \ldots, y_m$)-terms in the similarity type of the reduct $(\mathcal{E}; \preceq; \text{pair}, g)$, for some fixed $m \geq k + l$.

Every $k$-tuple $a \in \mathbb{E}^k$ induces a partition on $T$, namely $T = U_a \cup V_a$ where:

$U_a = \{t \in T | \text{there is a term } \bar{t} \in \widetilde{T} \text{ and a } m\text{-tuple } \bar{h} \in (\mathbb{N} \cup \{\Omega\})^m \text{ such that } t^E(a) = \bar{t}^F(\bar{b})\}$, \hspace{1cm} (§)

$V_a = \{t \in T | \text{there is no term } \bar{t} \in \widetilde{T} \text{ and no } m\text{-tuple } \bar{b} \in (\mathbb{N} \cup \{\Omega\})^m \text{ such that } t^E(a) = \bar{t}^F(\bar{b})\}$. \hspace{1cm} (§§)

By Propositions 2 and 3 in Appendix 1, it is not difficult to check that if $q = 3 \cdot 2^{(m-1)} - 1$, then for all $a \in \mathbb{E}^k$:

$\{t^F(a) | t \in U_a\} \subset \{n_i | n \in \mathbb{E}_0 \cup \ldots \cup \mathbb{E}_{m-1}, i < q\} \cup \{\Omega\}$, \hspace{1cm} (§)

$\{t^F(a) | t \in V_a\} \subset \{n_i | n \in \mathbb{E}_r, r \geq m, i \text{ arbitrary}\}$

$\cup \{n_i | n \in \mathbb{E}_0 \cup \ldots \cup \mathbb{E}_{m-1}, i \geq q\}$. \hspace{1cm} (§§)

By an argument similar to that used in the case of $\Sigma$, we can again show here that every $l$-($x_1, \ldots, x_k$)-term $t \in V_a$ cannot involve parameters in $\mathbb{E}_0 \cup \{\Omega\} = \mathbb{N} \cup \{\Omega\}$, nor therefore can it involve $\text{pred}$, $\text{proj}$, and $\Theta$. Hence such a $l$-($x_1, \ldots, x_k$)-term is in the similarity type of $(\mathcal{E}; \preceq; \text{pair}, g)$.

There is an added complication in the case of $\mathcal{E}$; namely, there are $k$-tuples $a \in \mathbb{E}^k$ such that $V_a$ is an infinite set. This happens, for example, if there is a term $t \in V_a$ such that $t^E(a) = n_i$ and $n \in \mathbb{E}_0 \cup \ldots \cup \mathbb{E}_{m-1}$ and $i < 3 \cdot 2^{(l-1)} - 1$; in such a situation there is a $l$-($x_1, \ldots, x_k$)-term $u(x)$ such that $t^E(a) = u^E(n) = n_i$, so that also $t^F(a) = u^F(g(t^F(a), t^F(a)))$. Then if we define the set of $l$-($x_1, \ldots, x_k$)-terms $\{v_j | j \in \omega\}$ inductively by:

1. $v_0 = g(t, t)$,
2. $v_{i+1} = g(v_i, v_i)$

then for all $j \in \omega$, $v_j^E(a) = n_i$, which in turn implies that $V_a$ is infinite.

Instead of working with $V_a$, we shall work with an appropriately defined finite subset $\hat{V}_a \subseteq V_a$. Given a term $t$, we denote by $\text{subterms}(t)$ the set of all subterms of $t$. We choose a fixed enumeration of all $l$-($x_1, \ldots, x_k$)-terms in the similarity type of $(\mathcal{E}; \preceq; \text{pair}, g)$, say: $t_0, t_1, \ldots, t_i, \ldots, i \in \omega$, satisfying the following condition:

for all $i, j \in \omega$, if $i < j$ then $|\text{subterms}(t_i)| \leq |\text{subterms}(t_j)|$.

For all $a \in \mathbb{E}^k$, we now define $\hat{V}_a$ as:

$\hat{V}_a = \{t_i | t_i \in V_a \text{ and there is no term } t_i, i < j, \text{ such that } t_i^F(a) = t_j^F(a)\}$. 
it is clear that \( \text{sub}(a, \hat{V}_a) = \text{sub}(a, V_a) \). Since \((E; \models; \text{pair}, g)\) is uniformly locally finite w.r.t. bounded space, there is an upper bound \(N\) such that for all \(a \in E^k\):

\[
|\hat{V}_a| = |\text{sub}(a, \hat{V}_a)| \leq N.
\]

2 Lemma. There is a finite collection \(\mathcal{V} = \{V_1, \ldots, V_p\}\) where each \(V_i\) is a finite set of \(l\)-\(x\)-terms in the similarity type of \((E; \models; \text{pair}, g)\), satisfying the following conditions:

(1) For all \(k\)-tuple \(a \in E^k\), \(\hat{V}_a = V_i \) for some \( V_i \in \mathcal{V}\).

(2) There is a loop-free program over \((E; \models; \text{pair}, \text{proj}, g)\) which on input \(a \in E^k\), determines the index \(i\) such that \(\hat{V}_a = V_i, i \leq i \leq p\).

Proof. (1) This is similar to part (1) of Lemma 1. We want to show that for all \(a \in E^k\) and all \(t \in \hat{V}_a\), \(\text{rank}(t) \leq N\). In contrast to the proof of Lemma 1, we do not need to distinguish between the cases when the \(k\) components of \(a\) are distinct and when they are not, because of the ordering we have imposed on the enumeration of all \(l\)-\(x\)-terms in the similarity type of \((E; \models; \text{pair}, g)\).

We consider an arbitrary term \(t \in \hat{V}_a\) and the corresponding set \(\text{subterm}(t) = \{t_1, \ldots, t_n\}\). By the definition of \(\hat{V}_a\), each of the elements in:

\[
\{t_1^i(a), \ldots, t_n^i(a)\}
\]

must be distinct, so that \(\text{rank}(t) \leq n \leq N\), as desired.

If we denote by \(T_N\) the following finite set of \(l\)-\(x\)-\(1\), \(x_k\)-terms:

\[
T_N = \{t|\text{rank}(t) \leq N, t \text{ is in the similarity type of } (E; \models; \text{pair}, g)\}.
\]

then for all \(a \in E^k\), we have that \(\hat{V}_a \subseteq T_N\).

(2) Similar to part (2) of Lemma 1. If \(q = 3 \cdot 2^{(m-1)} - 1\), there are \(m\)-\(x\)-terms involving only function symbol \(g\), \(t_0(x), t_1(x), \ldots, t_{q-1}(x)\), such that for all \(n \in \bar{E}\):

\[
t_0(n) = n, t_{l}^{i}(n) = n_1, \ldots, t_{q-1}(n) = n_q - 1.
\]

Let us also define the following special \(m\)-\(y\)-\(1\), \(y_m\)-terms, where we denote \((y_1, \ldots, y_m)\) by \(y\):

\[
u_0(y) = y_1
\]

\[
u_1(y) = \text{pair}(y_1, y_2)
\]

\[
u_2(y) = \text{pair}(\text{pair}(y_1, y_2), \text{pair}(y_2, y_3))
\]

\[
\vdots
\]

\[
u_{m-1}(y) = \text{the } m\text{-}y\text{-term } t(y_1, \ldots, y_m) \text{ defined in the proof of Lemma 5.11, with } l \text{ changed to } m.
\]

Clearly if \(n \in \bar{E}_r\), with \(0 \leq r < m\), then there is a \(m\)-tuple \(b \in (\mathbb{N} \cup \{l\})^m\) such that \(u_r^l(b) = n\).
The desired loop-free program over \((E; \Rightarrow; \text{pair, proj, } g)\) has the following form:

\[
\begin{align*}
\text{if } x_1 &= g(g(x_1, x_1), g(x_1, x_1)) \text{ then } \bar{x}_1 := x_1 \text{ else } \bar{x}_1 := g(x_1, x_1); \\
& \vdots \\
\text{if } x_k &= g(g(x_k, x_k), g(x_k, x_k)) \text{ then } \bar{x}_k := x_k \text{ else } \bar{x}_k := g(x_k, x_k); \\
\text{for } i \in \{1, 2, \ldots, k\} \text{ do} \\
& \quad \text{begin } y_{i1} := \text{proj}(0, \bar{x}_i); \\
& \quad \quad y_{i2} := \text{proj}(1, \bar{x}_i); \\
& \quad \quad \vdots \\
& \quad \quad y_{im} := \text{proj}(m-1, \bar{x}_i) \\
& \quad \text{end}; \\
V &:= \emptyset; \\
\text{for } t \in T_N (\text{as defined in (1)}) \text{ do} \\
& \quad \text{if } t^E(x) \notin \{t^E(u^E(b)) | 0 \leq i < q, 0 \leq j < m, b \in \{y_{11}, y_{12}, \ldots, y_{km}\}^*\} \text{ then} \\
& \quad V := V \cup \{t\}
\end{align*}
\]

In this procedure the values stored in \(\bar{x}_1, \ldots, \bar{x}_k\) are always from \(E \cup \{\Omega\} \subset E\), while those stored in \(y_{11}, y_{12}, \ldots, y_{km}\) are from \(N \cup \{\Omega\}\). On input \(a \in E^k\), this program returns in variable \(V\) the value of \(V_a \cap T_N\), not that of \(\hat{V}_a\). To obtain \(\hat{V}_a\) which is a subset of \(V_a \cap T_N\), we list the members of the latter set in the order specified before the lemma, eliminating every term \(t'\) which maps \(a\) to the same value as another term \(t\) already listed. Since there is a finite upper bound (independent of \(a\)) on the size of \(V_a \cap T_N\), we can thus determine \(\hat{V}_a\) in a loop-free fashion. \(\square\)

An immediate consequence of the preceding lemma is that, i.e., all \(a \in E^k\) and all \(l\)-x-term \(t\), it is decidable whether \(t \in U_a\) or \(t \in V_a\). First we can effectively determine the set \(\hat{V}_a\), by part (2) of the lemma, the structure \((E; \Rightarrow; \text{pair, proj, } g)\) being computable. Second, \(t\) is in \(V_a\) if and only if \(t\) does not involve \(\text{pred, proj, and } \hat{\Theta}\), and there is a term \(f \in \hat{V}_a\) such that \(t^E(a) = t^E(f(a))\).

We leave it to the reader to check that for all \(a \in E^k\), the set of terms \(\hat{V}_a\) determines up to isomorphism the substructure \(\text{sub}(a, \hat{V}_a) = \text{sub}(a, V_a)\).

**Proof of Theorem 5.2.** This proof follows closely the earlier proof of Theorem 5.1. We point out the differences between the two proofs.

Instead of \(V_a\), we use here \(\hat{V}_a\). As in 5.1, we first show that for each \(\hat{V} \in \Gamma\), every iterative program \(P\) over \(E\) is equivalent to a loop-free program over \(E\) when inputs are restricted to \(\{a \in E^k | \hat{V}_a = \hat{V}\}\). Here \(\hat{V}\) is a finite set of \(l\)-x-terms, say \(\hat{V} = \{v_1, v_2, \ldots, v_n\}\); we denote \(V_a\) by \(V\), and \(U = T - V\).
As in 5.1, we define \( \text{fed}(S) \) from \( P \), and then another \( \text{fed}B \) from \( \text{fed}(S) \). The appropriate sequence of assignments introducing \( B \) is now:

\[
\text{input } x_1, \ldots, x_k
\]

\[
\text{for } i \in \{1, \ldots, n\} \text{ do if } g(v_i, v_i) \in V \text{ then } y_i := g(v_i, v_i) \text{ else } y_i := \Omega
\]

\[
y_{n+1} := \hat{\Theta}(v_1); \ldots; y_{2n} := \hat{\Theta}(v_n)
\]

\[
y_{2n+1} := [v_1]_q; \ldots; y_{3n} := [v_n]_q
\]

\[
\text{for } i, j \in \{1, \ldots, n\} \text{ do } z_{ij} := \text{proj} \; (v_i, v_j)
\]

In the above program scheme, \( q = 3 \cdot 2^{(m-1)} - 1 \). An assignment of the form \( y := \lfloor v \rfloor_q \) is a short-hand notation for the following operation (when we interpret the scheme in \( \mathcal{E} \)): "If \( v^E(\alpha) \in \bar{\mathcal{E}} \) and \( v^E(\alpha) = \langle b_1, \ldots, b_v \rangle \) for some \( r \geq q \), then \( y := \langle b_1, \ldots, b_v \rangle \) else \( y := \Omega \)". That is, if \( v^E(\alpha) \) is a sequence whose length exceeds \( q \), then \( \lfloor v^E(\alpha) \rfloor_q \) consists of the first \( q \) components in \( v^E(\alpha) \). It is easily checked that, for a fixed \( q \), the function \( \lfloor \; \rfloor_q \) can be computed by a loop-free program over \( \mathcal{E} \).

The values stored in \( \{y_1, \ldots, y_{3n}, z_{11}, z_{12}, \ldots, z_{nn}\} \) are always in \( \{t^E(\alpha) | t \in U\} \).

The input variables of \( B \) are \( \{y_1, \ldots, y_{3n}, z_{11}, \ldots, z_{nn}\} \cup \{x_i | 1 \leq i \leq k \text{ and } x_i \in U\} \).

In the definition of \( B \), step (a) becomes:

(a) Replace every term \( v \in V \) which appears in either \( \alpha \) or \( t \) by an equivalent term \( \hat{v} \in \hat{V} \) (\( v \) and \( \hat{v} \) are 'equivalent' if \( v^E(\alpha) = \hat{v}^E(\alpha) \)).

Steps (b), (c), and (d) are identical to (b), (c), and (d) in the proof of 5.1—except that \( \hat{V} \) here replaces \( V \) there.

Step (e) now becomes:

(c) For every \( v_i \in \hat{V} \), replace in \( \beta \) every subterm of the form \( \text{pred}(v_i) \) by \( \Omega \); every subterm of the form \( \text{pair}(v_i, \ast) \) or \( \text{pair}(\ast, v_i) \) not in \( \hat{V} \) by \( \Omega \); every subterm of the
form \( g(v_i, *) \) or \( g(*, v_i) \) not in \( \hat{V} \) by \( \Omega \) if \( * \neq v_i \), and by \( y_i \) if \( * = v_i \); and every subterm of the form \( \tilde{\Theta}(v_i) \) by \( y_{n+i} \). Further, for all \( v_i, v_j \in \hat{V} \), replace in \( \beta \) every subterm of the form \( \text{proj}(v_i, *) \) with \( * \in U \) by \( \Omega \); every subterm of the form \( \text{proj}(*, v_i) \) with \( * \in U \) by \( \text{proj}(*, y_{2n+i}) \); and every subterm of the form \( \text{proj}(v_i, v_j) \) by \( z_{ij} \).

Step (f) here is identical to (f) in the proof of 5.1, with \( \hat{V} \) replacing \( V \).

The rest of the proof is now similar to the proof of 5.1, with \( \hat{V} \) replacing \( V \) throughout. □

Acknowledgment

Special thanks are due to my colleagues Rohit Parikh and Leonid Levin for numerous suggestions, critical comments, and corrections, at different times in the course of the research for this paper. I also thank Fred Abramson, Ravel Urzyczyn and two anonymous referees who made several suggestions that are now incorporated in this version of the paper. I would also like to thank Thomas Orowan who was responsible for putting the paper in its final typewritten form.

References


