Transformations of Normal and Inverted Function Tables[1]

J.I. Zucker

Department of Computer Science and Systems,
McMaster University, Hamilton, Ontario L8S 4K1, Canada
zucker@mcmaster.ca

Abstract

We develop a theory of function tables, similar to, and inspired by, that given in the work of D. Parnas. We consider, in particular, two classes of function tables: normal and inverted. We study effective transformations between tables of these two classes, as well as transformations which change the dimension of a table. We also consider the interrelationship between these three types of transformation.

[1]Research supported by grants from the Natural Sciences and Engineering Research Council of Canada, and by funding by the Telecommunications Research Institute of Ontario.

Typeset by $\mathcal{AMS}$-TEX

0 Introduction

The method of tabular representations, developed by David Parnas and his collaborators, has been found to be very useful for the formal documentation and inspection of software systems.

The first application of this technique was in the documentation for the revised flight software for the US Navy's A-7 aircraft in the late seventies ([Hen80], [HKP78]). Another large project which used the tabular notation was the documentation of the shutdown systems of the Darlington Nuclear Power Generating Station in Ontario, Canada, required by the Atomic Energy Control Board of Canada for that station's licensing, in the late eighties ([PAM91], [Par94]). These two projects served both as testing grounds for the tabular method, and as incentives for its further development.

The tabular method is also useful in the documentation of simple programs, as demonstrated in [PMI92]. Some examples of its use in system documentation are given in [WiT95]. A survey of the method is given in [JPZ96].

The tabular notation is, essentially, a useful and perspicuous method for defining functions on many sorted algebras. In the course of the projects described above, many kinds of tables were developed, and were found to be useful. A systematic exposition of ten kinds of tabular expressions was given in [Par92].

In the consideration of such a variety of methods, many questions arise naturally: Given a function, what is the best kind of table to represent it? Given two tables (of the same or different kinds), can we see whether they define the same function? Can we (or under what conditions can we) perform certain useful manipulations on tables, such as transforming a table to a "simplest" form of the same kind, or transforming one kind of table to another?

The present paper focuses on the last question. We consider two of the many kinds of tables treated in [Par92], namely *normal function tables* and *inverted function tables*. We study various methods for effectively transforming one kind of table to the other.

The formalisation of tables given in [Par92] has been modified to some extent, leading to greater clarity. In particular, the syntax of any given table is based on the *signature* of the corresponding structure.

The outline of the rest of this paper is as follows. Section 1 contains preliminary material on many-sorted signatures and algebras. Section 2 describes the language (terms and conditions) out of which tables will be built. Thereafter the syntax and semantics of tables are given: normal tables in Section 3, and inverted tables in Section 4. Section 5 explains the notion of transformation of tables. Sections 6, 7 and 8 describe in turn three types of transformation: changing the dimension of a table, inverting a normal table, and normalising an inverted table. Section 9 investigates the interrelationship between these three types of transformation. Section 10 deals with the issue of decidability (or computability) of the semantic functions on tables defined in Sections 3 and 4. Section 11 contains some suggestions for further research.

A talk based on an early version of this paper was presented at a meeting of SERG

(the Software Engineering Research Group of the Communications Research Laboratory of MacMaster University, led by David Parnas). I am grateful to Dr Parnas for suggesting this investigation and for many discussions on this topic, and to members of SERG and others for their helpful comments, particularly T Jiang, J Madey and his students, A Malton, D Peters, H Shen, J V Tucker and an anonymous referee.

The algorithms described here have been implemented by H Shen ([She95]).

1 Many-sorted signatures and algebras

1.1 Many-sorted signatures

A *(many-sorted) signature* has the form $\Sigma = (\boldsymbol{Sort}(\Sigma),\ \boldsymbol{Func}(\Sigma))$ where $\boldsymbol{Sort}(\Sigma)$ is a finite set of *sorts of* Σ, and $\boldsymbol{Func}(\Sigma)$ is a finite set of *function symbols of* Σ, each of a fixed *type* $s_1 \times \cdots \times s_m \to s$, with *arity* $m \geq 0$, *domain sorts* $s_1, \ldots, s_m$ and *range sort* s in $\boldsymbol{Sort}(\Sigma)$. (For $m = 0$, this gives a *constant* of sort s.)

We present three examples of signatures, each of which incorporates the preceding ones.

Example 1 (*Booleans*). The signature $\Sigma_{\mathbb{B}}$ of booleans is defined as follows: $\boldsymbol{Sort}(\Sigma_{\mathbb{B}}) = \{\mathsf{bool}\}$, and $\boldsymbol{Func}(\Sigma_{\mathbb{B}})$ consists of

$$\begin{aligned} \wedge,\ \vee :\ & \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool} \\ \neg :\ & \mathsf{bool} \to \mathsf{bool} \\ \mathsf{true},\ \mathsf{false} :\ & \to \mathsf{bool} \end{aligned}$$

Example 2 (*Reals and booleans*). The signature $\Sigma_{\mathbb{R}}$ of reals and booleans is defined as follows: $\boldsymbol{Sort}(\Sigma_{\mathbb{R}}) = \{\mathsf{real},\ \mathsf{bool}\}$, and $\boldsymbol{Func}(\Sigma_{\mathbb{R}})$ consists of $\boldsymbol{Func}(\Sigma_{\mathbb{B}})$ (above), and further:

$$\begin{aligned} 0,\ 1 :\ & \to \mathsf{real} \\ +,\ -,\ \times,\ \div :\ & \mathsf{real} \times \mathsf{real} \to \mathsf{real} \\ \sqrt{\ } :\ & \mathsf{real} \to \mathsf{real} \\ =,\ < :\ & \mathsf{real} \times \mathsf{real} \to \mathsf{bool} \end{aligned}$$

Example 3 (*Stacks of reals*). The signature $\Sigma_{\mathbb{S}}$ of stacks of reals is defined as follows: $\boldsymbol{Sort}(\Sigma_{\mathbb{S}}) = \{\mathsf{real},\ \mathsf{stack},\ \mathsf{bool}\}$, and $\boldsymbol{Func}(\Sigma_{\mathbb{S}})$ consists of $\boldsymbol{Func}(\Sigma_{\mathbb{R}})$ (above), and further:

$$\begin{aligned} \mathsf{emptystack} :\ & \to \mathsf{stack} \\ \mathsf{isempty} :\ & \mathsf{stack} \to \mathsf{bool} \\ \mathsf{push} :\ & \mathsf{real} \times \mathsf{stack} \to \mathsf{stack} \\ \mathsf{pop} :\ & \mathsf{stack} \to \mathsf{stack} \\ \mathsf{top} :\ & \mathsf{stack} \to \mathsf{real} \\ =_{\mathsf{stk}} :\ & \mathsf{stack} \times \mathsf{stack} \to \mathsf{bool} \end{aligned}$$

The signature $\Sigma_{\mathbb{R}}$ of Example 2 (or something like it) is the one most widely used in the practical work listed in the previous Section, as well as in the examples throughout the present paper. The signature $\Sigma_{\mathbb{S}}$ of stacks of reals is used in this Section only, to illustrate the notions of many-sorted signature and algebra.

1.2 Many-sorted algebras

Let Σ be a many-sorted signature. A Σ-algebra A has, for each sort s of Σ, a (non-empty) *carrier set* A_s, and for each function symbol F of Σ of type $s_1 \times \cdots \times s_m \to s$, a function $F^A : A_{s_1} \times \cdots \times A_{s_m} \to A_s$, the *interpretation* or *meaning* of F in A.

Example: Algebra of stacks of reals. For the signature $\Sigma_{\mathbb{S}}$ given above, the $\Sigma_{\mathbb{S}}$-algebra A has the three carriers $\mathbb{R}$ (of reals), $\mathbb{B}$ (of truth values $\mathtt{tt}$ and $\mathtt{ff}$), and $\mathbb{S}$ (of stacks of reals). Further, it has 0^A and $\mathsf{emptystack}^A$ as distinguished elements of $\mathbb{R}$ and $\mathbb{S}$ respectively, the operations of *addition*, *subtraction*, *multiplication* and *division* and *square root extraction* on the reals

$$\begin{aligned} +^A,\ -^A,\ \times^A,\ \div^A &:\ \mathbb{R}^2 \to \mathbb{R}, \\ \sqrt{}^A &:\ \mathbb{R} \to \mathbb{R}, \end{aligned}$$

the *order* and *equality* relations on the reals

$$<^A,\ =^A\ :\ \mathbb{R}^2 \to \mathbb{B},$$

the stack operations

$$\begin{aligned} \mathsf{push}^A &:\ \mathbb{R} \times \mathbb{S} \to \mathbb{S} \\ \mathsf{pop}^A &:\ \mathbb{S} \to \mathbb{S} \\ \mathsf{top}^A &:\ \mathbb{S} \to \mathbb{R}, \end{aligned}$$

the test for *emptiness*

$$\mathsf{isempty}^A\ :\ \mathbb{S} \to \mathbb{B},$$

and so on for the other function symbols of $\Sigma_{\mathbb{S}}$.

We will sometimes omit the superscript 'A', using the same notation for the symbol as for its interpretation, when this will not lead to confusion.

Note that in this paper we will work with total functions. In this example, therefore, we must assume that the functions are made total by default values, defining, *e.g.*, $x \div 0 = 0$ for real x, $\sqrt{x} = 0$ for negative x, $\mathsf{top}(\mathsf{emptystack}) = 0$ and $\mathsf{pop}(\mathsf{emptystack}) = \mathsf{emptystack}$.

A more refined treatment would involve *error values* instead of default values, which could be handled by some form of *three valued logic*, as in [TuZ88], or by the two-valued logic of [Par93]. We should note that the work in this paper on transformations would adapt easily to such a framework. For the sake of simplicity, we wish to postpone such issues to a later paper.

A good introduction to the theory of many-sorted algebras is given in [MeT92].

Now let Σ be an (arbitrary but fixed) signature, and A a Σ-algebra. We are interested in defining functions on A by means of tables over Σ.

All the syntax below will be defined relative to Σ, although this will not always be stated explicitly. (Thus by "sort", *e.g.*, we will mean sort of Σ.) Also all the semantics will be defined relative to a (fixed but unspecified) Σ-algebra A.

2 Expressions: Terms and conditions

We assume we are working in a language $\boldsymbol{Lang}(\Sigma)$ over Σ, which produces *expressions* over Σ or *Σ-expressions.* These include the classes

- $\boldsymbol{Term}(\Sigma)$ of Σ-terms $t, \ldots,$
- $\boldsymbol{Cond}(\Sigma)$ of Σ-conditions $C, \ldots,$
- $\boldsymbol{Tab}(\Sigma)$ of Σ-tables $T, \ldots.$

Terms and *conditions* are used in the construction of tables. *Tables* are used to define functions, "tabular functions".

We will study *terms* and *conditions* (syntax and semantics) in this section, and *tables* of two kinds (normal and inverted) in the following two sections.

2.1 Defined functions over Σ

Before giving the definitions of $\boldsymbol{Term}(\Sigma)$, we must introduce the concept of *defined function symbol.* We may assume that we have, in addition to the set $\boldsymbol{Func} = \boldsymbol{Func}\,(\Sigma)$ of primitive function symbols of Σ, also a set $\boldsymbol{DeFunc} = \boldsymbol{DeFunc}(\Sigma)$ of *defined function symbols over Σ.* These are symbols for *functions defined over Σ* in some formalism. Typically, these will be *tabular functions*, *i.e.*, functions defined by tables, as described in this paper below. However, they may include functions defined from those in $\boldsymbol{Func}$ by some other formal method, such as *definitional schemes*, including, *e.g.*, *explicit definition*, *definition by cases* and *primitive recursion.*

These defined function symbols are incorporated in the construction of terms and conditions (see §§2.2, 2.3), which are, in turn, incorporated in the construction of tables (Sections 3 and 4).

Note that the theory of table transformations described in this paper does not depend at all on the definition of $\boldsymbol{DeFunc}$. (It could even be empty.) If, however, we want the tabular functions to be computable on A, then we must at least assume that all the functions in $\boldsymbol{DeFunc}$ are computable on A. (See also §2.3, Remark 2.)

2.2 Syntax of terms

$\boldsymbol{Term} = \boldsymbol{Term}(\Sigma)$ is the class of terms over Σ, and for each Σ-sort s, $\boldsymbol{Term}_s$ is the class of terms of of sort s. These are generated in the well know way, as follows. Assume given a countable set of *variables* $x^s, y^s, \ldots$ for each sort s of Σ. Then:

(a) a constant of Σ of sort s is in $\boldsymbol{Term}_s$, (*b*) a variable of sort s is in $\boldsymbol{Term}_s$,

(*c*) if $F \in \boldsymbol{Func} \cup \boldsymbol{DeFunc}$ of type $s_1 \times \cdots \times s_m \to s$ $(m > 0)$ and $t_i \in \boldsymbol{Term}_{s_i}$ for $i = 1, \ldots, m$ then $F(t_1, \ldots, t_m) \in \boldsymbol{Term}_s$.

We write $t : s$ for $t \in \boldsymbol{Term}_s$.

2.3 Syntax of conditions

The syntax of the class $\boldsymbol{Cond} = \boldsymbol{Cond}(\Sigma)$ of conditions over Σ does not have to be specified precisely.

Assuming that bool is one of the sorts of Σ (which is always the case in our examples), $\boldsymbol{Cond}$ could, on the one hand, be taken simply as the set of *booleans* over Σ, *i.e.*, the set $\boldsymbol{Term}_{\mathsf{bool}}$.

On the other hand, the definition of $\boldsymbol{Cond}$ could be extended to include other constructions, such as bounded quantification over integers (assuming Σ includes a sort of integers).

In fact, in order for the theory of transformations described in this paper to hold, the only assumption we need make on $\boldsymbol{Cond}$ is the following.

CLOSURE ASSUMPTION. $\boldsymbol{Cond}$ is closed under *conjunction* and *disjunction*, and contains true and false.

REMARKS. (1) Closure under negation is not necessary, although it may be helpful (see *e.g.* Section 7, Remark 3).

(2) Another desirable assumption on the class $\boldsymbol{Cond}$ is *computability*: $\boldsymbol{Cond}$ should be such that its semantics (see below) is *computable* on A, in order that the tabular functions be computable on A. In other words, there should be an algorithm which, given a condition C and a state σ for C, can decide whether $\sigma \models C$ (as explained below in §2.4) or not. However this assumption is relevant to the *computational aspects* of tables, rather than their *transformational aspects*. In this paper we concentrate on the transformational aspects. In a future work we intend to explore the computational aspects. (Computability on many sorted algebras is investigated in [TuZ88]).

(3) It should be emphasized that whether the syntax of $\boldsymbol{Cond}$ coincides with that of $\boldsymbol{Term}_{\mathsf{bool}}$ or not, their roles are quite different, as we will see below.

2.4 States; Semantics of terms and conditions

We need some preliminary notions.

DEFINITIONS. (1) For any expression (term, condition or table) E, $\boldsymbol{var}(E)$ is the set of free variables in E. Similarly, $\boldsymbol{var}(E_1, \ldots, E_n)$ is the set of free variables in the expressions $E_1, \ldots, E_n$.

(2) For any set V of variables, a *state over V (in A)* is a function σ with domain containing V, such that for all $x \in V$, if $x : s$ then $\sigma(x) \in A_s$.

(3) For any expression E, a *state over* E is a state over $\boldsymbol{var}(E)$.

Now a function symbol $F \in \boldsymbol{Func} \cup \boldsymbol{DeFunc}$ of type $s_1 \times \cdots \times s_m \to s$ has an *interpretation* on A:

$$F^A : \ A_{s_1} \times \cdots \times A_{s_m} \to A_s.$$

For the primitive function symbols, this is given by the definition of A (see §1.2 above). For the defined function symbols, such a semantics has to be provided. In particular, the interpretation of a function symbol defined by a table will be discussed below (in Sections 3 and 4).

Hence every term or condition E has a value $[\![E]\!]^A\sigma$ in A, relative to a state σ over E in A. If $E \in \boldsymbol{Term}_s$ then $[\![E]\!]^A\sigma \in A_s$, and if $E \in \boldsymbol{Cond}$ then $[\![E]\!]^A\sigma \in \mathbb{B}$. The definition of $[\![E]\!]^A$ is by induction on the complexity of E. We omit the inductive clauses here. (For more details on the semantics of terms, see, *e.g.*, [MeT92].)

We will usually drop the superscript 'A'. In our semantic considerations, we will also not usually refer explicitly to the algebra A. Thus 'state' will mean 'state in A', etc.

DEFINITION 4. If σ is a state over a condition C, then $\sigma \models C$ (σ *satisfies* C or C *holds at* σ) iff $[\![C]\!]\sigma = \mathtt{tt}$.

3 Normal tables

We will define the class $\boldsymbol{Tab}_N(\Sigma)$ of *normal (function) tables over* Σ.

3.1 Preliminary notions

(1) The set $\{1, \ldots, n\}$ of positive integers (for some $n \geq 1$) is called the *segment up to* n, denoted $\boldsymbol{seg}(n)$. Its *length* is n.

(2) An *indexed set* is a function F. The domain of F is an *index set* $I = \boldsymbol{ind}(F)$. An *index* of F is an element of I. The value of F at an index $i \in I$ is usually denoted F_i. We sometimes write a family F with index I as

$$F \ = \ \langle F_i \mid i \in I \rangle.$$

The *contents* of F is its range $\boldsymbol{cont}(F) = \{F_i \mid i \in I\}$. The elements of $\boldsymbol{cont}(F)$ are called the *entries* or (loosely) *elements* of F.

In this paper, all index sets and (hence) indexed sets will be *finite*. Also the entries of indexed sets will be *expressions*.

(3) A *variant* $F\langle i/e \rangle$ of an indexed set F, with entry e at index i, is the indexed set F' with the same index set as F, such that $F'_j = F_j$ for all indices $j \neq i$, and $F'_i = e$.

(4) A *tuple* is an indexed set for which the index set is a segment. Its *length* is the length of the segment. We write the tuple $\langle e_i \mid i \in \boldsymbol{seg}(n) \rangle$ as $(e_1, \ldots, e_n)$.

(5) Given a tuple of positive integers $(l_1, \ldots, l_n)$, the elements of the cartesian product $\boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_n)$ can be *enumerated* as a single segment of length $l_1 \cdot \ldots \cdot l_n$. In

other words, there is a bijection between $\boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_n)$ and $\boldsymbol{seg}(l_1 \cdot \ldots \cdot l_n)$. Let

$$\boldsymbol{enum}_{l_1,\ldots,l_n} : \ \boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_n) \ \rightarrow \ \boldsymbol{seg}(l_1 \cdot \ldots \cdot l_n)$$

be any such bijection. We could, *e.g.*, take

$$\boldsymbol{enum}_{l_1,\ldots,l_n}(k_1,\ldots,k_n) \ = \ \sum_{i=1}^{n-1}((k_i - 1) \cdot l_{i+1} \cdot \ldots \cdot l_n) + k_n.$$

We will generally use the notation

$$[k_1,\ldots,k_n] \ =_{df} \ \boldsymbol{enum}_{l_1,\ldots,l_n}(k_1,\ldots,k_n)$$

where the parameters $l_1,\ldots,l_n$ are understood from the context.

We use the notation '$\equiv$' for *syntactic identity.*

3.2 Syntax of normal tables

DEFINITION 1. A *grid* G is an indexed set such that for certain positive integers n and $l_1,\ldots,l_n$, the index set of G is the Cartesian product

$$\boldsymbol{ind}(G) \ = \ \boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_n),$$

and the entries of G are *expressions.* Further, with this notation:

- $\boldsymbol{dim}(G) = n$ is the *dimensionality* of G, and G is an n-*dimensional* grid;
- $\boldsymbol{len}_i(G) = l_i$ (for $1 \leq i \leq n$) is the *length* of G in its ith dimension;
- the *shape* of G is the tuple $\boldsymbol{shape}(G) = (l_1,\ldots,l_n)$;
- the *size* of G is the cardinality of its index set, *i.e.*,

$$\boldsymbol{size}(G) = \boldsymbol{card}(\boldsymbol{ind}(G)) = l_1 \cdot \ldots \cdot l_n;$$

- a *cell* of G is an *index* of G, *i.e.*, a tuple $\boldsymbol{i} = (i_1,\ldots,i_n) \in \boldsymbol{ind}(G)$;
- the *entry* in the cell $\boldsymbol{i}$ of G is (of course) $G_{\boldsymbol{i}}$;
- a k-*strip of* G (for $1 \leq k \leq n$) is a tuple of entries of G in dimension k of length $\boldsymbol{len}_k(G)$, *i.e.*, a tuple

$$(G_{\boldsymbol{i}\langle k/1\rangle},\ldots,G_{\boldsymbol{i}\langle k/l_k\rangle})$$

  for some index $\boldsymbol{i}$ of G, where $l_k = \boldsymbol{len}_k(G)$.
- the k-*slice of* G *at level* j (for $1 \leq k \leq n$ and $1 \leq j \leq l_k$) is the indexed set S where

$$\boldsymbol{ind}(S) \ = \ \{(i_1,\ldots,i_n) \in \boldsymbol{ind}(G) \mid i_k = j\}$$

and for all $\boldsymbol{i} \in \boldsymbol{ind}(S)$, $S_{\boldsymbol{i}} = G_{\boldsymbol{i}}$. (Think of the k-slices of G as $(n-1)$-dimensional subgrids of G, "orthogonal" to its k-strips.[2])

Note that a tuple is just a 1-dimensional grid.

DEFINITION 2. A *normal (function) table* $T \equiv (G, H^1, \ldots, H^n)$ $(n \geq 1)$ over Σ consists of a grid G whose entries are all Σ-terms of the same sort, and *coordinate headers* $H^1, \ldots, H^n$, where $n = \boldsymbol{dim}(G)$ and for $1 \leq i \leq n$, H^i is a tuple of Σ-*conditions* of length $\boldsymbol{len}_i(T)$. (H^k is the header of T in dimension k.) We also write

$\boldsymbol{ind}(T) = \boldsymbol{ind}(G)$,
$\boldsymbol{dim}(T) = \boldsymbol{dim}(G)$,
$\boldsymbol{len}_i(T) = \boldsymbol{len}_i(G)$ for $1 \leq i \leq \boldsymbol{dim}(T)$,
$\boldsymbol{shape}(T) = \boldsymbol{shape}(G)$,
$\boldsymbol{size}(T) = \boldsymbol{size}(G)$,
$\boldsymbol{cont}(T) = \boldsymbol{cont}(G)$,
a *k-strip of T* is a k-strip of G,
the *k-slice of T* at level j is the k-slice of G at level j, and
$T_{\boldsymbol{i}} \equiv G_{\boldsymbol{i}}$, the *entry* in the cell $\boldsymbol{i}$ of T (*i.e.*, cell $\boldsymbol{i}$ of G).

The j-th entry of header H^k is said to *correspond to* the k-slice of T at level j.

Note that for a 2-dimensional table T, *rows* are 1-slices (which are the same as 2-strips), and *columns* are 2-slices (which are the same as 1-strips); and H^1 and H^2 as the *row* and *column headers* respectively. Then, for an index (i,j) of T, $T_{(i,j)}$ (also written $T_{i,j}$) is the entry in *row* i and *column* j. (This is consistent with the well-known convention for matrices.)

[2] In [She95] a more general notion of "slice" is defined, in which an arbitrary number of the index components are held fixed.

3.3 Properness

We are looking for a condition on tables which will make their semantics unproblematical.

DEFINITION 1. Let C be a condition. A tuple $(C_1, \ldots, C_n)$ of conditions is called

(a) *disjoint relative to* C if every state over $\boldsymbol{var}(C, C_1, \ldots, C_n)$ which satisfies C, satisfies *at most one of* $C_1, \ldots, C_n$.

(b) *universal relative to* C if every state over $\boldsymbol{var}(C, C_1, \ldots, C_n)$ which satisfies C, satisfies *at least one of* $C_1, \ldots, C_n$.

(c) *proper relative to* C if it is both disjoint and universal relative to C.

Equivalently, $(C_1, \ldots, C_n)$ is proper relative to C iff every state over $\boldsymbol{var}(C, C_1, \ldots$ $\ldots, C_n)$ which satisfies C, satisfies *precisely one of* $C_1, \ldots, C_n$.

An important special case of the above concepts is given by the following

DEFINITION 2. A tuple $(C_1, \ldots, C_n)$ of conditions is called (respectively) *disjoint*, *universal* or *proper* if it is (respectively) disjoint, universal or proper relative to the condition true.

Equivalently, $(C_1, \ldots, C_n)$ is *proper* iff every state over $\boldsymbol{var}(C_1, \ldots, C_n)$ satisfies *precisely one of* $C_1, \ldots, C_n$.

Note that these concepts (disjointness, universality and properness) are all relative to the Σ-algebra A. For example, the tuple $(x < 0,\ x = 0,\ 0 < x)$ is not proper in all algebras (of the appropriate signature), but only in those algebras in which the interpretation of '$<$' satisfies the trichotomy law. On the other hand, the tuple $(x < 0,\ x \not< 0)$ is proper in all algebras (of the appropriate signature).

Let T be a normal table.

DEFINITION 3. T is *proper* if all its headers are proper.

REMARK. In a more refined semantic analysis, we would require for properness of T, not only the condition stated in Definition 3, but also (recursively) the properness of all tables used as tabular function symbols in the construction of T. The above definition is sufficicient for our present purposes.

Table 1 is an example of a proper normal table. Note that this table, and other examples of tables used in this paper, are based on the signature $\Sigma_{\mathbb{R}}$ of Section 1.

Note again that properness of a table is relative to the Σ-algebra A. We will usually not mention A explicitly.

$y = 10$	$y > 10$	$y < 10$

H^2

$x \geq 0$
$x < 0$

H^1

0	y^2	$-y^2$
x	$x + y$	$x - y$

G

TABLE 1. A proper normal table

3.4 Semantics of normal tables

Suppose T is a proper normal table. By definition, all the entries in the grid of T have the *same sort*, say s.

Suppose $T \equiv (G, H^1, \ldots, H^n)$ with shape $(l_1, \ldots, l_n)$, index set $I = \boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_n)$, grid $G \equiv \langle t_{\boldsymbol{i}} \mid \boldsymbol{i} \in I \rangle$, where the $t_{\boldsymbol{i}}$ are terms of sort s, and headers $H^j \equiv (C_1^j, \ldots, C_{l_j}^j)$, where the C_i^j are conditions.

For any state σ over T in A, the *meaning* $[\![T]\!]^A\sigma$ *of* T *relative to* σ (also written $[\![T]\!]\sigma$) is an element of A_s, given by the following *operational semantics*: For each header H^j, find the index i_j for which the entry $C_{i_j}^j$ holds at σ. Since T is proper, there is a unique such index for each header. These indices determine the cell $\boldsymbol{i} = (i_1, \ldots, i_n)$ of T, for which the entry $t_{\boldsymbol{i}}$ gives the required value $[\![T]\!]\sigma = [\![t_{\boldsymbol{i}}]\!]\sigma$.

Regarding the *semantics of improper tables*, the situation is more complicated.

(i) If any header of T is *not universal*, then for some state *no entry* in that header will hold.

(ii) If, on the other hand, any of the headers is *not disjoint*, then for some state *more than one entry* in that header will hold.

In such cases, we may still be able to assign an operational semantics to tables as deterministic functions, by stipulating either default or error values as output. (Alternatively, in case (ii), we might stipulate an *order* in which the the entries of the headers are to be read. Or it may be possible in case (ii) to assume that the values agree on overlapping conditions given by the non-disjoint headers.)

The transformations described in this paper adapt easily to the situation of case (i), *i.e.*, with tables where the headers are disjoint but not universal. They may or may not work in case (ii), depending on the semantics adopted for improper tables.

Nevertheless, an analysis even of case (i) require a careful treatment of *error semantics*, which we wish to postpone to a future paper. We will therefore not consider the semantics of improper tables further in this paper.

Suppose, then, that T is a proper normal table. We may, equivalently, give the semantics of T in the following way. We can think of T as representing a *term* incorporating (typically) a huge definition by cases at the top level, which can be written (in an informal but suggestive notation):

$$\boldsymbol{term}(T) \;\equiv\; \bigwedge_{i_1=1}^{l_1} \cdots \bigwedge_{i_n=1}^{l_n} \big(C^1_{i_1} \wedge \ldots \wedge C^n_{i_n} \;\rightarrow\; t_{i_1,\ldots,i_n}\big). \tag{1}$$

Note that properness of T is equivalent to the condition that for any state σ over T, there is precisely one index $(i_1, \ldots, i_n)$ for which $\sigma \models C^1_{i_1} \wedge \ldots \wedge C^n_{i_n}$. This determines the value of T at σ as

$$[\![T]\!]\sigma \;=\; [\![\boldsymbol{term}(T)]\!]\sigma \;=[\![t_{i_1,\ldots,i_n}]\!]\sigma.$$

Next we will see how a table, relative to a list of variables, can define a *function.* First we need some definitions.

DEFINITION 1. A list $\boldsymbol{x}$ of variables is said to *cover* T if it includes all of $\boldsymbol{var}(T)$, *i.e.*, if $\boldsymbol{var}(T) \subseteq \boldsymbol{x}$.

Now let $\boldsymbol{x} = x_1, \ldots, x_m$ be any list of variables which covers T, with $x_i : s_i$ for $i = 1, \ldots, m$. Then relative to $\boldsymbol{x}$, T *names* or *defines* a *tabular function symbol*

$$f_{T,\boldsymbol{x}} : \; s_1 \times \cdots \times s_m \rightarrow s$$

with interpretation on A

$$f^A_{T,\boldsymbol{x}} : \; A_{s_1} \times \cdots \times A_{s_m} \rightarrow A_s,$$

as follows. For all $a_1 \in A_{s_1}, \ldots, a_m \in A_{s_m}$, let σ be the state over T defined by $\sigma(x_i) = a_i$ for $i = 1, \ldots, m$. Then

$$f^A_{T,\boldsymbol{x}}(a_1, \ldots, a_m) \;=\; [\![T]\!]\sigma.$$

We may also write, suggestively,

$$f_{T,\boldsymbol{x}} \;\equiv\; \lambda \boldsymbol{x} \cdot T.$$

Typically, the tabular function symbol $f_{T,\boldsymbol{x}}$ is included in $\boldsymbol{DeFunc}(\Sigma)$, to be used in the construction of terms appearing in other tables.

REMARKS. (1) The free variables of T are *not* free in $f_{T,\boldsymbol{x}}$. (They are bound by 'λ' in the notation '$\lambda \boldsymbol{x} \cdot T$'.) They therefore do not contribute to the free variables of any table containing T.

(2) Note that changing the covering variable list will, in general, completely alter the semantics of a given table. For example, if $\boldsymbol{x}'$ is a permutation of $\boldsymbol{x}$, then the function $f^A_{T,\boldsymbol{x}'}$ will generally be quite different from the function $f^A_{T,\boldsymbol{x}}$.

Definition 2 (Semantic equivalence of tables). Let T_1 and T_2 be two proper normal tables. $T_1 \approx_A T_2$ (T_1 and T_2 are *semantically equivalent on* A) iff for all states σ over $\boldsymbol{var}(T_1, T_2)$ in A, $[\![T_1]\!]^A\sigma = [\![T_2]\!]^A\sigma$.

Remarks. (3) Semantic equivalence is defined here only as a relation between *proper tables.*

(4) Two tables may be semantically equivalent without having the same set of free variables. For example, let T be any proper normal table, take a variable $y \notin \boldsymbol{var}(T)$, and let T' be formed from T by replacing all the conditions C in all the headers of T by $C \wedge (y = y)$. Then T' is a proper normal table and $T' \approx_A T$, but $\boldsymbol{var}(T') = \boldsymbol{var}(T) \cup \{y\}$.

(5) Note that $T_1 \approx_A T_2$ iff for any list $\boldsymbol{x}$ of variables covering T_1 and T_2,

$$f^A_{T_1,\boldsymbol{x}} \;=\; f^A_{T_2,\boldsymbol{x}}\,.$$

(6) The important questions of the decidability (or computability) of *properness* of tables, and of *semantic eqivalence* between proper tables, are treated in Section 10.

(7) Although semantic equivalence depends on the algebra A, we will usually drop the subscript 'A', and leave the dependence on A implicit.

4 Inverted tables

In this section we consider the class $\boldsymbol{Tab}_I(\Sigma)$ of *inverted (function) tables over* Σ. Such a table T differs from a normal table in the following way:

(1) One of its coordinate headers, say H^k, is called the *value header*. It contains *terms*, all of the same sort, instead of conditions. The other headers are called *normal*, and contain conditions as before.

(2) The cells of T contain *conditions* instead of terms.

The dimension k of the value header is called the *value dimension*. The dimensions of the normal headers are called *normal dimensions*.

The idea (or operational semantics) for T is as follows. For a given state σ over T, search along the normal headers

$$H^1, \ldots, H^{k-1}, H^{k+1}, \ldots, H^n$$

until you find the index in each header for which the entry holds at σ. These indices determine a k-strip. Search along this strip for the cell whose entry has the value $\mathtt{tt}$. The corresponding entry in H^k then gives the value of the function.

The desirability of this search always producing a unique value, leads to the following definitions. Let T be an inverted table.

Definition 1. T is *proper* if

(i) all its normal headers are proper, and

(ii) every k-strip in T (where k is the value dimension) is proper *relative to the conjunction of the entries in the normal headers corresponding to that strip.*

Table 2 is an example of a proper inverted table. Here the value dimension is 1, so that the value header is the row header H^1, and the tuples of cells in this dimension are the columns of the table. Notice that the columns are *not* proper, but *are* proper relative to the corresponding entries in the column header H^2.

$y \geq 0$	$y < 0$

H^2

$x + y$
$x - y$
$y - x$

H^1

$x < 0$	$x < y$
$0 \leq x < y$	$y \leq x < 0$
$x \geq y$	$x \geq 0$

G

Table 2. A proper inverted table

Suppose $T \equiv (G, H^1, \ldots, H^n)$, with shape $(l_1, \ldots, l_n)$ and value dimension k. So it has an index set $I = \boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_m)$, grid $G \equiv \langle C_{\boldsymbol{i}} \mid \boldsymbol{i} \in I \rangle$ with conditions $C_{\boldsymbol{i}}$, value header $H^k \equiv (t_1, \ldots, t_{l_k})$ with terms t_i, all of the same sort, and normal headers $H^j \equiv (C^j_1, \ldots, C^j_{l_j})$ $(j \neq k)$ with conditions C^j_i. Clause (ii) in Definition 1 says that for any index $\boldsymbol{i} = (i_1, \ldots, i_n)$, the k-strip $(C_{\boldsymbol{i}\langle k/1 \rangle}, \ldots, C_{\boldsymbol{i}\langle k/l_k \rangle})$ is proper relative to

$$C^1_{i_1} \wedge \ldots \wedge C^{k-1}_{i_{k-1}} \wedge C^{k+1}_{i_{k+1}} \wedge \ldots \wedge C^n_{i_n}.$$

Proposition 1. *(With T as above, and k the value dimension:) T is proper iff for any state σ over T, there is a unique index $(i_1, \ldots, i_n)$ of T such that*

(i) $\sigma \models C^j_{i_j}$ for $j = 1, \ldots, n$, $j \neq k$; and

(ii) $\sigma \models C_{i_1, \ldots, i_n}$.

We get a stronger notion for inverted tables than properness, namely *strict properness*, if we strengthen clause (ii) in Definition 1:

DEFINITION 2. T is *strictly proper* if

(i) all its normal headers are proper, and

(ii) every k-strip in T (where k is the value dimension) is proper.

PROPOSITION 2. *(With T as above, k the value dimension, and i_j ranging over $\boldsymbol{seg}\,(l_j)$ for $j = 1, \ldots, n$:) T is strictly proper iff for any state σ over T,*

(i) for $j = 1, \ldots, n,\ j \neq k,$ there is a unique i_j such that $\sigma \models C^j_{i_j}$; and

(ii) for all $i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n,$ there is a unique i_k such that $\sigma \models C_{i_1,\ldots,i_k,\ldots,i_n}$.

The more fundamental notion for our purposes is *properness*, since it is that which permits a total, deterministic (functional) semantics for inverted tables. However strict properness also features in our results (see Theorem 1 in Section 6 below).

Assume now that T is proper. We may give its semantics equivalently in the following way. Assume also (for notational convenience) that the value dimension is 1. Then T represents the following term:

$$\boldsymbol{term}(T) \;\equiv\; \bigwedge_{i_2=1}^{l_2} \cdots \bigwedge_{i_n=1}^{l_n} \Big(C^2_{i_2} \wedge \ldots \wedge C^n_{i_n} \;\rightarrow\; \bigwedge_{i_1=1}^{l_1} (C_{i_1 i_2 \ldots i_n} \;\rightarrow\; t_{i_1})\Big).$$

which can be rewritten (more simply, but less perspicuously in terms of the operational semantics given above) as

$$\bigwedge_{i_1=1}^{l_1} \bigwedge_{i_2=1}^{l_2} \cdots \bigwedge_{i_n=1}^{l_n} (C_{i_1 i_2 \ldots i_n} \wedge C^2_{i_2} \wedge \ldots \wedge C^n_{i_n} \;\rightarrow\; t_{i_1}). \tag{2}$$

By Proposition 1, properness of T is equivalent to the condition that for any state σ over T, there is a unique index $(i_1, \ldots, i_n)$ of T for which

$$\sigma \models C_{i_1 i_2 \ldots i_n} \wedge C^2_{i_2} \wedge \ldots \wedge C^n_{i_n}.$$

This determines the value of T at σ as

$$[\![T]\!]\sigma \;=\; [\![\boldsymbol{term}(T)]\!]\sigma \;=[\![t_{i_1}]\!]\sigma.$$

Again, relative to any list of variables $\boldsymbol{x}$ which covers T, this defines a function

$$f_{T,\boldsymbol{x}} \;\equiv\; \lambda \boldsymbol{x} \cdot T.$$

Semantic equivalence is defined for proper inverted tables just as for proper normal tables (§3.4, Definition 2).

REMARK. In the formula for $\boldsymbol{term}$(T) above, we assumed that the value dimension was 1. Likewise, in the examples involving 2-dimensional inverted tables below, we will always assume that the value dimension is 1; in other words, that the value header is the row header H^1. This is only for convenience; it is not part of the definition of inverted table.

5 Transformations of tables

We are interested in transforming tables to other, semantically equivalent tables, which may be easier to work with. We will define transformations

$$\phi:\ \mathcal{C}\ \to\ \mathcal{C}'$$

of tables from one class $\mathcal{C}$ to another class $\mathcal{C}'$. These transformations will satisfy the following two properties:

(1) ϕ is *semantics preserving*, in the sense that (although in general not all tables in $\mathcal{C}$ are proper), if $T\in\mathcal{C}$ is proper, then so is $\phi(T)$, and $\phi(T)\approx T$.

(2) ϕ is *effective* or *computable*.

If $\phi(T)=T'$, then T' is the *transform* of T under ϕ.

Most of the rest of this paper is devoted to examples of such transformations: changing the dimensionality of a table (Section 6), inverting a normal table (Section 7), and normalising an inverted table (Section 8).

6 Changing the dimensionality of a table

Note first that any n-dimensional (normal or inverted) table can be trivially transformed to an $(n+1)$-dimensional table, by adding an $(n+1)$-th coordinate header with a single entry, 'true'.

More interestingly: given a normal n-dimensional table, we can transform it to an $(n-1)$-dimensional table, by "combining" two of the dimensions, *i.e.*, combining two of the headers into a single header, using the enumeration function of §3.1(5).

More precisely, consider a normal table $T=(G,H^1,\dots,H^n)$ $(n\geq 2)$ with shape $(l_1,\dots,l_n)$, grid entries $G_{i_1,\dots,i_n}$ and headers $H^j\equiv(C^j_1,\dots,C^j_{l_j})$. This is transformed to a table $\tilde{T}\equiv(\tilde{G},H^1,\dots,H^{n-2},\tilde{H}^{n-1})$ with shape $(l_1,\dots,\ \dots,l_{n-2},\tilde{l}_{n-1})$, where $\tilde{l}_{n-1}=l_{n-1}\cdot l_n$, new header $\tilde{H}^{n-1}\equiv(\tilde{C}^{n-1}_1,\dots,\tilde{C}^{n-1}_{\tilde{l}_{n-1}})$, where

$$\tilde{C}^{n-1}_{[i,j]}\ \equiv\ C^{n-1}_i\wedge C^n_j,$$

headers $H^1,\dots,H^{n-2}$ unchanged, and grid entries

$$\tilde{G}_{i_1,\dots,i_{n-2},[i_{n-1},i_n]}\ \equiv\ G_{i_1,\dots,i_{n-2},i_{n-1},i_n}.$$

Remark 1. In the above description we combined the dimensions $n-1$ and n. This was just for notational convenience. We could combine any two dimensions of the table.

$x \geq 0 \wedge y = 10$	0
$x \geq 0 \wedge y > 10$	y^2
$x \geq 0 \wedge y < 10$	$-y^2$
$x < 0 \wedge y = 10$	x
$x < 0 \wedge y > 10$	$x + y$
$x < 0 \wedge y < 10$	$x - y$
H^1	G

TABLE 3. 1-dimensional transform of Table 1

By iterating the above process, we can transform any normal table T to a 1-dimensional normal table of length ***size*** (T). This could be useful in the generation of program code from tables.

As an example of this, Table 1 can be transformed to the 1-dimensional Table 3.

Note next that the above transformation could also be applied to an *inverted* table of dimensionality > 2, by combining two of its *normal* dimensions. By iterating this process, we can transform any such inverted table down to 2 dimensions.

Finally, we can transform a 2-dimensional inverted table down to 1 dimension as follows. Consider (for example) a 2-dimensional inverted table, such as Table 4, with value header H_1. This can be transformed to the 1-dimensional inverted table shown as Table 5.

t_1	$(C_1^2 \wedge C_{11}) \vee (C_2^2 \wedge C_{12})$
t_2	$(C_1^2 \wedge C_{21}) \vee (C_2^2 \wedge C_{22})$
t_3	$(C_1^2 \wedge C_{31}) \vee (C_2^2 \wedge C_{32})$
H^1	G

TABLE 5. 1-dimensional transform of Table 4

REMARK 2. This transformation (from Table 4 to Table 5) will be considered again from a different perspective in Section 8 (see Remark there).

C^2_1	C^2_2

H^2

t_1
t_2
t_3

H^1

C_{11}	C_{12}
C_{21}	C_{22}
C_{31}	C_{32}

G

TABLE 4. 2-dimensional inverted table

For the transformations considered in this Section, the following holds.

THEOREM 1. *Suppose T is proper, and T' is any of the tables transformed as above. Then*

(a) T' is proper;

(b) $T' \approx T$.

We omit a proof. However, the truth of the theorem should be apparent from the example.

7 Inverting a normal table

We consider the problem of inverting a normal table, *i.e.*, transforming a normal table to a semantically equivalent inverted one. We present two methods (or algorithms) for this.

For the first method, we begin with a simple example, from which the general procedure should be clear. Consider the case of a 2-dimensional 3×3 normal table, such as Table 6. (The header entries C^i_j are conditions, and the grid entries t_{ij} are terms.)

This table can be "inverted along dimension 1" (say), to produce Table 7.

We now derive the general formula for inversion by Method 1. Consider a normal table $T \equiv (G, H^1, \ldots, H^n)$ with shape $(l_1, \ldots, l_n)$, grid entries $t_{i_1,\ldots,i_n}$ and headers $H^j \equiv (C^j_1, \ldots, C^j_{l_j})$. This is inverted along dimension 1 (say) to form a table $\tilde{T} \equiv (\tilde{G}, \tilde{H}^1, H^2, \ldots, H^n)$ with shape $(\tilde{l}_1, l_2, \ldots, l_n)$, where $\tilde{l}_1 = \boldsymbol{size}(T) = l_1 \cdot \ldots \cdot l_n$, value header $\tilde{H}^1 \equiv (\tilde{t}_1, \ldots \tilde{t}_{\tilde{l}_1})$, with $\tilde{t}_{[i_1,\ldots,i_n]} \equiv t_{i_1,\ldots,i_n}$, normal headers $H^2, \ldots, H^n$ unchanged from T, and grid entries $\tilde{C}_{j_1,j_2,\ldots,j_n}$, where

$$\tilde{C}_{[i_1,\ldots,i_n],j_2,\ldots,j_n} \equiv \begin{cases} C^1_{i_1} & \text{if } i_2 = j_2, \ldots, i_n = j_n \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

C^2_1	C^2_2	C^2_3

H^2

C^1_1
C^1_2
C^1_3

H^1

t_{11}	t_{12}	t_{13}
t_{21}	t_{22}	t_{23}
t_{31}	t_{32}	t_{33}

G

TABLE 6

In general, a normal table can be inverted along any dimension k to produce an inverted table with value header $\tilde{H}^k$, and the other headers unchanged. The practical value of this transformation is, however, unclear, since the new table is much bigger than the original. (The length of the value header in the new table has increased to the size of the original table!)

We now consider a second method for inversion, which is appropriate for a normal table T in which $\boldsymbol{card}(\boldsymbol{cont}(T))$ (the number of *distinct* terms in its grid) is *small.* Suppose, *e.g.*, the grid in Table 6 contains only 2 terms, say t_1 and t_2, as shown in Table 8. According to Method 2, we invert T along dimension 1 to produce Table 9.

The table produced by this method is much smaller than that produced by Method 1 (under the assumption that $\boldsymbol{card}(\boldsymbol{cont}(T))$ is small), since the length of the value header ($\tilde{H}^1$ in Figure 8) is equal to $\boldsymbol{card}(\boldsymbol{cont}(T))$, and the other headers remain unchanged. However the conditions in the grid of the inverted table can be quite large disjunctions. It may happen that these disjunctions can be simplified by inspection. Otherwise we can make a *tradeoff* between length of value header and complexity of these conditions, by "splitting disjunctions" as in Table 10. The value header $\tilde{H}^1$ must then be lengthened by *repeating* each entry in it $\boldsymbol{len}_1(T)$ times, this being an upper bound for the lengths of the disjunctions in the grid in Table 9.

REMARKS. (1) The inversion given by Table 10 is a variant of the one given by Table 9 (Method 2). It is however quite different from the one shown as Table 7 (Method 1). In that one, the length of the value header was equal to $\boldsymbol{size}(T)$. Here (Table 10) its length is equal to $\boldsymbol{len}_1(T) \times \boldsymbol{card}(\boldsymbol{cont}(T))$, which may still be considerably less than $\boldsymbol{size}(T)$ if $\boldsymbol{card}(\boldsymbol{cont}(T))$ is small.

(2) Splitting disjunctions may not always be useful in simplifying conditions, since a disjunction $C_1 \vee C_2$ can often be simplified to a formula simpler than either C_1 or C_2, *e.g.*, if C_1 and C_2 are respectively of the form $D \wedge E$ and $D \wedge \neg E$.

C_1^2	C_2^2	C_3^2

H^2

t_{11}
t_{21}
t_{31}
t_{12}
t_{22}
t_{32}
t_{13}
t_{23}
t_{33}

$\tilde{\mathsf{H}}^1$

C_1^1	false	false
C_2^1	false	false
C_3^1	false	false
false	C_1^1	false
false	C_2^1	false
false	C_3^1	false
false	false	C_1^1
false	false	C_2^1
false	false	C_3^1

$\tilde{\mathsf{G}}$

TABLE 7. Inversion of Table 6: Method 1

(3) Assuming T is proper, we could, alternatively, simplify the conditions in Table 9 by replacing '$C_1^1 \vee C_2^1 \vee C_3^1$' by 'true', '$C_1^1 \vee C_2^1$' by '$\neg C_3^1$', and '$C_1^1 \vee C_3^1$' by '$\neg C_2^1$'.[3]

(4) Further, we can simplify Table 10 by deleting rows with only 'false' entries.

The following theorem holds for all the inversion transformations (and variants) considered in this Section.

THEOREM 2. *Suppose T is a proper normal table, and $\tilde{T}$ is the inverted table obtained from T by Method 1 or Method 2. Then*

(a) $\tilde{T}$ is strictly proper;

(b) $\tilde{T} \approx T$.

[3] But see Remark 3 in Section 9.

H^1 \ H^2	C^2_1	C^2_2	C^2_3
C^1_1	t_1	t_1	t_2
C^1_2	t_2	t_1	t_2
C^1_3	t_1	t_2	t_2

G

TABLE 8. A special case of Table 6

$\tilde{\mathsf{H}}^1$ \ H^2	C^2_1	C^2_2	C^2_3
t_1	$C^1_1 \vee C^1_3$	$C^1_1 \vee C^1_2$	false
t_2	C^1_2	C^1_3	$C^1_1 \vee C^1_2 \vee C^1_3$

$\tilde{\mathsf{G}}$

TABLE 9. Inversion of Table 8: Method 2

PROOF: For Method 1, the theorem follows from a comparison of ***term***(T) (formula (1) in Section 3) and ***term***($\tilde{T}$), which is constructed by applying formula (2) in Section 4 to the table $\tilde{T}$ as defined above. For Method 2, we omit a formal proof; however the truth of the theorem in this case should be apparent (at least for 2-dimensional tables) from the examples given. $\square$

8 Normalising an inverted table

We now come to the converse problem, that of transforming an inverted table to a normal one. The method we present is simple, but not completely satisfactory, since the normal table it produces is 1-dimensional.

As a simple example, consider again the 2-dimensional 3×2 inverted table shown as Table 4, with value header H^1. This can be "normalised" to a 1-dimensional table, shown as Table 11.

C_1^2	C_2^2	C_3^2

H^2

t_1
t_1
t_1
t_2
t_2
t_2

$\tilde{\mathsf{H}}^1$

C_1^1	C_1^1	false
C_3^1	C_2^1	false
false	false	false
C_2^1	C_3^1	C_1^1
false	false	C_2^1
false	false	C_3^1

$\tilde{\mathsf{G}}$

Table 10. Inversion of Table 8: Variant of Method 2

$(C_1^2 \wedge C_{11}) \vee (C_2^2 \wedge C_{12})$
$(C_1^2 \wedge C_{21}) \vee (C_2^2 \wedge C_{22})$
$(C_1^2 \wedge C_{31}) \vee (C_2^2 \wedge C_{32})$

$\hat{\mathsf{H}}^1$

t_1
t_2
t_3

$\hat{\mathsf{G}}$

Table 11. Normalisation of Table 4

Remark. The (1-dimensional) normal Table 11 is essentially the same as the (1-dimensional) inverted Table 5, which was formed by another tranformation of the inverted Table 4, namely lowering its dimensionality to 1 (see Remark 2 in Section 6). These two 1-dimensional tables are transformed to each other simply by interchanging their row header and grid.

Note that the header $\hat{H}^1$ of the transformed Table 11 still has length $\boldsymbol{len}_1(T)$. However the conditions in this header are, as they stand, quite complicated. Again, it may happen that these disjunctions can be simplified by inspection. Otherwise we can again effect a tradeoff between complexity of conditions and header length by "splitting" disjunctions, as in Table 12. This, however, increases the header length to $\boldsymbol{size}(T)$ (in the worst case).

$C_1^2 \wedge C_{11}$
$C_2^2 \wedge C_{12}$
$C_1^2 \wedge C_{21}$
$C_2^2 \wedge C_{22}$
$C_1^2 \wedge C_{31}$
$C_2^2 \wedge C_{32}$

$\hat{\mathsf{H}}^1$

t_1
t_1
t_2
t_2
t_3
t_3

$\hat{\mathsf{G}}$

TABLE 12. Normalisation of Table 4: Variant of Table 11

The general formula for normalising as in Table 11 is as follows. Consider an inverted table $T \equiv (G, H^1, \ldots, H^n)$ with shape $(l_1, \ldots, l_n)$, grid entries $C_{i_1,\ldots,i_n}$, value header $H^1 \equiv (t_1, \ldots, t_{l_1})$, and normal headers $H^j \equiv (C_1^j, \ldots, C_{l_j}^j)$ $(j = 2, \ldots, n)$. This is transformed to the 1-dimensional normal table $\hat{T} \equiv (\hat{G}, \hat{H}^1)$ with shape (l_1), grid entries $t_1, \ldots, t_{l_1}$, and header $\hat{H}^1 \equiv (\hat{C}_1, \ldots, \hat{C}_{l_1})$, where for $i = 1, \ldots, l_1$

$$\hat{C}_i \;\equiv\; \bigvee_{i_2=1}^{l_2} \cdots \bigvee_{i_n=1}^{l_n} (C_{i,i_2,\ldots,i_n} \wedge C_{i_2}^2 \wedge \ldots \wedge C_{i_n}^n).$$

For this transformation, or the variant given by Table 12, we have

THEOREM 3. *Suppose T is a proper inverted table, and $\hat{T}$ is the normal table obtained from T as above. Then*

(a) $\hat{T}$ is proper;

(b) $\hat{T} \approx T$.

PROOF: This follows from a comparison of $\boldsymbol{term}$(T) (formula (2) in Section 4) and $\boldsymbol{term}(\hat{T})$, which is constructed by applying formula (1) of Section 3 to $\hat{T}$ as defined above. It should be clear that the same result holds for the variant transformation of Table 12. $\square$

Another (more complex) normalisation algorithm for inverted tables, which preserves dimensionality, was devised and investigated in [She95].

9 Interrelationship between transformations

We can find a connection between the three types of transformation considered in this paper (changing dimensionality, inverting and normalising), as follows. First we define:

Definition 1. An *elementary transformation* of a table is any one of the following operations: permuting components of a conjunction or disjunction (in a cell), distributing a conjunction over a disjunction, simplifying conditions '$C \wedge \mathsf{false}$' to 'false', and '$C \vee \mathsf{false}$' to 'C', repeating a slice by "splitting a disjunction" in the corresponding header entry, permuting 2 slices together with their corresponding header entries, and deleting a slice with 'false' in the corresponding header entry.

Note that in two dimensions the last three operations amount to repeating, permuting and deleting rows or columns.

Definition 2. *Elementary equivalence* of tables is the equivalence relation on $\boldsymbol{Tab}_N$ and $\boldsymbol{Tab}_I$ generated by the class of elementary transformations. In other words, two tables are *elementarily equivalent* if one can be obtained from the other by a sequence of (0 or more) elementary transformations or their inverses.

Remark 1. For *proper tables*, elementary transformations are easily seen to be semantics-preserving; hence elementary equivalence implies semantic equivalence (but not conversely).

Now, given a normal table T, an *inversion* (by either of the two methods given in Section 7) followed by a *normalisation* (by the method of Section 8) is *essentially* the same as the transformation of T to 1 dimension (by the method of Section 6); and conversely, starting with an inverted table T, normalisation followed by inversion is essentially the same as transformation of T to 1 dimension. This is made precise in the following theorem.

Theorem 4. *(a) If T is a normal table, then the result of an inversion of T (as in Section 7) followed by a normalisation (as in Section 8) is elementarily equivalent to the the 1-dimensional transform of T (as in Section 6).*

(b) If T is an inverted table, then the result of a normalisation of T (as in Section 8) followed by an inversion (as in Section 7) is elementarily equivalent to the the 1-dimensional transform of T (as in Section 6).

Proof: These can be proved (at least for inversions by Method 1) by examining the formulas for inverting and normalising tables given in Sections 7 and 8. $\square$

Remarks. (2) The second transformation in part (*b*) of Theorem 4 is the trivial transformation between 1-dimensional normal and inverted tables (see the Remark in Section 8).

(3) Theorem 4 holds for all the variants of the algorithms considered in the Sections indicated above, except for those in Remark 3 of Section 7 (which assumes properness of the table).

10 Decidability of properness and of semantic equivalence

10.1 Decidability of properness

The meaning of tables $[\![T]\!]^A\sigma$ has been defined only for proper tables T (normal or inverted). In fact, the motivation for defining properness of tables was to isolate that condition for which a total, deterministic table function could be defined.

It is therefore important to investigate whether it is possible, in principle, to test all tables (of a given kind) for properness. In other words, we must ask whether this properties (for normal or inverted tables) is *algorithmically decidable* or *computable*. The answer to this will depend on (1) the signature Σ, (2) the composition of $\boldsymbol{DeFunc}(\Sigma)$ (see §2.1), (3) the composition of $\boldsymbol{Cond}(\Sigma)$ (see §2.3), and (4) the Σ-algebra A.

It is clear from the definitions that the problem of the decidability of the *properness* of normal and inverted tables is equivalent to the problem of the decidability of the *properness* and *relative properness* of *tuples* of conditions. Properness of such a tuple $(C_1, \ldots, C_n)$ can be expressed in $\boldsymbol{Lang}(\Sigma)$ as

$$\forall x_1 \ldots \forall x_m \big(\bigvee_{1 \le i \le n} C_i \wedge \bigwedge_{1 \le i < j \le n} \neg(C_i \wedge C_j)\big), \tag{3a}$$

and properness relative to a condition C_0 as

$$\forall x_1 \ldots \forall x_m \big[C_0 \to \big(\bigvee_{1 \le i \le n} C_i \;\wedge \bigwedge_{1 \le i < j \le n} \neg(C_i \wedge C_j)\big)\big], \tag{3b}$$

where $x_1, \ldots, x_n$ are all the variables occurring in the C_i. The problem then is the decidability of the truth in A of arbitrary sentences of the form $(3a)$ or $(3b)$.

Here are two examples, with different answers.

Example 1. Let A be the ring $\mathbb{R}$ of reals, with signature Σ containing 0, 1, $+$, $\times$, $-$, $<$ and $=$ (as in §1.1, Example 2, but without $\div$ and $\sqrt{\ }$). Let $\boldsymbol{DeFunc}(\Sigma)$ contain symbols for tabular functions. Let $\boldsymbol{Cond}(\Sigma)$ consist of all *quantifier-free formulas* of $\boldsymbol{Lang}(\Sigma)$, that is, all booleans over Σ. Then the truth of sentences of the form (3) is decidable, by the quantifier-elimination theorem for real closed fields (see, *e.g.*, Chapter 4 of [KrK71]).

In fact, by the same argument, this decidability result holds if $\boldsymbol{Cond}(\Sigma)$ consists of *any set* of first-order formulas over Σ.

Example 2. Let A be the ring $\mathbb{Z}$ of integers, with signature Σ containing (at least) 0, 1, $+$, $\times$, $-$ and $=$. Let $\boldsymbol{Cond}(\Sigma)$ consist of (at least) all *quantifier-free formulas* of $\boldsymbol{Lang}(\Sigma)$. Then the truth of sentences of the form (3) is (in general) undecidable. This follows from the undecidability of the truth of universal (or existential) sentences over Σ (or, equivalently, the undecidability of the existence of solutions to Diophantine equations, proved by Davis, Matiyasevich, Putnam and Robinson; see, *e.g.*, [Mat93]).

10.2 Decidability of semantic equivalence

Another important problem is the decidability of the relation of semantic equivalence $T \approx T'$ for proper tables T and T' (normal or inverted). This is an example of a problem of *relative decidability*:

Is there an algorithm α which, given any two tables T and T' (of the requisite class), can decide whether $T \approx_A T'$ or not, assuming both T and T' are proper?

(We don't care what α does if either T or T' is not proper.) Again, this will depend on Σ, $\boldsymbol{DeFunc}(\Sigma)$ $\boldsymbol{Cond}(\Sigma)$, and A.

Here we can use the same examples as in the previous subsection.

Observe first that this problem (for normal or inverted tables) is solvable iff the truth in A of sentences of the form

$$\forall x_1 \ldots \forall x_m (C \to t_1 = t_2) \tag{4}$$

is decidable, where $C \in \boldsymbol{Cond}$ and $t_i \in \boldsymbol{Term}$. (Remember that $\boldsymbol{Cond}$ is closed under conjunction; see §2.3.)

Example 1. As in Example 1 of §10.1. Sentences of the form (4) can be decided, again by quantifier elimination. So in this case semantic equivalence is *decidable*.

Example 2. As in Example 2 of §10.1. The decidability problem for universal sentences over this signature can be *reduced* to the decidability problem for sentences of the form (4), since

$$\forall x_1 \ldots \forall x_m C \iff \forall x_1 \ldots \forall x_m (\neg C \to 0 = 1).$$

So in this case semantic equivalence is *undecidable*.

Remark. Of course, even in those cases where the above two problems are decidable, one should investigate the *complexity* of such a decision procedure.

11 Directions for further research

Here are some ideas for further investigation:

(1) Extend the methods and results of this paper to some of the other kinds of tables given in [Par92], or to the more general framework of [Jan95].

(2) There seems to be a mismatch in the transformations defined in this paper: *proper* inverted tables are transformed to *proper* normal tables, but *proper* normal tables are transformed to *strictly proper* inverted tables. How, or under what conditions, can we transform proper normal tables to (not necessarily strictly) proper inverted tables?

(3) Note that *elementary equivalence* between tables, as defined in Section 9, is a *refinement* of semantic equivalence, *i.e.*, any two elementarily equivalent tables are semantically equivalent (but not conversely). Furthermore it is a *decidable* relation. One

should make a *systematic study of elementary equivalence*, or some other refinement of semantic equivalence suitable for dealing with the problems posed in (5) and (6) below.

(4) Develop tests for *properness* of tables. Even in those cases where this is decidable (Section 10), there may not be a feasible algorithm. Therefore it may make more sense to develop tests for properness in *special cases*, or tests for useful *sufficient conditions* for properness, for which there are feasible algorithms.

(5) Similarly, develop tests for *semantic equivalence* of tables, or (where this is impossible or inefficient) tests for special cases, or for some suitable *refinement* of semantic equivalence such as elementary equivalence. In this connection, we may ask:

(6) Can we develop a notion of *canonical* or *semi-canonical* normal (or inverted) tables? That is, canonical with respect to elementary equivalence, or some other suitable refinement of semantic equivalence.

(7) By developing a suitable theory of error semantics, give a semantic theory for improper tables.

(8) The tabular formalism, as presented in this paper, provides (only) a convenient method for *explicit definition* of functions from other functions (primitive, or previously defined). This situation changes radically if we permit *recursive tables*, *i.e.*, tables T with entries possibly containing a symbol $f_{T,\boldsymbol{x}}$ for the function defined by T itself. Such tables provide an interesting model of computation, and should be studied.

(9) Transformation algorithms provide an interesting field of study, from the viewpoint of (*i*) complexity, and (*ii*) deciding which transformations will lead to simpler tables – "simpler" in some sense yet to be made precise. (Note that size is not the only criterion of simplicity – any table is trivially equivalent to a table of one cell only, with a very complex entry!) A start to such an analysis of transformation algorithms is given in [She95].

References

[Hen80] K.L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Transactions on Software Engineering* SE-6 (1980), 2–13.

[HKP78] K.L. Heninger, J. Kallander, D.L. Parnas & J.E. Shore, "Software requirements for the A-7E Aircraft," United States Naval Research Laboratory, Washington DC, NRL Memorandun Report 3876, 1978.

[PAM91] D.L. Parnas, G.J.K. Asmis & J. Madey, "Assessment of safety-critical software in nuclear power plants," *Nuclear Safety* 32 (1991), 189–198.

[Par94] D.L. Parnas, "Inspection of safety-critical software using program-function tables," in *Proceedings of the IFIP World Congress, August 1994, Volume III*, 1994, 270–277.

[PMI92] D.L. Parnas, J. Madey & M. Iglewski, "Formal documentation of well-structured programs," Communications Research Laboratory, McMaster University, CRL Report 259, 1992.

[WiT95] A.J. Wilder & J.V. Tucker, "System Documentation Using Tables — A Short Course," Communications Research Laboratory, McMaster University, CRL Report 306, 1995.

[JPZ96] R. Janicki, D.L. Parnas & J.I. Zucker, "Tabular representations in relational documents," in *Relational Methods in Computer Science*, C. Brink & G. Schmidt, eds., in cooperation with R. Albrecht. Springer-Verlag, 1996, To appear.

[Par92] D.L. Parnas, "Tabular representation of relations," Communications Research Laboratory, McMaster University, CRL Report 260, 1992.

[She95] H. Shen, "Implementation of Table Inversion Algorithms," Department of Electrical and Computer Engineering, McMaster University, M.Eng. Thesis, 1995.

[TuZ88] J.V. Tucker & J.I. Zucker, *Program Correctness over Abstract Data Types, with Error-State Semantics*, CWI Monographs #6, North-Holland, 1988.

[Par93] D.L. Parnas, "Predicate logic for software engineering," *IEEE Transactions on Software Engineering* 19 (1993), 856–862 .

[MeT92] K. Meinke & J.V. Tucker, "Universal algebra," in *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay & T. Maibaum, eds., Oxford University Press, Vol. 1, 1992, 189–411.

[KrK71] G. Kreisel & J.L. Krivine, *Elements of Mathematical Logic*, North-Holland, 1971.

[Mat93] Y.V. Matiyasevich, *Hilbert's Tenth Problem*, The MIT Press, 1993.

[Jan95] R. Janicki, "Towards a formal semantics of Parnas tables," in *Proceedings of the 17th International Conference on Software Engineering, Seattle, April 1995*, ACM/IEEE, 1995, 231–240.