DETERMINISTIC AND NONDETERMINISTIC COMPUTATION, AND HORN PROGRAMS, ON ABSTRACT DATA TYPES

J. V. TUCKER* AND J. I. ZUCKER†

We investigate the notion of ""semicomputability,"" intended to generalize the notion of recursive enumerability of relations to abstract structures. Two characterizations are considered and shown to be equivalent: one in terms of ""partial computable functions"" (for a suitable notion of computability over abstract structures) and one in terms of definability by means of Horn programs over such structures. This leads to the formulation of a ""Generalized Church-Turing Thesis"" for definability of relations on abstract structures.

1. INTRODUCTION

We will examine computation and specification by means of Horn clauses on abstract data types, using a general theory of computable functions and relations on abstract data types. In this theory, an abstract data type is modelled semantically by a many-sorted algebra $A$, considered unique up to isomorphism, and various equivalent models of

---

*Department of Mathematics and Computer Science, University College of Swansea, Swansea SA2 8PP, Wales. The research of J.V.T. was partially supported by SERC Research Grants GR/F 10606 (under the Alvey Programme) and GR/F 59070.

†Department of Computer Science and Systems, McMaster University, Hamilton, Ontario L8S 4K1, Canada. The research of J.I.Z. was supported by the National Science Foundation under grant no. DCR-8504296, by SERC Research Grant GR/F 10606 (under the Alvey Programme), by a grant from the Science and Engineering Research Board of McMaster University, by a grant from the Natural Sciences and Engineering Research Council of Canada, and by an academic travel grant from the British Council.

Address correspondence to Jeffery I. Zucker, Department of Computer Science and Systems, McMaster University, Hamilton, Ontario L8S 4K1, Canada.

Accepted January 1991.
computation are used to define effectively computable functions and sets on $A$. Usually these models are generalizations to $A$ of sequential deterministic methods of computing on the natural numbers, or finite strings; however we use a parallel deterministic formalism to study computability on $A$.

We will formulate a concept of Horn clause computability that applies to any many-sorted algebra $A$ and consider the following questions:

Does Horn clause computability define only, and all, the "effectively calculable" relations and functions on $A$?

Does Horn clause computability, with its nondeterminism and potential for parallelism, effect the Generalized Church-Turing Thesis for computation on $A$?

And, since abstract data types may be axiomatically specified,

What is the relation between an algebraic specification for $A$ and a Horn clause program over $A$?

In order to answer these questions fully, we must extend the theory of computability to discuss the notion of specifiability. We begin to determine the scope and limits of effective specification, and postulate an appropriate thesis for specification that complements the Generalized Church-Turing Thesis for computation on abstract data types. We show that Horn clause "computability" defines more than the effectively calculable sets on $A$, in general. We also show that algebraic specifications and Horn clause definability are complementary and equivalent specification tools.

This paper is divided into two parts. Part A (Sections 1–4) sets the background, with an extensive discussion of the computability and semicomputability of functions and relations (see below). Part B (Sections 5–9) relates all this to Horn clause definability.

In Section 1 we give various definitions relating to the many-sorted algebra $A$. This algebra is assumed to be standard, i.e., contain a standard model of arithmetic $\mathbb{N}$. We also introduce the algebra $A^*$ which extends $A$ with sets of finite sequences of the elements of the sets of $A$. This structure $A^*$ will be important in the development of our work.

Note that any structure can be standardized and "starred," by adjoining the sets $\mathbb{N}$ and $\mathbb{B}$ and the sets of finite sequences, together with the appropriate operations, so we do not lose generality in focussing on standard and starred structures.

In Section 2 we sketch the theory of the cov inductively computable partial functions on $A$, introduced in [19]. These are defined from the basic operations of $A$ by composition, simultaneous course-of-values recursion and least number search. We also introduce the star inductively computable partial functions on $A$, which are defined by simultaneous primitive recursion (instead of course-of-values recursion) on $A^*$. These two classes are shown to be equivalent, and henceforth we work mainly with the star inductively computable functions.

We believe that these functions are appropriate to establish the scope and limits of deterministic computation over an abstract data type. Accordingly, we record a Generalized Church-Turing Thesis for deterministic computation.

In Section 3 we turn to semicomputability. Horn clauses are designed to compute relations, rather than functions, so we define a relation $R$ on $A$ to be (i) star semicomputable if it is the domain of a star computable partial function on $A$, and (ii) projectively star semicomputable if it is a projection of a star semicomputable
relation, i.e., if for some semicomputable relation $S$,
\[
R(x) \leftrightarrow \exists y S(x, y).
\]
These notions are distinct in general, because projectivity involves nondeterministic choice, but are equivalent on the important class of class of minimal algebras (such as $\mathbb{N}$), which are generated by elements named as constants.

In Section 4 we present another approach to computability and semicomputability, via imperative programs—generally, some version of 'while' programs. For the case corresponding to projective semicomputability we define the new construct of initialization of search variables. We also treat computation trees and Engeler’s Lemma, which has important theoretical consequences, including a proof of the fact that projective semicomputability is not equivalent to semicomputability. Finally we consider definability via 'while' programs with random assignments, which is again equivalent to projective star semicomputability.

In Part B we turn to Horn clause computability over $A$. This exercise involves a clarification of the interplay between functions, assumed as operations of $A$, and new relations, defined by a program, and is relevant to the design theory of logic programming languages.

In Section 5 we give the relevant definitions, and in Section 6 we prove:

**Theorem.** A relation $R$ on $A$ is definable by Horn clauses over $A^*$ iff $R$ is projectively star-semicomputable. If $A$ is minimal then $R$ is definable by Horn clauses over $A^*$ iff $R$ is star-semicomputable.

In Section 7 we consider other well-known notions of definability, due to Montague and Moschovakis, show their equivalence to Horn clause computability via results of Gordon [9] and, in particular, Fitting [5].

In Section 8 we consider the important case when an abstract data type is axiomatically specified under initial algebra semantics.

Let $\Sigma$ be a signature and $E$ a set of Horn clauses (e.g., conditional equations) over $\Sigma$. Let $\text{Term}(\Sigma)$ be the algebra of closed or ground terms over $\Sigma$ and let $\text{Init}(\Sigma, E)$ be the initial algebra of the specification or theory $(\Sigma, E)$. Assume $\text{Init}(\Sigma, E)$ is standard, i.e., contains a copy of the natural numbers. Let $P$ be a set of Horn clauses over $\Sigma$.

**Theorem.** A computation by a Horn clause program $P$ on $\text{Init}(\Sigma, E)$ is equivalent to a computation by the Horn clause program $H \cup E$ on $\text{Term}(\Sigma)$.

Horn clause computability applied to $\text{Init}(\Sigma, E)$ is thus equivalent to standard Horn clause computability on $\text{Term}(\Sigma)$ (see [11]). Hence the above result may be applied in an implementation technique for abstract data types in the context of logic programming; see also [7].

In Section 9 we will review our answers to the general questions above.

The relationship between computability on $\mathbb{N}$ and logic programming has been considered by Tärnlund [18], Sebelik and Stepánek [16] and Fitting [6]. Many-sorted logic programming has been considered in Cohn [2] and Walther [25] with efficiency of implementation in mind. The basic results in this paper were announced in [20]. Some related results on single-sorted partial structures with countable domains can be found in [17].
PART A: DETERMINISTIC AND NONDETERMINISTIC COMPUTATION

1. ABSTRACT STRUCTURES AND ABSTRACT DATA TYPES

1.1. Standard Signatures, Standard Structures, and Classes of Structures

A standard signature $\Sigma$ specifies (1) a finite set of sorts: algebraic sorts $1, \ldots, r$ (for some $r \geq 0$), and the numerical sort $N$ and boolean sort $B$; and (2) finitely many function symbols $F$, each having a type $(i_1, \ldots, i_m; i)$, where $m \geq 0$ is the arity of $F$, $i_1, \ldots, i_m$ are the domain sorts and $i$ is the range sort (including the case $m = 0$ for constant symbols). These include symbols for certain standard operations associated with the sorts $N$ and $B$: (a) arithmetical operations, namely the constant '0', successor operation $S$ and order relation '$<$' on the natural numbers; and (b) boolean operations, including a complete set of propositional connectives, the constants true and false, and an equality operator $eq_i$ at some sorts $i$, including (at least) $i = N$ and $i = B$. We call those sorts $i$ with the equality operator $eq_i$, equality sorts. (We do not want to assume that all sorts necessarily have a computable equality.)

Our signatures do not explicitly include relation symbols; relations are interpreted for now as boolean-valued functions. (This will change in Part B.)

We make one further assumption on $\Sigma$:

Instantiation Assumption. Each sort of $\Sigma$ is instantiated, i.e., there is a closed term of each sort.

We will see later where this assumption is used.

A $\Sigma$-structure $A$ has, for each sort $i$ of $\Sigma$, a carrier set $A_i$, and for each function symbol $F$ of type $(i_1, \ldots, i_m; i)$, a function $F^A : A_{i_1} \times \cdots \times A_{i_m} \to A_i$. The structure $A$ is standard if $A_N = \mathbb{N}$, the set of natural numbers, $A_B = \mathbb{B} = \{tt, ff\}$, the set of truth values, and the standard operations have their standard interpretations on $\mathbb{N}$ and $\mathbb{B}$, so that, in particular, the equality symbol is interpreted as identity on each sort.

We will only consider standard signatures and structures.

Note that any many-sorted structure $B$ can be standardized to such a structure $A$ by the adjunction of the sets $\mathbb{N}$ and $\mathbb{B}$, together with their standard operations, so that $B$ is a reduct of $A$ to the signature of $B$.

This notion of reduct is important in what follows, so we define it here.

Definition. Let $\Sigma$ and $\Sigma'$ be signatures with $\Sigma \subseteq \Sigma'$. If $A$ is a $\Sigma'$ algebra, then the $\Sigma$-reduct of $A$ is the algebra

$$A \mid \Sigma$$

of signature $\Sigma$ consisting of the carriers of $A$ named by the sorts of $\Sigma$ and equipped with the functions and relations of $A$ named by the function and relation symbols of $\Sigma$. 
1.2. Strictly Standard Signatures and Structures

We consider a notion stricter than standardness, namely \textit{strict standardness}.

The signature $\Sigma$ is \textit{strictly standard} if the only operations with range sort $N$ are the \textit{standard operations} listed in §1.1. The structure $A$ is \textit{strictly standard} if its signature is.

Actually, strictly standard signatures and structures are easy to come by—whenever we standardize a structure, it is automatically strictly standard! (If $A$ contains the carrier $\mathbb{N}$ with non-standard operations on it, then the standardized version of $A$ will contain another copy of $\mathbb{N}$ with only the standard operations on it.)

Strictly standard structures have some interesting properties, as we will see in §4.4.

Now fix a (not necessarily strictly) standard signature $\Sigma$. We will consider classes $\mathcal{K}$ of $\Sigma$-structures. We impose no restriction on $\mathcal{K}$ other than that it be \textit{closed under isomorphism}. Such a class can be thought of as an \textit{abstract data type}.

Fix such a class $\mathcal{K}$, and consider a particular $\Sigma$-structure $A \in \mathcal{K}$. We will extend $A$ in two stages.

1.3. Unspecified Value $u$ and Structures $A^u$ of Signature $\Sigma^u$

Given a standard $\Sigma$-structure $A$, for each sort $i$ let $u_i$ be a new object, representing an "unspecified value," and let $A^u_i = A_i \cup \{u_i\}$. For each function symbol $F$ of $\Sigma$ of type $(i_1, \ldots, i_m; i)$, extend its interpretation $F^A$ on $A$ to a function $F^{A^u}: A^u_i \times \cdots \times A^u_{i_m} \to A^u_i$ by \textit{strictness}—i.e., the value is defined as $u$ whenever any argument is $u$. Then the structure $A^u$ with signature $\Sigma^u$, contains:

(i) the original carriers $A_i$ of sort $i$, and functions $F^A$ on them;
(ii) the new carriers $A^u_i$ of sort $i^u$, and functions $F^{A^u}$ on them;
(iii) a constant $\text{unspec}_i$ of type $i^u$ to denote $u_i$ as a distinguished element of $A^u_i$;
(iv) a boolean-valued function $\text{Unspec}_i$ of type $(i; \mathbb{B})$, the characteristic function of $u_i$;
(v) an \textit{embedding function} $i_j$ of type $(i; i^u)$ to denote the embedding of $A_j$ into $A^u_i$, and an inverse function $j_i$ of type $(i^u; i)$, which maps $u_i$ to the denotation of some closed term in $A_j$ (this being possible by the Instantiation Assumption) for each sort $i$; and finally
(vi) an \textit{equality operator} on $A^u_i$ for each equality sort $i$.

Also, $\mathcal{K}^u$ is the class of structures $A^u$ for $A \in \mathcal{K}$.

REMARKS.

(1) The structure $A$ is the $\Sigma$-reduct of $A^u$.

(2) \textit{(Two- and three valued boolean operations.)} $A^u$ is itself a standard structure. However it contains the carrier $\mathbb{B}^u = \{tt, ff, u\}$ as well as $\mathbb{B}$, with associated extensions of the original standard boolean operations, leading to a \textit{weak three-valued logic} (see [19], §1.1.6). Further, there are \textit{two equality operations} on $A^u_i$ for each equality sort $i$: (a) the extension by strictness of $\text{eq}_i^A$ to a
three-valued function

\[ \text{eq}^u_{i^*}: A^u_i \times A^u_i \to \mathbb{B}^u \]

which has the value \( u_B \) if either argument is \( u_j \); (b) the "standard (two-valued) equality" on \( A^u_i \)

\[ \text{eq}^u_i: A^u_i \times A^u_i \to \mathbb{B}, \]

which we will usually denote by \( \cdot = \) in infix.

(3) The structure \( A^u \) has some resemblance to those considered in \([8]\).

I.4. Structures \( A^* \) of Signature \( \Sigma^* \)

Define, for each sort \( i \), the carrier \( A^*_i \) to be the set of pairs \( a^* = (\xi, l) \) where \( \xi: \mathbb{N} \to A^u_i, \, l \in \mathbb{N} \) and for all \( n \geq l, \, \xi(n) = u_j \). So \( l \) is a witness to the "finiteness" of \( \xi \), or an "effective upper bound" for \( a^* \). The elements of \( A^*_i \) have "starred sort" \( i^* \), and can be considered as finite sequences or arrays. The resulting structures \( A^* \) have signature \( \Sigma^* \), which extends \( \Sigma^u \) by including, for each sort \( i \), the new "starred sorts" \( i^* \) as well as \( i^u \), and also the following new function symbols:

(i) The \( \text{Lgth}_i \) operator, of type \((i^*, \mathbb{N})\), where

\[ \text{Lgth}^A_i((\xi, l)) = l. \]

(ii) the application operator \( \text{Ap}_i \) of type \((i^*, N; i^u)\), where

\[ \text{Ap}^A_i((\xi, l), n) = \xi(n), \]

(iii) the null array \( \text{Null}_i \) of type \( i^* \), where

\[ \text{Null}^A_i = (\lambda n \cdot u_j, 0) \in A^*_i. \]

(iv) The \( \text{Update}_i \) operator of type \((i^*, N; i^*)\), where for \((\xi, l) \in A^*_i, \, n \in \mathbb{N} \) and \( x \in A^u_i \), \( \text{Update}^A_i((\xi, l), n, x) \) is the array \((\eta, l) \in A^*_i \) such that for all \( k \in \mathbb{N} \),

\[ \eta(k) = \begin{cases} \xi(k) & \text{if } k < l, \, k \neq n \\ x & \text{if } k < l, \, k = n \\ u_j & \text{otherwise}. \end{cases} \]

(v) the \( \text{Newlength}_i \) operator of type \((i^*, \mathbb{N}; i^*)\), where \( \text{Newlength}^A_i((\xi, l), m) \) is the array \((\eta, m) \) such that for all \( k \),

\[ \eta(k) = \begin{cases} \xi(k) & \text{if } k < m \\ u_j & \text{if } k \geq m, \end{cases} \]

(vi) the equality operator on \( i^* \), for each equality sort \( i \).

(Thus justification for this is that if a sort \( i \) has computable equality, then clearly so has the sort \( i^* \), since it amounts to testing equality of finitely many pairs of objects of sort \( i \), up to a computable length.)

For \( a^* \in A^*_i \) and \( n \in \mathbb{N} \), we write \( a^*[n] \) for \( j^A_i(\text{Ap}^A_i(a^*, n)). \) (So \( a^*[n] \) is the element of \( A_i^u \) 'corresponding to' \( \text{Ap}(a^*, n) \in A^u_i \).)

The standardness of \( A^* \) follows from (vi) and the standardness of \( A^u \).

Also, \( \mathcal{A}^* \) is the class of structures \( A^* \) for \( A \in \mathcal{A} \).
REMARKS.

(1) The structure $A^u$ is the $\Sigma^u$-reduct of $A^*$, and $A$ is the $\Sigma$-reduct of $A^*$.

(2) The "dynamic update" operator. The 'Update' operator defined above "ignores" updates at points greater than the length of the array, i.e., if $\text{Update}(a^*, n, x) = b^*$ where $n \geq \text{Lgth}(a^*)$, then $Ap(b^*, x) = u$, not $x$. We might prefer a "dynamic" operator $\text{Update}^D$ which, in such cases, makes the required update and simultaneously increases the length of the array appropriately, thus: $\text{Update}^D(a^*, n, x) = b^*$ where $\text{Lgth}(b^*) = \max(\text{Lgth}(a^*), n + 1)$, and for all $k$.

$$Ap(b^*, k) = \begin{cases} Ap(a^*, k) & \text{if } k \neq n \\ x & \text{if } k = n. \end{cases}$$

In fact $\text{Update}^D$ can be defined from our 'Update' by:

$$\text{Update}^D(a^*, n, x) = \text{Update}(a^*, n, x)$$

where

$$a_i^* = \text{Newlength}(a^*, \max(\text{Lgth}(a), n + 1))$$

Conversely, 'Update' can be defined from 'Update$^D$' using definition by cases (see §2.1).

(3) Comparison with the definitions in book by Tucker and Zucker [19]. In that book we included addition, multiplication and predecessor among the standard operations on $\mathbb{N}$; these are unnecessary, at least in the present context. Furthermore, the definition of $A^u$ there was such that the carriers $A_i$ were "represented" by $A_i^*$ in $A^u$ (op. cit., §1.1.5 and §3.2.1, Remarks (2) and (3)). The present definition has some conceptual advantages: for example, $A$ is now a $\Sigma$-reduct of $A^u$ and $A^*$.

1.5. Mappings and Projections

We collect some definitions and notation. Consider again a structure $A$ of signature $\Sigma$.

(1) If $\vec{k} = k_1, \ldots, k_m$ ($m \geq 0$) is a list of sorts then $A[\vec{k}]$ denotes $A_{k_1} \times \cdots \times A_{k_m}$.

(2) A function on $A$ of type $(\vec{k} ; l)$ is a partial function from $A[\vec{k}]$ to $A$ (by "function" we will always mean partial function), and a relation on $A$ of type $\vec{k}$ is a subset of $A[\vec{k}]$.

(3) A tuple $k = k_1, \ldots, k_n$ is an equality type if $k_i$ is an equality sort for $i = 1, \ldots, n$.

(4) A relation on $A$ of type $\vec{k}$ is algebraic if none of its arguments is of sort $\mathbb{N}$, i.e., for $i = 1, \ldots, n$, $k_i \neq \mathbb{N}$.

(5) Given two lists of sorts $\vec{k} = k_1, \ldots, k_m$ and $\vec{l} = l_1, \ldots, l_n$, let $f_i$ be a function of type $(\vec{k}_i ; l_i)$ for $i = 1, \ldots, n$. Then the vector of functions $(f_1, \ldots, f_n)$ forms a (partial) mapping $f : A[\vec{k}] \to A[\vec{l}]$, of type $\vec{k}$.

(6) Let $\vec{k}$ be a list of sorts $k_1, \ldots, k_m$, and $\vec{i}$ a list of numbers $i_1, \ldots, i_r$ such that $1 \leq i_1 < \cdots < i_r \leq m$. Then $\vec{k}(\vec{i})$ denotes the restriction of $\vec{k}$ to $\vec{i}$. $k_{i_1}, \ldots, k_{i_r}$.

If $R$ is a relation on $A$ of type $\vec{k}$, then $\text{proj}(\vec{k}(\vec{i})R)$ is the projection of $R$
onto $\vec{i}$ (or onto the subspace $A[\vec{k} \mid \vec{i}]$), i.e., the relation
\[
\{(x_{i_1}, \ldots, x_{i_t}) \in A[\vec{k} \mid \vec{i}] \mid \exists x_{j_1}, \ldots, x_{j_t} \in A[\vec{k} \mid \vec{j}] : R(x_1, \ldots, x_m)\}
\]
of type $\vec{k} \mid \vec{i}$, where $\vec{j} = j_1, \ldots, j_t$ lists $\{1, \ldots, m\} \setminus \vec{i}$.

2. COMPUTABLE FUNCTIONS

In this section, we review, for the most part, work in Chapter 4 of [19]. We use induction schemes $\alpha$ (over $\Sigma$) to define functions $\alpha^A$ over $A$, or, more generally, families of functions $\{\alpha^A \mid A \in \mathbb{A}\}$ uniformly over $\mathbb{A}$. These induction schemes generalize the schemes for partial recursive functions over $\mathbb{N}$ in [10].

2.1. Model with Bounded Memory: Inductive Computability

The induction schemes for this class define, on each $A \in \mathbb{A}$, (partial) functions as follows:

(i) Initial functions and constants corresponding to all the operations of $\Sigma$.

(ii) Projection:
\[
f(\vec{x}) = x_i
\]
of type $(\vec{k}; k_i)$, where $\vec{x}$ is a tuple of variables of sorts $\vec{k}$.

(iii) Composition:
\[
f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x}))
\]
where $g_1, \ldots, g_m$ and $h$ are cov-inductive computable (of suitable type).

(iv) Definition by cases:
\[
f(b, x, y) = \begin{cases} 
x & \text{if } b = \mathtt{tt} 
y & \text{if } b = \mathtt{ff}
\end{cases}
\]
of type $(\mathbb{B}, k, k; k)$, where $x$ and $y$ are variables of sort $k$.

(v) Simultaneous primitive recursion on $\mathbb{N}$: This defines, on each $A \in \mathbb{A}$, for fixed $m > 0$ (the degree of simultaneity), $n \geq 0$ (the number of parameters), and sorts $\vec{k} = k_1, \ldots, k_n$ and $\vec{l} = l_1, \ldots, l_m$, an $m$-tuple of functions $f = (f_1, \ldots, f_m)$ with $f_i$ of type $(\mathbb{N}, \vec{k}; l_i)$, such that for all $\vec{x} \in A[\vec{k}]$ and $i = 1, \ldots, m$,
\[
f_i(0, \vec{x}) = g_i(\vec{x})
f_i(z + 1, \vec{x}) = h_i(z, \vec{x}, f_1(z, \vec{x}), \ldots, f_m(z, \vec{x}))
\]
where for $i = 1, \ldots, m$, $g_i$ and $h_i$ are computable (of suitable type).

(vi) Least number operator:
\[
f(\vec{x}) = \mu z [g(\vec{x}, z) = \mathtt{tt}]
\]
of type $(\vec{k}; \mathbb{N})$, where $g$ is a cov-inductive function of type $(\vec{k}, \mathbb{N}; \mathbb{B})$. Here $f(\vec{x}) \downarrow z$ if, and only if, $g(\vec{x}, y) \downarrow \mathtt{tt}$ for each $y < z$ and $g(\vec{x}, z) \downarrow \mathtt{tt}$.

The last two schemes are the interesting ones, which use the standardness of the structures, i.e., the carrier $\mathbb{N}$. (In fact, the schemes of definition-by-cases and least number operator also use the carrier $\mathbb{B}$.)
We have not given the exact coding of syntactic schemes (or Gödel numbering) corresponding to each of the above defining principles. The exact coding is unimportant; for definiteness, we may take the one in [19], §4.1.5.

It turns out, however, that this is not the class we are looking for (for reasons that will become clear later). We need a broader class.

2.2. Models with Unbounded Memory: Cov-Inductive Computability

The induction schemes described above are modified by replacing the scheme (v) for simultaneous primitive recursion by a scheme (v') for simultaneous cov ('course-of-values') recursion. This defines, on each $A \in \mathbb{X}$, for fixed $m > 0$ (the degree of simultaneity), $d > 0$ (the degree of the cov recursion), and $n \geq 0$ (the number of parameters), and sorts $k = k_1, \ldots, k_n$ and $l = l_1, \ldots, l_m$, an $m$-tuple of functions $f = (f_1, \ldots, f_m)$ with $f_i$ of type $(N, k, l_i)$ such that for all $x \in A[k]$ and $i = 1, \ldots, m$,

$$f_i(0, x) = g_i(\bar{2})$$

and for $z > 0$,

$$f_i(z, x) = h_i(z, x, f_i(\hat{h}_1(z, x), x), \ldots, f_m(\hat{h}_d(z, x), x), \ldots, f_1(\hat{h}_1(z, x), x), \ldots, f_m(\hat{h}_d(z, x), x))$$

where for $i = 1, \ldots, m$, $g_i$ and $h_i$ are cov-computable (of suitable type), and for $i = 1, \ldots, d$, $\hat{h}_i$ is defined by

$$\hat{h}_i(z, x) = \min(h_i(z, x), z - 1).$$

for some cov-computable $h_i$, so that for $z > 0$, $\hat{h}_i(z, x) < z$.

(Over $\mathbb{X}$, all these schemes are equivalent to simple primitive recursion: see [10, §46].)

The class of functions thus characterized is more satisfactory, in the sense that (i) many of the theorems of classical recursion theory hold with this class, for example, a universal function theorem, recursion theorem and $S_n^m$ theorem; and (ii) many important independent models of computation can be characterized in terms of them. We were therefore led to formulate a generalized Church-Turing Thesis for this class of functions (see below).

First, however, we wish to consider one more notion of computability (not discussed explicitly in [19]).

2.3. New Model with Unbounded Memory: Star-Inductive Computability

A function on $A$ is star-inductively computable (''star computable'' for short) if it is defined by an induction scheme over $\Sigma^*$, interpreted on $A^*$ (i.e., using starred sorts in its definition).

Theorem. Let $f$ be a function on $A$. Then

$f$ is cov-inductively computable $\iff$ $f$ is star-inductively computable,

uniformly effectively over $\mathbb{X}$. 
This will follow from equivalences which will be given in §4.1.
In brief, this shows the equivalence of ‘cov’ and ‘star’ for computable functions.

REMARK. What about iterating the star operation? A "doubly starred structure" $A^{**}$ contains, essentially, two-dimensional arrays. Such arrays can, however, be represented effectively as one-dimensional arrays, by use of the \texttt{Lgth} operation. Hence $A^{**}$ can be effectively coded in $A^*$. Thus we have:

Corollary. Cov-inductive computability and inductive computability coincide on $A^*$.

PROOF. A function on $A^*$ which is cov-inductively computable on $A^*$ is star-inductively computable on $A^*$ (by the Theorem applied to $A^*$), i.e., inductively computable on $A^{**}$, and hence (by the above Remark) inductively computable on $A^*$. 

### 2.4. Generalized Church-Turing Thesis for Deterministic Computation

In Chapter 4 of [19], the cov inductively definable functions, and their equivalents, were examined as possible formalizations of "effective calculability" over abstract data types. It was argued that "effective calculability" is ill-defined as an informal idea, when generalized; but that the informal ideas of deterministic computation and operational semantics are meaningful and equivalent in an abstract setting. This led to the postulation of a generalized Church-Turing Thesis, which (in the present context) can be formulated as follows.

Generalized Church-Turing Thesis for Computation. Computability of functions on standard structures by deterministic algorithms can be formalized by cov-inductive computability or star-inductive computability.

Note again that any structure or class of structures can be standardized and "starred."

Further equivalent models of computation, in support of this thesis, will be given in §4.

In Part B, we will give a second version of this thesis for specification, involving the types of formalism considered below for relations.

### 2.5. Two Facts about Inductive Computability

We state here two results from Chapter 4 of [19] that will be needed later. The first concerns term evaluation.

**Proposition 1.** Term evaluation on $A$ is uniformly cov inductively computable on $A$.

More precisely: Fix a list of variables $\tilde{w}$ of sorts $j_1, \ldots, j_r$. Let $\text{Term}(\tilde{w})$ be the class of program terms over $\Sigma$ with variables among $\tilde{w}$ only, and let $\tilde{t} \in \tilde{w}$ denote the Gödel number of the term $t \in \text{Term}(\tilde{w})$. Then for $i = 1, \ldots, r$, the map $t_{E}^{A_i}: \mathbb{N} \times A[\tilde{k}] \rightarrow A_{j_i}$, where $t_{E}^{A_i}(i, \tilde{a})$ is the value of $t$ when $\tilde{w}$ is evaluated as $\tilde{a}$, is cov inductively computable, uniformly for $A \in \mathcal{X}$.

By the equivalence of 'cov' and 'star', we also have:
Proposition 1*. Term evaluation on A is uniformly star inductively computable on A.

The second (obvious) proposition concerns the classical partial recursive functions on N [10].

Proposition 2. Any partial recursive function on N is inductively computable, hence (certainly) cov and star inductively computable, on A.

3. NOTIONS OF SEMICOMPUTABILITY

Our main topic of investigation in this paper is semicomputability of relations, intended to generalize to A (and N) the notion of recursive enumerability over N. In each of the following three subsections, we will consider three different notions of semicomputability (all nine coinciding in the classical case over N). Let R be a relation on A.

3.1. Semicomputable Relations

Definition 1. R is semicomputable iff R is the domain of an inductively computable function.

This is the most basic notion. We have a version of Post’s Theorem:

Proposition 1. A set on A has inductively computable characteristic function iff both it and its complement are semicomputable.

Definition 2. R is projectively semicomputable iff R is a projection of a semicomputable relation.

Definition 3. R is range-semicomputable iff R is the range of an inductively computable mapping.

(A mapping is said to be computable if its component functions are.)

We can define a system of indices or “relation schemes” for each of these notions of semicomputability. For instance, we can take the index of a semicomputable relation to be simply the index (i.e., scheme) of the computable function of which it is the domain; the index of a projective semicomputable relation to be the index of such a function together with the tuple of coordinates along which it is projected; and the index of a range-semicomputable relation to be the tuple of indices of functions comprising the mapping of which it is the range.

Such an index actually defines a family of relations on N.

The three concepts of semicomputability coincide, of course, in the classical theory over N. In the abstract case, the second and third are equivalent, assuming the relation has equality type. (This was defined in §1.5(3).)

Proposition 2. Let R be a relation on A with equality type. Then

R is projectively semicomputable ↔ R is range-semicomputable,
uniformly effectively over N.
Actually, all the equivalences proved in this paper will be uniform effective over \( \mathcal{N} \), in the sense of defining effective transformations between indices.

PROOF.

\((\Rightarrow)\) Let \( R = \text{proj}[\tilde{k} | \tilde{l}](\text{dom}(f)) \), where \( f \) is inductively computable, of type \((\tilde{k}; \tilde{l})\).

Let us take a special case, for notational simplicity. Suppose \( \tilde{k} = k_1, \ldots, k_5 \) and \( \tilde{l} = (1, 2, 3) \). So

\[
R = \{(x_1, x_2, x_3) | \exists x_4, x_5. f(x_1, \ldots, x_5) \}\.
\]

Now we must find computable \( \tilde{g} = g_1, g_2, g_3 \) such that \( R = \text{ran}(\tilde{g}) \). Here is an informal algorithm for \( g_i \) of type \((k; k_i)\) \((i = 1, 2, 3)\).

With \text{input} \( \tilde{x} \):

If and when \( f(\tilde{x}) \downarrow \), output \( x_i \).

Then \( g_i \) is inductively computable. This follows from the Generalized Church-Turing Thesis. However, we can give an induction scheme for \( g_i \), based on those for \( f \), using definition by cases, and the equality operator on \( \mathbb{N} \):

\[
g_i(\tilde{x}) = \begin{cases} 
  x_i & \text{if zero}(f(\tilde{x})) = 0 \\
  x_i & \text{otherwise}
\end{cases}
\]

where \text{zero} is a function of suitable type with constant value 0.

\((\Leftarrow)\) Suppose, conversely, that \( R \), of type \( \tilde{k} \), is the range of some computable mapping \( \tilde{g} \), of type \((\tilde{l}; k)\). Note first that the graph of \( \tilde{g} \) is semicomputable, since it is the domain of the computable function \( h \) of sort \((k, l; \mathbb{N})\) defined by

\[
h(\tilde{x}, \tilde{y}) = \begin{cases} 
  0 & \text{if } \tilde{g}(\tilde{y}) = \tilde{x} \\
  \uparrow & \text{otherwise}
\end{cases}
\]

where '\( \uparrow \)' (divergence) is inductively computable as \( \mu z[\text{false}] \). (Note the testing of equality on the type of \( R! \))

Further, \( R \) is a projection of the graph of \( \tilde{g} \), since

\[
R = \{\tilde{x} | \exists \tilde{y}. (\tilde{g}(\tilde{y}) = \tilde{x})\},
\]

from which the result follows.

What about the relationship between "ordinary" and projective semicomputability? In general, these are \textit{not} equivalent. We return to this topic in the next subsection. However the special case of existential quantification over \( \mathcal{N} \) can be dealt with now:

**Proposition 3.** Suppose \( R \subseteq A[\tilde{k}; \mathbb{N}] \) is semicomputable. Then so is its projection on \( A[\tilde{k}] \), that is, \( \{\tilde{x} | \exists z R(\tilde{x}, z)\} \).

**PROOF.** As in the classical case, we can effectively "search" for the existentially quantified \( z \) by means of the \( \mu \) operator: Suppose \( R = \text{dom}(f) \), where \( f \) is semicomputable. Then its projection on \( \mathcal{N} \) is the domain of the semicomputable function \( g \), given by

\[
g(\tilde{x}) = f(\tilde{x}, \mu z[\text{zero}(f(\tilde{x}, z)) = 0]).
\]
3.2. Cov-Semicomputable Relations

Now let us repeat the work of the last subsection, replacing ‘simple’ by ‘cov’ semicomputability. Let $R$ be a relation on $A$.

**Definition 1'**. $R$ is **cov-semicomputable** iff $R$ is the **domain** of a cov-inductively computable function.

Again we have, for this notion, a version of Post’s Theorem:

**Proposition 1'**. A set on $A$ has cov-computable characteristic function iff both it and its complement are cov-semicomputable.

**Definition 2'**. $R$ is **projectively cov-semicomputable** iff $R$ is a projection of a cov-semicomputable relation.

**Definition 3'**. $R$ is **range-cov-semicomputable** iff $R$ is the range of a cov-semicomputable mapping.

Now all the results in §3.1 carry over to this framework—e.g., the second and third concepts defined above coincide:

**Proposition 2'**. Let $R$ be a relation on $A$ with equality type. Then

$R$ is projectively cov-semicomputable $\Leftrightarrow$ $R$ is range-cov-semicomputable,

uniformly effectively over $\mathbb{R}$.

We return to the relationship between “ordinary” and projective cov-semicomputability. In general, these are not equivalent:

**Counterexample.** Let $A$ be the field of reals, and consider the relation $R(x, y) \equiv_{df} x = y^2$ on $A$. Assume the reals form an equality sort. Then $R$ is semicomputable, but the projection of $R$ on the first argument:

$$\{ x \mid \exists y (x = y^2) \}$$

i.e., the set of all nonnegative reals, is not semicomputable, or even cov-semicomputable. The proof of this depends on the following result:

**Lemma 1**. Any cov-semicomputable subset of the field of reals is either countable or cofinite in that field.

The proof of this, which depends on the notion of computation trees, is postponed until §4.5.

On the other hand, the following Selection Lemma gives a sufficient condition for the two concepts to coincide on an abstract structure $A$.

**Lemma 2** (Selection Lemma). Let $R \subseteq A[\vec{k}, \vec{l}]$ be cov-semicomputable. Then the following are equivalent:

(i) There is a cov-computable mapping $\hat{f} : A[\vec{k}] \rightarrow A[\vec{l}]$ (a “selection function”)
such that for all $\bar{x} \in A[\bar{k}]$:
\[ \exists \bar{y} \in A[\bar{l}] \ R(\bar{x}, \bar{y}) = R\left(\bar{x}, \bar{f}(\bar{x})\right), \]

(ii) for all $\bar{x} \in A[\bar{k}]$:
\[ \exists \bar{y} \in A[\bar{l}] \ R(\bar{x}, \bar{y}) = \exists \bar{y} \in A[\bar{l}] \left(R(\bar{x}, \bar{y}) \text{ and } y \in (\bar{x})\right). \]

(Here $\langle \bar{x} \rangle$ denotes the substructure of $A$ generated by $\bar{x}$.)

**Proof.**

(i) $\Rightarrow$ (ii): By structural induction on the induction scheme for $f$. In the case of simultaneous primitive recursion, use a secondary induction on the numerical argument.

(ii) $\Rightarrow$ (i): This uses the cov-inductive computability of term evaluation (§2.5, Proposition 1). By means of this and the least number operator, we can effectively search in $(\bar{x})$ for a solution $\bar{y}$ to the relation $R(\bar{x}, \bar{y})$.

**Corollary 1.** If either of the conditions in Lemma 2 holds, then the projection of $R$ on $A[\bar{k}]$ is cov-semicomputable.

Taking the special case $\bar{l} = N$ (since $\emptyset$ is included in any substructure of $A$), we obtain closure of cov-semicomputable relations under existential qualification over $\emptyset$ (as with semicomputable relations: see Proposition 3 in §3.1).

As another example, on minimal structures (i.e., structures in which all elements of the carriers are named by closed terms), the two concepts of ordinary and projective cov-semicomputability coincide:

**Corollary 2.** If $A$ is minimal, then

$R$ is cov-semicomputable $\iff$ $R$ is projectively cov-semicomputable.

### 3.3. Star-Semicomputable Relations

Now let us again repeat the work of the last subsection, replacing 'cov' by 'star' semicomputability. Let $R$ be a relation on $A$.

**Definition 1*.** $R$ is star-semicomputable iff $R$ is the domain of a star-inductively computable function.

**Remark 1.** We can give a "structural" characterization for star-semicomputable relations, among those which are algebraic on strictly standard structures (Corollary of Engeler's Lemma, Version 2, in §4.4).

In any case, we have for this notion, a version of Post's Theorem:

**Proposition 1*.** A set on $A$ has star-computable characteristic function iff both it and its complement are star-semicomputable.

**Definition 2*.** $R$ is projectively star-semicomputable iff $R$ is a projection (in $A$) of a semicomputable relation on $A^*$. 
**Definition 3*.** $R$ is **range-star-semicomputable** iff $R$ is the range (in $A$) of an inductively computable mapping on $A^*$.

Note that 'star' is again equivalent to 'cov' in the case of the first definition ("'simple'" semicomputability):

**Theorem.** Let $R$ be a relation on $A$. Then

\[ R \text{ is cov-semicomputable } \Rightarrow \text{ R is star-semicomputable, uniformly effectively over } \mathbb{K}. \]

**Proof.** This is immediate from the Theorem in §2.3. □

**Remark 2.** However 'star' is stronger than 'cov' in the cases of the second and third definitions ("'projective'" and "'range'" semicomputability), since in these cases we are projecting off starred (as compared to unstarred) sorts. (Intuitively, this corresponds to existentially quantifying over a finite, but unbounded, sequence of elements.) An example of a relation that is projectively star-semicomputable but not projectively cov-semicomputable is given in [24].

In any case, all the results in §3.2 adapt nicely to a starred framework—e.g., the second and third concepts defined above coincide:

**Proposition 2*.** Let $R$ be a relation on $A$ with equality type. Then

\[ R \text{ is projectively star-semicomputable } \Rightarrow \text{ R is range-star-semicomputable, uniformly effectively over } \mathbb{K}. \]

Again, this notion is stronger than "'ordinary'" star semicomputability, with the same counterexample (the non-negative reals) as before:

**Lemma 1*.** Any star-semicomputable subset of the field of reals is either countable or co-finite in that field.

By the equivalence of cov and star semicomputability, this is just a restatement of Lemma 1 of §3.2. It will be proved in §4.5.

Similarly, there is a starred version (Lemma 2*) of the Selection Lemma, which uses the star-inductive computability of term evaluation (§2.5):

**Corollary 2*.** If $A$ is minimal, then

\[ R \text{ is star-semicomputable } \Rightarrow \text{ R is projectively star-semicomputable.} \]

### 3.4. Some Other Notions of Semicomputability

We briefly mention here two related notions.

#### 3.4.1. Projective Computability

A relation $R$ on $A$ is **projectively computable** if it is a projection of a computable relation on $A$, i.e., a relation whose characteristic function is computable. Similarly for projective cov and star computability. In [22] we showed that (on equality types) projective star computability is equivalent to projective star semicomputability.
3.4.2. $\Sigma^*_i$ Definability

Consider the first-order language over $\Sigma^*_i$. A **bounded quantifier** has the form $\forall k < t$ or $\exists k < t$, where $t$ is a term of sort $N$; an **elementary formula** is one with only bounded quantifiers; and a $\Sigma^*_i$ **formula** is formed by prefixing an elementary formula with a string of existential quantifiers and bounded universal quantifiers (in any order). (Such formulas were used for proof-theoretical investigations in [21] and [23].)

$\Sigma^*_i$ definability is also equivalent to projective star semicomputability (on equality types). This is actually implicit in an expressibility theorem for ‘while’ computable functions in [19], §3.5.2. Again, details are given in [22].

3.5. Looking Ahead

In Part B we will investigate which (if any) of the four main notions considered in this section: simple semicomputability, cov (= star) semicomputability, projective cov semicomputability and projective star semicomputability, corresponds to the notion of “effective specifiability,” as determined by Horn clause definability.

But first we present, in the next section, another approach to the computability of functions and semicomputability of relations, by means of ‘while’ programs, which involves an analysis of initialization of variables.

4. DEFINABILITY OF FUNCTIONS AND RELATIONS BY IMPERATIVE PROGRAMS

4.1. Computability of Functions by i/o-Programs

An **i/o-program** over $\Sigma$ is defined to be a triple $[S, \bar{v}, w]$, consisting of a deterministic program $S$ in some programming language over $\Sigma$, together with a list of **input variables** $\bar{v}$ and an **output variable** $w$ (of sorts $k$ and $l$, say). Such a triple defines (in an obvious way) a function $[S, \bar{v}, w]$ on $A$ of type $(k; l)$, or a family of such functions on $\mathbb{A}$.

Note that there may also be **auxiliary variables** in $S$ (distinct from the input and output variables), which we assume to be completely uninitialized.

The i/o-program $[S, \bar{v}, w]$ is assumed to be $\mathcal{A}$-**functional**, which means that on any $A \in \mathcal{A}$, and for any values of the input variables $\bar{v}$ on $A$, the program will (deterministically) either halt, with a given output, or diverge (leading to an undefined value for the function at that argument), but never abort.

To clarify this: we assume that all variables, apart from the input variables, are **uninitialized**, i.e., have the ‘unspecified value’ $u$ at the start of execution. If, during execution, an expression was called which contains such a variable, still uninitialized, then the program would abort, or halt in an “error state.” So an i/o-program is said to be functional if this cannot occur for any initialization of the input variables.

**Remark.** Since $[S, \bar{v}, w]$ is $\mathcal{A}$-functional, it will define the same family of functions on $\mathcal{A}$ if some or all of the auxiliary variables are initialized arbitrarily, by the **monotonicity property** of programs [19], §2.2.8).
4.1.1. ‘while’-Computability

Now consider a ‘while’ programming language. For convenience, we take the syntax as in Chapter 2 of [19], and repeat the definitions here. First we define the class of program terms $t^i, \ldots$ of sort $i$ over $\Sigma$:

$$t^i ::= \text{skip} \mid a^i[t^N] \mid \text{if } b \text{ then } t^i \text{ else } t^i \text{ fi}$$

where $F$ is a function symbol in $\Sigma$ of type $(i_1, \ldots, i_m; i)$, and $b$ is a term of sort $B$.

(The sort superscript is often dropped.)

It is worth considering the particular case $i = B$, namely the class of program terms of sort $B$ or boolean terms, denoted either $t^B, \ldots$ (as above) or $b, \ldots$:

$$b ::= \text{true} \mid \text{not } b \mid (b_1 \text{ and } b_2) \mid \text{if } b \text{ then } b' \text{ else } b' \text{ fi}$$

where $F$ is a function symbol of $\Sigma$ with range sort $B$, and $i$ is any sort of $\Sigma$.

Remark. Note that because of the ‘if ... fi’ construction, the set of closed program terms over $\Sigma$ is more extensive than the set of closed terms over $\Sigma$ as usually defined. We will see examples of the usefulness of the ‘if ... fi’ construction later.

Now we define the class of ‘while’ program statements $S, \ldots$ over $\Sigma$:

$$S ::= \text{skip} \mid v^i := t^i \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } b \text{ do } S \text{ od}$$

where $b$ is a term of sort $B$.

This approach to the definition of computable functions is equivalent to that in §2.1:

Theorem. Let $f$ be a function on $A$. Then

$f$ is ‘while’-computable iff $f$ is inductively computable.

In fact there are uniform effective transformations between induction schemes and ‘while’ programs, which define the same families of functions on $\mathbb{R}$.

The proof is given in [19].

4.1.2. ‘while’-Array Computability

Suppose now that the ‘while’ programs can contain indexed variables. (Of course, the input and output variables will be simple; only the auxiliary variables may be indexed.)

We could formalize the notion of finite array in two ways: (i) exactly as in [19], in which arrays are functions on $\mathbb{N}$ with values equal to $u$ almost everywhere; or (ii) as above (§1.3), where arrays are taken as pairs, including explicit upper bounds or ‘lengths.’ Both these approaches turn out to be equivalent, at least with regard to defining functions. The first approach seems to be more convenient. So the syntax is now

$$t^i ::= \ldots \mid a^i[t^N]$$

$$S ::= \ldots \mid a^i[t^N] := t^i$$

where $a^i$ is an array variable of sort $i$ and the index $t^N$ is of sort $N$.

Note that the input and output variables of an i/o-program are still simple; only the auxiliary variables may be indexed.
Again, this approach matches that in §2.2:

**Theorem.** Let $f$ be a function on $A$. Then

$f$ is ‘while’-array-computable $\iff$ $f$ is cov-inductively computable.

(Again, this is uniform effective over $\mathbb{K}$.)

The proof of this is also given in [19].

### 4.1.3. ‘while’-Star Computability

A function on $A$ is ‘while’-star-computable if it is computable by a ‘while’ program on $A^\ast$. (Of course, the input and output variables will be of unstarred sorts; only the auxiliary variables may be of starred sorts.)

Once again, this notion coincides with that in §2.3, by the Theorem in 4.1.1 (applied to $A^\ast$ or $A^\mathbb{K}$):

**Theorem.** Let $f$ be a function on $A$. Then

$f$ is ‘while’-star-computable $\iff$ $f$ is star-inductively computable,

uniform effectively over $\mathbb{K}$.

Next, we want to show that cov-inductive computability on $A$ is equivalent to star-inductive computability on $A$. By the Theorems in §4.1.2 and §4.1.3, it is sufficient to show that ‘while’-array programs on $A$ define the same functions as simple ‘while’ programs on $A^\ast$:

**Lemma.** If $f$ be a function on $A$. Then

$f$ is ‘while’-array computable $\iff$ $f$ is ‘while’-star computable,

uniform effectively over $\mathbb{K}$.

**Proof (Outline).** The basic idea is very simple—starred variables can be viewed as finite arrays, and an assignment to (or test on) a starred variable can be simulated by a finite sequence of assignments to (or tests on) indexed variables, with index ranging (or looping) over the length of the array. However the details are rather tricky. The main problem is that $A^\ast$ is built up over $A^\mathbb{K}$, not over $A$, and the Instantiation Assumption must be used. The proof is given elsewhere in full.

The Theorem stated in §2.3 ("the equivalence of cov and star") follows immediately from this lemma and the Theorems in §4.1.2 and §4.1.3 above.

**Remark.** We have so far shown the following equivalences, in support of the Generalized Church-Turing Thesis for Computation (§2.4):

- cov-inductive computability
- star-inductive computability
- ‘while’-array computability
- ‘while’-star computability.
4.2. Definability of Relations by i-Programs

An i-program (input program) is defined to be a pair \([S, \vec{v}]\) consisting of a deterministic program \(S\) in the language of \(\Sigma\), together with a list of input variables \(\vec{v}\) (but no output variables). Such a pair defines a relation on \(A\), namely the halting set of \([S, \vec{v}]\) on \(A\), that is, the set \(\vec{a}\) of tuples of elements of \(A\), such that when \(\vec{v}\) is initialized to \(\vec{a}\) then \(S\) halts.

There may also be auxiliary variables in \(S\) (distinct from the input variables), which we assume to be completely uninitialized.

The i-program \([S, \vec{v}]\) is assumed to be \(\mathbb{R}\)-relational, which means that on any \(A \in \mathbb{R}\), and for any values of the input variables on \(A\), the program will (deterministically) either halt or diverge, but never abort. (This is analogous to \(\mathbb{R}\)-functionality of i/o-programs defined above.)

Remark 1. Hence, again, such an i-program will define the same halting set on \(\mathbb{R}\) if some or all of the auxiliary variables are initialized arbitrarily, by the monotonicity property of programs (see the Remark in §4.1).

Let us review some terminology. A program variable is called (i) simple if it is a simple variable of some sort in \(\Sigma\); (ii) indexed if it is an indexed or subscripted variable of some sort in \(\Sigma\); and (iii) starred if it is a simple variable of some starred sort in \(\Sigma^*\).

We will assume that \(S\) is a 'while' program, with or without indexed or starred variables. So \(S\) may be a 'while' or 'while'-array program over \(A\), or a 'while' program over \(A^*\). However, we will always assume that the input variables \(\vec{v}\) are simple, so that in all cases \(S\) defines a relation on \(A\).

Definitions. Let \(R\) be a relation on \(A\), defined by \([S, \vec{v}]\).

(1) \(R\) is 'while' definable (on \(A\), by \([S, \vec{v}]\)) if \(S\) is a 'while' program over \(A\).

(2) \(R\) is 'while'-array definable if \(S\) is a 'while'-array program over \(A\) in which the auxiliary variables may be indexed.

(3) \(R\) is 'while'-star definable if \(S\) is a 'while' program over \(A^*\) in which the auxiliary variables may be starred.

As simple consequences of the theorems in §4.1, we have:

Corollary. Let \(R\) be a relation on \(A\).

(1) \(R\) is 'while' definable \(\Rightarrow\) \(R\) is semicomputable.

(2) \(R\) is 'while'-array definable \(\Rightarrow\) \(R\) is cov-semicomputable.

(3) \(R\) is 'while'-star definable \(\Rightarrow\) \(R\) is star-semicomputable.

Remark 2. We know from the Lemma in §4.1.3 that (2) and (3) coincide. Also, these are (apparently) stronger than (1), although we have been unable to prove this.

4.3. Definability of Relations by 'while' Programs with Initialization

We now introduce a new feature: definability with the possibility of arbitrary initialization of search variables.

An i/s-program (input program with search variables) is defined to be a triple
[S, \bar{v}, \bar{z}] consisting of a deterministic program S in the language of \Sigma, together with a list of input variables \bar{v} and search variables \bar{z}. The relation defined by such a triple on A is the set \bar{x} of tuples of elements of A, such that when \bar{v} is initialized to \bar{x} then for some (nondeterministic) initialization of \bar{z}, S halts.

Again, there may also be auxiliary variables in S (distinct from the input and search variables), which we assume to be completely uninitialized.

And again, the i/s-program [S, \bar{v}, \bar{z}] is assumed to be \mathcal{R}-relational, which means that on any A \in \mathcal{R}, and for any values of the input and search variables on A, the program will (deterministically) either halt or diverge, but never abort.

Definitions. Let R be a relation on A, defined by [S, \bar{v}, \bar{z}].

(1.) R is 'while' definable with initialization (on A, by [S, \bar{v}, \bar{z}]) if S is a simple 'while' program over A.

(2.) R is 'while'-array definable with initialization if S is a 'while'-array program over A in which the search variables are simple, but the auxiliary variables may be indexed.

(3.) R is 'while'-star definable with initialization if S is a 'while' program over \mathcal{A} in which the search and auxiliary variables may be starred.

As further simple consequences of the theorems in §4.1, we have:

Corollary. Let R be a relation on A.

(1.) R is 'while' definable with initialization \iff R is projectively semicomputable.

(2.) R is 'while'-array definable with initialization \iff R is projectively cov-semi-computable.

(3.) R is 'while'-star definable with initialization \iff R is projectively star-semi-computable.

REMARK. To compare these three classes of relations with one another and with the three in §4.2: we know that (1) is strictly stronger than (2) and (3) of §4.2 (by the counterexample given in §3.2). Also (3) is stronger than (2), since (in the former case) we are initializing with, or projecting off, starred, as compared to simple, variables (see Remark 2 in §3.3). We do not know whether (2) is stronger than (1). (Compare Remark 2 in §4.2.)

One may ask how these classes of relations compare with the class defined when we permit, instead of nondeterministic initialization of variables, nondeterministic assignments (of the form \( v := ? \)) throughout the program. This will be discussed in §4.6 below.

4.4. Engeler's Lemma

One can define, for any 'while' program S over \Sigma, and vector \bar{v} of program variables such that \text{var}(S) \subseteq \bar{v}, the computation tree \mathcal{T}[S, \bar{v}], which is like an "unfolded flow chart" of S. (Full details are given in [22].)

For each leaf \lambda of this tree, there is a boolean term B[S, \bar{v}, \lambda], with free variables among \bar{v}, which expresses the conjunction of results of all the successive tests, that (the
current values of) the variables $\bar{v}$ must satisfy in order for the computation to "follow" the finite path from the root of the tree to $\lambda$.

Then if $\langle \lambda_j \mid j \geq 0 \rangle$ is an effective enumeration of the leaves of $\mathcal{T} [S, \bar{v}]$, the halting predicate $\text{Halt}[S, \bar{v}]$ of $S$ with respect to $\bar{v}$ is the infinite disjunction

$$\bigvee_{j=0}^{\infty} B[S, \bar{v}, \lambda_j]$$

which expresses that execution of $S$ eventually halts, if started in the initial state (represented by) $\bar{v}$.

Furthermore, the predicate $B[S, \bar{v}, \lambda_j]$ is effective in $S$, $\bar{v}$ and $j$. This means that there is a partial recursive function $\beta$ of three arguments such that $\beta(\bar{S}, [\bar{v}], j)$ is the Gödel number of $R[S, \bar{v}, \lambda_j]$. So $S$ halts on the initialization $\bar{x} \in A[k]$ iff for some $j$,

$$\text{te}_A \left( \beta(\bar{S}, [\bar{v}], j), \bar{x} \right) \cup \text{tt},$$

where $\text{te}_A$ is the term evaluation function on $A$, which is cov inductively computable (by Proposition 1 in §2.5).

This gives us (by §4.2, Corollary 1) the following form of Engeler's Lemma [4]:

**Theorem 1** (Engeler's Lemma, Version 1). Let $R$ be a semicomputable relation on a $\Sigma$-structure $A$. Then $R$ can be expressed as an effective (infinite) disjunction of boolean terms over $\Sigma$.

Now the halting set $\text{Halt}^A[S, \bar{v}]$ of $S$ on $A$ with respect to $\bar{v}$ is the set of tuples $\bar{a}$ from $A$ for which the predicate $\text{Halt}[S, \bar{v}]$ evaluates to $\text{tt}$ when $\bar{a}$ is assigned to $\bar{v}$. (Note that here $\bar{v}$ includes all the program variables of $S$, unlike the case in §4.1, where $\bar{v}$ referred to the input variables only.) So

$$\text{Halt}^A[S, \bar{v}] = \left\{ \bar{a} \in A[k] \mid \text{for some } j, \text{te}_A \left( \beta(\bar{S}, [\bar{v}], j), \bar{x} \right) \cup \text{tt} \right\},$$

which shows (by the propositions in §2.5 and Proposition 3 in §3.1) that $\text{Halt}^A[S, \bar{v}]$ is cov-semicomputable—which is not very helpful, since in fact we know (by Corollary 1 in §4.2) that it is semicomputable!

Actually, we need a stronger version of Engeler's Lemma, applied to 'while'-star programs, which we now formulate.

**Theorem 2** (Engeler's Lemma, Version 2). Let $R$ be an algebraic relation on a strictly standard $\Sigma$-structure $A$. Suppose $R$ is star-semicomputable (or, equivalently, cov-semicomputable) on $A$. Then $R$ can be expressed as an effective (infinite) disjunction of boolean terms over $\Sigma$.

The proof is given in [22]. (Algebraic relations were defined in §1.5(4).) We have the following result in the converse direction.

**Proposition.** Let $R$ be a relation on the $\Sigma$-structure $A$. If $R$ can be expressed as an effective (infinite) disjunction of boolean terms over $\Sigma$, then $R$ is cov-semicomputable (or, equivalently, star-semicomputable).

**Proof.** Exercise. (Use the Propositions in §2.5 and Proposition 3 in §3.1.)
Combining Engeler's Lemma (Version 2) with this result gives the following "structural" characterization of star-semicomputable relations, among those which are algebraic on strictly standard structures.

Corollary. Let R be an algebraic relation on a strictly standard structure. Then R can be expressed as an effective (infinite) disjunction of boolean terms over Σ iff R is cov- (or star-) semicomputable.

4.5. Proof of Lemmas in Section 3

We will now prove Lemma 1* in §3.3 (or, equivalently, Lemma 1 in §3.2), which gives an example (the nonnegative reals) of a projectively semicomputable set that is not star-semicomputable.

Let \( \mathbb{R} = (\mathbb{R}, \mathbb{N}, \mathbb{B}, \ldots) \) be the field of reals \( \mathbb{R} \), with constants 0 and 1, and operations \(+, \times, \text{ and } -\), standardized by the adjunction of the sets \( \mathbb{N} \) and \( \mathbb{B} \) together with their standard operations. Let \( C \) be a star-semicomputable subset of \( \mathbb{R} \). Then \( C \) is an algebraic relation in \( \mathbb{R} \), and so, by Engeler's Lemma (Version 2), \( C \) is given by an (effective) infinite disjunction

\[
C = \left\{ x \in \mathbb{R} \left| \bigvee b_j(x) \right. \right\}
\]

where for all \( i \), \( b_j(v) \) is in \( \text{Term}_R(v) \) (for \( v \) of sort \( R \)), i.e., an unstarred boolean term with free variable \( v \) (where "unstarred" means without any starred variables).

The disjunction in (1) can be transformed by the following three steps:

(i) Any boolean term in \( \text{Term}_R(v) \) can be rewritten as a boolean combination of equations between (unstarred) terms of sort \( R \).

(ii) Any equation between unstarred terms of sort \( R \) can be rewritten as a boolean combination of polynomial equations.

(iii) Next (by writing the boolean disjuncts in disjunctive normal form, and absorbing the disjunctions into the "big disjunction" (1)), the disjunction in (1) can be rewritten as

\[
\bigvee_j b_j^0(x)
\]

where each \( b_j^0(x) \) is a finite conjunction of polynomial equations and negations of polynomial equations over \( x \in \mathbb{R} \), with coefficients in \( \Sigma \).

Now by the Fundamental Theorem of Algebra, each such polynomial equation has at most finitely many roots in \( \mathbb{R} \).

Regarding the disjunction in (2), there are two cases:

Case 1. For some \( j \), \( b_j^0(x) \) contains only negations of equations. Then (for this \( j \)) \( b_j^0(x) \) holds for all but finitely many \( x \in \mathbb{R} \). Hence \( C \) is co-finite in \( \mathbb{R} \).

Case 2. For all \( j \), \( b_j^0(x) \) contains at least one equation. Then (for all \( j \)) \( b_j^0(x) \) holds for at most finitely many \( x \in \mathbb{R} \). Hence \( C \) is countable.

This completes the proof of Lemma 1* in §3.3.
### 4.6. ‘while’ Programs with Random Assignments

We now consider the ‘while’ programming language over $\Sigma$, extended by the random assignment $v_i' := ?$ for all sorts $i$ of $\Sigma$.

Let $[S, \bar{v}]$ be an i-program (§4.2) in this language. The halting set of $[S, \bar{v}]$ on $A$, or the relation defined by $[S, \bar{v}]$ on $A$, is the set $\bar{a}$ of tuples of elements of $A$ such that if $\bar{v}$ is initialized to $\bar{a}$, then for some values of the random assignments, $S$ halts.

**Definitions.** Let $R$ be a relation on $A$, defined by $[S, \bar{v}]$.

1. $R$ is ‘while'-random definable (on $A$, by $[S, \bar{v}]$) if $S$ is a ‘while’ program over $A$ with random assignments (to simple variables).
2. $R$ is ‘while'-array-random definable if $S$ is a ‘while'-array program over $A$ with random assignments (possibly to indexed variables).
3. $R$ is ‘while'-star-random definable if $S$ is a ‘while’ program over $A^*$ with random assignments (possibly to starred variables).

**Remark 1.** Clearly, (3) implies (2) (the “easy” direction of the proof of the Lemma in §4.1.3), which trivially implies (1).

**Remark 2.** Definability with random assignments can be viewed as a generalization of the notion of definability with initialization, since initialization amounts to random assignments at the beginning of the program. We may ask how the two notions of definability compare. We will see that, at least in the case of programs over $A^*$, the two notions coincide—both are equivalent to projectively star-semicomputability.

**Theorem.** Let $R$ be a relation on $A$. If $R$ is

1. ‘while’-random definable, or
2. ‘while’-random-array definable, or
3. ‘while’-star-random definable,

then $R$ is projectively star-semicomputable.

The proof is given in detail in [22]. Briefly, we define a computation tree for a ‘while’-random program. In such a tree, as a result of ‘?’-assignments, program variables “proliferate,” but can be represented by single-starred variables.

### PART B: HORN PROGRAMS AND DEFINABILITY

#### 5. HORN PROGRAM DEFINABILITY

Let $\Sigma$ be a signature which (unlike those considered so far) includes relation symbols $R$, each of fixed type. We define the following syntactic classes (all relative to $\Sigma$):

**Terms** $t$ are defined inductively by

$$t ::= v^i | F(t_1, \ldots, t_m),$$

where $v^i$ is a variable of sort $i$, $F$ is a function symbol of type $(i_1, \ldots, i_m)$, and $t_j$ has sort $i_j$. Atomic booleans or atoms $B$ may be either equational atoms $(t_1 = t_2)$ ($t_1$ and $t_2$ of the same sort) or relational atoms $R(t_1, \ldots, t_m)$ (where $R$ has type
(i_1, \ldots, i_m) and \( t_j \) has sort \( i_j \). Ground terms, ground atoms, etc., are terms, atoms, etc., without free variables. Horn clauses \( H \) have the form \( H \equiv B \leftarrow B_1, \ldots, B_m \) \((m \geq 0)\), with head\( (H) \equiv B \). Goal clauses \( G \) have the form \( G \equiv \leftarrow B_1, \ldots, B_m \) \((m \geq 0)\).

Now let \( \Sigma_0 \) be a signature with (as before) no relation symbols. A Horn program with goal relation \( R_0 \) over \( \Sigma_0 \) is a four-tuple

\[ \mathcal{P} = (P, R_0, \Sigma_0, \Sigma) \]

where \( \Sigma \) is \( \Sigma_0 \) together with the new relation symbols \( R_0, \ldots, R_p \) \((p \geq 0)\), and \( P \) is a finite sequence of Horn clauses over \( \Sigma \), each of which has, as head, a relational atom: \( R(t_1, \ldots, t_q) \).

Substitutions \( \theta \) (over \( \Sigma \)) are defined as in [11, §4]. In addition, for a \( \Sigma \)-structure \( A \), we define a valuation over \( A \) to be a function \( \sigma \) from variables to elements of \( A \) (of the correct sort).

Let \( \mathcal{P} = (P, R_0, \Sigma_0, \Sigma) \) be a Horn program over \( \Sigma \), with \( P = (H_1, \ldots, H_n) \). Assume \( P \) includes the equality axioms for \( R_0, \ldots, R_p \), i.e., the clauses

\[ R_i(x_1, \ldots, y_j, \ldots, x_m) \leftarrow x_j = y_j, R(x_1, \ldots, x_j, \ldots, x_m) \]

(where the variables have the correct sorts) for \( i = 1, \ldots, p \) and \( j = 1, \ldots, q_i \) (the arity of \( R_i \)).

Suppose \( R_0 \) has type \( k_1, \ldots, k_q \).

A computation (of \( R_0 \)) by \( \mathcal{P} \) is a finite or infinite sequence \( c = (c_0, \ldots, c_l) \) or \( c \equiv (c_0, \theta_1, c_2, \ldots) \) of four-tuples

\[ c_i = (G_i, \theta_i, j_i, h_i) \]

with \( \text{length}(c) = l \) or \( \infty \) respectively, where, for \( i < \text{length}(c) \), \( G_i \) is a goal clause, with \( G_0 \equiv \leftarrow R_0(\bar{u}) \), where \( \bar{u} \) a list of \( q \) new variables of sorts \( \bar{k} \), not in \( P \), \( 1 \leq h_i \leq h \), \( \theta_i \) is the most general unifier of the \( j_i \)-th formula of \( G_i \) with head\( (H_i) \), and \( G_{i+1} \) is the corresponding resolvent. Here \( H_i^j \) is formed from \( H_i \) by renaming all its variables in some systematic manner. If \( c \) is finite, with length \( l \), write \( G_{end} \) for the final goal clause \( G_l \).

A computation \( c \) is semisuccessful if (i) it is finite, and (ii) \( G_{end} \) contains none of the new relation symbols.

If, moreover, for some \( \Sigma_0 \)-structure \( A \), there exists a valuation \( \sigma \) on \( \text{var}(c) \) satisfying the equational atoms of \( G_{end} \) in \( A \) (in symbols, \( A \models \wedge (G_{end}) \sigma \), where \( \wedge (G_{end}) \) is the conjunction of the atoms in \( G_{end} \)), then the pair \((c, \sigma)\) is called a successful computation of \( \mathcal{P} \) over \( A \), with answer \( u \theta_0 \ldots \theta_{l-1} \sigma \in A[k] \).

**Remarks.**

1. The purpose of \( \mathcal{P} \) is (only) to define the goal relation \( R_0 \), together with the auxiliary relations \( R_1, \ldots, R_p \) which help in defining it. The program \( \mathcal{P} \) is not being used to define the "old" functions (from \( \Sigma_0 \)) or the equality relation, which are considered to be "known" and fixed in \( A \). There is thus no need to eliminate any equational atoms. That is why we stipulate that (i) the heads of all clauses in \( \mathcal{P} \) involve only the new relations, and (ii) a computation of \( \mathcal{P} \) succeeds when we end up with a goal \( G_{end} \), which is not necessarily empty, but contains only equational atoms, which are (furthermore) satisfied in \( A \) by some valuation \( \sigma \).

2. Note the "nonstandard" aspect of our computations (compared to [11, §7]): we
obtain an answer, in a successful computation, by applying to the variables in the computation not only a sequence of substitutions \( \theta_0, \ldots, \theta_j \), but also, in the last step, a valuation \( \sigma \) satisfying the final goal, so as to obtain a tuple of elements of \( A \). Notice also that the structure \( A \) only enters in this last step.

**Definition (Relation Definable by a Horn Program).**

1. Let \( A \) be a \( \Sigma_0 \)-structure, \( P = (P, R_0, \Sigma_0, \Sigma) \) as above, and let \( R^A \subseteq A[k] \). \( R^A \) is said to be (Horn) definable by \( P \) over \( A \) if: for all \( \bar{a} \in A[k] \), \( \bar{a} \in R^A \) iff \( \bar{a} \) is the answer given by a successful computation of \( P \) over \( A \).

2. Let \( \mathfrak{A} \) be a class of \( \Sigma_0 \)-structures, and \( R = \{ R^A \mid A \in \mathfrak{A} \} \) a family of relations of type \( k \) over \( \mathfrak{A} \). \( R \) is uniformly definable by \( P \) over \( \mathfrak{A} \) if for all \( A \in \mathfrak{A} \), \( R^A \) is definable by \( P \) over \( A \).

Finally, a Horn star program over \( \Sigma_0 \) is a Horn program \( P = (P, R_0, \Sigma_0^*, \Sigma^*) \) over \( \Sigma_0^* \), with the restriction that \( R_0 \) have "unstarred" type. Horn star definability is defined as above, but with reference to such a program.

6. EQUIVALENCE BETWEEN HORN-STAR DEFINABILITY AND PROJECTIVE STAR-SEMICOMPUTABILITY

We come to the main result of this paper.

**Theorem.** Let \( R \) be a relation on \( A \). Then

\[ R \text{ is projectively star-semicomputable } \Rightarrow \text{ } R \text{ is Horn star-definable, uniformly effectively over } \mathfrak{A}. \]

**Proof.**

\((\Rightarrow)\) This is the easy direction. Suppose \( R = \text{proj}[\hat{k} | \hat{i}](\text{dom}(\alpha^A)) \), where \( \alpha \) is an induction scheme on \( \Sigma^* \), \( \hat{k} \) is a list of (possibly) starred sorts, and \( \hat{i} \) is a sublist of unstarred sorts. First construct, following the method of [16] for \( \mathfrak{F} \) (or see [11, §9]), a Horn program (over \( \Sigma^* \)) which defines the graph \( F \) of \( \alpha^A \). This is done by induction on the complexity of \( \alpha \). (We should remember that our schemes include simultaneous instead of simple recursion, but the technique in [16] still works.) We add the final line

\[ R_0(\bar{u}) \leftarrow F(\bar{w}, z) \]

where \( \bar{u} \) is a sublist of \( \bar{w} \) of sorts \( \hat{k} | \hat{i} \), corresponding to the given projection.

It is not clear (in this direction) why we need starred sorts at all! That will be clarified in considering the reverse direction.

\((\Leftarrow)\) We will use the two facts about inductive computability presented in §2.5.

So suppose \( R \) is definable by the Horn program \( P = (P, R_0, \Sigma_0^*, \Sigma^*) \), where \( R_0 \) has type \( \hat{k} \). We must find a cov induction scheme \( \alpha \) such that \( R \) is a projection of the domain of \( \alpha^A \).

Let \( \bar{u} \) be the list of variables, of sorts \( \hat{k} \), in the goal clause \( G_0 \equiv \leftarrow R_0(\bar{u}) \), and suppose all variables of \( P \) are included in the list \( \bar{v} = v_1, \ldots, v_m \), of sorts \( \hat{l} = l_1, \ldots, l_m \), disjoint from \( \bar{u} \).

Suppose first, for simplicity, that no variables in \( \bar{v} \) are starred.
Now we want to define a computable function, such that \( R \) will be some projection of its domain. What should the type of such a function be?

An immediate problem is that any computation of \( R_0 \) contains many more variables than \( \vec{u} \cup \vec{v} \), because of the renaming of variables at each step. In fact we cannot delimit \textit{a priori} the number of variables in any computation, even a finite computation. The variables proliferate without bound. However, their \textit{sorts} remain restricted! This gives us the clue: we can represent \textit{arbitrarily many} variables of a given sort by a \textit{single} variable of the corresponding \textit{starred sort}. (This is the same strategy we use in defining the halting predicate for ‘while’ programs with random assignments—see §4.6.)

Thus, in defining the function, we ‘‘replace’’ the variables \( \vec{u} \) of sorts \( \vec{I} \) by the corresponding starred variables \( \vec{u}^* = v_1^*, \ldots, v_m^* \) of sorts \( \vec{I}^* = I_1^*, \ldots, I_m^* \). So we will define a star-computable function

\[
 f: A[\vec{k}] \times A[\vec{I}^*] \times \mathbb{N} \rightarrow \mathbb{N}
\]
such that \( R \) is the projection of \( \text{dom}(f) \) on \( A[\vec{k}] \).

First, let \((c_0, c_1, \ldots)\) be an enumeration of (Gödel numbers of) all \textit{semisuccessful computations} of \( R_0 \) by \( \mathcal{P} \). This is partial recursive and hence, by Proposition 2 of §2.5, star-computable. Now, with input \( \vec{a} \in A[k], \vec{b}^* \in A[I^*], n \in \mathbb{N} : f \) tests whether the final goal clause of \( c_n \) is satisfied by the (fixed) valuation which maps the variables \( \vec{u} \) to \( \vec{a} \) and \( \vec{v}^* \) to \( \vec{b}^* \). This test is star-computable, by Proposition 1 of §2.5. If the answer is yes, halt with output 0 (say). Otherwise diverge (by searching, \textit{e.g.}, for \( \mu i [ \text{false} ] \)).

The function \( f \) thus defined is star-computable. The projection of its domain along \( A[\vec{k}] \) then gives those \( q \)-tuples \( \vec{a} \) for which there exist a semi-successful computation of \( R_0 \), and a valuation of the variables in \( \mathcal{P} \), giving a successful computation of \( R_0 \) over \( A \), with answer \( \vec{a} \), as desired.

Suppose, finally, that the variables \( \vec{u} \) occurring in \( P \) may already be of starred sorts. Then we can represent sequences of these as ‘‘doubly starred’’ variables, or two-dimensional arrays, which can, however, be effectively coded as one-dimensional arrays (§2.3, Remark).

7. EQUIVALENCES WITH OTHER MODELS OF COMPUTABILITY

We present here a few more models of computability and consider their relationship to Horn-star definability. Their mutual relationship has already been considered in [5]. The context of all these models are \textit{single-sorted} structures which are either \textit{algebraic}

\[ A = (A, F_1, \ldots, F_k, c_1, \ldots, c_l) \]

where the \( F_i \) are functions on \( A \) of fixed type and the \( c_j \) are constants, or \textit{relational}

\[ B = (B, R_1, \ldots, R_k, c_1, \ldots, c_l) \]

where the \( R_i \) are relations on \( B \) of fixed arity and the \( c_j \) constants.

Such a structure, say \( B \), is ‘‘elaborated’’ or extended to a structure \( B^* \) (where \( * \) is one of ‘\( s \), ‘\( w \), or ‘\( p \)’), in one of three ways:

1. \( (\ast = 's') \) by embedding \( B \) in the set \( B^s \) of hereditarily finite sets over \( B \), with the operation \( x, y \rightarrow x \cup \{y\} \);
2. \( (\ast = 'w') \) by embedding \( B \) in the set \( B^w \) of finite words over \( B \), with the operation of concatenation;
3. \( (\ast = 'p') \) by adjoining a new object 0 and closing under a pairing operation, to form \( B^p \).
It is shown in [5] (working in the context of relational structures) that these three extensions are essentially equivalent. The models of computability given below were all originally defined relative to one or other of these three extensions.

We will consider how these concepts translate into the context of our many-sorted, standard algebraic structures. We note, first, that Fitting's notion of 'r.e. in rec(B*)' ([5], Chapter 1, §3 and §11) corresponds closely to our notion of Horn-star definability. We therefore call it Horn-star definability in Fitting's sense. We will give the exact connection with our notion in §7.3. First we consider (in §7.1) three other models of (semi)computability of relations, and also (in §7.2) their mutual equivalence.

Throughout this section, A denotes a single-sorted algebraic structure, B a single-sorted relational structure, and A* or B* one of the three extensions described above.

7.1. Three Models of Computability

7.1.1. \( \Sigma^* \) Definability of Montague

Definitions.

(1) An atomic set formula over B is an atomic formula in the signature of B or a formula \( x \in y \) (\( x, y \) ranging over \( B^* \)).

(2) A \( \Sigma^* \) formula over B is one built up from atomic set formulae over B, using conjunction, disjunction, existential quantification and bounded universal quantification (\( \forall x \in y \)).

(3) A relation \( R \) on B is \( \Sigma^* \) definable if it is definable by a \( \Sigma^* \) formula over B.

One can similarly define, for the structure \( B^w \), the notions of atomic word formula, using \( 'z = x^* y' \) (concatenation of words) instead of \( 'x \in y' \); and similarly \( \Sigma^w \) definability, with bounded universal quantification formulated as \( \forall x \text{seg} y \), expressing that the word x is a segment (not necessarily initial) of the word y.

Montague [13] investigated \( \Sigma^* \) definability, which he called \( \kappa_0 \) recursive enumerability. \( (\Sigma^* \text{ definability in models of set theory was investigated in [1]).} \)

7.1.2. Search Computability of Moschovakis

This was investigated in [14], for algebraic structures A. He considered two notions, working in the extension \( A^p \) of A: primitive computability over A, and search computability over A, formed by adjoining a (non-constructive) unordered search operator.

Definition. A relation on A is \( \sigma^0 \) if it is a projection (i.e., formed by a single existential quantification) of a relation on \( A^p \) whose characteristic function is search computable over A.

Remark. In the above definition, 'search computable' could be replaced by 'primitive computable,' because of a normal form theorem.

7.1.3. Existential Inductive Definability of Moschovakis

This was investigated in [15].
Definitions. Let $B$ be a relational structure, and $'R'$ a new relation symbol, not in the signature of $B$.

1. $\Sigma^*(R)$ formulae over $B$ are defined like $\Sigma^*$ formulae (§7.1), except that the atomic set formulae may also contain $'R'$.
2. A relation $R$ on $B$ is existential star-inductive on $B$ if it is the fixed point of a $\Sigma^*(R)$ inductive definition over $B$.

(The terminology in (2) is ours.)

7.2. Mutual Equivalence of these Models

In order to formulate these results, we need some definitions.

Definition 1. Let $A$ be an algebraic structure. The relational version of $A$ is the relational structure $A^{rel}$ formed by replacing the functions in $A$ by their graphs (and keeping the constants).

Definition 2. Let $B$ be a relational structure. The algebraic version of $B$ is the algebraic structure $B^{alg}$ formed from $B$ by replacing the relations in $B$ by their characteristic functions (and keeping the constants).

We are being a bit sloppy here; in order to define characteristic functions, either the signature of $B$ should contain at least two constants, or we must adjoin two constants to $B$: 0 and 1 say, or true and false. These constants can in fact be defined in the extensions $B^*$ of $B$, and are present in the “standardized” version of $B^{alg}$ of $B$ (see below).

Definition 3. Let $B = (B, R_1, \ldots, R_k, c_1, \ldots, c_l)$ be a relational structure. The complementary completion of $B$ is the relational structure $\bar{B} = (B, R_1, \ldots, R_k, \bar{R}_1, \ldots, \bar{R}_k, c_1, \ldots, c_l)$, where $\bar{R}$ is the complement of $R$ in $B$.

Now, to simplify matters, we will assume (since some of the results below require it):

Assumption 1. All structures contain the relation of equality (or its characteristic function).

Then the above three notions of computability are all equivalent to Horn-star definability (in Fitting’s sense). Here are the exact results and references:

1. $\Sigma^*$ definability (of Montague) over $B$ is equivalent to $\sigma_0^0$ (of Moschovakis) over $\bar{B}^{alg}$ ([9]).
2. $\Sigma^*$ definability (of Montague) over $B$ is equivalent to Horn-star definability (of Fitting) over $B$ ([5], Chapter 6, Theorem 5.4).
3. Horn-star definability (of Fitting) over $B$ is equivalent to existential-star inductive definability (of Moschovakis) over $\bar{B}$ ([5], Chapter 1, Propositions 13.6 and 13.9).
7.3. Correspondence with Our Work

In order to formulate this, we further assume (to satisfy our own Instantiation Assumption):

Assumption 2. All structures contain at least one constant.

Definition. If $A$ is a single-sorted algebraic structure, then $A_{N,B}$ is the standardization of $A$ formed by adjoining the sets $\mathbb{H}$ and $\mathbb{B}$ and their standard operations ($\S$1.1).

Theorem.

1. Let $A$ be an algebraic structure, and $R$ a relation on $A$. Then $R$ is Horn-star definable (in Fitting's sense) over $A^{rel}$ iff $R$ is Horn-star definable (in our sense) over $A_{N,B}$.

2. Let $B$ be a relational structure, and $R$ a relation on $B$. Then $R$ is Horn-star definable (in Fitting's sense) over $B$ iff $R$ is Horn-star definable (in our sense) over $B_{N,B}^{alg}$.

The basic idea of the proof is straightforward, but the details are lengthy, and are omitted.

Note that this theorem yields equivalences of our Horn-star definability with the other notions considered above. For example, we immediately have:

Corollary 1. For any relational structure $B$, a relation on $B$ is $\sigma_1^0$ over $B_{alg}$ (in the sense of Moschovakis) iff it is Horn-star definable over $B_{N,B}^{alg}$.

And, somewhat less immediately, we have the following, more direct, formulation of this equivalence:

Corollary 2. For any algebraic structure $A$, a relation on $A$ is $\sigma_1^0$ (in the sense of Moschovakis) iff it is Horn-star definable (in our sense) over $A_{N,B}$.

A more direct analysis of the relationship between Moschovakis' search computability and star-inductive computability (or semicomputability) would be interesting.

Some interesting results on search computability over certain structures are given in [17].

8. HORN CLAUSE AXIOMATIZABLE CLASSES

Now let us consider, as a special case, Horn axiomatizable classes of structures over a given signature $\Sigma_0$, i.e., classes $\mathfrak{K} = \text{Mod}(\Sigma_0, E)$ of $\Sigma_0$-structures axiomatizable by a set $E$ of Horn clauses (for example, conditional equations). Assume $E$ includes the equality axioms for $\Sigma_0$.

Such a class has an initial model $\text{Init}(\Sigma_0, E)$ ([12]). The carriers here consist of the congruence classes of the ground terms of $\Sigma_0$ with respect to $\Sigma_0$, where the congruence relation (interpreting equality) is generated from $E$.

The problem now is that Horn axiomatizable structures need not be standard! Even the initial model might not be standard. The problem, briefly, is that there may be a
function symbol $f$ in $\Sigma_0$ with range sort $N$, without corresponding axioms in $E$ capable of "reducing" a closed term of sort $N$ containing $f$ to a numeral. (In the terminology of [25], the specification $E$ may not be "sufficiently complete.""

We will therefore consider only those classes $\text{Mod}(\Sigma_0, E)$ in which (at least) $I = \text{df init}(\Sigma_0, E)$ is standard (e.g., if $\Sigma_0$ is strictly standard), and consider $I$ itself.

Since $I$ is a minimal algebra, a relation on $I$ is projectively star-semicomputable if and only if it is star-semicomputable (by §3.3, Corollary 2*).

Furthermore, the notion of Horn definability of relations on $I$, as defined in §5, reduces to the "standard" notion (as in [11, §7]), in the following sense.

**Theorem.** Let $\mathcal{P} \equiv (P, \ldots)$ be a Horn (or Horn star) program over $\Sigma_0$. Let $E \cup P$ be the sequence of Horn clauses formed by concatenating $E$ with $P$. Let $\bar{t} = t_1, \ldots, t_q$ be a tuple of ground terms of $\Sigma_0$, of sorts $\bar{k}$ (the type of $R_0$). Let $\bar{t} = t_1, \ldots, t_q$ be the tuple of denotations of $t_1, \ldots, t_q$ in $I$ (i.e., their congruence classes, as explained above). Then there is a successful computation of $\mathcal{P}$ over $I$, with answer $\bar{t}$, iff there is an SLD-refutation of $\leftarrow R_0(\bar{t})$ from $E \cup P$, i.e., a computation of $E \cup P \cup \{ \leftarrow R_0(\bar{t}) \}$ ending with the empty clause.

**PROOF.**

$(\Rightarrow)$ (This is the simpler direction.) Consider a successful computation $(c, \sigma)$ of $\mathcal{P}$ over $I$. Note first that the valuation $\sigma$ is a mapping from variables to elements of $I$, i.e., congruence classes of ground terms of $\Sigma_0$. Let $\sigma'$ be a corresponding mapping from the same variables to representatives of these congruence classes. Then $(c, \sigma)$ can be given a standard "syntactic" interpretation $c'$ by (in the notation of §5) replacing the goal clauses $G_i$ by $G_i \theta_1 \ldots \theta_1 \sigma'$. Now since $I \models \wedge (G_{\text{end}}) \sigma'$, therefore, by the definition of $I$,

$$E \leftarrow \wedge (G_{\text{end}}) \sigma'.$$

Hence $E \cup \{ G_{\text{end}} \sigma' \}$ is unsatisfiable, and so, by the completeness of SLD-resolution ([11], Chapter 2, Theorem 8.4), there is an SLD-refutation $d$ of $G_{\text{end}} \sigma'$ from $E$. Finally, combining $c'$ and $d$ gives us the required refutation of $\leftarrow R_0(\bar{t})$ from $E \cup P$.

$(\Leftarrow)$ Suppose we have an SLD-refutation of $\leftarrow R_0(\bar{t})$ from $E \cup P$, with mgu's $\theta_1, \ldots, \theta_q$. Let $\theta$ be the substitution defined by the mapping $\bar{u} \mapsto \bar{t}$ (where $\bar{u}$ is a fresh list of variables of sorts $\bar{k}$). Then by the Lifting Lemma ([11], Chapter 2, Lemma 8.2) there is an SLD-refutation of $\leftarrow R_0(\bar{u})$ from $E \cup P$, with mgu's $\theta_1', \ldots, \theta_q'$ and a substitution $\gamma$ such that $\theta_1 \ldots \theta_q = \theta_1' \ldots \theta_q' \gamma$. Next, by the Switching Lemma ([11], Chapter 2, Lemma 9.1) we can permute the resolution steps in this refutation so that resolutions of goal clauses with clauses having relational atoms as their head all precede those with clauses having equational atoms as their head. Now let $I$ (1 \leq l \leq q) be such that $G_i$ is the first goal clause without any relational atoms. Then we can construct a semi-successful computation $c \equiv (c_0, \ldots, c_l)$ where $c_i \equiv (G_i, \theta_i', j_i, h_i)$ (still in the notation of §5). Put $G_{\text{end}} = G_l$ and $\sigma' = \theta_l' \ldots \theta_q' \gamma$. Finally, let $\sigma$ be the valuation on $I$ which corresponds (in an obvious way) to $\sigma'$, i.e., it maps each variable $u$ in $c$ to the congruence class of the ground term $\sigma'(u)$. Then $(c, \sigma)$ is the required successful computation of $\mathcal{P}$ over $I$. \[\blacksquare\]
This result should be compared with Theorem 1 in [7].

The user's facility to model, specify, and program, using abstract data types, can be added to a conventional logic programming language and implemented by compiling into "standard" Horn clause programs for term structures and employing established techniques.

The importance of initial models in the specification of abstract data types has been clearly demonstrated in [8]. For more recent work in this area, see [3].

9. CONCLUSION: IMPLICATIONS FOR A GENERALIZED CHURCH-TURING THESIS

In §2.4 we formulated a generalized Church-Turing Thesis for deterministic computations. In the present paper, we have studied various nondeterministic formalisms, including programs with initialization and, most notably, Horn computability, which, as we have shown in the Theorem of Section 6 and the counterexample of §3.2, extends deterministic computation by requiring a nondeterministic search or selection over the structure (in the last stage).

These nondeterministic formalisms are best viewed as formalisms for definition or specification of relations, rather than for computation. It is an interesting fact that their theory is distinct from that of the deterministic formalisms that are the starting point of programming. Of course, in the case that $A$ is minimal, the two formalisms are equivalent, namely: every specification can be implemented and every implementation can be specified.

In thinking about specification, implementation and computation, a first step is to assume that relations are specifications of problems, and functions are specifications of algorithms for their solutions. Typically, we have:

* Specification of Problem. Let $R \subseteq A \times B$. For each $a \in A$ find $b \in B$ such that $R(a, b)$, if such $b$ exists.

* Specification and Algorithm for Solution. Devise some algorithm for a function $f: A \to B$ such that for $a \in A$, if $b$ exists such that $R(a, b)$, then $f(a) \downarrow$ and $R(a, f(a))$.

An obvious requirement on the specification of the problem is that $R$ be effective in some sense so that the $b$ found for given $a$ can be effectively checked. This leads directly to the projections of semicomputable sets in the present context. (Recall §3.2, §3.3.) As we know (see §6), other intuitive models of specifications can be formulated that lead to logic languages, nondeterministic algorithms, programs with initialization, and other concepts that can be proved equivalent.

In view of the above considerations, and the equivalences proved in this paper, we would like to formulate a parallel thesis for nondeterministic specifications of relations:

* Generalized Church-Turing Thesis for Specification. Definability or specifiability of relations on standard structures by effective specifications or nondeterministic algorithms can be formalized by any one of:

  * projective star semicomputability
• 'while'-star definability with initialization
• Horn star definability.

Note again that any structure or class of structures can be standardized and starred. In connection with this Church-Turing Thesis for Specification, many points remain to be discussed; an extensive analysis appears in [22].

We would like to thank J. Derrick and M. Fairtlough (Leeds) for valuable discussions, and J.-L. Lassez for inviting us to give a detailed exposition of these results.

REFERENCES

20. Tucker, J. V., and Zucker, J. I., Horn programs and semicomputable relations on abstract


