Halting sets of programs over an algebra

H. Peter Gumm

Abstract

We show that the halting set of a program over a (possibly partial) universal algebra $\mathcal{A}$ is a countable directed union of basic sets, where a basic set is a subset of (a power of) $\mathcal{A}$, definable by a conjunction of atomic and negated atomic formulae in the first order language of $\mathcal{A}$.

If $\mathcal{A}$ is "satisfaction semicomplete", and standard Peano arithmetik can be defined in $\mathcal{A}$ then the halting sets of programs over $\mathcal{A}$ are the same as the output sets of programs over $\mathcal{A}$.

This generalizes and also simplifies a result of Blum, Shub, and Smale [BSS], and thus provides an elementary proof for the fact that most familiar fractals are undecidable sets.

1 Introduction

In [BSS] Blum, Shub and Smale have defined the notion of a "machine over R", where R is an ordered ring or field. Amongst many other interesting results they show that most Julia sets are not halting sets of machines over $\mathbb{R}$, and further, that for real closed fields $\mathbb{K}$ the output sets of machines over $\mathbb{K}$ coincide with the halting sets of machines over $\mathbb{K}$.

Their definition of machines over R as connected graphs with *input nodes, computation nodes, decision nodes* and *output nodes* is rather difficult to treat theoretically. Essentially those machines are flowchart schemes over the data type $\mathcal{R} = (R; +, -, *, /, (r)_{r\in R}; <)$ in the sense of Manna [M].

Loops in such flowcharts are the biggest hurdle in their theoretical treatment, since there is not a natural way of uniquely decomposing an arbitrary machine into smaller units.

For this reason, we shall consider a *while-language* over R (or over any universal algebra, for that matter). While-programs are inductively defined so each program can be uniquely decomposed into smaller programs, with simple assignments forming the atomic constituents. It is not surprising then that all definitions and proofs are done inductively and proceed rather straightforwardly, once the right concepts are prepared. The theory of while-programs over arbitrary data types, their semantics and their proof theory is well developed, see [MS], [B], [H], or the survey by Apt [A]. We will, nevertheless, develop here all notions and lemmata needed for our results which will generalize the following two results from [BSS]:

Theorem 1 (Blum, Schub, Smale) *The halting set of a machine over a ring R is a countable union of basic sets, where a basic set is given by a finite number of polynomial inequalities:*

$$\begin{array}{ll} h_i(\vec{x}) \leq 0, & i = 1, \ldots m \\ h_j(\vec{x}) < 0, & j = 1, \ldots, n \end{array}$$

Theorem 2 (Blum,Schub, Smale) *If $\mathbb{K}$ is a real closed field, then the output sets of machines over $\mathbb{K}$ are precisely the halting sets of machines over $\mathbb{K}$.*

For an interesting application, consider the construction of the familiar Cantor set $\mathcal{C}$. To construct it, start with the closed interval $[0, 1] \subseteq \mathbb{R}$ and call it $\mathcal{C}_0$. From $\mathcal{C}_0$ remove the middle third (the open interval $(\frac{1}{3}, \frac{2}{3})$) and obtain $\mathcal{C}_1 = \mathcal{C}_0 - (\frac{1}{3}, \frac{2}{3})$. $\mathcal{C}_1$ now consists of two disjoint intervals, so in the next step remove the "middle thirds" of each of these intervals, obtaining $\mathcal{C}_2$. In general then $\mathcal{C}_n$ consists of 2^n disjoint closed intervals and $\mathcal{C}_{n+1}$ is obtained by removing the "middle thirds" of each of these intervals. The Cantor set $\mathcal{C}$ is defined as the intersection of all $\mathcal{C}_n$. $\mathcal{C}$ has uncountably many elements and is totally disconnected. The following figure shows $\mathcal{C}_4$.

Thinking in terms of actually computing $\mathcal{C}$ (disregarding problems of real number representations or computing errors,) it seems clear that for a point p outside of $\mathcal{C}$ we shall after a finite time have determined that $p \notin \mathcal{C}$, but for a point $q \in \mathcal{C}$ we shall at no finite time be able to say with certainty that $q \in \mathcal{C}$. In short: *One needs an eraser to draw a Cantor set.*

The above two theorems indeed make this situation precise: We cannot have a machine over $\mathbb{R}$ that stops if and only if its input is from $\mathcal{C}$, nor can we have a machine over $\mathbb{R}$ whose potential outputs (when inputs range over all of $\mathbb{R}^n$) form precisely $\mathcal{C}$. This follows from the above theorems and the observation that each basic set over $\mathbb{R}$ has only finitely many connected components.

One might conjecture that the halting set $H(\mathtt{P})$ of a program $\mathtt{P}$ over $\mathbb{R}$ could somehow be determined by its intersection with $\mathbb{Q}$. For $\mathbb{Q}$ we could then simply refer to the standard notion of computability. To see that this is not the case, consider the following two examples:

Example 1.1 *Let $r \in \mathbb{R}$ so that the sequence of digits of r (in its standard representation) is a non-computable sequence. Then both the set $(-\infty, r]$ and its complement are halting sets over $\mathbb{R}$. If their intersections with $\mathbb{Q}$ were enumerable in the sense of classical recursive function theory, they would be decidable, allowing us to construct an algorithm generating the sequence of digits of r.*

Example 1.2 *Clearly $\mathbb{R}$ is a halting set for a program over $\mathbb{R}$. The set of all irrational numbers is not a halting set over $\mathbb{R}$, though its intersection with $\mathbb{Q}$ is trivially decidable.*

In the following we shall give a generalization of theorems 1 and 2, replacing the ring R with an arbitrary algebra. For theorem 2 the condition of being "real closed" will be replaced by the condition of "satisfaction semi-completeness", a notion coming from "Constraint Logic Programming".

2 While-programs over universal algebras

Assume a first order language $\mathcal{L}$ be given where $\mathcal{L}$ consists of collections of operation symbols $\mathcal{F}$ and relation symbols $\mathcal{R}$, each associated with a fixed arity. For simplicity of exposition we shall work with homogeneous (one-sorted) structures only, although in practice most (data)-structures are heterogeneous (many-sorted). There is no obstacle other than the more complex notation when treating the heterogeneous case. Also we shall allow our algebraic structures to be partial, in that for some $f \in \mathcal{F}$ or some $r \in \mathcal{R}$, $f^{\mathcal{A}}(a_1, \ldots, a_n)$ or $r^{\mathcal{A}}(a_1, \ldots, a_n)$ could be undefined for certain elements $(a_1, \ldots, a_n)$.

Assume a countable set $X = \{x_1, \ldots, x_n, \ldots\}$ of variables be given. We refer to [BS],chapter V, for a definition of *terms of type $\mathcal{L}$ over X*, *atomic formulas of type $\mathcal{L}$ over X* and *first order formulas of type $\mathcal{L}$ over X*, (also called $\mathcal{L}$-formulas). We shall only have to deal with *open* $\mathcal{L}$-formulas, i.e. $\mathcal{L}$-formulas without quantifiers. An $\mathcal{L}$-formula that is atomic or the negation of an atomic formula is called a *literal*, and a finite conjunction of literals is called a $\wedge$-*clause*.

Next define a programming language over $\mathcal{L}$ using the the following inductive definition of programs :

Definition 2.1 (Assignment) *An $\mathcal{L} -$ assignment is an expression of the form "x`:=` t", where x is a variable and t an $\mathcal{L}$-term*

Definition 2.2 ($\mathcal{L}$-Programs)
(i) Every $\mathcal{L}$-assignment is a program over $\mathcal{L}$.

If `P` *and* `Q` *are programs over $\mathcal{L}$, and if c is an open $\mathcal{L}$-formula, then the following are programs over $\mathcal{L}$:*

(ii) `P ; Q`,	*(sequencing)*
(iii) `IF` c `THEN P ELSE Q`,	*(conditional)*
(iv) `WHILE` c `DO P`	*(while loop)*

In the next section we shall supply a precise definition for the meaning $\mathcal{M}$ of a program `P`. The following remarks, however should already be clear from an intuitive understanding:

(1) The right hand side of an assignment could be restricted to be either a variable or a function symbol applied to variables. Together with (ii) all term-functions can still be computed.

(2) The condition c in the if-then-else construction could be restricted to be an atomic formula. The constructions

`IF` c_1 `THEN P ELSE IF` c_2 `THEN P ELSE Q`,
`IF` c_1 `THEN IF` c_2 `THEN P ELSE Q ELSE Q`,
`IF` c_1 `THEN Q ELSE P`,

could be used in place of `IF` c `THEN P ELSE Q` when c is of the form $c_1 \vee c_2$, $c_1 \wedge c_2$, $\neg c_1$, respectively. This is also known as "short circuit evaluation".

(3) We shall frequently refer to the following two trivial programs :

`SKIP` stands for the program "x`:=` x" , and
`LOOP` stands for the program "`WHILE` $x = x$ `DO SKIP`",
moreover we use
"`IF` c `THEN P`" as a short form of "`IF` c `THEN P ELSE SKIP`".

3 Semantics

Given an $\mathcal{L}$-structure $\mathcal{A} = (A, \mathcal{F}, \mathcal{R})$, the *meaning* of terms and formulas is defined relative to interpretations of the variables from X in $\mathcal{A}$. An *interpretation* is any map $\sigma : X \to A$. This map naturally extends to a homomorphism $\overline{\sigma} : \mathcal{T}_{\mathcal{L}}(X) \to \mathcal{A}$, where $\mathcal{T}_{\mathcal{L}}(X)$ is the algebra of all $\mathcal{L}$-terms over X, and further to a (homo)morphism $\overline{\sigma} : \mathcal{F}_{\mathcal{L}}(X) \to \mathbb{B}$, from the set (Lindenbaum-algebra) of all open $\mathcal{L}$-formulas over X to the Boolean algebra $\mathbb{B} = (\{T, F\}; \vee, \wedge, \neg, F, T)$ of truth values.

In the context of programming an interpretation is called a *state*. In its most trivial form, a variable can be thought of as a memory location.

We set $\mathcal{S} = [X \to A]$, the set of all maps from X to A. The only way a state can be changed is by modifying the value of a variable. If σ is a state and a in A, then we denote by $\sigma + \{x \to a\}$ the updated state σ_0 with

$$\sigma_0(v) = \begin{cases} a, & if\ v = x \\ \sigma(v), & otherwise. \end{cases}$$

The following is a preliminary definition of the meaning $\mathcal{M}$ of programs not containing **while**, and where all operations and relations of $\mathcal{A}$ are assumed total. As is common practice, we enclose the program argument to $\mathcal{M}$ in double brackets.

Definition 3.1 (Meaning) *:*

$$\begin{array}{rl} (i) & \mathcal{M}[\![x\texttt{:= }t]\!](\sigma) = \sigma + \{x \rightarrow \overline{\sigma}(t)\} \\ (ii) & \mathcal{M}[\![\texttt{P ; Q}]\!](\sigma) = \mathcal{M}[\![\texttt{Q}]\!](\mathcal{M}[\![\texttt{P}]\!](\sigma)), \\ (iii) & \mathcal{M}[\![\texttt{IF } c \texttt{ THEN P ELSE Q}]\!](\sigma) = \left\{ \begin{array}{ll} \mathcal{M}[\![\texttt{P}]\!](\sigma), & if\ \overline{\sigma}(c) = T \\ \mathcal{M}[\![\texttt{Q}]\!](\sigma), & if\ \overline{\sigma}(c) = F \end{array} \right. \end{array}$$

Note that (ii) states that $\mathcal{M}[\![\texttt{P ; Q}]\!] = \mathcal{M}[\![\texttt{Q}]\!] \circ \mathcal{M}[\![\texttt{P}]\!]$, and that $\mathcal{M}[\![\texttt{SKIP}]\!] = \mathcal{M}[\![x\texttt{:= }x]\!] = \mathcal{M}[\![y\texttt{:= }y]\!] = id_{\mathcal{S}}$ for any $x, y \in X$.

The while-construct may introduce nonterminating programs, hence the meaning function $\mathcal{M}$ may become partial. We shall make it formally total by adding a new fictitious state $\perp_{\mathcal{S}}$ to the set of all states. Operationally, $\perp_{\mathcal{S}}$ represents the state the machine is in when it is caught in an infinite loop. In a similar vein, we shall make partial operations and relations over $\mathcal{A}$ total by adding to $\mathcal{A}$ the new element $\perp_{\mathcal{A}}$ and to $\mathbb{B}$ the new element $\perp_{\mathbb{B}}$, thus we get $\mathcal{S}^+ = \mathcal{S} \cup \{\perp_{\mathcal{S}}\}$, $\mathbb{B}^+ = \{\perp_{\mathbb{B}}, T, F\}$ and $\mathcal{A}^+ = \mathcal{A} \cup \{\perp_{\mathcal{A}}\}$. We shall set $\perp_{\mathcal{S}}(t) = \perp_{\mathcal{A}}$ and $\perp_{\mathcal{S}}(c) = \perp_{\mathbb{B}}$ for every term t and every open formula c.

A *complete partial order (cpo)* can be defined on each of these sets, that is a partial order $\sqsubseteq$ *(less defined than)* with the $\perp$-element as the smallest element, so that suprema of directed sets always exist. We simply define

$$x \sqsubseteq y \Leftrightarrow (x = y) \vee (x = \perp).$$

The resulting cpo is called *flat*. The operations $(f \in \mathcal{F})$ and relations $(r \in \mathcal{R})$ are extended to the $\perp$-element by setting $f(a_1, \ldots, a_n) = \perp_{\mathcal{A}}$ (resp. $r(a_1, \ldots, a_n) = \perp_{\mathbb{B}}$), if $\perp_{\mathcal{A}} \in \{a_1, \ldots, a_n\}$ or if f (resp. r) is undefined on $(a_1, \ldots, a_n)$. An important point is that the operations become *continuous* with respect to $\sqsubseteq$. For the boolean operations $\vee$ and $\wedge$ there is a second natural way of continuously extending them to $\mathbb{B}^+$ as

$$x \vee y = \left\{ \begin{array}{ll} T & \text{if } x = T \\ F & \text{if } x = F \text{ and } y = F \\ \perp_{\mathbb{B}} & \text{otherwise} \end{array} \right. .$$

$\wedge$ is extended analogously. In computing practice this extension of the boolean operations is called *short circuit evaluation*. This is the interpretation of $\wedge$ and $\vee$ that we adopt. Note that now $T = T \vee \perp_{\mathbb{B}} \neq \perp_{\mathbb{B}} \vee T = \perp_{\mathbb{B}}$. Thus, for instance, $(x \neq 0) \wedge (1/x \neq 0)$ is never $\perp_{\mathbb{B}}$, even though $1/x = \perp_{\mathcal{A}}$ for $x = 0$. The distributive laws and deMorgan's laws remain valid, so that we still can write each open formula as a disjunction of $\wedge$-clauses.

If all operations and relations of $\mathcal{A}$ were total, or if their domain could be described by an open formula, (as is the case for division in $\mathbb{R}$) then we would not need $\bot_{\mathcal{A}}$, nor $\bot_{\mathbb{B}}$. An assignment x`:=` t could be replaced with `IF` *defined*(t) `THEN` x`:=` t `ELSE LOOP`, similarly could the test in an if-then-else be protected.

The updating operation for states is extended by setting $\sigma + \{x \to a\} = \bot_{\mathcal{S}}$ in the cases where $\sigma = \bot_{\mathcal{S}}$ or $a = \bot_{\mathcal{A}}$.

The new meaning function $\mathcal{M}$ will belong to $[\mathcal{S}^+ \to \mathcal{S}^+]$, the cpo of continuous maps from $\mathcal{S}^+$ to $\mathcal{S}^+$. Its least element Ω is given by $\Omega(\sigma) = \bot_{\mathcal{S}}$ for all $\sigma \in \mathcal{S}^+$.

Definition 3.2 (Meaning) *:*

$$\begin{array}{rl}
(i) & \mathcal{M}[\![x\texttt{:=}\ t]\!](\sigma) = \sigma + \{x \to \overline{\sigma}(t)\} \\
(ii) & \mathcal{M}[\![\texttt{P ; Q}]\!] = \mathcal{M}[\![\texttt{Q}]\!] \circ \mathcal{M}[\![\texttt{P}]\!], \\
(iii) & \mathcal{M}[\![\texttt{IF } c \texttt{ THEN P ELSE Q}]\!](\sigma) = \begin{cases} \mathcal{M}[\![\texttt{P}]\!](\sigma), & if\ \overline{\sigma}(c) = T \\ \mathcal{M}[\![\texttt{Q}]\!](\sigma), & if\ \overline{\sigma}(c) = F \\ \bot_{\mathcal{S}} & if\ \overline{\sigma}(c) = \bot_{\mathbb{B}} \end{cases} \\
(iv) & \mathcal{M}[\![\texttt{WHILE } c \texttt{ DO P}]\!] = \bigsqcup \phi_i, where \\
 & \begin{cases} \phi_0 & = \Omega \\ \phi_{k+1}(\sigma) & = \begin{cases} \bot_{\mathcal{S}} & if\ \overline{\sigma}(c) = \bot_{\mathbb{B}} \\ \sigma & if\ \overline{\sigma}(c) = F \\ \phi_k(\mathcal{M}[\![\texttt{P}]\!](\sigma)) & if\ \overline{\sigma}(c) = T \end{cases} \end{cases}
\end{array}$$

In order for the supremum in (iv) to exist, one must check that the ϕ_i form an increasing chain. Since $\mathcal{S}^+$ is flat, this is equivalent to

$$\forall \sigma \in \mathcal{S}^+ \phi_i(\sigma) = \tau \neq \bot_{\mathcal{S}} \Rightarrow \phi_{i+1}(\sigma) = \tau.$$

Note that in particular $\mathcal{M}[\![\texttt{LOOP}]\!] = \Omega$. Also, by induction over the structure of programs it is easily seen that $\mathcal{M}[\![\texttt{P}]\!](\bot_{\mathcal{S}}) = \bot_{\mathcal{S}}$ for every program `P`.

It is shown in [B] that the above denotational semantics $\mathcal{M}[\![\texttt{P}]\!]$ is equivalent to the operational semantics $\mathcal{O}[\![\texttt{P}]\!]$ where $\mathcal{O}[\![\texttt{P}]\!] = \mathcal{M}[\![\texttt{P}]\!]$ for the case where `P` is an assignment, a sequencing or an if-then-else. For a while-loop the operational semantics is defined as

$$(iv') \quad \mathcal{O}[\![\texttt{WHILE } c \texttt{ DO P}]\!](\sigma) = \begin{cases} \tau & \text{if there exists an } n \text{ and } \sigma_0, \ldots, \sigma_n \in \mathcal{S} \\ & \text{such that } \sigma_0 = \sigma,\ \sigma_n = \tau,\ \overline{\sigma}_n(c) = F, \\ & \overline{\sigma}_i(c) = T \text{ and } \mathcal{O}[\![\texttt{P}]\!](\sigma_i) = \sigma_{i+1} \\ & \text{for } i = 0, \ldots, n-1 \\ \bot_{\mathcal{S}} & \text{otherwise} \end{cases}$$

For the next chapter we shall need a better understanding of the ϕ_i in the previous definition. First let us define for any program `P` and natural number k the k-fold iterate of `P` as:

$$\begin{aligned} \texttt{P}^0 &= \texttt{SKIP} \\ \texttt{P}^{k+1} &= \texttt{P}; \texttt{P}^k \end{aligned}$$

Lemma 3.3 *With the notation of definition 3.2 we have:*

$$\phi_{i+1} = \mathcal{M}[\![(\texttt{IF c THEN P})^{\texttt{i}} \,; \texttt{IF c THEN LOOP}]\!]$$

Proof: Use induction over i. In the inductive step we need to use associativity of "**;**" as follows :

$$\begin{aligned} &\mathcal{M}[\![(\texttt{IF c THEN P})^{k+1}; \texttt{IF c THEN LOOP}]\!] \\ &= \mathcal{M}[\![\texttt{IF c THEN P}; (\texttt{IF c THEN P})^k; \texttt{IF c THEN LOOP}]\!] \\ &= \phi_{k+1} \circ \mathcal{M}[\![\texttt{IF c THEN P}]\!] \\ &= \phi_{k+2}. \end{aligned}$$

4 Halting Sets

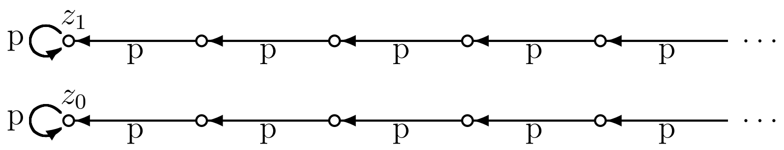
We shall now define the *halting set* of a program `P` as the set of all states $\sigma \in \mathcal{S}$ such that `P` will eventually halt when started in state σ. (If `P` contains variables $x_1, \ldots, x_n$ the halting set may be considered as a subset of $\mathcal{A}^n$).

Definition 4.1 (Halting set) $H(\texttt{P}) = \{\sigma \in \mathcal{S} | \mathcal{M}[\![\texttt{P}]\!](\sigma) \neq \bot_{\mathcal{S}}\}$

Note that $H(\texttt{LOOP}) = \{\}$, $H(\texttt{SKIP}) = \mathcal{S}$ and $\bot_{\mathcal{S}} \notin H(\texttt{P})$.

A first order description of $H(\texttt{P})$ in the language $\mathcal{L}$ may well be impossible. M. Wand in [W] gave an example of a data structure whose Hoare logic is incomplete. A simplified version of this datatype can be used to construct a program whose halting set $H(\texttt{P})$ is not first order definable:

Example 4.2 *Let $\mathcal{N} = (\mathbb{N};p)$ be the natural numbers with the predecessor operation : $p(0) = 0$ and $p(n+1) = n$. Let $\mathcal{N} + \mathcal{N}$ be the algebra consisting of two disjoint copies of $\mathcal{N}$ and let z_0 and z_1 be unary relations denoting the zero-elements in the respective copies. Then the halting set of the program "*`WHILE` *not $z_0(x)$* `DO` *x*`:=` *$p(x)$" is the first copy of $\mathcal{N}$ which is not first order definable within $\mathcal{N} + \mathcal{N}$.*



Definition 3.2 and lemma 3.3 show how the semantics of single while-loops are approximated by k-fold iterations of conditionals and the `LOOP` construct. We shall need to approximate complete programs `P` (perhaps containing nested while-loops), by a one-parameter family of programs not containing while-loops. To do this we introduce `LOOP` as a primitive construct with the semantic definition $\mathcal{M}[\![\texttt{LOOP}]\!] = \Omega$. A program that is constructed from assignments, sequencing, conditionals and `LOOP` will be called *almost straight line.* For any natural number k we define now :

Definition 4.3 *Let* `P` *be a while-program and k a natural number, then we define an almost-straight-line program $\texttt{P}_{(k)}$ inductively as:*

$(x{:=}t)_{(k)}$	$= x{:=}t$
$(\texttt{R}\,;\texttt{S})_{(k)}$	$= \texttt{R}_{(k)}\,;\texttt{S}_{(k)}$
$(\texttt{IF c THEN R ELSE S})_{(k)}$	$= \texttt{IF c THEN R}_{(k)}\texttt{ ELSE S}_{(k)}$
$(\texttt{WHILE c DO R})_{(k)}$	$= (\texttt{IF c THEN R}_{(k)})^{k}\,;\texttt{IF c THEN LOOP}$

Then we get :

Theorem 4.4 *For each program* `P` *we have*

(i) $H(\texttt{P}_{(k)}) \subseteq H(\texttt{P}_{(k+1)})$
(ii) $H(\texttt{P}) = \bigcup H(\texttt{P}_{(k)})$

Before we can prove this we need the following lemma:

Lemma 4.5 *For all programs* `P` *and natural numbers* k *we have:*

$$\begin{array}{ll} (i) & \mathcal{M}[\![\mathtt{P}_{(k)}]\!] \sqsubseteq \mathcal{M}[\![\mathtt{P}_{(k+1)}]\!] \\ (ii) & \mathcal{M}[\![\mathtt{P}]\!] = \bigsqcup \mathcal{M}[\![\mathtt{P}_{(k)}]\!] \end{array}$$

<u>Proof :</u> We shall show by induction on the structure of programs that

$$\mathcal{M}[\![\mathtt{P}_{(k)}]\!](\sigma) = \tau \neq \bot_{\mathcal{S}} \Rightarrow \mathcal{M}[\![\mathtt{P}_{(k+1)}]\!](\sigma) = \tau.$$

If `P` is an assignment, then the claim is obvious. If `P` is a sequencing, $\mathtt{P} = \mathtt{R}\ ;\ \mathtt{S}$, then $\mathcal{M}[\![(\mathtt{R}\,;\mathtt{S})_{(k)}]\!](\sigma) = \tau \neq \bot_{\mathcal{S}}$ implies that $\mathcal{M}[\![\mathtt{R}_{(k)}]\!](\sigma) = \alpha \neq \bot_{\mathcal{S}}$ and $\mathcal{M}[\![\mathtt{S}_{(k)}]\!](\alpha) = \tau \neq \bot_{\mathcal{S}}$, hence $\mathcal{M}[\![\mathtt{R}_{(k+1)}]\!](\sigma) = \alpha$ and $\mathcal{M}[\![\mathtt{S}_{(k+1)}]\!](\alpha) = \tau$, so $\mathcal{M}[\![(\mathtt{R}\,;\mathtt{S})_{(k+1)}]\!](\sigma) = \tau$. The case where `P` is a conditional is similar. Now let $\mathtt{P} = \mathtt{WHILE}\ c\ \mathtt{DO}\ \mathtt{S}$.

$$\begin{aligned} \mathcal{M}[\![\mathtt{P}_{(k)}]\!](\sigma) &= \mathcal{M}[\![\mathtt{IF\ c\ THEN\ S_{(k)}})^k;\mathtt{IF\ c\ THEN\ LOOP}]\!](\sigma) \\ &= \mathcal{M}[\![\mathtt{IF\ c\ THEN\ LOOP}]\!](\mathcal{M}[\![(\mathtt{IF\ c\ THEN\ S_{(k)}})^k]\!](\sigma)) \\ &= \tau \neq \bot_{\mathcal{S}}. \end{aligned}$$

Hence $\mathcal{M}[\![(\mathtt{IF\ c\ THEN\ S_{(k)}})^k]\!](\sigma) = \tau \neq \bot_{\mathcal{S}}$ and $\overline{\tau}(\sigma) = F$, so

$$\begin{aligned} \mathcal{M}[\![(\mathtt{IF\ c\ THEN\ S_{(k)}})^k]\!](\sigma) &= \mathcal{M}[\![(\mathtt{IF\ c\ THEN\ S_{(k)}})^{k+1}]\!](\sigma) \\ &= \mathcal{M}[\![((\mathtt{IF\ c\ THEN\ S})^{k+1})_{(k)}]\!](\sigma) \\ &= \tau \end{aligned}$$

By the previous cases, and using the induction hypothesis on `S` this is equal to

$$\begin{aligned} \mathcal{M}[\![((\mathtt{IF\ c\ THEN\ S})^{k+1})_{(k+1)}]\!](\sigma) &= \mathcal{M}[\![(\mathtt{IF\ c\ THEN\ S_{(k+1)}})^{k+1}]\!](\sigma) \\ &= \mathcal{M}[\![\mathtt{P}_{(k+1)}]\!](\sigma) \\ &= \tau. \end{aligned}$$

(ii) : From (i) it follows that $\bigsqcup \mathcal{M}[\![\mathtt{P}_{(k)}]\!]$ exists. It is a simple exercise to show by induction that $\mathcal{M}[\![\mathtt{P}_{(k)}]\!] \sqsubseteq \mathcal{M}[\![\mathtt{P}]\!]$. We now show by structural induction that

$$\mathcal{M}[\![\mathtt{P}]\!](\sigma) = \tau \neq \bot_{\mathcal{S}} \Rightarrow \exists k\ \mathcal{M}[\![\mathtt{P}_{(k)}]\!](\sigma) = \tau.$$

With (i) the conclusion is actually equivalent to

$$\exists k\ \forall k' \ \geq k\ \mathcal{M}[\![\mathtt{P}_{(k')}]\!](\sigma) = \tau.$$

The cases where $\mathtt{P}$ is an assignment, sequencing or conditional are again easy. Now suppose $\mathcal{M}[\![\mathtt{WHILE\ c\ DO\ S}]\!](\sigma) = \tau \neq \bot_{\mathcal{S}}$. Then there exists a ϕ_k (see definition 3.2) such that $\phi_k(\sigma) = \tau \neq \bot_{\mathcal{S}}$. By lemma 3.3

$$\mathcal{M}[\![(\mathtt{IF\ c\ THEN\ S})^k; \mathtt{IF\ c\ THEN\ LOOP}]\!](\sigma) = \tau \neq \bot_{\mathcal{S}},$$

hence

$$\mathcal{M}[\![(\mathtt{IF\ c\ THEN\ S})^k]\!](\sigma) = \tau \neq \bot_{\mathcal{S}},$$

and $\overline{\sigma}(c) = F$. This means that there exist $\sigma = \sigma_1, \ldots, \sigma_n = \tau$ so that $\mathcal{M}[\![\mathtt{IF\ c\ THEN\ S}]\!](\sigma_i) = \sigma_{i+1}$. By induction hypothesis, for each i there is a k_i so that $\mathcal{M}[\![\mathtt{IF\ c\ THEN\ S_{k_i}}]\!](\sigma_i) = \sigma_{i+1}$. For $m = max(k_1, \ldots, k_n)$ we have $\mathcal{M}[\![\mathtt{IF\ c\ THEN\ S_{(m)}}]\!](\sigma_i) = \sigma_{i+1}$, and, finally $\mathcal{M}[\![\mathtt{P}_{(m)}]\!](\sigma) = \tau$.

Lemma 4.6 *Let* $\mathtt{P}$ *be an almost straight line program. Then there exists an open* $\mathcal{L}$*-formula* $h_{\mathtt{P}}$ *so that* $H(\mathtt{P}) = \{\sigma \in \mathcal{S} \mid \overline{\sigma}(h_{\mathtt{P}}) = T\}$.

Proof: For any program $\mathtt{P}$ and first order formula q define $H(P, q) = \{\sigma \in \mathcal{S} \mid \mathcal{M}[\![\mathtt{P}]\!](\sigma) = \tau \neq \bot_{\mathcal{S}}$ and $\overline{\tau}(q) = T\}$. Now for any almost straight line program $\mathtt{P}$ and any open formula q we define an open formula pre$(\mathtt{P}, q)$ inductively as follows:

(i) pre$(x\mathtt{:=}\ t, q) = (t = t) \wedge q\{t/x\}$
(ii) pre$(\mathtt{P}\ ;\ \mathtt{Q}, q) =$ pre$(\mathtt{P},$pre$(\mathtt{Q}, q))$
(iii) pre$(\mathtt{IF}\ c\ \mathtt{THEN\ P\ ELSE\ Q}, q) = (c \wedge$ pre$(\mathtt{P}, q)) \vee (\neg c \wedge$ pre$(\mathtt{Q}, q))$
(iv) pre$(\mathtt{LOOP}, q) = F$

$q\{t/x\}$ is the formula obtained from q by substituting each occurrence of x in q by t. Note that $\overline{\sigma}(t = t)$ is T iff $\overline{\sigma}(t)$ is defined, and $\bot_{\mathbb{B}}$ otherwise.

It can now be checked that for any almost-straight-line program $\mathtt{P}$ and any state σ we have that $\sigma \in H(\mathtt{P}, q) \Leftrightarrow \overline{\sigma}(pre(\mathtt{P}, q)) = T$. To finish the proof of the lemma, note that $H(\mathtt{P}) = H(\mathtt{P}, T)$, so $H(\mathtt{P})$ can be axiomatized by the open formula pre$(\mathtt{P}, T)$.

Theorem 4.7 *Let* $\mathtt{P}$ *be any program in the language* $\mathcal{L}$. *Let* $x_1, \ldots, x_n$ *be the variables occurring in* $\mathtt{P}$. *The halting set of* $\mathtt{P}$ *over* $\mathcal{A}$ *is a directed union of subsets of* $\mathcal{A}^n$*, each of which can be axiomatized by an open formula. Equivalently, the halting set can be axiomatized by an countable disjunction of* $\wedge$*-clauses.*

5 Output Sets

Dually to halting sets we could define the “output-set” of a program `P` as

$$O(\texttt{P}) = \{\tau \in \mathcal{S} | \exists \sigma \in \mathcal{S}\ \mathcal{M}[\![P]\!](\sigma) = \tau\}.$$

It is more common though, to restrict $O(\texttt{P})$ to a certain subset of variables, say $z_1, \ldots z_k$ which are designated as output variables. We define :

Definition 5.1 (output set) *A subset $O \subseteq A^n$ is an* output set *for a program* `P`*, if for some variables $z_1, \ldots, z_n \in X$ we have*

$$O = \{(\tau(z_1), \ldots, \tau(z_n)) \mid \exists \sigma \in \mathcal{S}\ [\ \mathcal{M}[\![\texttt{P}]\!](\sigma) = \tau \neq \bot_{\mathcal{S}}\ ]\}.$$

We denote this set by $\mathcal{O}(\texttt{P} : z_1, \ldots, z_n)$

Halting sets can also be viewed as subsets of $\mathcal{A}^k$, in particular if $x_1, \ldots, x_k$ are the only variables occurring in `P`. (If more variables occurred in P, we would have to make sure that the execution of P does not depend on these variables, e.g. by having them initialized at the beginning.) We may now ask, whether halting sets and output sets coincide.

Classical recursive function theory uses the data type $(\mathbb{N}; succ, pred, 0; =)$. In this case the halting sets are the same as the output sets of programs over the same data structure. This fact is due to the enumerability of $\mathbb{N}^k$. Given any output set $O \subseteq A^k$ and the associated program `P`, we let `P`’ be the program that takes an element from $\mathbb{N}^k$ and waits until this element appears in the output of `P`. The halting set of `P`’ is clearly the same as the output set of `P`.

We let now $\mathcal{A}$ be any data structure with at least two different elements, named 0 and 1. As noted in [BSS], one direction is always true: Let `P` be a machine (program) with halting set $H \subseteq \mathcal{A}^k$, then a program `P`’ can be constructed having H as output set: Let $x_1, \ldots, x_n$ be the variables occurring in `P`. Let $z_1, \ldots, z_n$ be new variables intended to contain the output when `P`’ terminates. Simply set

$$\texttt{P}' = z_1{:=}x_1; \ldots; z_n{:=}x_n; \texttt{P}.$$

The other direction, given an output set O for program `P`, to construct a program `P`’ with halting set $H(\texttt{P}) = O$, is not quite as easy; indeed we can prove the equivalence only in the case that $\mathcal{A}$ is *satisfaction semicomplete* in the sense defined below. In the case where $\mathcal{A}$ is a real closed field, or where $\mathcal{A} = \mathbb{N}$, this condition is satisfied.

Definition 5.2 (satisfaction semicomplete) *Let $\mathcal{L}$ be a language and $\mathcal{A}$ an $\mathcal{L}$-structure. $\mathcal{A}$ is satisfaction semicomplete, if there exists a while-program over $\mathcal{A}$ which will recognize all satisfiable $\wedge$-clauses over $\mathcal{L}$.*

In other words, for some representation of $\wedge$-clauses by elements of $\mathcal{A}$ there must be a while-program over $\mathcal{A}$ that will terminate with output $z = 1$ if and only if it received as input a representation of a satisfiable $\wedge$-clause.

Note that this is a weakening of the notion of *satisfaction completeness* which is of importance in the field of "Constraint Logic Programming", see Jaffar, Lassez [JL].

By a result of Tarski [T], real closed fields are satisfaction complete, that is, the satisfiable open formulas are even decidable. In the data structure $\mathbb{N}$, the satisfiable open formulas are semidecidable. Simply enumerate all elements of the domain until you find one satisfying the formula. For the same reason, the data structure $(\mathbb{Z}; +, *; <)$ is satisfaction semicomplete, but not satisfaction complete, see [C].

We need to do some extra calculations too, so we need to assume further that programs over $\mathbb{N}$ can be "emulated" by programs over $\mathcal{A}$. To this end we simply stipulate that a standard model of Peano-arithmetic can be defined in $\mathcal{A}$. Then we shall show:

Theorem 5.3 *Let $\mathcal{A}$ be an $\mathcal{L}$-structure that is satisfaction semicomplete. Further suppose that a standard model of Peano-arithmetic can be defined in $\mathcal{A}$. Then each output set of a program* `P` *over $\mathcal{L}$ is the halting set of some other program* `P'` *over $\mathcal{L}$. Thus halting sets and output sets over $\mathcal{A}$ coincide.*

Proof: Let `P` be a program and let u be a variable not occurring in `P`. For any natural number k we construct a program $\mathtt{P}_{[k]}$ by recursively replacing every `WHILE` c `DO S` by

$$(\texttt{IF c THEN S}_{[\texttt{k}]})^k \; ; \; \texttt{IF c THEN u:=1}$$

The following lemma can be proven analogous to the results in the previous chapter:

Lemma 5.4 *For any program* $\mathtt{P}$ *and any open formula* q *we have :*

(*i*) $H(\mathtt{P}, q) = \bigcup H(\mathtt{u{:=}0}\,;\mathtt{P}_{[\mathtt{k}]}, q \wedge (u = 0))$

(*ii*) $(a_1, \ldots, a_n) \in \mathcal{O}(\mathtt{P} : z_1, \ldots, z_n) \Leftrightarrow H(\mathtt{P}, z_1 = a_1 \wedge \ldots \wedge z_n = a_n) \neq \{\}$

Using this and lemma 4.6 we find that $(a_1, \ldots, a_n) \in \mathcal{O}(\mathtt{P} : z_1, \ldots, z_n)$ iff for some k, pre$(\mathtt{u{:=}0}\,;\mathtt{P}_{[\mathtt{k}]}, (z_1 = a_1\wedge, \ldots, \wedge a_n = u_n \wedge u = 0))$ is satisfiable.

Our assumptions about $\mathcal{A}$ allow us to test this: Since Peano-arithmetic can be defined in $\mathcal{A}$, we can represent $\mathcal{L}$-formulas in $\mathcal{A}$. We then calculate pre$(\mathtt{P}, q)$ according to the algorithm of lemma 4.3, and finally, using the assumption of satisfaction semicompleteness, we can probe whether pre$(\mathtt{u{:=}0}\,;\mathtt{P}_{[\mathtt{k}]}, T)$ is satisfiable. By the standard dovetailing technique, we can arrange it so that the satisfiability of pre$(\mathtt{u{:=}0}\,;\mathtt{P}_{[\mathtt{k}]}, T)$ is tested concurrently for increasing k. Thus, if for some k pre$(\mathtt{u{:=}0}\,;\mathtt{P}_{[\mathtt{k}]}, T)$ is satisfiable, we will eventually know. If it is not, our program will never stop, hence its halting set is precisely the output set O we started from.

6 Conclusion

We have, by elementary means, characterized the halting sets of programs over arbitrary universal algebras and have given a criterion for algebras $\mathcal{A}$ whose output sets coincide with their halting sets. These results generalize and simplify corresponding results of Blum, Shub and Smale [BSS].

7 References

[A] K.R. Apt, *Ten Years of Hoare's Logic: A Survey - Part I*, ACM Trans. Progr. Lang. and Systems, **3**, (1981), 431-483.

[B] J. de Bakker, *Mathematical Theory of Program Correctness* Prentice Hall International, 1980.

[BSS] L. Blum, M. Shub, S. Smale, *On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines.* Bulletin of the AMS **21** (1989), 1-46.

[BS] S. Burris, H.P. Sankappanavar, *A Course in Universal Algebra*, Springer Verlag, 1981.

[C] J. Cohen, *Constraint Programming Languages*,Communications of the ACM,**33** (1990),52-68.

[H] C.A.R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM, **12** (1969), 576-580.

[JL] J. Jaffar, J-L. Lassez, *Constraint Logic Programming*, in: Proceedings of the Fourteenth ACM Symposium of the Principles of Programming Languages, (Munich 1987), pp. 111-119.

[M] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill Computer Science Series,McGraw-Hill, New York, 1974.

[MS] R.E. Milne, C. Strachey, *A Theory of Programming Language Semantics*,Chapman and Hall, 1976.

[T] A. Tarski, *A decision method for elementary algebra and geometry*, 2nd revised ed., University of California Press, 1948.