Recursive Tables and Effective Definition Schemes

A. J. Wilder

Department of Computer Science,
University of Wales Swansea,
Singleton Park,
Swansea, SA2 8PP,
United Kingdom.

Abstract

This paper explores the expressive power of algebraic tables via an adaption of H. Friedman’s effective definition schemes (eds). The separation of tables, for documentation roles, from eds, for a model of computation, is used to give a clean semantics of algebraic tables independent of the type of table.

We define (1) the class of recursive algebraic tables and (2) recursive eds. By specialising techniques developed by R. Janicki, we show how a recursive algebraic table $T$ can be mapped to recursive eds, i.e. the list of decisions of $T$. Both operational and denotational semantics are given for recursive eds and they are shown to agree. The expressive power of recursive tables is compared to that of while-array program schemes: both soundness and adequacy are shown.

1 Introduction

Algebraic tables are simply tables that contain terms over a signature $\Sigma$ and variables $X$ and can be used for a variety of purposes, see Zucker [1996], Wilder and Tucker [1998] and §6.1. Although a variety of tables have been well reported in the literature from the software engineering (Heninger et al. [1978], Heninger [1980], Heitmeyer et al. [1998], Faulk et al. [1992], Faulk

*E-mail: A.J.Wilder@swansea.ac.uk
et al. [1994], Parnas et al. [1994] and Parnas and Wang [1989]), semantics (Janicki [1995], Abraham [1997], Janicki [1997] and Heitmeyer et al. [1996]) and transformation (Zucker [1996] and von Mohrenschlicht [1997]) perspectives little consideration has been given to their computational strength. This paper aims to redress this imbalance by asking in the terminology of Bergstra and Tucker [1987], for an algebra $A$:

1. Are algebraic tables sound for the class $\text{Comp}(A)$ of computable functions on $A$?

2. Are algebraic tables adequate for $\text{Comp}(A)$?

For our purposes a function $f : A^n \rightarrow A^m$ is in $\text{Comp}(A)$ when it is computable by a while-array program scheme, see Tucker and Zucker [1988] and Tucker and Zucker [1999]. There are other models of computation that can also be used to characterise $\text{Comp}(A)$ but our choice of the while-array program schemes is motivated by the fact that it can be used to formulate a Church-Turing thesis.

Building on Janicki [1995] we have developed four classes of algebraic tables.

Simple tables: a simple table is finite in size and its cells contain terms over a $(\Sigma, X)$.

Nested tables: a nested table is finite in size and its cells contain either a term over a $(\Sigma, X)$ or another nested table.

Infinite tables: an infinite table is one whose length of one or more dimensions may be equal to $\omega$ and cells contain terms over a $(\Sigma, X)$.

Recursive tables: a recursive table is finite in size and its cells contain terms over the $(\Sigma \cup \Gamma, X)$. The signature $\Sigma$ names the operations which we use as building blocks to define new operations named in $\Gamma$. Each operation $\gamma \in \Gamma$ is defined by a recursive table.

Although in this paper we focus on recursive tables we will show the following.

**Theorem.**

Let $f$ be a function on a many sorted $\Sigma$-algebra then

(a) the following are equivalent:

i. $f$ is computed by a **straight line** program;

ii. $f$ is documented by a simple table; and
iii. $f$ is documental by a nested table;

(b) and the following are equivalent

i. $f$ is computed by a \textbf{while-array} program;

ii. $f$ is documental by an infinite table; and

iii. $f$ is documental by a recursive table.

We note that part (a) of the theorem demonstrates the inadequacy of algebraic finite non-recursive tables for $\text{Comp}(A)$. To prove the equivalences we will not directly compare algebraic tables with program schemes, rather we will use an intermediate model of computation to which we will translate algebraic tables. This intermediate model is chosen because of its close similarities with algebraic tables as follows. It is clear (from Janicki [1993]) that each component of a table contains either tests or results and, furthermore, that the table provides a regular display for combining these to form decisions. Put succinctly, algebraic tables are a graphical representation for a list of decisions, each decision being a pair

$$b \rightarrow t$$

of a test boolean term $b$ and a result $s$-sorted term $t$. It is these lists of decisions that we will use as the intermediate model of computation with which to study the class of functions definable by algebraic tables.

Computing with such lists of decisions is not a new idea: Friedman [1971] defines a model of computation called effective definition schemes (eds) that uses a recursively enumerable set of clauses (a clause is Friedman’s terminology for a decision) to define a function. Effective definition schemes have been used for the analysis of programming languages, see Tjuryn [1980]. Here we work with the equivalent definition of an eds as an infinite list of clauses.

The four different classes of algebraic tables produce different types of eds as summarised below.

<table>
<thead>
<tr>
<th>Table</th>
<th>Model of Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>finite eds</td>
</tr>
<tr>
<td>nested</td>
<td>eds</td>
</tr>
<tr>
<td>infinite</td>
<td>recursive eds</td>
</tr>
<tr>
<td>recursive</td>
<td></td>
</tr>
</tbody>
</table>

The paper is organised as follows. In Section 2 we summarise the tools of universal algebra necessary for this paper. A running example using Ackermann’s function is introduced. In Section 3, we introduce recursive tables
and recursive eds. The operational and denotational semantics and their
equivalence is the topic of Section 4. We explore the expressive power of re-
cursive eds in Section 5 demonstrating soundness by the use of a while-array
implementable stack machine and showing adequacy by compiling course-of-
value schemes to recursive eds systems. In Section 6 we briefly discuss (1)
the use of algebraic tables in the literature, (2) related work on the semantics
of tables, and (3) summarise the consequences of these results.

I would like to thank John Tucker for proposing the use of effective def-
inition schemes as the model of computation for tables and supervising my
research studentship at Swansea. I would also like to thank Jeff Zucker for
his assistance with the semantics of the recursive eds during a two week visit
to McMaster University in December 1995.

2 Preliminaries

We assume the reader is familiar with the elements of algebra and abstract
data type theory. In this section we briefly summarise the notations and
tools of universal algebra used in this paper. In particular we require the
following: (in §2.1) signatures; standard, $N$-standard and array algebras; (in
§2.2) variables and assignments; terms and term evaluations; substitution of
a term’s variables; (in §2.3) given and defined signatures; $\Gamma$-system. Further
details on the material contained in §§2.1 and 2.2 can be found in Wirasing
[1990], Meinke and Tucker [1992], Tucker and Zucker [1988], Tucker and

2.1 Algebras

A signature is a pair $\Sigma = (S, Funcs)$ of a set $S$ of sorts and a $S^n \times S$-indexed
family $Funcs$ of sets of typed function names; a function $f \in Funcs_{w \rightarrow s}$ is
typed $w \rightarrow s$, where $w = s_1 x s_2 \times \cdots \times s_n \in S^n$ and $s \in S$. We identify
the special functions $f \in Funcs_{\lambda \rightarrow s}$, for $\lambda$ denoting the empty string of sorts,
as constants. A vector type $u \rightarrow v$, where $u = u_1 x u_2 \times \cdots \times u_n \in S^n$ and
$v = v_1 x v_2 \times \cdots \times v_m \in S^m$, is the type of a vector of functions typed $u \rightarrow v_i$, for
$1 \leq i \leq m$.

An $\Sigma$-algebra $A$ provides a semantics for the sorts, constants and functions
named in $\Sigma$. We assume $A$ is a total algebra, i.e. every function in $A$ is total.
We call an algebra standard just in case it is an expansion of the algebra $B$
of booleans displayed in Figure 1. We call an algebra $A$ $N$-standard just in
case it is standard and an expansion of the algebra $N$ displayed in Figure 2.
Figure 1: The algebra $\mathcal{B}$ of booleans.

Figure 2: The algebra $\mathcal{N}$ of natural numbers.

Modeling finite unbounded sequences or arrays of data from a $\Sigma$-algebra $A$ is accomplished by defining the algebra $A^u$. The steps in constructing $A^u$ are to (1) add a special value for accessing an array with an index beyond its bounds by augmenting the sorts of $\Sigma$ with a unique undefined value to form the algebra $A^n$; (2) add the indexing set of the arrays by $N$-standardising $A^n$ to the algebra $A^{n,N}$; and (3) for each sort $s$ of $\Sigma$ defining the starred sort $s^*$ whose carrier $A^*_s$ contains pairs $a = (\alpha, l)$ of a function

$$\alpha : \mathbb{N} \to A^*_s$$

and a witness $l \in \mathbb{N}$ to the finiteness of $a$ such that, for all $i \geq l$, $\alpha(i) = u$. Appropriate operations are added to $A^u$ to operate on starred sorts and include: the empty array constant; a discriminator; equality; finding and altering the length of an array; and accessing and updating the contents of an array. A full account of constructing $A^u$ from $A$ can be found in either Tucker and Zucker [1988] or Tucker and Zucker [1999].
2.2 Terms

We will use a set $X$ of variables typed from the sorts in $\Sigma$. An assignment of the variables in $X$ is a map $\sigma : X \rightarrow A$; $\sigma(x^i) \in A_s$ denotes the value held by the variable $x^i$. The function space $[X \rightarrow A]$ is the set of all possible assignments. For a list $\vec{x} = x_1^{i_1}, x_2^{i_2}, \ldots, x_n^{i_n}$ of variables, $\sigma[\vec{x}] = (\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n)) \in A^\vec{i}$. Two assignments $\sigma, \sigma' : X \rightarrow A$ are equal relative to $\vec{x}$, $\sigma \approx \sigma'$ (rel $\vec{x}$), when, for every variable $x_i$ in the list $\vec{x}$, $\sigma(x_i) = \sigma'(x_i)$.

A term of sort $s$ over a signature $\Sigma$ and a set $X$ of typed variables is either a variable $x^s \in X$, a constant $c^s$ from $\Sigma$ or an application $f(t_1, \ldots, t_n)$ of a function, whose co-domain is typed $s$, to appropriately typed terms. Let $\mathcal{T}(\Sigma, X)$, denote the set of all $s$-sorted terms and, for any term $t \in \mathcal{T}(\Sigma, X)_s$, $\text{var}(t) \subseteq X$ be the set of all variables occurring in $t$. We assume the instantiation principle applies to $\mathcal{T}(\Sigma)_s$ allowing us to select, for any sort $s$, an instantiation term $t \in \mathcal{T}(\Sigma)_s$.

A term $t \in \mathcal{T}(\Sigma, X)_s$ has a semantic value $a \in A_s$ from a $\Sigma$-algebra $A$ under an assignment $\sigma : X \rightarrow A$ given by the term evaluation map

$$te_s : \mathcal{T}(\Sigma, X)_s \times [X \rightarrow A] \rightarrow A_s$$

defined by structural induction on $t$ in the usual manner.

Let $t \in \mathcal{T}(\Sigma \cup \Gamma, X)_s$ be a term where $\text{var}(t) = \{x_1^{i_1}, \ldots, x_n^{i_n}\}$ and select a term $t_i \in \mathcal{T}(\Sigma \cup \Gamma, Y)_s$ for each variable $x_i \in \text{var}(t)$. The substitution of $t$’s variables $x_1, \ldots, x_n$ with terms $t_1, \ldots, t_n$, written as

$$t(t_1, \ldots, t_n/x_1, \ldots, x_n),$$

is also defined by structural induction on the complexity of $t$ in the usual way.

2.3 Models of Computation

Hiding sorts and operations are useful for providing modular algebras. To this end we will use a signature $\Sigma$ to define the interfaces for known given operations. The interfaces for the functions to be defined from those named in $\Sigma$ are held within a signature $\Gamma$. The sorts of $\Gamma$ are contained in the sorts of $\Sigma$ and the functions of two signatures are disjoint, i.e. for any type $w \rightarrow s$, $\Sigma_{w \rightarrow s} \cap \Gamma_{w \rightarrow s} = \emptyset$.

Example 2.3.1 (Ackermann’s interface).
The given signature $\Sigma_{\text{Ack}}$ is displayed below.
The defined signature $\Gamma_{\text{Ack}}$ is displayed below.

<table>
<thead>
<tr>
<th>Signature</th>
<th>$\Sigma_{\text{Ack}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import</td>
<td>$\Sigma_{\text{bool}}$</td>
</tr>
<tr>
<td>Sort names</td>
<td>$\text{n}@t$</td>
</tr>
<tr>
<td>Constant names</td>
<td>0, 1 : $\text{n}@t$</td>
</tr>
<tr>
<td>Function names</td>
<td>$+1, -1 : \text{n}@t \to \text{n}@t$</td>
</tr>
<tr>
<td></td>
<td>$= : \text{n}@t \times \text{n}@t \to \text{bool}$</td>
</tr>
</tbody>
</table>

Suppose a method $\mathcal{M}$ for defining a function $f : A^w \to A_s$ named in $\Gamma$ using operations named in $\Sigma$. A $\Gamma$-system of $\mathcal{M}$ is a family

$$S = \langle m_\gamma \in M| \gamma : w \to s \rangle$$

of definitions using $\mathcal{M}$; one definition $m_\gamma$ selected for each operation $\gamma \in \Gamma$. The display of a $\Gamma$-system of $\mathcal{M}$ is shown in Figure 3. A $\Gamma$-system $S$ of $\mathcal{M}$

Figure 3: The display of a $\Gamma$-system $S$ of $\mathcal{M}$ using given operations from $\Sigma$. defines a $\Gamma$-algebra $B$ whose carrier sets are taken from the $\Sigma$-algebra $A$ and the functions of $B$ are taken from $S$. We observe that $B$ may often be a partial algebra (i.e. some or all of its functions are partial) due to how $\mathcal{M}$ defines new functions.

Furthermore, suppose, for $\mathcal{M}$, there is a mapping $[\cdot] : M \to [A^w \to A_s]$ and, for $1 \leq i \leq m$, there are defined operations $\gamma_i : A^w \to A_{s_i}$ from a $\Gamma$-system $S$ of $\mathcal{M}$. The strict extension of $[\cdot]$ to operate on the vector
\((m_{\gamma_1}, \ldots, m_{\gamma_m})\) is defined, for \(a \in A^w\), as
\[
\llbracket m_{\gamma_1}, \ldots, m_{\gamma_m} \rrbracket (a) = \begin{cases} 
\llbracket m_{\gamma_1} \rrbracket (a), \ldots, \llbracket m_{\gamma_m} \rrbracket (a) & \text{if each } \llbracket m_{\gamma_i} \rrbracket (a) \downarrow \\
\uparrow & \text{if one } \llbracket m_{\gamma_i} \rrbracket (a) \uparrow.
\end{cases}
\]

3 Recursive Tables and Recursive Effective Definition Schemes

In §3.1 we define what a recursive function table is and show how it documents a list of decisions. The function documented by a recursive function table is specified by the table’s list of decisions. The pure mathematical structure of these decisions gives rise to the recursive effective definition scheme which we consider in §3.2.

3.1 Recursive Tables

A recursive table has a shape that is a non-empty list \(l_1, \ldots, l_n\) of integers greater than zero. For a recursive table \(T\) with shape \(l_1, \ldots, l_n\) we define the following terms: the dimensionality \(\text{Dim}(T)\) of \(T\) is the length \(n\) of the list; a dimension of \(T\) is an integer \(1 \leq i \leq \text{Dim}(T)\); the length and index set of a dimension \(i\) of \(T\) is the integer \(l_i\) from the list and the set \(I_i = \{1, 2, \ldots, l_i\}\), respectively; and the size \(\text{Size}(T)\) of \(T\) is the product \(l_1 \times l_2 \times \cdots \times l_n\).

**Definition 3.1.1 (Recursive table).**

An (algebraic) recursive table with shape \(l_1, \ldots, l_n\) is a \(n + 1\) tuple
\[
T = (header_1, \ldots, header_n, grid)
\]
of total functions
\[
header_i : I_i \rightarrow \mathcal{T}(\Sigma \cup \Gamma, X)_{s_i},
\]
for each dimension \(i\) of \(T\), and
\[
grid : I_1 \times I_2 \times \cdots \times I_n \rightarrow \mathcal{T}(\Sigma \cup \Gamma, X)_{s}.
\]

We write \(RTable(\Sigma, \Gamma, X)_{s_1, \ldots, s_n, s}\) for the set of all recursive tables.

A recursive table \(T\) is displayed with the grid central and the headers as satellites, see Figure 4. The Cartesian product \(I_T = I_1 \times \cdots \times I_n\) is called the index set of \(T\). When \(I_T\) is unambiguously associated with a recursive table \(T\) we may omit the subscript.
\[
\begin{array}{|c|c|}
\hline
y = 0 & \text{Not}(y = 0) \\
\hline
x = 0 & y + 1 \\
\text{Not}(x = 0) & \text{Ack}(x - 1, 1) \\
\hline
y + 1 & y + 1 \\
\end{array}
\]

Figure 4: An algebraic recursive table with shape 2, 2.

For a recursive table \(T\) we define the set \(C_T = \{h_1, \ldots, h_n, g\}\) of syntactic names for the components of \(T\). We often abuse notation by writing, for example, \(h_j(i)\) instead of \(\text{header}_j(i_j)\), for an index \(i = (i_1, \ldots, i_n) \in I\).

A table alone can be read in numerous ways: a reading strategy needs to be defined. Our approach specialises the generalised semantics of tables of Janicki [1995] and its extension by Abraham [1997], see §6.2. The components of \(T\) can be partitioned into input and output classes and, furthermore, by simply listing the output components \(\vec{c} = c_1, \ldots, c_m\) of \(T\) the set of input components can be described as \(C_T \setminus \{\vec{c}\}\). The list \(\vec{c}\) satisfies the property that either each component \(c_i\) is a header or there is only one component \(c_1\) which is the grid. For the purposes of this paper we fix the table predicate rule to be conjunction and the table relation rule to vectorisation, see Definition 3.1.4. The choice to vector the cells of the output components in the context of a many-sorted algebra induces a constraint on the order of the output components in the construction of a recursive function table.

**Definition 3.1.2 (Recursive function table).**

A recursive function table of type \(u \rightarrow v\) is a triple \(T = (\vec{x}, \vec{c}, T)\) of a list \(\vec{x} = x_1, \ldots, x_n\) of input variables, a list \(\vec{c} = c_1, \ldots, c_m\) of output components of \(T\) and an algebraic recursive table \(T\) such that

1. for every component \(c \in C_T\) and every index \(i \in I\), \(\text{var}(c(i)) \subseteq \{\vec{x}\}\);
2. the contents of the output component \(c_i\) are terms of sort \(v_i\), i.e. \(c_i(I) \subseteq \mathcal{T}(\Sigma \cup \Gamma, X)_{v_i}\); and
3. the contents of any input component are boolean terms, i.e. for \(c \in C_T \setminus \{\vec{c}\}\), \(c(I) \subseteq \mathcal{T}(\Sigma \cup \Gamma, X)_{\text{bool}}\).

Let \(\text{RTTable}(\Sigma, \Gamma, X)_{u \rightarrow v}\) be the set of all function recursive tables of type \(u \rightarrow v\).

\footnote{We believe this a simpler and more direct alternative to the cell connection graph of Janicki [1995].}
Every recursive function table in a $\Gamma$-system $\mathcal{RT}$ of recursive function
tables will typed $w \to s$ to define a function $\gamma : w \to s$ in $\Gamma$. This means
that every recursive function table in $\mathcal{RT}$ has exactly one output component
$c = c_1$.

**Example 3.1.3** (Ackermann’s function as a recursive table).
We can define Ackermann’s function by a recursive table. The $\Gamma_{\text{Ack}}$-system
$S$ of recursive tables for Ackermann’s functions is displayed below.

<table>
<thead>
<tr>
<th>System</th>
<th>$S_{\text{Ack}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin Tables</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>$\text{Ack} : \text{nat} \times \text{nat} \to \text{nat}$</td>
</tr>
<tr>
<td>Variables</td>
<td>$x, y$</td>
</tr>
<tr>
<td>Output components</td>
<td>$g$</td>
</tr>
<tr>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>$x = 0$</td>
<td></td>
</tr>
<tr>
<td>$\text{Not}(x = 0)$</td>
<td>$y + 1$</td>
</tr>
<tr>
<td>$\text{Ack}(x - 1, 1)$</td>
<td></td>
</tr>
</tbody>
</table>

A table provides a structured display medium for lists of decisions.

**Definition 3.1.4** (Decisions of a recursive function table).
Let $T = (\bar{x}, \bar{c}, \bar{T})$ be a recursive function table of type $u \to v$. The decisions $\text{Dec}(T)$ of $T$ is a list

$$(b_1, \ldots, 1 \to \bar{b}_1, \ldots, 1), \ldots, (b_{i_1, \ldots, i_n} \to \bar{b}_{i_1, \ldots, i_n})$$

of length $\text{Size}(T)$ of pairs indexed by indices $i \in I_T$. Each pair $(b_i, \bar{b}_i)$, for $i \in I_T$, is comprised of a boolean term

$$b_i = \bigwedge_{c \in \bar{c}(i)} c(i)$$

formed from the conjunction of terms from the input components of $T$ and a $m$-tuple $\bar{b}_i = (c_1(i), \ldots, c_m(i))$ of terms from the output components of $T$.

**Example 3.1.5** (Decisions of Ack recursive table).
Supposing header one runs down the side of the table, the decisions of the Ack
recursive function table are:

$$(1, 1) \quad x = 0 \land y = 0 \to y + 1,$$
$$(1, 2) \quad x = 0 \land \text{Not}(y = 0) \to y + 1,$$
$$(2, 1) \quad \text{Not}(x = 0) \land y = 0 \to \text{Ack}(x - 1, 1)$$
and
$$(2, 2) \quad \text{Not}(x = 0) \land \text{Not}(y = 0) \to \text{Ack}(x - 1, \text{Ack}(x, y - 1)).$$
The function documented by a recursive function table $T$ is intimately linked to its decisions.

**Definition 3.1.6 (Documentability of recursive tables).**
A recursive function table $T \in RFTable(\Sigma, \Gamma, X)_{w \rightarrow s}$ documents a function $f : A^u \rightarrow A^v$ when its decisions $Dec(T)$ specify $f$.

### 3.2 Recursive Effective Definition Schemes

We introduce *recursive effective definition schemes* as a model of computation based on Friedman [1971] eds. It is easy to translate recursive tables to this new model of computation using the decisions of the table.

**Definition 3.2.1 (Clause).**
A clause of sort $s \in S$ over a signature $\Sigma$ and a set $X$ of typed variables is a pair

$$(b, t) \in T(\Sigma, X)_{\text{bool}} \times T(\Sigma, X)_s$$

of terms over $(\Sigma, X)$ — the first, $b$, of sort $\text{bool}$ and the second, $t$, of sort $s$. For a clause $(b, t)$ we call $b$ the *test* of the clause and $t$ the *result* of the clause. We will often write a clause $(b, t)$ as

$$b \rightarrow t.$$ 

Let $\mathcal{C}(\Sigma, X)_s = T(\Sigma, X)_{\text{bool}} \times T(\Sigma, X)_s$ be the set of all $s$-sorted clauses over $(\Sigma, X)$.

**Definition 3.2.2 (Recursive effective definition scheme).**
A recursive effective definition scheme (recursive eds) of type $s_1 \times s_2 \times \cdots \times s_n \rightarrow s$ is a triple $(\vec{x}, l, g)$ of a list $\vec{x} = x_1^{s_1}, \ldots, x_n^{s_n}$ of *input* variables from $X$, a *length* $l > 0$ of the recursive eds and a total function

$$g : \{1, \ldots, l\} \rightarrow \mathcal{C}(\Sigma \cup \Gamma, \{\vec{x}\})_s$$

such that, for $1 \leq i \leq l$, the $i$-th clause of $g$ is

$$g(i) = b_i \rightarrow t_i.$$ 

We write $REDs(\Sigma, \Gamma, X)_{w \rightarrow s}$ for the set of all recursive eds $g$ of type $w \rightarrow s$.

The function of a recursive eds $(\vec{x}, l, g) \in REDs(\Sigma, \Gamma, X)_{w \rightarrow s}$ can be decomposed to a pair $g_b : \{1, \ldots, n\} \rightarrow T(\Sigma \cup \Gamma, X)_{\text{bool}}$ and $g_r : \{1, \ldots, n\} \rightarrow T(\Sigma \cup \Gamma, X)_s$ of unique functions where $g(i) = g_b(i) \rightarrow g_r(i)$.

For a tuple $\tilde{\gamma} = (\gamma_1 : u \rightarrow v_1, \ldots, \gamma_m : u \rightarrow v_m)$ of $\Gamma$ operations we define the tuple $g_{\tilde{\gamma}} = (g_{\gamma_1}, \ldots, g_{\gamma_m})$. 

11
Example 3.2.3 (Ackermann’s function as a recursive eds).
The $\Gamma_{\text{Ack}}$-system of recursive eds for Ackermann’s function is shown below.

<table>
<thead>
<tr>
<th>System</th>
<th>$S_{\text{Ack}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>$\text{Ack} : \text{n}at \times \text{n}at \rightarrow \text{n}at$</td>
</tr>
<tr>
<td>Var</td>
<td>$x, y$</td>
</tr>
</tbody>
</table>
| Reds   | $g_{\text{Ack}}(i) = \begin{cases} 
  x = 0 \rightarrow y + 1 & \text{if } i = 1 \\
  \text{Not}(x = 0) \text{ And } y = 0 \rightarrow \text{Ack}(x - 1, 1) & \text{if } i = 2 \\
  \text{Not}(x = 0) \text{ And Not}(y = 0) \rightarrow \text{Ack}(x - 1, \text{Ack}(x, y - 1)) & \text{if } i = 3 
\end{cases}$ |
| End Recursive eds |

4 Semantics of Recursive EDS

For a system $G$ of mutually recursive eds we need a semantics that meets our expectations of their behaviour, i.e. find the first clause of a recursive eds to be true and evaluate its paired result. In this section we give two semantics for recursive eds and conclude by showing that they agree.

4.1 Operational Semantics of Recursive EDS

We will describe the semantics of recursive eds by the use of structural operational semantics. Consider a $\Gamma$-system $G$ of recursive eds and $A$ a $\Sigma$-algebra.

We will define a partial operational semantic function

$$
O_s : T(\Sigma \cup \Gamma, X)_s \rightarrow [X \rightarrow A] \rightarrow A,
$$

so that, for any term $t \in T(\Sigma \cup \Gamma, X)_s$ and assignment $\sigma : X \rightarrow A$, when $O_s(t)(\sigma) \downarrow a$ then $t$ has semantic value $a \in A_s$ and when $O_s(t)(\sigma) \uparrow$ then $t$ has no semantic value in $A_s$. We choose to write $O_s(t)(\sigma) \downarrow a$ as $t \xrightarrow{G, s, \sigma} \sigma(x)$

Definition 4.1.1 (Operational semantic function).

For a $\Gamma$-system $G$ of and an assignment $\sigma : X \rightarrow A$ we define simultaneously for each sort $s \in S$ from $\Sigma$ the operational semantic function $O_s$ for any term $t \in T(\Sigma \cup \Gamma, X)_s$ by structural induction on the complexity of $t$.

I. For a term $t \equiv x^s$,

$$
\frac{(\text{var})}{x \xrightarrow{G, \sigma} \sigma(x)}
$$
II. For a term $t \equiv c^a$ from $\Sigma$,

$$
\begin{aligned}
(\text{cons}) \quad & c \xrightarrow{\sigma} c_A \\
\end{aligned}
$$

III. For a term $t \equiv f(t_1, \ldots, t_n)$, for $f : w \rightarrow s$ from $\Sigma$,

$$
\begin{aligned}
(\text{func}) \quad & t_1 \xrightarrow{\sigma_1, \ldots, t_n \xrightarrow{\sigma_n}} f(t_1, \ldots, t_n) \xrightarrow{\sigma} f_A(a_1, \ldots, a_m) \\
\end{aligned}
$$

IV. For a term $t \equiv \gamma(t_1, \ldots, t_n)$, for $\gamma : w \rightarrow s$ from $\Gamma$,

$$
\begin{aligned}
(\text{reds}) \quad & \bar{g}_{\gamma,b,k} \xrightarrow{\sigma} \gamma(t_1, \ldots, t_n) \xrightarrow{\sigma} a \\
\end{aligned}
$$

where we write $\bar{g}_{\gamma,j,i}$ for $g_{\gamma,j}(i)(t_1, \ldots, t_n/x_{\gamma,i})$, for $j \in \{b, t\}$ and $1 \leq i \leq l_\gamma$.

**Proposition 4.1.2.**

For any $s \in S$ sorted term $t \in T(\Sigma \cup \Gamma, X)$, if $\sigma \approx \sigma'$ (rel $\text{var}(t)$) then $O_s(t)(\sigma) \sim O_s(t)(\sigma')$.

Using $O_s$, we define the semantics of any recursive eds from a $\Gamma$-system $\mathcal{G}$.

**Definition 4.1.3 (Recursive eds function).**

Let $\mathcal{G}$ be a $\Gamma$-system of recursive eds. The partial function computed by a recursive eds $(\bar{x}_\gamma, t_\gamma, g_\gamma) \in \mathcal{G}$ is typed $[g_\gamma]_{os} : A^w \rightarrow A_s$ and defined, for any $a \in A^w$, as

$$
[g_\gamma]_{os}(a) \simeq O_s(\gamma(\bar{x}_\gamma))(\sigma)
$$

where $\sigma[\bar{x}_\gamma] = a$.

We strictly extend $[.]_{os}$ to operate on a vector $g_{\vec{z}}$ of recursive eds from $\mathcal{G}$.

**Definition 4.1.4 (Recursive eds computability).**

A function $f : A^w \rightarrow A^w$ is recursive eds computable if, and only if, there exists a $\Gamma$-system $\mathcal{G}$ of recursive eds from which a vector $g \in \text{REDS}(\Sigma, \Gamma, X)_{u \Rightarrow v}$ is chosen such that $f \simeq [g]_{os}$.
4.2 Denotational Semantics of Recursive EDS

We use ω-complete partial orders to describe a denotational semantics of recursive eds. Let $A^\perp$ denote the flat complete partial order derived from the algebra $A$. The map $\lbrack \cdot \rbrack : A \to A^\perp$ injects data from $A$ into its derived flat cpo as expected. The definition of $\lbrack \cdot \rbrack$ can be extended to work with assignments by simply setting $\lbrack \sigma \rbrack(x) = \lbrack (\sigma(x)) \rbrack$

**Definition 4.2.1 (Function environment space).**
Consider a defined signature $\Gamma$. For each function name $\gamma : w \to s$ in $\Gamma$ we construct the continuous function space cpo $\mathcal{P}_\gamma = \left(\left((A^\perp)^w \to A^\perp_s\right), \sqsubseteq, \bot \right)$ in the usual way. The function environment space is the cpo

$$P = \left( \prod_{\gamma : w \to s \in \Gamma} \left((A^\perp)^w \to A^\perp_s\right), \sqsubseteq_P, \bot_P \right)$$

of the $\Gamma$-indexed Cartesian product of function spaces, the partial order $\sqsubseteq_P$ defined co-ordinate wise, that is $p \sqsubseteq_P p'$ if, and only if, for every $\gamma : w \to s$ $p_\gamma \sqsubseteq_p p'_\gamma$, and the least element $\bot_P = (\bot, \bot, \bot)$ the tuple of the bottom function $\bot$, for every $\gamma : w \to s$.

We term an element $p \in P$ a function environment.

**Definition 4.2.2 (Variable environment space).**
A variable environment space is the cpo

$$V = \left(\left[X \to A^\perp\right], \sqsubseteq, \bot\right)$$

with the ordering $\sqsubseteq$ is the ordering such that $\sigma \sqsubseteq \sigma'$ if, and only if, for every variable $x^*$, $\sigma(x^*) \sqsubseteq \sigma'(x^*)$ and $\bot$ is the assignment that maps every variable in $X$ to $\bot$.

**Definition 4.2.3 (The term semantic function).**
The term semantic function

$$\mathcal{M}_s : \mathcal{T}(\Sigma \cup \Gamma, X)_s \to P \to V \to A^\perp_s$$

is defined, simultaneously over all sorts, for a term $t \in \mathcal{T}(\Sigma \cup \Gamma, X)_s$, a function environment $p \in P$ and a variable assignment $\sigma : X \to A^\perp$ by structural induction on the complexity of $t$.

I. If $t \equiv x$, for a variable name $x \in X$ of sort $s$, then $\mathcal{M}_s(t)(p)(\sigma) = \sigma(x);$  
II. If $t \equiv c$, for a base constant name $c \in \Sigma$ of sort $s$, then $\mathcal{M}_s(t)(p)(\sigma) = c_{A^\perp};$
III. If \( t \equiv f(t_1, \ldots, t_n) \), for a base operation name \( f : s_1 \times \cdots \times s_n \to s \) from \( \Sigma \), then
\[
\mathcal{M}_s(f(t_1, \ldots, t_n))(p)(\sigma) = f_{A^\perp}(\mathcal{M}_{s_1}(t_1)(p)(\sigma), \ldots \mathcal{M}_{s_n}(t_n)(p)(\sigma));
\]
and

IV. If \( t \equiv \gamma(t_1, \ldots, t_n) \), for a defined name \( \gamma : s_1 \times \cdots \times s_n \to s \) from \( \Gamma \), then
\[
\mathcal{M}_s(\gamma(t_1, \ldots, t_n))(p)(\sigma) = p_\gamma(\mathcal{M}_{s_1}(t_1)(p)(\sigma), \ldots \mathcal{M}_{s_n}(t_n)(p)(\sigma)).
\]

Next we define a recursive eds semantic function that maps a recursive eds \( g \in REDS(\Sigma, \Gamma, X)_{\rightarrow s} \) to a point \( a \in A^\perp_s \).

**Definition 4.2.4 (The recursive eds semantic function).**
We simultaneously define every \( w \to s \)-indexed recursive eds semantic function
\[
R_{w \to s} : REDS(\Sigma, \Gamma, X)_{\rightarrow s} \to P \to V \to A^\perp_s
\]
for a recursive eds \( (\vec{x}, g) \in REDS(\Sigma, \Gamma, X)_{\rightarrow s} \), a function environment \( p \in P \) and a variable assignment \( \sigma : X \to A^\perp \) as

\[
R_{w \to s}(g)(p)(\sigma) \text{ is } \begin{cases} 
\mathcal{M}_s(g_1(1))(p)(\sigma) & \text{if } \mathcal{M}_{\text{bool}}(g_0(1))(p)(\sigma) \\
\mathcal{M}_s(g_2(2))(p)(\sigma) & \text{if } \mathcal{M}_{\text{bool}}(g_0(2))(p)(\sigma) \text{ and } \\
\vdots & \not \mathcal{M}_{\text{bool}}(g_0(1))(p)(\sigma) \\
\mathcal{M}_s(g_n(n))(p)(\sigma) & \text{if } \mathcal{M}_{\text{bool}}(g_0(n))(p)(\sigma) \text{ and } \\
\not \mathcal{M}_{\text{bool}}(g_0(1))(p)(\sigma) \text{ and } \\
\not \mathcal{M}_{\text{bool}}(g_0(2))(p)(\sigma) \text{ and } \\
\vdots & \\
\not \mathcal{M}_{\text{bool}}(g_0(n-1))(p)(\sigma) \text{ otherwise }
\end{cases}
\]

Finally we define a continuous function \( F : P \to P \) to provide increasingly better approximations of the functions defined by the recursive eds in a \( \Gamma \)-system \( \mathcal{G} \).

**Definition 4.2.5 (The fixed point function).**
Let \( \mathcal{G} \) be a \( \Gamma \)-system of recursive eds. For every defined function name \( \gamma \in \Gamma \), of type \( w \to s \), we define \( f_\gamma : P \to P_\gamma \) for a function environment \( p \in P \) and a point \( a \in (A^\perp)^w \) as
\[
f_\gamma(p)(a) = R_{w \to s}(g_\gamma)(p)(\sigma(a/\vec{x}_\gamma)).
\]
The fixed point function \( F : P \rightarrow P \) is defined for a function environment \( p \in P \) by the \( \Gamma \)-indexed family
\[
F(p) = \langle f_\gamma(p) \mid \gamma \in \Gamma \rangle
\]
of functions.

**Lemma 4.2.6 (Continuity of \( \mathcal{M}_s, R_{w \rightarrow s} \) and \( F \)).**

The term semantic function \( \mathcal{M}_s(t) \), recursive eds semantic function \( R_{w \rightarrow s}(g) \) and fixed point function \( F \) are continuous functions.

It is necessary to prove that \( \mathcal{M}_s(t) \) is both monotone and limit preserving because a monotone function \( f : D \rightarrow F \) from a cpo \( D \) to a flat cpo \( F \) need not be continuous.

The least fixed point \( \mu F = \bigsqcup \{ F^n(\perp) \} \) of the fixed point function \( F \) is the semantics of the recursive eds in the \( \Gamma \)-system \( \mathcal{G} \).

**Definition 4.2.7 (Denotational semantics of a term).**

The denotational term semantic function
\[
\mathcal{D}_s : \mathcal{T}(\Sigma \cup \Gamma, X)_s \rightarrow V \rightarrow A^s_\perp
\]
is defined on any term \( t \in \mathcal{T}(\Sigma \cup \Gamma, X)_s \) and assignment \( \sigma : X \rightarrow A^\perp \) of the variables of \( X \) as
\[
\mathcal{D}_s(t)(\sigma) = \mathcal{M}_s(t)(\mu F)(\sigma).
\]

**Proposition 4.2.8.**

For any \( s \in S \) sorted term \( t \in \mathcal{T}(\Sigma \cup \Gamma, X)_s \) if \( \sigma \approx \sigma' \) (rel. \( \text{var}(t) \)) then \( \mathcal{D}_s(t)(\sigma) \approx \mathcal{D}_s(t)(\sigma') \).

**Definition 4.2.9 (Recursive eds function).**

Let \( \mathcal{G} \) be a \( \Gamma \)-system of recursive eds. The recursive eds function for a recursive eds \( (\vec{x}_\gamma, l_\gamma, g_\gamma) \), for \( \gamma : w \rightarrow s \), is typed \( \llbracket g_\gamma \rrbracket_{\mathcal{D}_s} : A^w \rightarrow A_s \) and defined, for any \( a \in A^w \) as
\[
\llbracket g_\gamma \rrbracket_{\mathcal{D}_s}(a) \approx \begin{cases} \mathcal{D}_s(\gamma(\vec{x}_\gamma))(\llbracket \sigma \rrbracket) & \text{if } \mathcal{D}_s(\gamma(\vec{x}_\gamma))(\llbracket \sigma \rrbracket) \neq \perp \\ \uparrow & \text{if } \mathcal{D}_s(\gamma(\vec{x}_\gamma))(\llbracket \sigma \rrbracket) = \perp \end{cases}
\]

where \( \sigma[\vec{x}_\gamma] = a \).

We strictly extend \( \llbracket \rrbracket_{\mathcal{D}_s} \) to operate on a vector \( g_\gamma \) of recursive eds from \( \mathcal{G} \).

**Definition 4.2.10 (Recursive eds computable).**

A function \( f : A^u \rightarrow A^w \) is recursive eds computable if, and only if, there exists a \( \Gamma \)-system \( \mathcal{G} \) of recursive eds from which a vector \( g \in REDS(\Sigma, \Gamma, X)_u \rightarrow w \) of recursive eds is chosen such that \( f \approx \llbracket g \rrbracket_{\mathcal{D}_s} \).
4.3 Equivalence of Semantics

Theorem 4.3.1 (Equivalence of semantics of recursive eds).
Let \( \mathcal{G} \) be a \( \Gamma \)-system of recursive eds then, for any \( g_\gamma \in \mathcal{G} \),

\[
\llbracket g_\gamma \rrbracket_{ds} \simeq \llbracket g_\gamma \rrbracket_{os}.
\]

Proof. By unfolding the definitions of \( \llbracket \cdot \rrbracket_{ds}, \llbracket \cdot \rrbracket_{os}, D \) and \( O \) we are required to show

\[
\mathcal{M}_s(\gamma(\vec{x}))(\mu F)([\sigma]) = [b] \neq \bot \iff \gamma(\vec{x}) \xrightarrow{G,\sigma} b \tag{1}
\]

where \( \sigma[\vec{x}] = a \in A^w \) and \( b \in A_s \).

In the proofs of either direction of (1) we replace \( \gamma(\vec{x}) \) with an arbitrary
term \( t \in \mathcal{T}(\Sigma \cup \Gamma, X)_s \).

\( \Rightarrow \) Let the variables of the term \( t \) be \( \text{var}(t) = \{x_1^n, \ldots, x_{n}^n\} \) and, furthermore, for each \( x_i \in \text{var}(t) \) a term \( t_i \in \mathcal{T}(\Sigma \cup \Gamma, Y)_s \) is selected. Then we proceed to show that, for any assignments \( \sigma : X \rightarrow A^\bot \) and \( \sigma' : Y \rightarrow A \)

where \( \sigma(x_i) = O_s(t_i)(\sigma') \) for all variables \( x_i \in \text{var}(t) \),

\[
\mathcal{M}_s(t)(F^n(\bot))(\sigma) = [b] \neq \bot \iff t(t_1/x_1, \ldots, t_n/x_n) \xrightarrow{G,\sigma'} b. \tag{2}
\]

The proof of (2) is constructed by induction on \( n \) and then \( t \). The use of a
term \( t_i \) to give a value to variable \( x_i \) by the operational semantics \( O \) makes it possible to apply the induction hypothesis in the case of \( n = i + 1 \) and \( t \equiv \gamma(t_1', \ldots, t_m') \) and is a consequence of the definition of the components \( f_\gamma \) of the fixed point function \( F \) (see Definition 4.2.5).

When \( X = Y \) and we select, for each variable \( x_i \in \text{var}(t) \), \( t_i \) to be \( x_i \) then

(2) simplifies to

\[
\mathcal{M}_s(t)(F^n(\bot))(\sigma') = [b] \neq \bot \iff t \xrightarrow{G,\sigma'} b. \tag{3}
\]

We can show that if \( \mathcal{M}_s(t)(\mu F)([\sigma']) = [b] \neq \bot \) then there exists an \( n \in \mathbb{N} \)
such that \( \mathcal{M}_s(t)(F^n(\bot))(\sigma') = [b] \neq \bot \). Combing this with (3), completes the proof \( \Rightarrow \) of (1).

\( \Leftarrow \) Proceed by structural induction on the complexity of \( t \). The interesting case is for a \( \Gamma \) function application. If \( t \equiv \gamma(t_1, \ldots, t_n) \) and \( t \xrightarrow{G,\sigma} b \) then the rules operational rule tells us that there exists a \( k \leq l \) such that \( \tilde{g}_{\gamma,b,k} \xrightarrow{G,\sigma} \tilde{t}t \) and \( \tilde{g}_{\gamma,b,k} \xrightarrow{G,\sigma} b \) and, for all \( i < k \), \( \tilde{g}_{\gamma,b,i} \xrightarrow{G,\sigma} ff \). By working

through the definition of the denotational semantics and replacing the denotational semantics of the sub-terms \( \vec{t} = t_1, \ldots, t_n \) for the values of the

17
variables \( \bar{x}_\gamma \) with the substitution of \( \bar{t} \) for the variables \( \bar{x}_\gamma \) in the terms of the clauses of \( g_\gamma \) by a substitution lemma we derive

\[
M_s(\gamma(\bar{t}))(\mu F)([\sigma]) = \begin{cases} 
M_s(\bar{g}_{\gamma,l_1}(1))(\mu F(\bot))(\sigma) \\
\text{if } M_\text{bool}(\bar{g}_{\gamma,b_1})(\mu F(\bot))(\sigma) \\
M_s(\bar{g}_{\gamma,l_2})(\mu F(\bot))(\sigma) \\
\text{if } M_\text{bool}(\bar{g}_{\gamma,b_2})(\mu F(\bot))(\sigma) \quad \text{and} \\
\text{not } M_\text{bool}(\bar{g}_{\gamma,b_1})(\mu F(\bot))(\sigma) \quad \text{and} \\
\text{not } M_\text{bool}(\bar{g}_{\gamma,b_2})(\mu F(\bot))(\sigma) \quad \text{and} \\
\vdots \\
\text{not } M_\text{bool}(\bar{g}_{\gamma,b_{l_{\gamma - 1}}})(\mu F(\bot))(\sigma) \quad \text{and} \\
\bot \quad \text{otherwise.}
\end{cases}
\]

The induction hypothesis applied to the sequents of the \textit{rules} rule forces the conditional equation (4) to evaluate the \( k \)-th case, i.e.

\[
M_s(\gamma(t_1, \ldots, t_n))(\mu F)([\sigma]) = M_s(\bar{g}_{\gamma,l,k})(\mu F)([\sigma]) = [b] \neq \bot
\]

from which we conclude

\[
\gamma(t_1, \ldots, t_n) \xrightarrow{\sigma} b \implies M_s(\gamma(t_1, \ldots, t_n))(\mu F)([\sigma]) = [b] \neq \bot.
\]

\( \square \)

5 Computational Completeness of Recursive EDS

We show the soundness and adequacy of recursive eds expressed by the theorem.

\textbf{Theorem.}

\textit{Let} \( A \) \textit{be a standard algebra then} \( \text{REDS}(A) = \text{While}^*_A \).
5.1 Simulating Recursive EDS by while-array

In this section we demonstrate an effective two-stage construction of a while-array program $P$ to simulate a $\Gamma$-system $\mathcal{G}$ of recursive eds. This construction shows that, for any standard algebra $A$,

$$\text{REDS}(A) \subseteq \text{While}^*(A).$$

The first stage defines a stack machine, which we call the reds machine, that eliminates $\Gamma$ symbols from a term in $\mathcal{T}(\Sigma \cup \Gamma, X)$ to produce a term in $\mathcal{T}(\Sigma, X)$ using semantic preserving transformations. The machine provides a stack implementation of the semantics of the recursive eds. The second stage is the implementation of the reds machine by a while-array program.

We will focus our attention on the first stage of building the stack machine rather than the precise details of writing a while-array program to simulate a reds machine.

5.1.1 The REDS Machine

Let $\mathcal{G}$ be a $\Gamma$-system of recursive eds and $\sigma : X \to A$ an assignment. The purpose of the reds machine is to eliminate $\Gamma$ symbols from any term $t \in \mathcal{T}(\Sigma \cup \Gamma, X)$ to produce a term $t' \in \mathcal{T}(\Sigma, X)$ such that $\mathcal{O}(t)(\sigma) = \text{te}(t', \sigma)$. This is formalised by Lemma 5.1.4 at the end of this subsection.

In tracking $\mathcal{O}$ we observe that $\mathcal{O}(t)(\sigma) \uparrow$ means either (1) the recursion defined by the $\Gamma$-system $\mathcal{G}$ of recursive eds is non-terminating for $t$ and $\sigma$; or (2) for a particular $\gamma$ none of the tests of the clauses of its recursive eds $g_\gamma$ evaluate to false. The first case is integral to the machine while the second case requires us to force the machine to be non-terminating when the final test of a recursive eds returns false. This special behaviour is associated with the FTest instruction described below.

We will now outline how the machine eliminates $\Gamma$ symbols. For a rigorous mathematical definition of the machine’s behaviour we direct the reader to Appendix A.

The machine works on term-instruction pairs $(t, i)$ of a term $t$ from $\mathcal{T}(\Sigma \cup \Gamma, X)$ and an instruction from $\mathcal{T}^2$. The machine has two stacks, a processed and work stack. The processed stack stores partially processed term-instructions and the work stack stores term-instructions yet to be processed. The state of the reds machine is a pair $(ps, ws)$ of the states of the processed stack $ps$ and the work stack $ws$.

To eliminate $\Gamma$ symbols from a term $t \in \mathcal{T}(\Sigma \cup \Gamma, X)$ the machine is started with an empty processed stack and the work stack containing the

---

2A full definition of $\mathcal{T}$ can be found in Appendix A
single term-instruction \((t, \text{Halt})\). The \text{Halt} instruction causes the machine to stop when, and if, all \(\Gamma\) symbols have been removed from \(t\).

The machine operates by inspecting the term-instruction \(t_i\) on the top of the work stack. There are three cases determined by the term \(t\) of \(t_i\):

1. \(t\) can be a term without any \(\Gamma\) symbols;
2. \(t\) can be a function application \(f\) from \(\Sigma\) applied to sub-terms that may contain \(\Gamma\) symbols; and
3. \(t\) can be a function application \(\gamma\) from \(\Gamma\).

In the first case the machine does not have to eliminate any \(\Gamma\) operations and its behaviour is then determined by the instruction \(i\) paired with \(t\). In the other two cases the top of the work stack is moved to the processed stack and additional term-instructions are placed on the work stack. The details of these two cases follow.

If \(t \equiv f(t_1, \ldots, t_n)\) then its sub-terms are pushed onto the work stack, see Figure 5. The \(j\)-th sub-term \(t_j\), for \(1 \leq j < n\), is paired with a Sub-\(j\)

<table>
<thead>
<tr>
<th>(f(t_1, \ldots, t_n))</th>
<th>(t_1)</th>
<th>Sub-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ps)</td>
<td>(t_2)</td>
<td>Sub-2</td>
</tr>
<tr>
<td></td>
<td>(\vdots)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(t_{n-1})</td>
<td>Sub-(n - 1)</td>
</tr>
<tr>
<td></td>
<td>(t_n)</td>
<td>FSub</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(w)</td>
</tr>
</tbody>
</table>

Figure 5: Pushing the sub-terms onto the work stack.

instruction. When this instruction is considered all \(\Gamma\) symbols in the sub-term \(t_j\) have been eliminated leaving a term \(t_j' \in \mathcal{T}(\Sigma, X)\) on the top of the work stack. The machine substitutes \(t_j'\) for \(t_j\) in the term \(f(\ldots)\) on the top of the processed stack and then removes the term-instruction \((t_j', \text{Sub-}j)\) from the top of the work stack to continue with the next sub-term \(t_{j+1}\), see Figure 6.

The final sub-term \(t_n\) is paired with a FSub instruction. When this instruction is considered the machine will have eliminated all \(\Gamma\)-symbols from the sub-terms of \(f(\ldots)\) and this new term, with its paired instruction, must
(a) Need to eliminate \( \Gamma \)-symbols from \( t_j \).

(b) Eliminated \( \Gamma \) symbols from \( t_j \) to produce \( t'_j \).

(c) Substituting \( t'_j \) for \( t_j \) in \( f(t'_1, \ldots, t'_{j-1}, t_j, \ldots, t_n) \) and considering next sub-term.

Figure 6: The stages of eliminating \( \Gamma \) symbols from a sub-term \( t_j \), for \( 1 \leq j < n \), of \( f(t_1, \ldots, t_n) \).
be considered next. The machine must replace the final sub-term \( t_n \) in \( f(\ldots) \) with its \( \Gamma \) eliminated derivative \( t_n' \) before moving \( (f(t'_1, \ldots, t'_n), i) \) from the processed stack to the work stack.

If \( t \equiv \gamma(t_1, \ldots, t_n) \), the tests and results of the clauses of the recursive eds defining \( \gamma \) are pushed on to the work stack, see Figure 7. All but the last test

\[
\begin{array}{c|c}
\gamma(t_1, \ldots, t_n) & i \\
p_s & \\
\end{array}
\]

Processed
stack

\[
\begin{array}{c|c}
\bar{g}_{\gamma,b,l} & \text{Test} \\
\bar{g}_{\gamma,t,i} & \text{Result}-l_i - 1 \\
\vdots & \\
\bar{g}_{\gamma,b,1} & \text{Test} \\
\bar{g}_{\gamma,t,1} & \text{Result}-1 \\
\bar{g}_{\gamma,t,0} & \text{FTest} \\
\bar{g}_{\gamma,t,i} & \text{Result}-0 \\
w_s & \\
\end{array}
\]

Work stack

Figure 7: The tests and results of the recursive eds defining \( \gamma \) are pushed on to the work stack.

is paired with a Test instruction; the final test \( \bar{g}_{\gamma,b,l} \) is paired with the FTest instruction. When either a Test or FTest instruction is considered a test \( \bar{g}_{\gamma,b,i} \), for \( 1 \leq i \leq l_i \), has a value denoted by a term \( \bar{g}_{\gamma,b,i} \in \mathcal{T}(\Sigma, X) \). The machine’s behaviour is determined by (a) the value \( \bar{g}_{\gamma,b,i} \) takes and (b) the instruction. For either instruction if \( \bar{g}_{\gamma,b,i} \) evaluates to true then the machine removes the test from the work stack and continues with the following results, see Figure 8(a) for the Test instruction and Figure 8(b) for the FTest instruction. Otherwise, if \( \bar{g}_{\gamma,b,i} \) takes the value false, a Test instruction causes the machine to remove the test and its paired result from the work stack and continue with the next test, see Figure 8(c), while a FTest instruction causes the machine to enter an infinite loop.

The result of the \( i \)-th clause is paired with a Result-\( j \) instruction, where \( j = l_i - i \). When this instruction is considered the result \( \bar{g}_{\gamma,t,i} \) has a value expressible by a term \( \bar{g}_{\gamma,t,i} \in \mathcal{T}(\Sigma, X) \). In this case the machine will replace the term \( \gamma(\ldots) \) on the top of the work stack with \( \bar{g}_{\gamma,t,i} \), pop the remaining \( 2j + 1 \) term-instructions off the work stack and finally move the term-instruction \( (\bar{g}_{\gamma,t,i}, i) \) onto the work stack to be considered next.

**Definition 5.1.1.**

The machine value of a term \( t \in \mathcal{T}(\Sigma \cup \Gamma, X) \) for an assignment \( \sigma : X \to A \)
Figure 8: Eliminating the $\Gamma$ symbols from the test of a recursive eds.
is given by the function

$$mv : \mathcal{T}(\Sigma \cup \Gamma, X) \times [X \rightarrow A] \rightarrow A$$

defined as $$mv(t, \sigma) \downarrow te(t', \sigma)$$ where $$t'$$ is the $$\Gamma$$ eliminated term of $$t$$ computed by running the reds machine from the initial state $$(empty, (t, Halt))$$ until and if it halts in a final state $$(empty, [t', Halt])$$. If the machine does not halt then $$mv(t, \sigma) \uparrow$$.

**Example 5.1.2.**

Using the Ack recursive eds from Example 3.2.3 we illustrate the steps of the machine on the term $$+1(Ack(0, +1(0)))$$. For the purposes of this example we are writing terms in full using the prefix notation. Figure 9 shows the steps the machine takes before halting starting with Figure 9(a) showing the initial state.

Because $$+1(\ldots)$$ is a $$\Sigma$$ operation and its sub-term contains a $$\Gamma$$ operation the first step of the machine is to eliminate the $$\Gamma$$ symbols from the sub-term. Figure 9(b) illustrates the state of the machine after pushing the sub-term onto the work stack.

Next, because the term $$Ack(\ldots)$$ on the top of the work stack is a $$\Gamma$$ operation the machine places the tests and result of $$gAck$$ on the work stack with the sub-terms of $$Ack(\ldots)$$ used to denote the values taken by the variables $$x_{Ack}$$ of $$gAck$$, see Figure 9(c).

The term $$= (0, 0)$$ (after applying the substitution) belongs to $$\mathcal{T}(\Sigma, X)$$ so the machine inspects the instruction Test paired with it. This causes $$= (0, 0)$$ to be evaluated to the value true which causes the machine to remove the test thus causing its paired result to be considered next, see Figure 9(d).

The term $$+1(+1(0))$$ belongs to $$\mathcal{T}(\Sigma, X)$$ so the instruction Result-2 is considered. This forces the term $$Ack(\ldots)$$ on the top of the processed stack to be replaced by $$+1(+1(0))$$ and the five entries on the work stack to be remove before moving the top of the processed stack onto the work stack, see Figure 9(e).

The term $$+1(+1(0))$$ is now paired with a FSub instruction that causes $$+1(+1(0))$$ to be substituted for the final (and only) sub-term of the term $$+1(\ldots)$$ on the top of the processed stack. The top of the work stack is removed and the top of the processed stack is moved to the top of the work stack, see Figure 9(f).

Finally, there the term $$+1(+1(0)))$$ is in $$\mathcal{T}(\Sigma, X)$$ and the Halt instruction causes the machine to cease its operations. Hence the machine value $$mv(+1(Ack(0, +1(0))), \sigma)$$ is given by $$te(+1(+1(0))), \sigma) = 3.$$
The initial state of the reads machine with the term loaded:

| empty          | $+1(\text{Ack}(0,+1(0))) \mid \text{Halt}$ |

Processing the sub-term of the function $+1$:

| $+1(\text{Ack}(0,+1(0))) \mid \text{Halt}$ | $\text{Ack}(0,+1(0)) \mid \text{FSub}$ |

(c) Processing the first test of the Ack recursive reads:

| $\text{Ack}(0,+1(0)) \mid \text{FSub}$ | $\text{Ack}(0,+1(0)) \mid \text{FSub}$ |

(d) Processing the first result of the Ack recursive reads:

(e) Reconsidering the sub-term of $+1(\ldots)$ with the Ack($\ldots$) term replaced by a $\Gamma$ eliminated term from $\mathcal{T}(\Sigma, X)$:

| $+1(\text{Ack}(0,+1(0))) \mid \text{Halt}$ | $+1(+1(0)) \mid \text{FSub}$ |

(f) Reconsidering the term $+1(\ldots)$ with its sub-term replaced with a $\Gamma$ eliminated term.

Figure 9: The execution trace of the reads machine for the term $+1(\text{Ack}(0,+1(0)))$. 

25
Definition 5.1.3.
The machine semantics function $\llbracket \bullet \rrbracket_M : A^w \to A$, for any recursive eds $(\overline{x}_γ, t_γ, g_γ)$ is defined as $[g_γ]_M(\sigma) = m(v(\gamma(\overline{x}_γ), \sigma))$ where $\sigma[\overline{x}_γ] = a$.

We strictly extend $\llbracket \bullet \rrbracket_{ds}$ to operate on a vector $g_γ$ of recursive eds from $\mathcal{G}$.

Lemma 5.1.4.
Let $\mathcal{G}$ be a $\Gamma$-system of recursive eds then, for any recursive eds $g_γ$ of $\mathcal{G}$,

$$[g_γ]_{os} \simeq [g_γ]_M.$$  

5.1.2 While-array Program of a REDS Machine

To write a while-array program to simulate the operations of the reds machine we use

1. arrays to implement the stacks of the reds machine; and
2. an effective Gödel numbering $\phi : \mathbb{N} \to \mathcal{T} \mathcal{T}$ of the set of term-instructions.

Such a program must use computable tests on the representation $n$ of a term-instruction $(t, i)$ to determine

1. if the term $t$ contains a symbol from $\Gamma$;
2. if the term $t$ is a $\Gamma$ constant or function;
3. if the instruction $i$ is the Halt instruction;
4. if the instruction $i$ is a Sub-$j$ instruction and, in addition, a function $\text{Sub-j} : \mathbb{N} \to \mathbb{N}$ defined as $\text{Sub-j}(n) = j$;
5. if the instruction $i$ is a FSub instruction and, furthermore, a function $\text{FSub} : \mathbb{N} \to \mathbb{N}$ defined by $\text{FSub}(n) = \text{arity of the } \Sigma \text{ function on the top of the processed stack};$
6. if the instruction $i$ is a Test or FTest instruction; and
7. if the instruction $i$ is a Result-$j$ instruction and a function $\text{Result-j} : \mathbb{N} \to \mathbb{N}$ defined as $\text{Result-j}(n) = j$.

The program will also need to be able to manipulate a term $t$ within the Gödel number representation to

1. extract the sub-terms $t_1, \ldots , t_n$ of a function application $f(t_1, \ldots , t_n)$ or $\gamma(t_1, \ldots , t_n)$;
2. replace a sub-term \( t_j \) with a term \( t' \) in \( f(\ldots) \); and

3. apply a simultaneous substitution \( \langle t_1, \ldots, t_n/x_1, \ldots, x_n \rangle \) of terms for variables to \( t \).

All of the tests and projection functions for term-instructions and term manipulation operations and the representation functions of the recursive eds in \( G \) are computable by a \textbf{while} program on a \( N \)-standard algebra due to the effective Gödel numbering of term-instructions.

Although the program needs to evaluate terms at various control flow junctions it is shown in Tucker and Zucker [1988] that term evaluation is computable on any standard algebra \( A \) by a \textbf{while-array} program.

5.2 Simulating while-array programs by Recursive ED-S

It is shown in Tucker and Zucker [1988] that a \textbf{while-array} program can be simulated by a course-of-values (cov) scheme \( \alpha \in \mu CR(\Sigma) \). We show below that any \( \mu CR(\Sigma) \) scheme can be simulated by recursive eds thus demonstrating the adequacy of recursive eds, i.e.

\[ \text{While}^t(A) \subseteq REDS(A). \]

Recall that a cov scheme is built from the base constructions of: (1) \( \Sigma \) functions, (2) projection and (3) conditionals and the inductive constructions of: (4) vectorisation, (5) composition, (6) cov recursion and (7) least number search. Cov schemes are a generalisation of induction schemes pioneered by Kleene [1952]. They were advocated first in Tucker and Zucker [1988], but details can also be found in §7 of Tucker and Zucker [1999].

For a cov scheme \( \alpha \in \mu CR(\Sigma)_{u \rightarrow v} \), we define a \( \Gamma_\alpha \)-indexed system \( G_\alpha \) of recursive eds from which a selection \( g_{\gamma_\alpha} = g_{\alpha,1}, \ldots, g_{\alpha,m} \) of recursive eds track the function defined by \( \alpha \), i.e. for all \( a \in A^u \)

\[ [\alpha](a) = [g_{\gamma_\alpha}](a). \]

We call \( g_{\gamma_\alpha} \) the recursive eds of \( \alpha \).

A template showing how \( G_\alpha \) can be displayed is shown in Figure 10. The \( \alpha \) \texttt{reds} option lists the names \( \gamma_\alpha \) of the recursive eds of \( \alpha \). If \( \alpha \) is inductively built from existing schemes \( \beta = \beta_1, \ldots, \beta_k \) then the \texttt{import} option names the recursive eds systems for \( \beta \). Importing \( G_{\beta_i} \) includes every recursive ed in \( G_{\beta_i} \) not only the recursive eds of \( \beta_i \).
Figure 10: Template for displaying a $\Gamma_\alpha$-indexed system $G_\alpha$ for a cov scheme $\alpha$.

1. *(Operations from $\Sigma_\ast$)* If $\alpha \equiv \text{sig}(f) \in \mu CR(\Sigma)_{w \rightarrow s}$, for $f \in \Sigma_{w \rightarrow s}$, then we define $G_{\text{sig}(f)}$ as

   System $G_{\text{sig}(f)}$
   $\alpha$ Red $\gamma_\alpha$
   Begin Recursive Eds
   Op $\gamma_\alpha : u \rightarrow s$
   Var $\vec{x}_\alpha : u$
   Red $g_{\gamma_\alpha}(1) = \text{true} \rightarrow f(\vec{x}_\alpha)$
   End Recursive Eds

2. *(Projection.)* If $\alpha \equiv U_{u,i} \in \mu CR(\Sigma)_{u \rightarrow s \ast}$ then we define $G_{U_{u,i}}$ as

   System $G_{U_{u,i}}$
   $\alpha$ Red $\gamma_\alpha$
   Begin Recursive Eds
   Op $\gamma_\alpha : u \rightarrow s_i$
   Var $\vec{x}_\alpha : u$
   Red $g_{\gamma_\alpha}(1) = \text{true} \rightarrow x_i$
   End Recursive Eds

3. *(Condition.)* If $\alpha \equiv dC_s \in \mu CR(\Sigma)_{\text{bool}_a \rightarrow s \ast}$ then we define $G_{dC_s}$ as
4. (Vectorisation.) If \( \alpha = \text{vect}_{u,v}(\beta_1, \ldots, \beta_m) \in \mu CR(\Sigma)_{u \rightarrow v} \), where \( v = v_1 \times \cdots \times v_m \) and, for \( 1 \leq i \leq m \), \( \beta_i \in \mu CR(\Sigma)_{u_i \rightarrow v_i} \), then we define \( G_{\text{vect}_{u,v}(\beta_1, \ldots, \beta_m)} \) as

\[
\begin{array}{l}
\text{System } G_{\text{vect}_{u,v}(\beta)} \\
\text{Reds } \gamma_{\beta_1, \ldots, \beta_m} \\
\text{Import } G_{\beta_1, \ldots, G_{\beta_m}}
\end{array}
\]

End Recursive Eds

Note that in this case no new recursive eds are defined. Instead the recursive eds of the \( \beta_i \) schemes are gathered together and used to define \( \text{vect}_{u,v}(\beta_1, \ldots, \beta_m) \).

5. (Composition.) If \( \alpha \equiv \text{comp}_{u,v,w}(\beta_1, \beta_2) \in \mu CR(\Sigma)_{u \rightarrow w} \), where \( v = v_1 \times \cdots \times v_m \) and \( w = w_1 \times \cdots \times w_o \) and \( \beta_1 \in \mu CR(\Sigma)_{u \rightarrow v_1} \) and \( \beta_2 \in \mu CR(\Sigma)_{v \rightarrow w} \), then we define \( G_{\text{comp}_{u,v,w}(\beta_1, \beta_2)} \) as

\[
\begin{array}{l}
\text{System } G_{\text{comp}_{u,v,w}(\ldots)} \\
\text{Reds } \gamma_{\alpha,1} : u \rightarrow w_1 \\
\text{Import } G_{\beta_1}, G_{\beta_2} \\
\text{Begin Recursive Eds} \\
\text{Op } \gamma_{\alpha,1} : u \rightarrow w_1 \\
\text{Var } \vec{x}_\alpha : u \\
\text{Reds } g_{\gamma_{\alpha,1}}(1) = \text{true} \rightarrow \gamma_{\beta_1,1}(\gamma_{\beta_2,1}(\vec{x}_\alpha), \ldots, \gamma_{\beta_2,m}(\vec{x}_\alpha)) \\
\quad \vdots \\
\text{Op } \gamma_{\alpha,o} : u \rightarrow w_o \\
\text{Var } \vec{x}_\alpha : u \\
\text{Reds } g_{\gamma_{\alpha,o}}(1) = \text{true} \rightarrow \gamma_{\beta_1,1}(\gamma_{\beta_2,1}(\vec{x}_\alpha), \ldots, \gamma_{\beta_2,m}(\vec{x}_\alpha)) \\
\text{End Recursive Eds}
\end{array}
\]
6. \textit{(Simultaneous course-of-values recursion.)} If \( \alpha \equiv \text{cv}_{\nu,d}(\beta_1, \beta_2, \delta) \in \mu CR(\Sigma)_{\text{nat} \times u \rightarrow \text{nat}} \), where \( v = v_1 \times \cdots \times v_m \) and \( \beta_1 \in \mu CR(\Sigma)_{u \rightarrow w}, \beta_2 \in \mu CR(\Sigma)_{\text{nat} \times u \times v' \rightarrow w} \), and \( \delta = \delta_1, \ldots, \delta_d \) such that, for \( 1 \leq i \leq d, \delta_i \in \mu CR(\Sigma)_{\text{nat} \times u \rightarrow \text{nat}} \), then we define \( G_{\text{cv}_{\nu,d}(\beta_1, \beta_2, \delta)} \) as

\[
\text{System } \ G_{\text{cv}_{\nu,d}(\beta_1, \beta_2, \delta)} \\
\alpha \text{ Red } \gamma_{\alpha,1}, \ldots, \gamma_{\alpha,m} \\
\text{Import } G_{\beta_1}, G_{\beta_2}, G_{\delta_1}, \ldots, G_{\delta_d} \\
\text{Begin Recursive Eds} \\
\text{Op } \gamma_{\alpha,1} : \text{nat} \times u \rightarrow v_1 \\
\text{Var } n, \bar{x}_\alpha : \text{nat}, u \\
\text{Reds } g_{\gamma_{\alpha,1}}(1) = n = 0 \rightarrow \gamma_{\beta_1}(\bar{x}_\alpha) \\
\qquad \quad g_{\gamma_{\alpha,1}}(2) = \text{Not}(n = 0) \rightarrow \\
\qquad \qquad \gamma_{\beta_{2,1}}(n, \bar{x}_\alpha, \gamma_\alpha(\min(\gamma_{\delta_1}(n, \bar{x}_\alpha), n - 1), \bar{x}_\alpha), \\
\qquad \qquad \qquad \ldots, \gamma_\alpha(\min(\gamma_{\delta_d}(n, \bar{x}_\alpha), n - 1), \bar{x}_\alpha)) \\
\vdots \\
\text{Op } \gamma_{\alpha,m} : \text{nat} \times u \rightarrow v_m \\
\text{Var } n, \bar{x}_\alpha : \text{nat}, u \\
\text{Reds } g_{\gamma_{\alpha,m}}(1) = n = 0 \rightarrow \gamma_{\beta_{1,m}}(\bar{x}_\alpha) \\
\qquad \quad g_{\gamma_{\alpha,m}}(2) = \text{Not}(n = 0) \rightarrow \\
\qquad \qquad \gamma_{\beta_{2,m}}(n, \bar{x}_\alpha, \gamma_\alpha(\min(\gamma_{\delta_1}(n, \bar{x}_\alpha), n - 1), \bar{x}_\alpha), \\
\qquad \qquad \qquad \ldots, \gamma_\alpha(\min(\gamma_{\delta_d}(n, \bar{x}_\alpha), n - 1), \bar{x}_\alpha)) \\
\text{End Recursive Eds}
\]

In the above we have shortened \( \gamma_{\alpha,1}(\ldots), \ldots, \gamma_{\alpha,m}(\ldots) \) to \( \gamma_\alpha(\ldots) \).

7. \textit{(Least number search.)} If \( \alpha \equiv \text{lst}_u(\beta) \in \mu CR(\Sigma)_{u \rightarrow \text{nat}} \), where \( \beta \in \mu CR(\Sigma)_{\text{nat} \times u \rightarrow \text{bool}^+} \), then we define \( G_{\text{lst}_u(\beta)} \) as

\[
\text{System } \ G_{\text{lst}_u(\beta)} \\
\alpha \text{ Red } \gamma_{\alpha,2} \\
\text{Import } G_{\beta} \\
\text{Begin Recursive Eds} \\
\text{Op } \gamma_{\alpha,1} : \text{nat} \times u \rightarrow \text{nat} \\
\text{Var } n, \bar{x}_\alpha : \text{nat}, u \\
\text{Reds } g_{\gamma_{\alpha,1}}(1) = \gamma_\beta(n, \bar{x}_\alpha) \rightarrow n \\
\qquad \quad g_{\gamma_{\alpha,1}}(2) = \text{Not}(\gamma_\beta(n, \bar{x}_\alpha)) \rightarrow \gamma_{\alpha_1}(n + 1, \bar{x}_\alpha) \\
\text{Op } \gamma_{\alpha,2} : u \rightarrow \text{nat} \\
\text{Var } \bar{x}_\alpha : u \\
\text{Reds } g_{\gamma_{\alpha,2}}(1) = \text{true} \rightarrow \gamma_{\alpha,1}(0, \bar{x}_\alpha) \\
\text{End Recursive Eds}
\]
We note that in case 6 when defining the recursive eds we use the predecessor \( +1 : \text{nat} \to \text{nat} \) and minimum \( \min : \text{nat}^2 \to \text{nat} \) operations. Neither of these operations are necessarily included in \( \Sigma \) and so may need to be defined by recursive eds. Both are simply defined via the use of a searching mechanism, for example the predecessor of \( x \) is found by increasing \( n \) from 0 until \( n + 1 = x \). The reader should note that the error case of \( x = 0 \) needs to excluded before the start of the search to make the predecessor function total.

6 Related Work and Concluding Remarks

6.1 Tables

The use of tables, specifically in the guise of decision tables, as a technique for aiding software developers was recognized by early researchers during the sixties and seventies, see King [1968] for a survey of decisions tables in the sixties. Decision tables are still an active research topic, see Leveson et al. [1994], Hoover and Chen [1995] and Day et al. [1997]. We believe that decision tables are as expressive as simple algebraic tables.

In 1978 D. Parnas and others worked with tables to document the computational requirements of a complex embedded system component of the A-7E naval aircraft as part of the pilot Software Cost Reduction (SCR) project, Heninger et al. [1978] and Heninger [1980]. Only three distinct types of tables are used in the SCR project and they are collectively referred to as the SCR tables. The SCR tables have remained unchanged since their original conception in 1978, see Heitmeyer et al. [1998]. We believe that SCR tables are as expressive as simple algebraic tables. This belief is strengthened by the transformation of tables in Zucker [1996] and the closure of composition result in von Mohrenschildt [1997].

Simple algebraic tables have been used in Zucker [1996] (and the sequel Zucker and Shen [1998]) for describing transformations of tables. In Wilder and Tucker [1998] nested algebraic tables are used for documenting the next state function \( n : S \times C \to S \) that defines the behaviour of a reactive system modelled by the sets \( S \) and \( C \) of states and commands, respectively.

6.2 Related Work

To use tables effectively for describing computations a semantics is necessary. An initial description of the syntax and semantics of a variety of tables is given in Parnas [1992]. Heitmeyer et al. [1996] provide a semantics for the
SCR formalism that includes the SCR tables. However, both of these are unsatisfactory from a tables perspective because, for each new type of table, a separate semantics is given that describes how the cells of the table combine to define a function or relation.

Janicki [1995] proposes a cell connection, a table predicate rule and a table relation rule for homogenising the semantics of tables that document an input-output relation $R \subseteq X \times Y$ (equivalent to a multi-valued partial function). The cell connection separates the components of a table into input or output. The predicate and relation rules combine the cells of the input and output components (respectively) located at a given index $\alpha \in I$ to form a single test $P_\alpha$ and result $R_\alpha$. The union of all $P_\alpha \land R_\alpha$ defines the $R$. Abraham [1997] extends the work of Janicki [1995] to homogenise the semantics of a wider class of tables including SCR, vector and decision tables by considering operations other than set union on $P_\alpha \land R_\alpha$.

However, none of the above semantics explicitly or adequately manage recursive tables. The PVS system Owre et al. [1992] has been extended with a table constructor, see Owre et al. [1995]. Currently, the PVS system incorporates facilities for defining Parnas, SCR and decision tables only. The PVS system permits recursive functions including those defined by tables. Unfortunately there are no formal semantics of the recursion of PVS, see Owre and Shankar [1999].

6.3 Concluding Remarks

In this paper we have reinforced the relationship between tables and decisions, for which Janicki [1995] and Abraham [1997] describe transformations from tables to lists of decisions. This provides a clean separation of the semantics for lists of decisions from tabular documentation.

Using a specialisation of Janicki [1995], we have demonstrated that recursive algebraic tables are as expressive as while-array program schemes. It is known that finite algebraic tables representing finite eds are only as expressive as straight-line programs, see Thiery [1980] and Wilder [1997]. For a standard algebra $A$, Figure 11 summarises (1) the equivalence of eds, recursive eds and while-array programs; (2) the equivalence of finite eds and straight-line programs; (3) the injection of straight-line to while to while-array; and (4) how, if $A$ is $N$-standard and has the TEP then while is as expressive as while-array. It is possible to give direct proofs relating eds and recursive eds to while programs in the presence of the TEP on $A$, as is done in Wilder [1997].

From Section 6.1 it appears that the tables reported in the literature are all based around an expressively weak documentation technique. However,
the use of tables is predominantly focused on the early requirements life-cycle phase where high-level abstractions are widely and beneficially used. It is likely that these abstractions are hiding complex functions for which a stronger model of computation and documentation is required. By the results of this paper, such functions can be documented by recursive tables.

References

Abraham [1997]

Bergstra and Tucker [1987]

Day et al. [1997]

Faulk et al. [1992]
Faulk et al. [1994]

Friedman [1971]

Heitmeyer et al. [1996]

Heitmeyer et al. [1998]

Heninger et al. [1978]

Heninger [1980]

Hoover and Chen [1995]

Janicki [1995]
Janicki [1997]

King [1968]

Kleene [1952]

Leveson et al. [1994]

Meinke and Tucker [1992]

Owre and Shankar [1999]

Owre et al. [1992]

Owre et al. [1995]

Parnas and Wang [1989]
Parnas et al. [1994]

Parnas [1992]

Tiuryn [1980]

Tucker and Zucker [1988]

Tucker and Zucker [1999]

von Mohrenschildt [1997]

Wechler [1992]

Wilder and Tucker [1998]

Wilder [1997]
Wirsing [1990]

Zucker and Shen [1998]

Zucker [1996]

## A Semantics of REDS Machine

### A.1 The REDS Machine’s State

**Definition A.1.1 (Instructions).**
The instructions for the reds machine are
\[
I = \{\text{Halt}\} \cup \{\text{Sub-j} | j \in \mathbb{N}\} \cup \{\text{FSub, Test, FTest}\} \cup \{\text{Result-j} | j \in \mathbb{N}\}.
\]

**Definition A.1.2 (Term-instructions).**
A term-instruction is a pair \((t, i)\) of a term \(t \in T(\Sigma, X)\) and an instruction \(i \in I\). Let \(TI\) be the set of all possible term-instructions.

**Definition A.1.3 (Machine state).**
The set of states of a reds machine is
\[
M = \text{Stack}(TI) \times \text{Stack}(TI),
\]

namely the processed and the work stack. The machine is always in a state \(m = (ps, ws) \in M\).

### A.2 The REDS Machine’s Behaviour

We now present a reds machine state transformer \(n : M \to M\) to describe rigorously the behaviour of the machine. For an informal account of the reds machine behaviour the reader should consult Section 5.1.1.

The reds machine state transformer \(n\) is defined by a structural induction on the complexity of the term-instruction currently on the top of the work stack, \(top(ws) = (t, i)\). There are three primary cases dependent on \(t\).
1. When $t \in T(\Sigma, X)$ we consider the instruction $i$ paired with $t$.
   
   (a) When $i \equiv \text{Sub}$ we define
   \[
   n(ps, ws) = \left( \text{push}([f(t_1, \ldots, t_{j-1}, t, t_{j+1}, \ldots, t_n), i'], \text{pop}(ps)), \text{pop}(ws) \right)
   \]
   where $\text{top}(ps) = [f(t_1, \ldots, t_n), i'].$
   
   (b) When $i \equiv \text{FSub}$ we define
   \[
   n(ps, ws) = \left( \text{pop}(ps), \text{push}((f(t_1, \ldots, t_{n-1}, t), i'), \text{pop}(ws)) \right)
   \]
   where $\text{top}(ps) = [f(t_1, \ldots, t_n), i'].$
   
   (c) When $i \equiv \text{Test}$ the machine considers $te(t, \sigma)$.
      
      i. If $te(t, \sigma) = \tt$ we define
      \[
      n(ps, ws) = (ps, \text{pop}(ws)); \text{otherwise}
      \]
      
      ii. $te(t, \sigma) = \ff$ and we define
      \[
      n(ps, ws) = (ps, \text{pop}^2(ws)).
      \]
   
   (d) When $i \equiv \text{FTest}$ the machine considers $te(t, \sigma)$.
      
      i. If $te(t, \sigma) = \tt$ we define
      \[
      n(ps, ws) = (ps, \text{pop}(ws)); \text{otherwise}
      \]
      
      ii. $te(t, \sigma) = \ff$ and we define
      \[
      n(ps, ws) = (ps, ws).
      \]

   (e) When $i \equiv \text{Result}$ the machine combines the term from the top of the stack with the instruction from the top of the processed stack. If $\text{top}(ps) = (t', i')$ we define
   \[
   n(ps, ws) = (\text{pop}(ps), \text{push}((t, i'), \text{pop}^{2j+1}(ws))).
   \]

   (f) When $i \equiv \text{Halt}$ the machine halts.

2. When $t \in T(\Sigma \cup \Gamma, X)$ the machine considers if $t \equiv f(t_1, \ldots, t_n)$, for $f \in \Sigma$. If so we define
   \[
   n(ps, ws) = \left( \text{push}(\text{top}(ws), ps), \right.
   \]
   \[
   \text{push}((t_1, \text{Sub}), \ldots, \text{push}((t_{n-1}, \text{Sub} - n - 1),
   \]
   \[
   \text{push}((t_n, \text{FSub}), \text{pop}(ws))) \ldots ) \right).
   \]
3. If \( t \in \mathcal{T}(\Sigma \cup \Gamma, X) \) and \( t \equiv \gamma(t_1, \ldots, t_n) \), for \( \gamma \in \Gamma \), we define

\[
\begin{align*}
n(ps, ws) &= (\text{push}(\text{top}(ws), ps), \\
&\quad \text{push}(\bar{\gamma}, \text{Test}), \text{push}(\bar{\gamma}, \text{Result}-1), \ldots) \\
&\quad \text{push}(\bar{\gamma}, \text{Result}-1), \text{push}(\bar{\gamma}, \text{Result}-0), \text{pop}(ws)))\ldots).
\end{align*}
\]

**Definition A.2.1 (Length of reds machine computation).**

The length of the computation of a reds machine on a term \( t \in \mathcal{T}(\Sigma \cup \Gamma, X) \) by \( \text{mv}(t, \sigma) \) is the least \( l \in \mathbb{N} \) such that \( n(l, \text{empty}, (t, \text{Halt})) = (ps', ws') \) and \( \text{top}(ws') \in \mathcal{T}(\Sigma, X) \times \{\text{Halt}\} \).

The length of a reds machine computation is undefined if a recursion that keeps unfolding or all of the tests of a reds produce terms that evaluate to false.

## B Proof of Equivalence of Operational and Machine Semantics

Lemma 5.1.4 states that, for any recursive eds \( g_\gamma \) from a system \( \mathcal{G} \) of recursive eds and data \( a \in A^m \),

\[
\llbracket g_\gamma \rrbracket_{\sigma^i}(a) \simeq [g_\gamma]_M(a).
\]  

The proof of this is outlined in Appendix B.2. To facilitate the proof we note in Appendix B.1 a simple property of the reds machine’s ability to remove \( \Gamma \) operations.

### B.1 Deterministic \( \Gamma \) Elimination

For a fixed \( \Gamma \)-system \( \mathcal{G} \) of recursive eds and assignment \( \sigma : X \rightarrow A \) it is necessary for the reds machine to deterministically eliminate occurrences of \( \Gamma \) symbols from a term \( t \in \mathcal{T}(\Sigma \cup \Gamma, X) \) on the top of the work stack independent of its paired instruction, the processed stack and the remaining contents of the work stack.

**Lemma B.1.1 (Deterministic \( \Gamma \) elimination).**

Consider a fixed \( \mathcal{G} \) and \( \sigma : X \rightarrow A \) and let \( t \in \mathcal{T}(\Sigma \cup \Gamma, X) \) and \( t' \in \mathcal{T}(\Sigma, X) \) be terms, \( i, i' \in I \) be instructions and \( m = (ps, ws), m' = (ps', ws') \in M \) be machine states then the reds machine runs from

\[
(ps, \text{push}((t, i), ws)) \text{ to } (ps, \text{push}((t', i), ws))
\]
just in those cases that it also runs from

\((ps', \text{push}((t, i), ws'))\) to \((ps', \text{push}((t', i'), ws'))\).

Proof. By induction on the length of computation’s of the reds machine and the definition of the reds machine state transformed \(n : M \to M\). \(\square\)

A simple corollary of Lemma B.1.1 used frequently in Appendix B.2 is the following.

**Corollary B.1.2.**

For a fixed \(\mathcal{G}\) and \(\sigma : X \to A\) let \(t \in \mathcal{T}(\Sigma \cup \Gamma, X)\) be a term then \(m v(t, \sigma) = a\) just in those cases when, for an instruction \(i\) and a machine state \(m = (ps, ws) \in M\), the machine runs from \((ps, \text{push}((t, i), ws))\) to \((ps, \text{push}((t', i), ws))\) and, furthermore, \(t e(t', \sigma) = a\).

## B.2 The Proof

To prove (5) we must show both directions of

\[
\mathcal{O}_i(\gamma(\overline{x})) (\sigma) \Downarrow b \iff m v(\gamma(\overline{x}), \sigma) \Downarrow b. \tag{6}
\]

In the following we refer to the structural cases defining the reds machine state transformed \(n : M \to M\) from Appendix A.2. To see \(\Rightarrow\) of (6) we proceed by induction on the complexity of the derivation of \(t \xrightarrow{\mathcal{G}, \sigma} b\).

1. **Base cases.** There are two cases to consider; the \textit{var} and \textit{cons} rules. Both cases are similar; for convenience we will show only the case of the \textit{var} rule. \(\mathcal{O}_i(x)(\sigma) \Downarrow \sigma(x)\) by the rule

\[
\frac{x \xrightarrow{\mathcal{G}, \sigma} \sigma(x)}{}
\]

Running the machine from \(m v(x, \sigma)\) case \(\text{If}\) applies and the machine terminates immediately leaving \(x\) on the top of the work stack for which \(t e(x, \sigma) = \sigma(x)\).

2. **Inductive cases.** There are two cases to consider; the \textit{func} and \textit{reds} rules. Both cases are similar; we will show the more interesting \textit{reds} case. \(\mathcal{O}_i(\gamma(t_1, \ldots, t_n))(\sigma) \Downarrow b\) by the rule

\[
\begin{align*}
\exists k \leq l & \ \text{s.t.} \ y_{r, b, 1} \xrightarrow{\mathcal{G}, \sigma} ff, \ldots, y_{r, b, k - 1} \xrightarrow{\mathcal{G}, \sigma} ff, \\
y_{r, b, k} & \xrightarrow{\mathcal{G}, \sigma} tt, y_{r, t, k} \xrightarrow{\mathcal{G}, \sigma} a
\end{align*}
\]

\[
\gamma(t_1, \ldots, t_n) \xrightarrow{\mathcal{G}, \sigma} a
\]
By applying the induction hypothesis to the sequents we derive
\[ O_i(\bar{\gamma}, b, i)(\sigma) \downarrow ff \implies mv(\bar{\gamma}, b, i, \sigma) \downarrow ff, \]
for \(1 \leq i < k\),
\[ O_i(\bar{\gamma}, b, k)(\sigma) \downarrow tt \implies mv(\bar{\gamma}, b, k, \sigma) \downarrow tt \]
and
\[ O_i(\bar{\gamma}, l, k)(\sigma) \downarrow b \implies mv(\bar{\gamma}, b, k, \sigma) \downarrow b. \]

Running the machine as \(mv(\gamma(t_1, \ldots, t_n), \sigma)\) invokes case 3 leaving the machine in a state as illustrated in Figure 12.

<table>
<thead>
<tr>
<th>(\gamma(t_1, \ldots, t_n))</th>
<th>Halt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed stack</td>
<td></td>
</tr>
<tr>
<td>(\bar{\gamma}, b, 1)</td>
<td>Test</td>
</tr>
<tr>
<td>(\bar{\gamma}, b, i)</td>
<td>Result-L_i - 1</td>
</tr>
<tr>
<td>(\bar{\gamma}, b, l_1)</td>
<td></td>
</tr>
<tr>
<td>(\bar{\gamma}, b, l_{i-1})</td>
<td>Test</td>
</tr>
<tr>
<td>(\bar{\gamma}, b, l_i)</td>
<td>Result-1</td>
</tr>
<tr>
<td>(\bar{\gamma}, b, l_{i-1})</td>
<td>FTest</td>
</tr>
<tr>
<td>(\bar{\gamma}, b, l_i)</td>
<td>Result-0</td>
</tr>
</tbody>
</table>

Work stack

Figure 12: The state of the reds machine after loading the tests and results of the reds \(\gamma\).

By \(k - 1\) applications of (7), Corollary B.1.2 and case 1(c)ii the machine pops 2\((k - 1)\) term-instructions from the work stack corresponding to the first \(k - 1\) tests evaluating to false. Then, by (8) and Corollary B.1.2 case 1(c)i applies if \(k < l\), or, if \(k = l\), case 1(d)i applies. In either case the machine pops off the \(k\)-th test leaving the \(k\)-th result on the top of the work stack. By (9) and Corollary B.1.2 case 1e applies and the machine enters a state
\[(empty, (t', Halt))\]
where \(te(t', \sigma) = b\).

To see \(\Rightarrow\) of (6) we proceed by induction on the length of terminating machine computations.
1. **Base case.** The machine can terminate after \( n = 0 \) steps when \( t \in \mathcal{T}(\Sigma, X) \) by case 1f. In this case it is easy to see by structural induction on the complexity of \( t \) that if \( tc(t, \sigma) = b \) then \( t \xrightarrow{\sigma} b \).

2. **Inductive cases.** When \( t \in \mathcal{T}(\Sigma \cup \Gamma, X) \) there are two cases to consider, (1) if \( t \equiv f(t_1, \ldots, t_n) \) with occurrences of \( \Gamma \) symbols in one or more of the sub-terms; and (2) if \( t \equiv \gamma(t_1, \ldots, t_n) \). For convenience we will demonstrate only the case of \( f(t_1, \ldots, t_n) \); the case of \( \gamma(t_1, \ldots, t_n) \) proceeds using similar arguments.

When \( t \equiv f(t_1, \ldots, t_n) \) case 2 applies and the state of the machine becomes that as illustrated in Figure 13. For the machine to halt on

\[
\begin{array}{c|c}
  f(t_1, \ldots, t_n) & \text{Halt} \\
  \text{Processed stack} & \hline \\
  t_1 & \text{Sub-1} \\
  t_2 & \text{Sub-2} \\
  \vdots & \\
  t_{n-1} & \text{Sub-} n - 1 \\
  t_n & \text{FSub} \\
\end{array}
\]

Figure 13:

\((t, \text{Halt})\) there must exist a \( t_j' \in \mathcal{T}(\Sigma, X) \), for \( 1 \leq j < n \), such that the machine runs from

\[
((f(\ldots), \text{Halt}), (t_j, \text{Sub-j}) \circ \cdots \circ (t_n, \text{FSub})) \quad \text{to} \quad ((f(\ldots), \text{Halt}), (t'_j, \text{Sub-j}) \circ \cdots \circ (t_n, \text{FSub})).
\]  

(11)

By case 1a the machine substitutes \( t_j' \) for \( t_j \) in \( f(\ldots) \) on the top of the processed stack before popping \((t'_j, \text{Sub-j})\) off the work stack. The machine repeats this activity for the term-instructions \((t_j, \text{Sub-j}), j = 2, \ldots, n - 1 \) until the term-instruction \((t_n, \text{FSub})\) is on the top of the work stack. Again, to terminate on \( f(\ldots) \) there must exist a \( t_n' \in \mathcal{T}(\Sigma, X) \) such that the machine runs from

\[
((f(\ldots), \text{Halt}), (t_n, \text{FSub})) \quad \text{to} \quad ((f(\ldots), \text{Halt}), (t'_n, \text{FSub})).
\]  

(12)

By case 1b the machine (a) substitutes \( t'_n \) for \( t_n \) in \( f(\ldots) \) before popping \((t'_n, \text{FSub})\) off the work stack and (b) moves \((f(\ldots), \text{Halt})\) back onto the
work stack. As \( f(t'_1, \ldots, t'_n) \in \mathcal{T}(\Sigma, X) \) case 1f applies and the machine halts. If \( te(t'_j, \sigma) = b_j \), for \( 1 \leq j \leq n \), then Corollary B.1.2 applied to (11) and (12) gives

\[
mv(t_j, \sigma) = b_j. \tag{13}
\]

Because

\[
mv(t, \sigma) = te(f(t'_1, \ldots, t'_n), \sigma) = \]
\[
f_A(te(t'_1, \sigma), \ldots, te(t'_n, \sigma)) = f_A(b_1, \ldots, b_n)
\]

and \( t_j \xrightarrow{\sigma} b_j \) can be deduced from the induction hypothesis and (13) then the

\[
\frac{t_1 \xrightarrow{\sigma} b_1, \ldots, t_n \xrightarrow{\sigma} b_n}{f(t_1, \ldots, t_n) \xrightarrow{\sigma} f_A(b_1, \ldots, b_n)} \quad (\text{func})
\]

rule applies and we conclude that \( O_A(t)(\sigma) = mv(t, \sigma). \)