PROJECTIONS OF SEMICOMPUTABLE RELATIONS ON ABSTRACT DATA TYPES

J.V. TUCKER

Department of Mathematics and Computer Science,
University College of Swansea, Swansea SA2 8PP, Wales

and

J.I. ZUCKER

Department of Computer Science and Systems,
McMaster University, Hamilton, Ontario L8S 4K1, Canada

ABSTRACT

We consider projections of semicomputable relations on abstract structures We show that they arise in several contexts, including nondeterministic extensions of '**while**' programs with arrays by means of arbitrary initializations and random assignments. They form a basis for an investigation of the concept of algorithmic specifications of relations with nondeterministic search. An important technique in this investigation is the study of computation trees for imperative programs, in order to prove characterization theorems for semicomputable sets, of a form first developed by E. Engeler.

Keywords: abstract data types, computability, many-sorted algebras

0. Introduction

The mathematical theory of computable functions and relations over classes of many-sorted algebras is a generalisation of classical recursion theory on the natural numbers. One application of this theory is to analyse the scope and limits of computation by deterministic programming languages over abstract data types. An abstract data type can be defined as a class $\mathbb{K}$ of algebras closed under isomorphism and, for example, a *generalized Church-Turing Thesis for deterministic computation on abstract data types* can be formulated and defended: see [1].

In this paper we will further develop the theory of computation on many-sorted algebras in order to analyse the scope and limits of nondeterministic specification over abstract data types.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

By considering the "effective" sets suitable for characterising specifications of algorithmic tasks, we are led to study *projections of semicomputable* sets on an arbitrary many-sorted algebra A, *i.e.*, sets defined by

$$x \in S \iff \exists y R(x,y)$$

where R is semicomputable, which means that R is the domain of a partial computable function on A. These concepts are defined uniformly over A from a class $\mathbb{K}$. There are many models of computation over many-sorted algebras that characterize formally the partial computable functions, and hence the semicomputable and projectively semicomputable sets, on A or $\mathbb{K}$. We choose to work with an imperative model based on 'while' programs, without and with arrays. In this model, a partial function is computable if it is definable by a 'while' program with arrays.

We show that the projective semicomputable sets can be characterized in many ways. We study computable and semicomputable sets over algebras A and A^*, being A augmented by finite arrays. We characterize these sets in terms of infinite disjunctions of quantifier-free formulae over A; these characterisation theorems are extensions of a theorem in [2]. We then prove that for R a relation on A, the following are equivalent:

R is projectively semicomputable over A^*;

R is projectively computable over A^*;

R is $\mathbf{\Sigma}_1$ definable over A^*;

R is definable by a 'while' program with random initialization over A^*;

R is definable by a 'while' program with random assignment over A^*.

Further equivalent characterizations are described in [3], including some in terms of Horn clauses from [4], and methods from Moschovakis [5], [6], Montague [7] and Fitting [8].

This paper is part of a series on computability theory on many-sorted algebras and its applications. In [1] the basic functional and imperative models of computation for many-sorted algebras were studied and many other models surveyed. In [3] and [4] we examined computation and specification by Horn clauses which led to the projective semicomputable sets. In [9] and [10] we studied selection functions for provable specifications. In [11] we studied semicomputable sets of real and complex numbers.

Basic computability theory on abstract structures has been developed primarily to tackle problems in programming language theory, starting with work of E. Engeler in 1965. A fairly detailed picture of the subject can be obtained from [12], [13] and [1].

We now give an outline of the paper.

In Section 1 we summarize definitions concerning many-sorted algebras, such as the construction of A^* from A, and projections of relations. In Section 2 we summarize material on partial functions computable by imperative programs, including a generalization of the *Church-Turing Thesis to deterministic computations on abstract data types.* In Section 3 we discuss semicomputable sets and their projections, and consider imperative programs with arbitrary initialization. We show that this notion characterizes projective semicomputability.

In Section 4 we discuss logical definability, and give a proof, under mild hypotheses, that projective semicomputability is equivalent to projective computability.

The main results of the paper are based on forms of Engeler's Lemma. In Section 5 we define computation trees for imperative programs. In Sections 6 and 7 we prove two versions of Engeler's Lemma for imperative programs on A and A^*, adding further characterizations of projective semicomputability. In Section 8 we give a second, more general, proof, using Engeler's Lemma, that projective semicomputability is equivalent to projective computability. In Section 9 we show that 'while' programs with random assignments characterize projective semicomputability. Finally, in Section 10, we review connections between projective semicomputability and specification languages, and formulate a complementary *generalized Church-Turing Thesis for specification on abstract data types.*

1. Abstract Data Types

In this and the following two sections, we briefly review, for the most part, some concepts defined and discussed in [1], [4], [11], [10], [3]. Detailed treatments can be found in Chapter 4 of [1] and Part A of [3].

1.1. *Standard Signatures; Standard Structures; Classes of Structures*

A *standard signature* Σ specifies (1) a finite set of *sorts*: *algebraic sorts* $1, \dots, r$ (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* $(i_1, \dots, i_m; i)$, where $m \geq 0$ is the arity of F, $i_1, \dots, i_m$ 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) $i = \mathsf{N}$ and $i = \mathsf{B}$. We call those sorts i with the equality operator eq_i, *equality sorts*.

We make one further assumption on Σ:

Instantiation Assumption. *Each sort of Σ is* instantiated, *i.e. there is a closed term of each sort.*

A Σ-structure A has, for each sort i of Σ, 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 $A_{\mathsf{N}} = \mathbb{N}$, the set of natural numbers, $A_{\mathsf{B}} = \mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$, 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 equality sort.

We only consider standard signatures and structures. Note however that any many-sorted structure can be *standardized* by the adjunction of the sets $\mathbb{N}$ and $\mathbb{B}$, together with their standard operations.

1.2. *Strictly Standard Signatures and Structures*

We also consider a notion stricter than standardness.

A standard signature Σ is *strictly standard* if the *only operations* with range sort N are the *standard operations* listed in §1.1. The structure A is *strictly standard* if its signature is.

Notice that any standardized structure is automatically strictly standard!

Now fix a (not necessarily strictly) standard signature Σ. An *abstract data types* of signature Σ is defined to be a class $\mathbb{K}$ of Σ-structures *closed under isomorphism.*

Fix any such class $\mathbb{K}$, and consider a particular standard Σ-structure $A \in \mathbb{K}$. We will extend A in two stages.

1.3. *The Unspecified Value* $\mathfrak{u}$*; Structures* A^{u} *of Signature* Σ^{u}

Given a standard Σ-structure A, for each sort i let $\mathfrak{u}_s$ be a new object, representing an "unspecified value", and let $A_i^{\mathsf{u}} = A_i \cup \{\mathfrak{u}_s\}$. For each function symbol F of Σ of type $(i_1, \ldots, i_m; i)$, extend its interpretation F^A on A to a function

$$F^{A,\mathsf{u}} : A_{i_1}^{\mathsf{u}} \times \cdots \times A_{i_m}^{\mathsf{u}} \longrightarrow A_i^{\mathsf{u}}$$

by *strictness* — *i.e.*, the value is defined as $\mathfrak{u}$ whenever any argument is $\mathfrak{u}$. Then the structure A^{u} with signature Σ^{u}, contains:

(i) the original carriers A_i of sort i, and functions F^A on them;

(ii) the new carriers A_i^{u} of sort i^{u}, and functions $F^{A,\mathsf{u}}$ on them;

(iii) a constant unspec_i of type i^{u} to denote $\mathfrak{u}_s$ as a distinguished element of A_i^{u};

(iv) a boolean-valued function Unspec_i of type $(i; \mathsf{B})$, the characteristic function of $\mathfrak{u}_s$;

(v) an *embedding function* i_i of sort $(i; i^{\mathsf{u}})$ to denote the embedding of A_i into A_i^{u}, and an inverse function j_i of sort $(i^{\mathsf{u}}; i)$, which maps $\mathfrak{u}_s$ to the denotation of some closed term in A_i (this being possible by the Instantiation Assumption) for each sort i; and finally

(vi) an *equality* operator on A_i^{u} for each equality sort i.

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

Remark. A^{u} can be *effectively coded within* A. Each element y of A_i^{u} is represented by the *pair* $(b, \mathsf{j}_i(y)) \in \mathbb{B} \times A_i$, where $b = \mathtt{tt}$ if $y \neq \mathbf{u}_s$ and $b = \mathtt{ff}$ otherwise. This induces, in an obvious way, a coding of the operations on A^{u} by operations on A.

1.4. Structures A^* of Signature Σ^*

Define, for each sort i, the carrier A_i^* to be the set of pairs $a^* = (\xi, l)$ where $\xi : \mathbb{N} \to A_i^{\mathsf{u}}$, $l \in \mathbb{N}$ and for all $n \geq l$, $\xi(n) = \mathbf{u}_s$. So l is a witness to the "finiteness" of ξ, 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 Σ^*, which extends Σ^{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 Lgth_i operator, of type $(i^*; \mathsf{N})$, where

$$\mathsf{Lgth}_i^A((\xi, l)) = l,$$

(ii) the application operator Ap_i of type $(i^*, N; i^{\mathsf{u}})$, where

$$\mathsf{Ap}_i^A((\xi, l), n) = \xi(n),$$

(iii) the null array Null_i of type i^*, where

$$\mathsf{Null}_i^A = (\lambda n \cdot \mathbf{u}_s, 0) \in A_i^*,$$

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

$$\eta(k) = \begin{cases} \xi(k) & \text{if } \ k < l, k \neq n \\ x & \text{if } \ k < l, k = n \\ \mathbf{u}_s & \text{otherwise,} \end{cases}$$

(v) the $\mathsf{Newlength}_i$ operator of type $(i^*, \mathsf{N}; i^*)$, where $\mathsf{Newlength}_i^A((\xi, l), m)$ is the array (η, m) such that for all k,

$$\eta(k) = \begin{cases} \xi(k) & \text{if } \ k < m \\ \mathbf{u}_s & \text{if } \ k \geq m, \end{cases}$$

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

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

We will call the sorts i *simple*, the sorts i^{u} *augmented* and the sorts i^* *starred*.

Remark. We do not have to introduce a structure A^{**} of "doubly starred" carrier sets containing "two-dimensional arrays"; such a structure can be effectively coded in A^*, since we can effectively code a finite sequence of starred objects of a given sort as a single starred object of the same sort, thanks to the explicit Lgth operation. More precisely, a sequence $x_0^*, \ldots, x_{k-1}^*$ of elements of A_h^* (for some sort h) can be coded as a *pair* $(y^*, n^*) \in A_h^* \times \mathbb{N}^*$, where $\mathsf{Lgth}(n^*) = k$, and for $0 \le j < k$, $n^*[j] = \mathsf{Lgth}(x_j^*)$, and $\mathsf{Lgth}(y^*) = n^*[0] + \cdots + n^*[k]$, and for $1 \le j \le k$ and $0 \le i < n^*[j]$, $y^*[n^*[0] + \cdots + n^*[j-1] + i] = x_j^*[i]$.

1.5. Types of Functions and Relations

We collect some definitions and notation. Consider again a structure A of signature Σ. Let $\vec{k} = k_1, \ldots, k_m$ $(m \ge 0)$ be a list of sorts of Σ.

(1) We write $\vec{t} : \vec{k}$ to mean that $\vec{t}$ is a vector of terms $t_1, \ldots, t_m$ of sorts $k_1, \ldots, k_m$ respectively.

(2) $A[\vec{k}]$ denotes $A_{k_1} \times \cdots \times A_{k_m}$.

(3) 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 $A[\vec{k}]$.

(4) Suppose R is a relation on A of type $\vec{k}, \vec{l}$. The *projection of R on $A[\vec{k}]$* is the relation R' on A of type $\vec{k}$ defined by

$$R'(\vec{x}) \iff \exists \vec{y} R(\vec{x}, \vec{y})$$

(where $\vec{x} : \vec{k}$ and $\vec{y} : \vec{l}$).

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

2. Computable Functions

We will use a model of computability of functions based on an imperative programming language. (We will mention other, equivalent, models below, in §2.3.)

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

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 $[S, \vec{v}, w]$ is assumed to be $\mathbb{K}$-*functional*, which means that on any $A \in \mathbb{K}$, and for any values of the input variables $\vec{v}$ on A, the program will (deterministically) either halt, with a uniquely determined 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.

2.1. 'while' and 'loop' Computability

Consider, now, a 'while' programming language. First we define the class of program terms $t^i, \ldots$ of sort i over Σ:

$$t^i ::= v^i \mid F(t_1^{i_1}, \ldots, t_m^{i_m}) \mid \mathsf{if}\ b\ \mathsf{then}\ t_1^i\ \mathsf{else}\ t_2^i\ \mathsf{fi}$$

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

Remark 1. Note that because of the '$\mathsf{if} \ldots \mathsf{fi}$' construction, the set of closed program terms over Σ is more extensive than the set of closed terms over Σ as usually defined. We will see examples of the usefulness of the '$\mathsf{if} \ldots \mathsf{fi}$' construction later (Section 7).

Now we define the class of 'while' program statements $S, \ldots$ over Σ:

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

where $b : \mathsf{B}$. The class of 'loop' programs is defined by replacing the 'while' construct by '$\mathsf{loop}\ t^{\mathsf{N}}\ \mathsf{do}\ S\ \mathsf{od}$'.

A function on A is 'while' *computable* if it is definable by an i/o-'while' program.

Similarly for 'loop' computability. Note that 'loop' computable functions are always *total*.

Remark 2. The semantics of 'while' programs is given in [1, §2.2]). In particular, it follows from the *monotonicity property* of 'while' programs (*op. cit.*, §2.2.8) and from the $\mathbb{K}$-*functionality* of $[S, \vec{v}, w]$, that $[S, \vec{v}, w]$ will define the same family of functions on $\mathbb{K}$ if some or all of the auxiliary variables are *initialized arbitrarily.* (The same applies, of course, to the 'loop' language.)

2.2. 'while^*' and 'loop^*' Computability

Consider now the 'while' and 'loop' programming languages over Σ^*.

We will call a program variable *simple*, *augmented* or *starred* according as its sort is simple i, augmented i^{u} or starred i^*.

A function on A is 'while^*' *computable* if it is computable by a 'while' program on A^*. Similarly for 'loop^*' computability.

Of course, the input and output variables will be simple; the auxiliary variables may be augmented or starred. Note that any program with augmented variables (as auxiliary variables) can be replaced by one with simple variables, by the device of coding A^{u} in A (see the Remark in §1.3).

*2.3. Three Lemmas about '*while**' Computability*

These will be needed later.

Lemma 1. *Term evaluation on A is '*loop**' computable, and hence '*while**' computable, on A, uniformly for $A \in \mathbb{K}$.*

More precisely: Fix a list of variables $\vec{w}$ of sorts $\vec{k} = k_1, ..., k_m$. Let $\boldsymbol{Term}(\vec{w})$ be the class of program terms over Σ with variables among $\vec{w}$ only, and let $\ulcorner t \urcorner \in \mathbb{N}$ denote the Gödel number of the term $t \in \boldsymbol{Term}(\vec{w})$. Then for $i = 1, ..., m$, the map

$$\boldsymbol{te}^i_A : \mathbb{N} \times A[\vec{k}] \to A_{k_i},$$

where $\boldsymbol{te}^i_A(\ulcorner t \urcorner, \vec{a})$ is the value of t when $\vec{w}$ is evaluated as $\vec{a}$, is 'loop^*' computable, uniformly for $A \in \mathbb{K}$.

Lemma 2. *Any partial recursive function on $\mathbb{N}$ is '*while*'-computable, and hence (certainly) '*while**' computable, on A.*

Lemma 3. *Any function on A^* which is '*while**' computable on A^* is '*while*' computable on A^*, uniformly for $A \in \mathbb{K}$.*

This follows from the computable coding of A^{**} in A^* (Remark in §1.4).

2.4. Computability by Schemes; the Generalized Church-Turing Thesis for Computation

We can use *induction schemes* α (over Σ) to define functions α^A over A, or, more generally, *families of functions* $\{\alpha^A \mid A \in \mathbb{K}\}$ uniformly over $\mathbb{K}$. These induction schemes generalize the schemes for *primitive recursive* and *μ-recursive functions* over $\mathbb{N}$ in [14].

The class of *PR computable* functions over $\mathbb{K}$ is generated by schemes for *projection*, *composition*, *definition by cases* and *simultaneous primitive recursion on* $\mathbb{N}$.

The class of *μPR computable* functions over $\mathbb{K}$ is formed by adding to the above schemes, a scheme for the μ or *least number operator*.

Notice that the last two schemes use the standardness of the structures, *i.e.* the carrier $\mathbb{N}$.

This approach to computability corresponds to that in §2.1:

Theorem 1. *Let f be a function on A. Then*

(a) $\qquad f$ *is* 'while' *computable* $\Longleftrightarrow f$ *is* μPR *computable,*

(b) $\qquad f$ *is* 'loop' *computable* $\Longleftrightarrow f$ *is PR computable,*

in both cases, uniformly effectively over $\mathbb{K}$.

Similarly, we can define the classes of functions "star computable" by schemes. A function on A is PR^* *computable* if it is defined by a PR scheme over Σ^*, interpreted on A^* (*i.e.*, using starred sorts in its definition).

Similarly, it is $\mu\mathrm{PR}^*$ *computable* if it is defined by a μPR scheme over Σ^*. Again, we have a correspondence:

Theorem 2. *Let f be a function on A. Then*

(a) $\qquad f$ *is* 'while*' *computable* $\Longleftrightarrow f$ *is* μPR^* *computable,*

(b) $\qquad f$ *is* 'loop*' *computable* $\Longleftrightarrow f$ *is* PR^* *computable,*

again, in both cases, uniformly effectively over $\mathbb{K}$.

On the basis of this and many other equivalences, the following thesis was postulated (for an equivalent formalism) in Chapter 4 of [1].

A Generalized Church-Turing Thesis for Deterministic Computation on Abstract Data Types. *Consider a deterministic programming language over an abstract data type* ***dt****. The set of functions on a structure A, representing an implementation of the abstract data type* ***dt****, that can be programmed in the language, is contained in the set of* 'while*' *computable functions on A. The class of functions over a class $\mathbb{K}$ of structures, representing a class of implementations of the abstract data type* ***dt****, that can be programmed in the language, uniformly over all implementations of $\mathbb{K}$, is contained in the class of* 'while*' *computable functions over $\mathbb{K}$.*

In Section 10 we will give a complementary version of this thesis for specification.

3. Semicomputable Relations

In this section we investigate various notions of *semicomputability* of relations, intended to generalize to A (and $\mathbb{K}$) the notion of *recursive enumerability* over $\mathbb{N}$.

An *i-program (input program)* is defined to be a pair $[S, \vec{v}]$ consisting of a deterministic program S in the language of Σ, 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.

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 $\mathbb{K}$-*relational*, which means that on any $A \in \mathbb{K}$, 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 $\mathbb{K}$-functionality of i/o-programs defined above.)

Remark. Hence, again, such an i-program will define the same halting sets on $\mathbb{K}$ if some or all of the auxiliary variables are initialized arbitrarily, by the *monotonicity property* of programs. (See Remark 2 in Section 2.)

Our i-program S may contain augmented or starred *auxiliary variables* (if it is a program over Σ^*). However, we will always assume that its *input variables* $\vec{v}$ *are simple*, so that in all cases S defines a relation on A.

3.1. 'while' *Semicomputability*

Let R be a relation on A.

Definition 1. (*a*) R is 'while' *computable* iff its (boolean-valued) characteristic function is 'while' computable.

(*b*) R is 'while^*' *computable* iff its characteristic function is 'while^*' computable.

Definition 2. Suppose R is the *halting set* on A of the i-program $[S, \vec{v}]$.

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

(*b*) R is 'while^*' *semicomputable* if S is a 'while' program over A^*.

Proposition 1. *(a) R is 'while' semicomputable $\iff$ R is the domain of a 'while' computable function.*

(b) R is 'while^*' semicomputable $\iff$ R is the domain of a 'while^*' computable function.

Proof. It is easy to define a (uniform, effective) transformation of an i-programs $[S, \vec{v}]$ which has R as its halting set, to an i/o-program $[S, \vec{v}, w]$ which has R as its domain, and vice versa. $\square$

We also have versions of Post's Theorem:

Proposition 2. *(a) R is 'while' computable iff both R and its complement are 'while' semicomputable.*

(b) R is 'while^*' computable iff both R and its complement are 'while^*' semicomputable.

3.2. Semicomputability 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 $[S, \vec{v}, \vec{z}]$ consisting of a deterministic program S in the language of Σ, together with a list of *input variables* $\vec{v}$ and *search variables* $\vec{z}$. The relation defined by such a triple on A is the set $\vec{x}$ of tuples of elements of A, such that when $\vec{v}$ is initialized to $\vec{x}$ then *for some (non-deterministic) initialization of* $\vec{z}$*,* 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 $[S, \vec{v}, \vec{z}]$ is assumed to be $\mathbb{K}$-*relational*, which means that on any $A \in \mathbb{K}$, and for any values of the input and search variables on A, the program will (deterministically) either halt or diverge, but never abort.

Remark. The semantics of i/s-programs can be obtained by a modification of the semantics of 'while' programs given in [1, §2.2.6], by taking the meaning of an i/s program to be a function from states *and initial values for the search variables* to computation sequences.

Definition 3. Let R be the relation on A defined by $[S, \vec{v}, \vec{z}]$.

(a) R is 'while' *semicomputable with initialization* (on A, by $[S, \vec{v}, \vec{z}]$) if S is a simple 'while' program over A.

(b) R is 'while*' *semicomputable with initialization* if S is a 'while' program over A^* in which the search and auxiliary variables may be starred.

3.3. Projective Semicomputability

Definition 1. (a) R is *projectively* 'while' *computable* iff R is a *projection* of a 'while' computable relation.

(b) R is *projectively* 'while*' *computable* iff R is a *projection* of a 'while*' computable relation.

Definition 2. (a) R is *projectively* 'while' *semicomputable* iff R is a *projection* of a 'while' semicomputable relation.

(b) R is *projectively* 'while*' *semicomputable* iff R is a *projection* of a 'while*' semicomputable relation.

The following equivalences follow easily from the definitions.

Proposition 1. *Let* R *be a relation on* A.

(a) R *is* 'while' *semicomputable with initialization* $\iff$ R *is projectively* 'while' *semicomputable.*

(b) R *is* 'while*' *semicomputable with initialization* $\iff$ R *is projectively* 'while*' *semicomputable.*

Remark 1. Projective 'while' semicomputability is strictly stronger than 'while' semicomputability. An example to show this, on the field of real numbers, is given in [11], which uses Engeler's Lemma (Section 7).

We do, however, have closure of semicomputability in the case of existential quantification over $\mathbb{N}$:

Proposition 2. *Suppose* $R \subseteq A[\vec{k}, \mathsf{N}\,]$ *is* 'while' *(or* 'while^*'*) semicomputable. Then so is its projection* $\{\vec{x} \mid \exists z\, R(\vec{x}, z)\}$ *on* $A[\vec{k}]$.

Remark 2. Projective 'while^*' semicomputability is strictly stronger than projective 'while' semicomputability (by another example, *loc. cit.*). In other words, projecting along "starred" sorts is stronger than projecting along "simple" sorts. Intuitively, this corresponds to existentially quantifying over a finite, but unbounded, sequence of elements. An example to show this, again on the real numbers, is given in [11].

We do, however, have the following equivalence.

Theorem (Projective Equivalence Theorem). *R is projectively* 'while^*' *semicomputable iff R is projectively* 'while^*' *computable.*

This will be proved twice: in Section 4, by use of the computation predicates and the notion of $\mathbf{\Sigma}_1^*$ definability (under the assumption that Σ has equality at all sorts), and again in Section 8, as a simple consequence of Engeler's Lemma (without any such assumption).

4. $\mathbf{\Sigma}_1^*$ Definability; The Computation Predicate

4.1. $\mathbf{\Sigma}_1^*$ *Definability*

Let $\boldsymbol{Lang}^* = \boldsymbol{Lang}(\Sigma^*)$ be the first order language with equality over Σ^*.

Let us call the sorts i *simple*, i^{u} *augmented* and i^* *starred*. There are variables of all these sorts.

We define some classes of formulae of $\boldsymbol{Lang}^*$.

The *atomic formulae* are equalities between terms of sort i, i^{u} and i^*, for all sorts i of Σ (whether equality sorts or not).

A *bounded quantifier* has the form '$\forall k < t$' or '$\exists k < t$', where $t : \mathsf{N}$; an *elementary formula* is one with only bounded quantifiers; a $\mathbf{\Sigma}_1^*$ *formula* is formed by prefixing an elementary formula with existential quantifiers only; and an *extended* $\mathbf{\Sigma}_1^*$ *formula* is formed by prefixing an elementary formula with a string of existential quantifiers and bounded universal quantifiers (in any order).

Proposition 1. *For any extended $\mathbf{\Sigma}_1^*$ formula P, there is a $\mathbf{\Sigma}_1^*$ formula Q which is equivalent to P over $\mathbb{K}$, in the sense that*

$$\mathbb{K} \models P \leftrightarrow Q.$$

Proof. The construction of Q from the P is given in [10]. (In that paper, the equivalence is actually shown relative to a formal system over $\mathbb{K}$ with $\mathbf{\Sigma}_1^*$ induction. However, we are not concerned with issues of provability in the present paper.) $\square$

Because of this result, we will use the term '$\mathbf{\Sigma}_1^*$' to denote (possibly) extended $\mathbf{\Sigma}_1^*$ formulae.

Proposition 2. *If P is an elementary formula all of whose variables are of equality sort, then the predicate defined by P is 'loop^*' computable.*

Proof. By structural induction on P. Equations between variables of equality sort are 'loop^*' computable, as are boolean operations and bounded quantification. $\square$

Remark. In general (as stated above), formulae over the structure A^* (or rather, over the signature Σ^*) may have variables of simple, augmented or starred sort. We will be interested in formulae with the property that all free variables are of *simple sorts*, since such formulae define relations on A. For such formulae, all bound variables of augmented sort may be replaced by bound variables of simple sort, by the effective coding of A^{u} in A (§1.3, Remark).

4.2. Computation Predicate

Given a 'while^*' i/o program $[S, \vec{v}, w]$ with $\vec{v} : \vec{k}$ and $w : l$, we can effectively construct a $\mathbf{\Sigma}_1^*$ formula $\boldsymbol{Comp}_{[S,\vec{v},w]}(\vec{x}, y)$, with free variables $\vec{x} : \vec{k}$ and $y : l$, which satisfies: for all $A \in \mathbb{K}$, for all $\vec{x}, y \in A[\vec{k}, l]$,

$$A \models \boldsymbol{Comp}_{[S,\vec{v},w]}(\vec{x}, y) \iff [S, \vec{v}, w]^A(\vec{x}) \downarrow y.$$

Here are two different constructions of $\boldsymbol{Comp}_{[S,\vec{v},w]}$:

(1) First we construct a $\mathbf{\Sigma}_1^*$ formula $\boldsymbol{Compu}_{[S,\vec{v},w]}(\vec{x}, y, \vec{z}^{\,*})$ with the meaning: "$\vec{z}^{\,*}$ represents a computation sequence generated by S, starting in a state in which the input variables have values $\vec{x}$, and ending in a state in which the output variable has value y". This is almost the same as the predicate $\boldsymbol{Compu}_S$ in [1, §2.6.11], constructed by structural induction on S (the difference being that the latter predicate refers to the initial and final values of all the variables in S, not just the input and ouput variables). By existentially quantifying over the variables $\vec{z}^{\,*}$ in $\boldsymbol{Compu}_{[S,\vec{v},w]}(\vec{x}, y, \vec{z}^{\,*})$, we obtain the $\mathbf{\Sigma}_1^*$ formula $\boldsymbol{Comp}_{[S,\vec{v},w]}(\vec{x}, y)$, as desired.

(2) Let α be the $\mu\mathrm{PR}^*$ scheme which corresponds to $[S, \vec{v}, w]$ according to the construction given by the proof of Theorem 2(a) of §2.4. By structural induction

on α, we construct the predicate $\boldsymbol{Comp}_\alpha$ ($= \boldsymbol{Comp}_{[S,\vec{v},w]}$), exactly as in [9] or [10, §5] (where the predicate is called P_α).

4.3. First Proof of the Projective Equivalence Theorem

We present a proof of the Projective Equivalence Theorem stated in §3.3, under the assumption that all sorts are equality sorts (§1.1). First we re-state the Theorem.

Theorem. *Suppose Σ has an equality operator at all sorts. Let R be a relation on A. Then the following are equivalent:*

*(i) R is projectively '*while**' computable;*

*(ii) R is projectively '*while**' semicomputable;*

(iii) R is $\boldsymbol{\Sigma}_1^*$ definable.

Proof. $(i) \Longrightarrow (ii)$: Trivial.

$(ii) \Longrightarrow (iii)$: Suppose

$$R(\vec{x}) \iff \exists \vec{x'} R_0(\vec{x}, \vec{x'})$$

where R_0 is 'while*' semicomputable. By Proposition 1 of §3.1, R_0 is the domain of a 'while*' computable function, say $[S, \vec{v}, w]^A$. Let $\boldsymbol{Comp}_{[S,\vec{v},w]}$ be the computation predicate for this function, as given in §4.2. Then

$$R_0(\vec{x}, \vec{x'}) \iff \exists y\ \boldsymbol{Comp}_{[S,\vec{v},w]}(\vec{x}, \vec{x'}, y).$$

Since $\boldsymbol{Comp}_{[S,\vec{v},w]}$ is $\boldsymbol{\Sigma}_1^*$, so also is R_0, and hence so also is R.

$(iii) \Longrightarrow (i)$: Suppose R is defined by the formula $\exists \vec{z}^* P(\vec{x}, \vec{z}^*)$, where P is elementary. Then R is a projection of P, which, by Proposition 2 of §4.1, is 'loop*' computable. $\square$

5. The Computation Tree of a 'while' Program

We will define, for any 'while' program S over Σ, and vector of program variables $\vec{v} \equiv v_1, \ldots, v_n$ such that $\boldsymbol{var}(()S) \subseteq \vec{v}$, the *computation tree* $\mathcal{T}[S, \vec{v}]$, which is like an "unfolded flow chart" of S.

The root of the tree $\mathcal{T}[S, \vec{v}]$ is labeled 's' (for 'start'), and the leaves are labeled 'e' (for 'end'). The internal nodes are labeled with assignment statements and boolean tests.

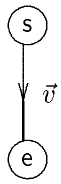
Furthermore, each edge of $\mathcal{T}[S, \vec{v}]$ is labeled with a *syntactic state*, *i.e.*, a vector of terms $\vec{t} \equiv t_1, \ldots, t_n \in \boldsymbol{Term}(\vec{v})$. Intuitively, $\vec{t}$ gives the current state, assuming execution of S starts in an initial state (represented by) $\vec{v}$.

In the course of the following definition we will make use of the *restricted tree* $\mathcal{T}^-[S, \vec{v}]$, which is just $\mathcal{T}[S, \vec{v}]$ without the 's' node.

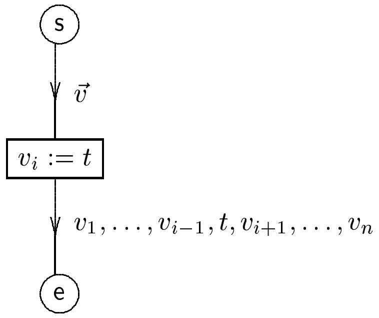
We also use the notation $\mathcal{T}[S,\vec{t}\,]$ for the tree formed from $\mathcal{T}[S,\vec{v}]$ by *simultaneous substitution* of $\vec{t}$ for $\vec{v}$ throughout.

The definition is by structural induction on S. (Recall the inductive definition of the syntax of 'while' programs in §2.1.)

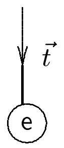
(i) $S \equiv$ skip. Then $\mathcal{T}[S,\vec{v}] \equiv$



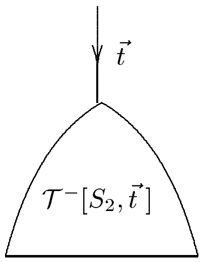
(ii) $S \equiv v_i := t$. Then $\mathcal{T}[S,\vec{v}] \equiv$



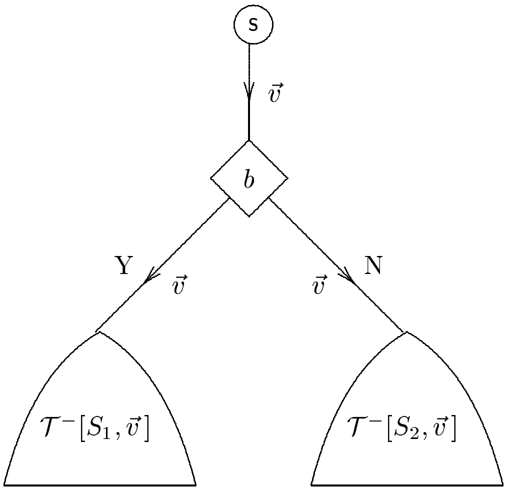
(iii) $T \equiv S_1;S_2$. Then $\mathcal{T}[S,\vec{v}]$ is formed from $\mathcal{T}[S_1,\vec{v}\,]$ by *replacing* each leaf



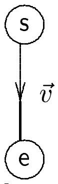
by



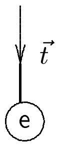
(iv) $S \equiv \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi}$. Then $\mathcal{T}[S, \vec{v}] \equiv$



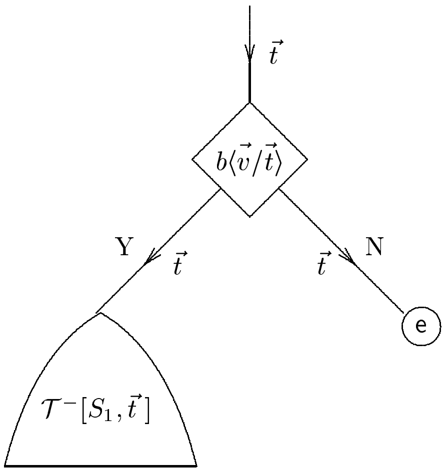
(v) $S \equiv \textsf{while } b \textsf{ do } S_1 \textsf{ od}$. Then $\mathcal{T}[S, \vec{v}]$ is the "limit" of the sequence of trees $\mathcal{T}_n[S, \vec{v}\,]$, where $\mathcal{T}_0[S, \vec{v}\,] \equiv$



and $\mathcal{T}_{n+1}[S, \vec{v}\,]$ is formed from $\mathcal{T}_n[S, \vec{v}\,]$ by *replacing* each leaf



by



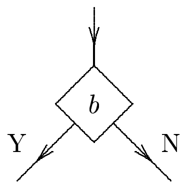
Remark. $\mathcal{T}[S, \vec{v}]$ can be constructed *effectively* from S and $\vec{v}$. More precisely: Assume some standard Gödel numbering $\ulcorner E \urcorner$ of syntactic objects E. Then $\mathcal{T}[S, \vec{v}]$ can be coded as a *recursive set* of numbers, with index primitive recursive in $\ulcorner S \urcorner$ and $\ulcorner \vec{v} \urcorner$.

6. The Halting Predicate for 'while' Programs; Engeler's Lemma for 'while' Semicomputable Relations

6.1. The Halting Predicate for 'while' Programs

For each leaf λ of the computation tree $\mathcal{T}[S, \vec{v}]$, there is a *boolean* (*i.e.*, a term of sort B in the 'while' programming language) $\boldsymbol{B}[S, \vec{v}, \lambda]$, with free variables among $\vec{v}$, which expresses the conjunction of results of all the successive tests, that (the current values of) the variables $\vec{v}$ must satisfy in order for the computation to "follow" the finite path from the root s to λ.

Consider, for example, a test node in $\mathcal{T}[S, \vec{v}]$:



If the path goes to the right here (say), then it contributes to $\boldsymbol{B}[S, \vec{v}, \lambda]$ the conjunct

$$\cdots \wedge \neg b\langle \vec{v}/\vec{t}\rangle \wedge \ldots$$

Next, let $\langle \lambda_j \mid j \geq 0\rangle$ be some *effective enumeration* of leaves of $\mathcal{T}[S, \vec{v}]$ (*e.g.*, in increasing depth, and, at a given depth, from left to right). Then the *halting predicate of S with respect to $\vec{v}$* is the infinite disjunction

$$\boldsymbol{Halt}[S, \vec{v}] \;\equiv_{df}\; \bigvee_{j=0}^{\infty} \boldsymbol{B}[S, \vec{v}, \lambda_j]$$

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

Note that the boolean $\boldsymbol{B}[S, \vec{v}, \lambda_j]$ is effective in S, $\vec{v}$ and j. More precisely, there is a partial recursive function β of three arguments such that $\beta(\ulcorner S \urcorner, \ulcorner \vec{v} \urcorner, j)$ is the Gödel number of $\boldsymbol{B}[S, \vec{v}, \lambda_j]$.

We will now rewrite $\boldsymbol{Halt}[S, \vec{v}]$ as a different infinite disjunction, as follows. For each j, the boolean $\boldsymbol{B}[S, \vec{v}, \lambda_j]$ is rewritten in disjunctive normal form, and its disjunctions absorbed into the "big disjunction", so that $\boldsymbol{Halt}[S, \vec{v}]$ has the form

$$\boldsymbol{Halt}[S, \vec{v}] \iff \bigvee_{k=0}^{\infty} \boldsymbol{B}^{\circ}[S, \vec{v}, k] \tag{1}$$

where $\boldsymbol{B}^{\circ}[S, \vec{v}, k]$ is a *finite conjunction of literals* (equations and inequations) over the signature Σ. We can think of this as a kind of "normal form" for the halting predicate.

Again, there is a partial recursive function β° of three arguments such that $\beta^{\circ}(\ulcorner S \urcorner, \ulcorner \vec{v} \urcorner, k)$ is the Gödel number of $\boldsymbol{B}^{\circ}[S, \vec{v}, k]$. In fact (by a standard technique of classical recursion theory) β° can be taken to be primitive recursive (assuming $\mathcal{T}[S, \vec{v}]$ has at least one leaf).

Now S *halts on the initialization* $\vec{x} \in A[\vec{k}]$ iff for some k,

$$\boldsymbol{te}_A(\beta^{\circ}(\ulcorner S \urcorner, \ulcorner \vec{v} \urcorner, k), \vec{x}) \downarrow \mathtt{tt},$$

where $\boldsymbol{te}_A$ is the *term evaluation function* on A, which is 'while*' computable (by Lemma 1 in §2.3).

Now the *halting set* $\boldsymbol{Halt}^A[S, \vec{v}]$ of S with respect to $\vec{v}$ on A is the set of tuples $\vec{x}$ from A for which the predicate $\boldsymbol{Halt}[S, \vec{v}]$ evaluates to $\mathtt{tt}$ when $\vec{x}$ is assigned to $\vec{v}$. So

$$\boldsymbol{Halt}^A[S, \vec{v}] = \{\vec{x} \in A[\vec{k}] \mid \text{for some } k,\ \boldsymbol{te}_A(\beta^{\circ}(\ulcorner S \urcorner, \ulcorner \vec{v} \urcorner, k), \vec{x}) \downarrow \mathtt{tt}\},$$

which shows (by Lemmas 1 and 2 in §2.3 and Proposition 2 in §3.3) that $\boldsymbol{Halt}^A[S, \vec{v}]$ is 'while*' semicomputable – which is not very helpful, since, by definition, it is 'while' semicomputable!

Remark. We have been treating all the variables of S as input variables. Suppose, more generally (to connect this with the framework of Section 3) that $\boldsymbol{var}(()S) \subseteq \vec{v}, \vec{w}$, where $\vec{w}$ is a list of auxiliary variables, and we are interested in the halting set as a relation on $\vec{v}$. According to the discussion in Section 3, the variables $\vec{w}$ should be uninitialized. However, by the *relationality* of $[S, \vec{v}]$ (see the Remark in Section 3), the halting set is not affected if we initialize the variables $\vec{w}$ to suitable closed terms (of the correct sorts) — which is always possible, by the Instantiation Assumption.

*6.2. Engeler's Lemma for '*while*' Semicomputable Relations*

Now suppose R is a 'while' semicomputable relation on a Σ-structure A. That means, by definition, that $R = \boldsymbol{Halt}^A[S, \vec{v}]$ for a suitable 'while' program S. Hence, by Eq. (1) of §6.1, we immediately derive the following form of *Engeler's Lemma* [2]:

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

Actually, we need a stronger version of Engeler's Lemma, applied to 'while*' programs, which we will derive in the next section.

7. Engeler's Lemma for 'while*' Semicomputable Relations

It is straightforward to repeat the work of Sections 5 and 6 for 'while*' programs. Consider a relational 'while*' program $[S, \vec{v}]$, with (unstarred) input variables $\vec{v} : \vec{k}$, unstarred auxiliary variables $\vec{w}$ and starred auxiliary variables $\vec{w}^*$ (and no search variables). Now we can construct a *computation tree* $\mathcal{T}[S, \vec{v}, \vec{w}, \vec{w}^*]$, *effectively* in S, $\vec{v}$, $\vec{w}$ and $\vec{w}^*$.

Again, we can form the *halting predicate* $\boldsymbol{Halt}[S, \vec{v}]$ of S relative to $\vec{v}$, by initializing the auxiliary variables suitably (see the Remark at the end of §6.1). Then, by the same reasoning as led to equation (1) in §6.1, we derive

$$\boldsymbol{Halt}[S, \vec{v}] \iff \bigvee_{i=0}^{\infty} b_i(\vec{v}) \tag{1}$$

where $b_i(\vec{v})$ is a finite conjunction of literals (equations and inequations) over the signature Σ^*. Note that the booleans $b_i(\vec{v})$, although they are not of starred sort, may contain subterms of starred sort — for example, they may be equations between terms of starred sort.

Again, there is a partial recursive function β^* such that (suppressing all arguments of β^* except i) $\beta^*(i)$ is the Gödel number of $b_i(\vec{v})$.

Now S *halts on the initialization* $\vec{x} \in A[\vec{k}]$ iff for some i,

$$\boldsymbol{te}^*_A(\beta^*(i),\, \vec{x}) \downarrow \mathtt{tt}$$

where $\boldsymbol{te}^*_A$ is the *term evaluation function* on A^*, which is 'while*' computable on A^* (by Lemma 1 in §2.3), and hence (by Lemma 3 in §2.3) 'while*' computable on A.

Hence the *halting set* of S with respect to $\vec{v}$ on A is given by

$$\boldsymbol{Halt}^*_A[S, \vec{v}] = \{\vec{x} \in A[\vec{k}] \mid \text{for some } i,\ \boldsymbol{te}^*_A(\beta^*(i),\, \vec{x}\,) \downarrow \mathtt{tt}\},$$

which shows (again from Lemmas 1 and 2 in §2.3 and Proposition 2 in 3.3) that it is 'while*' semicomputable, which — again — follows by definition!

However we will use (1) to prove a structure theorem in the case of *algebraic* relations on *strictly standard* structures (see §1.5(5) and §1.2 for definitions.)

Remark 1. The above conditions are very general, since if R is a relation on *any* structure which is then *standardized*, then, trivially, the new structure is strictly standard, and R is algebraic on it!

Theorem (Engeler's Lemma, Version 2). *Let R be an algebraic relation on a strictly standard structure A. Suppose R is 'while^*' semicomputable on A. Then R can be expressed as an effective infinite disjunction of booleans over Σ.*

The proof occupies most of the rest of this subsection.

By assumption, R is the halting set $\boldsymbol{Halt}^*_A[S, \vec{v}]$ for a relational 'while^*' program $[S, \vec{v}]$ over A, with $\vec{v} : \vec{k}$ say. Since R is algebraic, none of the sorts k_i in $\vec{k}$ is N.

Let $\boldsymbol{Term}^*(\vec{v})$ be the class of program terms of unstarred sort in the language of Σ^*, containing (at most) the program variables $\vec{v}$, and let $\boldsymbol{Term}^*_i(\vec{v})$ be the class of such terms of sort i.

Note that although the terms in $\boldsymbol{Term}^*(\vec{v})$ are of unstarred sort, they may contain subterms of starred sort.

By an *unstarred term* we mean a term without any subterms of starred sort. Let $\boldsymbol{Term}(\vec{v})$ be the class of unstarred terms in $\boldsymbol{Term}^*(\vec{v})$.

For $t \equiv t(\vec{v}) \in \boldsymbol{Term}^*(\vec{v})$ and $x \in A[\vec{k}]$, we write $t[\vec{x}]$ for the interpretation of t in A with $\vec{x}$ assigned to $\vec{v}$.

From (1),

$$R = \boldsymbol{Halt}^*_A[S, \vec{v}] = \{\vec{x} \in A[\vec{k}] \mid \bigvee_i b_i[\vec{x}]\} \tag{2}$$

where for all i, $b_i(\vec{v}) \in \boldsymbol{Term}^*_{\mathsf{B}}(\vec{v})$.

In order to convert this to the desired form, we first show that each boolean $b_i(\vec{v})$ is equivalent to an unstarred boolean. In fact we will show that any term in $\boldsymbol{Term}^*(\vec{v})$ is equivalent to an unstarred term (Lemma below).

We must first define the notion of *maximum value* of a term in $\boldsymbol{Term}^*_{\mathsf{N}}(\vec{v})$. This is the maximum possible numerical value that such a term could have, under any valuation of the variables $\vec{v}$.

We must actually define the maximum value of terms $t \in \boldsymbol{Term}^*(\vec{v})$ of sort N *or* N^{u}, simultaneously by structural induction on t. (Assume that $\mathsf{j}_{\mathsf{N}}(\mathsf{unspec}_{\mathsf{nat}}) = 0$: see cases (3) and (5) below.)

Definition. For $t \in \boldsymbol{Term}^*(\vec{v})$ of sort N or N^{u}, its maximum value $\boldsymbol{maxval}(t) \in \mathbb{N}$ is defined by structural induction on t. There are 7 cases:

(1) $t \equiv 0$ *or* $t \equiv 0^{\mathsf{u}}$. $\boldsymbol{maxval}(t) = 0$

(2) $t \equiv \mathsf{S}t_0$ *or* $t \equiv \mathsf{S}^{\mathsf{u}}t_0$. $\boldsymbol{maxval}(t) = \boldsymbol{maxval}(t_0) + 1$

(3) $t \equiv \mathsf{i}_{\mathsf{N}}(t_0)$ *or* $t \equiv \mathsf{j}_{\mathsf{N}}(t_0)$. $\boldsymbol{maxval}(t) = \boldsymbol{maxval}(t_0)$

(4) $t \equiv \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \mathsf{fi}.$ $\quad \boldsymbol{maxval}(t) = \max(\boldsymbol{maxval}(t_1),\ \boldsymbol{maxval}(t_2))$

(5) $t \equiv \mathsf{unspec}_{\mathsf{nat}}.$ $\quad \boldsymbol{maxval}(t) = 0$

(6) $t \equiv \mathsf{Ap}(s, t_0)$, where s is of sort N^*. There are 4 subcases, according to the form of s:

(i) $s \equiv \mathsf{Null}.$ $\quad \boldsymbol{maxval}(t) = 0$

(ii) $s \equiv \mathsf{Update}(s_0, t_1, t_2).$

$$\boldsymbol{maxval}(t) = \max(\boldsymbol{maxval}(\mathsf{Ap}(s_0, t_0)),\ \boldsymbol{maxval}(t_2))$$

(iii) $s \equiv \mathsf{Newlength}(s_0, t_1).$ $\quad \boldsymbol{maxval}(t) = \boldsymbol{maxval}(\mathsf{Ap}(s_0, t_0))$

(iv) $s \equiv \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}.$

$$\boldsymbol{maxval}(t) = \max(\boldsymbol{maxval}(\mathsf{Ap}(s_1, t_0)),\ \boldsymbol{maxval}(\mathsf{Ap}(s_2, t_0)))$$

(7) $t \equiv \mathsf{Lgth}(s)$, where s is of starred sort. There are 4 subcases, according to the form of s:

(i) $s \equiv \mathsf{Null}.$ $\quad \boldsymbol{maxval}(t) = 0$

(ii) $s \equiv \mathsf{Update}(s_0, t_1, t_2).$ $\quad \boldsymbol{maxval}(t) = \boldsymbol{maxval}(\mathsf{Lgth}(s_0))$

(iii) $s \equiv \mathsf{Newlength}(s_0, t_1).$ $\quad \boldsymbol{maxval}(t) = \boldsymbol{maxval}(t_1)$

(iv) $s \equiv \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}.$

$$\boldsymbol{maxval}(t) = \max(\boldsymbol{maxval}(\mathsf{Lgth}(s_1)),\ \boldsymbol{maxval}(\mathsf{Lgth}(s_2)))$$

Remark 2. This definition, which is used in the syntactic transformation described in the Lemma below, is only possible under our two assumptions: (i) R is algebraic, and (ii) A is strictly standard. We can give a counterexample with the absence of each condition. For (i): suppose the type $\vec{k}$ of R includes the sort $k_j = \mathsf{N}$, for some j, $1 \leq j \leq n$. Now the inductive generation of $\boldsymbol{Term}^*_{\mathsf{N}}(\vec{v})$ must include the case $t \equiv v_j$. But in this case, it is clearly impossible to define $\boldsymbol{maxval}(t)$! For (ii): suppose Σ includes the sort Z of integers, and the function symbol F of sort $(\mathsf{Z}; \mathsf{N})$, with the interpretation $F^A(z) = \max(z, 0)$. Then Σ and A are not strictly standard. Suppose now that the type $\vec{k}$ of R includes the sort $k_j = \mathsf{Z}$. Then the inductive generation of $\boldsymbol{Term}^*_{\mathsf{N}}(\vec{v})$ must include the case $t \equiv F(v_j)$. But again, in this case, it is clearly impossible to define $\boldsymbol{maxval}(t)$!

Lemma. *Any term in* $\boldsymbol{Term}^*(\vec{v})$ *is equivalent to an unstarred term.*

Proof. We will define a syntactic transformation from $\boldsymbol{Term}^*(\vec{v})$ to $\boldsymbol{Term}(\vec{v})$. First notice that if a term in $\boldsymbol{Term}^*(\vec{v})$ contains a subterm of starred sort, then it must contain a (maximal) subterm s of starred sort in one of the three contexts:

$$s = s', \qquad \mathsf{Ap}(s,t), \qquad \mathsf{Lgth}(s).$$

We will show how to eliminate each of these three contexts in turn.

Step 1. Transform all contexts of the form $s_1 = s_2$ (s_i of starred sort) to:

$$\mathsf{Lgth}(s_1) = \mathsf{Lgth}(s_2) \;\wedge\; \bigwedge_{k=1}^{M} \big(\mathsf{Ap}(s_1,\bar{k}) = \mathsf{Ap}(s_2,\bar{k})\big).$$

where $M = \boldsymbol{maxval}(\mathsf{Lgth}(s_1))$, and $\bar{k}$ is the numeral for k (that is, '0' preceded by 'S' k times).

Now all (maximal) occurrences of a subterm s of starred sort are in a context *either* $\mathsf{Ap}(s,t)$ *or* $\mathsf{Lgth}(s)$.

Step 2. Transform all contexts of the form $\mathsf{Ap}(s,t)$, by structural induction on s. There are 4 cases, according to the form of s:

(i) $s \equiv \mathsf{Null}$:

$$\mathsf{Ap}(s,t) \quad\longmapsto\quad \mathsf{unspec}.$$

(ii) $s \equiv \mathsf{Update}(s_0,t_0,t_1)$:

$$\mathsf{Ap}(s,t) \quad\longmapsto\quad \mathsf{if}\ t = t_0 < \mathsf{Lgth}(s_0)\ \mathsf{then}\ t_1\ \mathsf{else}\ \mathsf{Ap}(s_0,t)\ \mathsf{fi}.$$

(iii) $s \equiv \mathsf{Newlength}(s_0,t_0)$:

$$\mathsf{Ap}(s,t) \quad\longmapsto\quad \mathsf{if}\ t < t_0\ \mathsf{then}\ \mathsf{Ap}(s_0,t)\ \mathsf{else}\ \mathsf{unspec}\ \mathsf{fi}.$$

(iv) $s \equiv \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}$:

$$\mathsf{Ap}(s,t) \quad\longmapsto\quad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{Ap}(s_1,t)\ \mathsf{else}\ \mathsf{Ap}(s_2,t)\ \mathsf{fi}.$$

Remark 3. Note the introduction of the '$\mathsf{if} \cdots \mathsf{fi}$' construction in cases (ii) and (iii). (See Remark 1 in §2.1.)

Step 3. Transform all contexts of the form $\mathsf{Lgth}(s)$, by structural induction on s. Again there are 4 cases, according to the form of s:

(i) $s \equiv \mathsf{Null}$:

$$\mathsf{Lgth}(s) \quad\longmapsto\quad 0.$$

(ii) $s \equiv \mathsf{Update}(s_0,t_0,t_1)$:

$$\mathsf{Lgth}(s) \quad\longmapsto\quad \mathsf{Lgth}(s_0).$$

(iii) $s \equiv \mathsf{Newlength}(s_0,t_0)$:

$$\mathsf{Lgth}(s) \quad\longmapsto\quad t_0.$$

(iv) $s \equiv \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}$:

$$\mathsf{Lgth}(s) \quad\longmapsto\quad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{Lgth}(s_1)\ \mathsf{else}\ \mathsf{Lgth}(s_2)\ \mathsf{fi}.$$

By these three steps, we transform any term in $\boldsymbol{Term}^*(\vec{v})$ into an unstarred term, as desired. $\square$

To complete the proof of the Theorem: for any boolean $b \in \boldsymbol{Term}^*_{\mathsf{B}}(\vec{v})$, let b' be the unstarred boolean associated with b by the Lemma.

The characterization (2) of R can now be rewritten as

$$R = \{\vec{x} \in A[\vec{k}] \mid \bigvee_i b'_i[\vec{x}]\}. \tag{3}$$

Because the disjunction in (2), and the transformation $b \mapsto b'$ of the Lemma, are both effective, the disjunction in (3) is also effective. This proves the Theorem.

Now for any relation R definable by an effective disjunction as in (3), there is a partial recursive function β' such that $\beta'(i)$ is the Gödel number of b'_i. Hence

$$R = \{\vec{x} \in A[\vec{k}] \mid \text{for some } i,\ \boldsymbol{te}_A(\beta'(i), \vec{x}\,) \downarrow \mathtt{tt}\}. \tag{4}$$

and so (by the same argument as before) we can see again that R is 'while^*' semicomputable. This gives us a partial converse to Engeler's Lemma:

Proposition. *Let R be a relation on the Σ-structure A. If R can be expressed as an effective infinite disjunction of booleans over Σ, then R is 'while^*' semicomputable.*

Combining Engeler's Lemma (Version 2) with this result, gives the following "structural" characterization of 'while^*' semicomputable relations, among those which are algebraic on strictly standard structures.

Corollary 1. *Suppose R is an algebraic relation on a strictly standard Σ-structure. Then R can be expressed as an effective (infinite) disjunction of booleans over Σ iff R is 'while^*' semicomputable.*

If, moreover, the function $\boldsymbol{te}_A$ is 'loop' (rather than 'loop^*') computable on A, then we can say more from equation (4):

Corollary 2. *Suppose R is an algebraic relation on a strictly standard Σ-structure. Suppose further that term evaluation on A is 'while' computable on A. Then the following assertions are equivalent:*

(i) R is 'while' semicomputable,

(ii) R is 'while^*' semicomputable,

(iii) R can be expressed as an effective infinite disjunction of booleans over Σ.

An example of a structure on which term evaluation is 'loop' (and hence 'while') computable is the (standardized version of) the reals. Other examples, with applications of Corollary 2, are given in [11].

8. Second Proof of the Projective Equivalence Theorem

We give a second proof of the Projective Equivalence Theorem stated in §3.3. (The first proof, in §4.3, used computation predicates and the notion of $\mathbf{\Sigma}_1^*$ definability, and needed an assumption about equality sorts.)

Suppose $R \in A[\vec{k}]$ is projectively 'while^*' semicomputable. Then there is a 'while^*' semicomputable relation R_1 on A^*, of type $(\vec{k}, \vec{l}^*)$ (for some list $\vec{l}^*$ of starred and unstarred sorts) such that

$$R(\vec{x}) \iff \exists \vec{y}^{\,*} R_1(\vec{x}, \vec{y}^{\,*})$$

By Engeler's Lemma, Version 1 (§6.2) applied to A^*, there is an effective sequence $b_i^*(\vec{x}, \vec{y}^{\,*})$ $(i = 0, 1, 2, \dots)$ of booleans over Σ^* (*i.e.*, effective in their Gödel numbers) such that $R_1(\vec{x}, \vec{y}^{\,*})$ is equivalent over $\mathbb{K}$ to their disjunction. So there is a partial recursive function β^* such that for all $i \geq 0$, $\beta^*(i)$ is a Gödel number for b_i^*, and

$$R_1(\vec{x}, \vec{y}^{\,*}) \iff \text{for some } i, \ \ \boldsymbol{te}_A^*(\beta^*(i),\, \vec{x}, \vec{y}^{\,*}) \downarrow \mathtt{tt}.$$

Further, β^* is total, and $\boldsymbol{te}_A^*$ is 'loop' computable on A^* (by Lemma 1 in §2.3), and hence also total. Hence the function g defined by

$$g(i, \vec{x}, \vec{y}^{\,*}) \ =_{df} \ \boldsymbol{te}_A^*(\beta^*(i),\, \vec{x}, \vec{y}^{\,*})$$

is total, and 'while^*' computable on A^* (by Lemma 2 in §2.3). Hence the relation

$$R_0(i, \vec{x}, \vec{y}^{\,*}) \iff_{df} g(i, \vec{x}, \vec{y}^{\,*}) = \mathtt{tt}$$

is 'while' computable on A^*, and so the relation

$$R(\vec{x}) \iff \exists \vec{y}^{\,*} \exists i \, R_0(i, \vec{x}, \vec{y}^{\,*})$$

is projectively 'while^*' computable on A.

9. Relations Definable by 'while' Programs with Random Assignments

We now consider the 'while' programming language over Σ, extended by the *random assignment* $v := ?$ for all sorts i of Σ. This is an example of *nondeterministic computation.*

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

Remark 1. The semantics of such programs can be obtained by a modification of the semantics of 'while' programs given in [1, §2.2.3], by taking the meaning of a statement S to be a function $\mathcal{M}_A(S)$ from states to *sets of* next possible states.

In the case that S is a random assignment $v_i := ?$, $\mathcal{M}_A(S)(\sigma)$ is the set of *all states* which agree with σ on all variables *except* v_i.

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

(1) R is 'while'-*random semicomputable* (on A, by $[S, \vec{v}]$) if S is a 'while' program over A with random assignments (to simple variables).

(2) R is 'while*'-*random semicomputable* if S is a 'while' program over A^* with random assignments (possibly to starred variables).

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 these two notions of definability compare. We will show that, at least over A^*, they coincide — both are equivalent to projective 'while*' semicomputability.

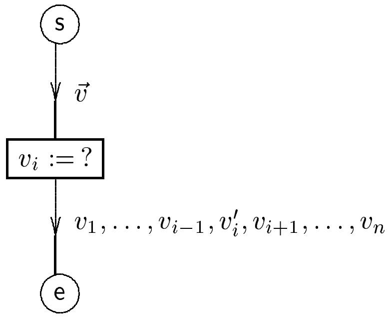
Theorem. *Let R be a relation on A. Then*

R is 'while*'-random semicomputable $\Leftrightarrow$ R is projectively 'while*' semicomputable. ∎

The direction ($\Longleftarrow$) follows from Remark 2. The proof of the direction ($\Longrightarrow$) occupies the rest of this section. For ease of exposition, we will first assume that R is 'while'-*random semicomputable.* So suppose that R is defined over A by a 'while'-random i-program $[S, \vec{v}]$, without starred variables.

We will define a *computation tree* $\mathcal{T}[S, \vec{v}]$ for "while-random" programs S with $\boldsymbol{var}(()S) \subseteq \vec{v}$, extending the definition for 'while' programs given in Section 5. There is one new case:

(vi) $S \equiv v_i := ?$. Then $\mathcal{T}[S, \vec{v}] \equiv$



So v_i is replaced by a *new variable* v_i' of the same sort.

Notice that for an '?'-assignment S, and for terms $\vec{t} \equiv t_1, \ldots, t_n$, the tree $\mathcal{T}[S, \vec{t}\,]$ is

$$\begin{tikzcd} \mathsf{s} \arrow[d, "\vec{t}"] \\ \boxed{v_i := ?} \arrow[d, "{t_1, \ldots, t_{i-1}, v_i', t_{i+1}, \ldots, t_n}"] \\ \mathsf{e} \end{tikzcd}$$

where v_i' does not occur in $\vec{t}$. The intuition here is that there is nothing we can say about the "new" value of v_i, so we can only represent it by a brand new variable v_i'. If this asignment is followed by another assignment $v_i := ?$, we introduce *another* new variable v_i'', and so on.

In this way *the variables proliferate*, so that we cannot simply express the halting predicate as an infinite disjunction of booleans $\boldsymbol{B}[S, \vec{v}, \lambda]$, as we did in §6.1.

The solution is to represent all the variables $v_i, v_i', v_i'', \ldots$ which arise in this way for each i $(1 \le i \le n)$ by a *single starred variable* v_i^* (with $v_i^*[0], v_i^*[1], v_i^*[2], \ldots$ representing $v_i, v_i', v_i'', \ldots$). Then to each leaf λ of $\mathcal{T}[S, \vec{v}]$ there corresponds (as in §6.1) a boolean $\boldsymbol{B}[S, \vec{v}, \lambda]$, but now in the *starred variables* $\vec{v}^{\,*} \equiv v_1^*, \ldots, v_n^*$ of sorts $\vec{k}^* \equiv k_1^*, \ldots, k_n^*$ (where $\vec{v}$ have sorts $\vec{k} \equiv k_1, \ldots, k_n$). Again we can define the *halting predicate* of S with respect to $\vec{v}$ as a countable disjunction of these:

$$\boldsymbol{Halt}[S, \vec{v}] \ \equiv_{df} \ \bigvee_{j=0}^{\infty} \boldsymbol{B}[S, \vec{v}, \lambda_j].$$

Note however that the program variables of $\boldsymbol{Halt}[S, \vec{v}]$ are now among $\vec{v}^{\,*}$, not $\vec{v}$!

Again, there is a partial recursive function β of three arguments, such that $\beta(\ulcorner S \urcorner, \ulcorner \vec{v} \urcorner, j)$ is the Gödel number of $\boldsymbol{B}[S, \vec{v}, \lambda_j]$.

Let us define, for $\vec{x} \in A[\vec{k}]$ and $\vec{y}^{\,*} \in A[\vec{k}^*]$,

$$\boldsymbol{Init}(\vec{x}, \vec{y}^{\,*}) \ \equiv \ \bigwedge \big(x_i = y_i^*(0)\big),$$

which says that "$\vec{x}$ gives the initial values of $\vec{y}^{\,*}$". Then

$$R = \boldsymbol{Halt}^A[S, \vec{v}] = \{\vec{x} \in A[\vec{k}] \mid \exists \vec{y}^{\,*}, j \ \big(\boldsymbol{Init}(\vec{x}, \vec{y}^{\,*}) \wedge \boldsymbol{te}_A^*(\beta(\ulcorner S \urcorner, \ulcorner \vec{v} \urcorner, j), \vec{y}^{\,*}) \downarrow \mathtt{tt}\big)\},$$

which (since $\boldsymbol{te}_A^*$ is 'while' computable on A^*, by Lemma 3 in §2.3) is projectively 'while^*' semicomputable. This proves the theorem for the case that S does not contain starred variables.

Assume, finally, that S is a program over A^*, and so contains random assignments to starred variables. Now we can represent a sequence of random assignments to such a variable by a single *doubly starred variable*, or two-dimensional array, which can then be effectively coded in A^* (Remark in §1.4), and proceed as before.

10. Scope and Limits of Specification

We have shown that projections of semicomputable sets over A^* arise when the definitions of relations allow some form or other of *nondeterministic search.* We mention, for example:

- projective 'while^*' computability (§3.3)
- 'while^*' computability with initialization (§3.3)
- 'while^*' computability with random assignments (§9)
- $\mathbf{\Sigma}_1^*$ definability (§4)
- Horn clause definability over A^* [3],

as well as older notions, such as $\mathbf{\Sigma}^*$ *definability* of Montague [7], *search computability* of Moschovakis [5] and *existential inductive definability* of Moschovakis [6]. (See the discussion in [3, §7].)

These notions are directly relevant to the theoretical analysis of specification languages for deterministic programming languages. This is because specification is founded on relations, and computation is founded on functions. More precisely, we suppose that

(1) each algorithm or program S computes a function f on A,

(2) each specification of a program is a relation R on A,

(3) program S meets specification R if the function f computed by S is a selection function for $\forall x \exists y R(x, y)$ *i.e.*,

$$\forall x R(x, f(x))$$

holds on A.

Thus the essential feature of specification is the definition of the projected relation $\exists y R(x, y)$, for which a *computable* partial selection function f may or may not exist.

Consider semantic criteria for the design of a specification language. Informally, we may require the language to be sufficiently

(a) *complex* to express all computations of interest; and

(b) *simple* to allow effective testing of computations against specifications.

On the basis of the theory of computable functions (recall the Church-Turing Thesis in §2.4) we may require formally that the specification language be sufficiently

(a') *complex* to define all partial computable functions as selection functions;

(b') *simple* to allow testing to be computable.

Using material from §4.2, for example, we see that each partial computable function f on A^* is the selection function of its graph, and that

$$(\vec{x}, y) \in \boldsymbol{graph}(f) \iff \exists \vec{z}^* \, \boldsymbol{Compu}_f(\vec{x}, y, \vec{z}^*).$$

In particular, we see that $\boldsymbol{graph}(f)$ is projectively 'while^*' computable.

For testing of the specification $\exists y R(x, y)$ to be computable, it is sufficient for R to be computable. This means that for any x and y (where y may be "guessed" nondeterministically) we can decide whether y is a valid output output for input x, according to the specification.

Combining these formal requirements leads to a specification language which defines exactly the projectively semicomputable or computable sets. Let us call a specification language that satisfies the computability conditions (a') and (b') on its semantics an *algorithmic specification language.*

The details of this informal analysis of specification languages, together with related requirements such as machine readable syntax etc., will be given elsewhere. It will be seen that the language of $\boldsymbol{\Sigma}_1^*$ formulae described in Section 4 meets the conditions (a') and (b').

With these considerations, we can formulate, analogous to the generalized Church-Turing thesis for deterministic computation on abstract data types stated in §2.3:

A Generalized Church-Turing Thesis for Specification on Abstract Data Types. *Consider an algorithmic specification language over an abstract data type* ***dt****. The set of relations on a structure A, representing an implementation of the abstract data type* ***dt****, that can be expressed or specified in the language, is contained in the set of projectively 'while^*' semicomputable relations on A. The class of relations over a class $\mathbb{K}$ of structures, representing a class of implementations of the abstract data type* ***dt****, that can be expressed in the language, uniformly over all implementations of $\mathbb{K}$, is contained in the class of 'while^*' semicomputable relations over $\mathbb{K}$.*

Acknowledgments

The research of the first author (JVT) was supported by SERC Research Grant GR/F 59070 and by an academic travel grant from the British Council. The research of the second author (JIZ) was supported by a grant from the Science & 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.

We thank an anonymous referee for constructive suggestions.

References

[1] J.V. Tucker & J.I. Zucker, *Program Correctness over Abstract Data Types, with Error-State Semantics*, CWI Monographs #6, North-Holland, 1988.

[2] E. Engeler, *Formal Languages: Automata and Structures*, Markham Publishing Co, 1968.

[3] J.V. Tucker & J.I. Zucker, "Deterministic and nondeterministic computation, and Horn programs, on abstract data types," *Journal of Logic Programming* 13 (1992), 23–55.

[4] J.V. Tucker & J.I. Zucker, "Horn programs and semicomputable relations on abstract structures," in *16th International Colloquium on Automata, Languages and Programming, Stresa, Italy, July 1989*, G. Ausiello, M. Dezani-Ciancaglini & S. Ronchi della Rocca, eds., Lecture Notes in Computer Science #372, Springer-Verlag, 1989, 745–760.

[5] Y.N. Moschovakis, "Abstract first order computability I," *Transactions of the American Mathematical Society* 138 (1969), 427–464.

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

[7] R. Montague, "Recursion theory as a branch of model theory," in *Logic, Methodology and Philosophy of Science III*, B. van Rootselaar & J.F. Staal, eds., North-Holland, 1968, 63–86.

[8] M. Fitting, *Fundamentals of Generalized Recursion Theory*, North-Holland, 1981.

[9] J.V. Tucker, S.S. Wainer & J.I. Zucker, "Provable computable functions on abstract data types," in *17th International Colloquium on Automata, Languages and Programming, Warwick University, England, July 1990*, M.S. Paterson, ed., Lecture Notes in Computer Science #443, Springer-Verlag, 1990, 660–673.

[10] J.V. Tucker & J.I. Zucker, "Provable computable selection functions on abstract structures," in *Proof Theory*, P. Aczel, H. Simmons & S.S. Wainer, eds., Cambridge University Press, 1993, 277–306.

[11] J.V. Tucker & J.I. Zucker, "Examples of semicomputable sets of real and complex numbers," in *Constructivity in Computer Science: Summer Symposium, San Antonio, Texas, June 1991*, J.P. Myers, Jr. & M.J. O'Donnell, eds., Lecture Notes in Computer Science #613, Springer-Verlag, 1992, 179–198.

[12] S.A. Greibach, *Theory of Program Structures: Schemes, Semantics, Verification*, Lecture Notes in Computer Science #36, Springer-Verlag, 1975.

[13] J.C. Shepherdson, "Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory," in *Harvey Friedman's Research on the Foundations of Mathematics*, L.A. Harrington, M.D. Morley, A. Ščedrov & S.G. Simpson, eds., North-Holland, 1985, 285–308.

[14] S.C. Kleene, *Introduction to Metamathematics*, North-Holland, 1952.