Table Transformations:
The Theory behind the Tools

J.I. Zucker

Department of Computing and Software,
McMaster University, Hamilton, Ontario L8S 4L7, Canada
zucker@mcmaster.ca

H. Shen[1]

Nortel Networks
Ottawa, Ontario, Canada
hongshen@nortelnetworks.com

Abstract

We work in a theory of function tables, similar to, and inspired by, that given in the work of D. Parnas and his colleagues. Table transformation algorithms transform one kind of table into another, preserving the semantics. We consider, in particular, two kinds of function tables: normal and inverted. We study effective transformations between tables of these two kinds, as well as transformations which change the dimension of a table. We also consider the interrelationship between these three types of transformation.

Most of these algorithms have been implemented as part of the Table Tool System developed by the Software Engineering Research Group at McMaster University. Some of the issues related to implementation are discussed.

Research supported by funding by the Telecommunications Research Institute of Ontario, and by a research grant from the Natural Sciences and Engineering Research Council of Canada.

[1]Work done while employed at the Software Engineering Research Group at McMaster University

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

1 Introduction

1.1 Tabular notation in documenting software systems

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 [HKPS78, Hen80]. 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].

The tabular notation is 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].

A natural question that arises is: Given a function, what is the best kind of table to represent it? It has been found that in practice one cannot always predict this. However a better question is: What is the best kind of table to represent a given function *for a specific purpose?* It turns out that the different kinds of table are best for different purposes, namely for developing the specifications, or for testing them, or for using the table.

Often however, when we construct a table, we may not know at the outset which table form is better, i.e. more comprehensible and perspicuous, with regard to the subject matter. Further, as the table is being developed, it may turn out to be larger or more unwieldy than expected. We will then want to know what the table would look like in other forms, which may be more suitable for the problem at hand.

Another point is this: the choice of table depends, in part, on the exact function or relation that one is trying to represent. We may start to construct a table before we know the function, in a form which is useful in the information gathering phase of the documentation. After we have that information, however, we might see that another form is more suitable for the given data. For example, there may be only a small number of expressions appearing repeatedly throughout the main grid, for which a second form would be better.

The following is an example of this situation from the A-7E project (personal communication, D. Parnas). This demonstrates the significance of table transformations, and the importance of mechanical implementations of these.

The software requirements document for this aircraft [AFB$^+$92] is one of many publications of the Software Cost Reduction (SCR) project conducted by the U.S. Naval Research Laboratory.

The HSI (Horizontal Situation Indicator) is an instrument that displays course, heading, distance and bearing information to the pilot. Our illustrative function (*cf.* Table 1.1) gives as output an external reference point which is used to measure the HSI range and bearing as a function of different modes and conditions. In the A-7E requirements specification document, a mode corresponds to a subset of the set of possible system states of the Operational Flight Program.

In constructing a table for this function, it was found that there were four possible outside reference points for the HSI: !*OAP*!, !*Fly-to-point*!, !*target*! and none. With six different modes, these reference points are determined by a set of four boolean conditions: $c1 : /FLYTOTW/ = 0$, $c2 : !Desig$, $c3 : !Before\ slewing!$ and $c4 : !Range! > 30$.

Just as in designing a flip-flop in digital circuits, we could describe the function in the form of a "normal table" by listing all the combinations of conditions. The result was the 16×6 normal table shown as Table 1.1.

After producing tables of this form, it was noticed that the same functions could also be described by "inverted tables", of size (in this case) 4×6 (the number of outside reference points times the number of modes), as in Table 1.2. This table, in a more compact form, tells the reader directly which reference point will be taken under a given mode, and so is more convenient for the user. (The concepts of normal and inverted table will be defined precisely later.)

In our example there were only 4 booleans and 6 modes, and so the original table was manageable. But in other cases, the normal table constructed in the form of Table 1.1 might have had over 20 booleans and 20 modes, with the result that it was huge, and had to be drawn on a number of sheets of paper spread out on a table. The difference between the normal and inverted table in these cases was quite dramatic.

When the A-7E requirements were originally written in 1977, the compact forms (such as Table 1.2) were prepared by hand from the original forms (such as Table 1.1) [AFB+92, p. 368]. Clearly, a tool for mechanical transformations of this kind would be preferable.

It therefore becomes important to be able to transform the format of a table from one kind to another. (Other examples of this nature are given in Remark 2 of Section 1.2. Furthermore, such a transformation should be effected mechanically, *i.e.*, implemented by a software tool. The Table Transformation Tool was developed at McMaster University for just this purpose [SERG97]

The present paper provides the theoretical framework to support such transformation tools.

Thanks are due to David Parnas for helpful comments on an earlier draft of this paper, and to some anonymous referees for help in improving the presentation.

1.2 Overview

In this subsection we illustrate the various table transformations with simple examples, so as to provide orientation. A formal development of these concepts will be given in the later sections.

We will consider two kinds of tables: normal and inverted. Tables 1.3 and 1.4 are examples of a normal table and inverted table respectively (not related to each other). As a rough orientation: the normal table 1.3 is read by finding the correct combination of conditions in the headers H^1 and H^2, and then looking up the expression in the corresponding cell of the grid G. The inverted table 1.4 is read by finding the correct condition in the condition header H^2, then finding the correct condition in the corresponding column of G, and finally looking up the term in the corresponding cell in the value header H^1.

H^2:

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

H^1 and G:

H^1	G		
$x < 0$	y^2	0	$-y^2$
$x \geq 0$	$x + y$	x	$x - y$

TABLE 1.3. A proper normal table

H^2:

$y < 0$	$y \geq 0$

H^1 and G:

H^1	G	
x	$x < y$	$x < 0$
$-y$	$y \leq x < 0$	$0 \leq x < y$
$x + y$	$x \geq 0$	$x \geq y$

TABLE 1.4. A proper inverted table

Both tables are *proper*, which is a requirement for a reasonable semantics. Exact definitions will be given later. Roughly, properness of Table 1.3 means properness of its headers H^1 and H^2, which means that the conditions they contain are non-overlapping and cover all cases. Properness of Table 1.4 means properness header H^2, as well as properness of each column of the main grid G *relative to the condition in the header of that column.*

We will sometimes need a condition stricter than properness for inverted tables, namely *strict properness*, which means (*e.g.*, with Table 1.5) that each column of the grid G is proper (independently of the condition in the header).

The transformations which we will be mainly concerned with will be transforming normal and inverted tables into the other kind, *i.e.*, *inverting* normalised tables and *normalising* inverted tables.

The inversion of Table 1.3 is shown as Table 1.6.

We consider two types of normalisation. The simpler, 1-D (one-dimensional) normalisation, transforms a proper inverted table to an equivalent 1-dimensional normal table.

	$y < 0$	$y \geq 0$
x	$x < y$	$x < y + 1$
$-y$	$y \leq x < y + 2$	$y + 1 \leq x < y + 3$
$x + y$	$x \geq y + 2$	$x \geq y + 3$

H^2 (column header), H^1 (row header), G (grid)

TABLE 1.5. A strictly proper inverted table (compare Table 1.4)

	$y < 10$	$y = 10$	$y > 10$
y^2	$x < 0$	false	false
$x + y$	$x \geq 0$	false	false
0	false	$x < 0$	false
x	false	$x \geq 0$	false
$-y^2$	false	false	$x < 0$
$x - y$	false	false	$x \geq 0$

H^2 (column header), H^1 (row header), G (grid)

TABLE 1.6. Inversion of Table 1.3

H^1	G
$(y < 0 \wedge x < y) \vee (y \geq 0 \wedge x < 0)$	x
$(y < 0 \wedge y \leq x < 0) \vee (y \geq 0 \wedge 0 \leq x < y)$	$-y$
$(y < 0 \wedge x \geq 0) \vee (y \geq 0 \wedge x \geq y)$	$x + y$

TABLE 1.7. 1-D normalisation of Table 1.4

For example, the 1-D (1-dimensional) normalisation of Table 1.4 is shown as Table 1.7.

Another type of normalisation, which preserves dimensionality, applies only to strictly proper inverted tables. For example, the m-D (multidimensional) normalisation of Table 1.5 is shown as Table 1.8.

Now the conditions in the header H^1 of Table 1.8 can be simplified, and for those of them which are found to reduce to 'false' (four, to be precise) , the corresponding rows can be deleted, to produce the equivalent, but simplified, Table 1.9.

	$y < 0$	$y \geq 0$
$(x < y) \wedge (x < y + 1)$	x	x
$(x < y) \wedge (y + 1 \leq x < y + 3)$	x	$-y$
$(x < y) \wedge (x \geq y + 3)$	x	$x + y$
$(y \leq x < y + 2) \wedge (x < y + 1)$	$-y$	x
$(y \leq x < y + 2) \wedge (y + 1 \leq x < y + 3)$	$-y$	$-y$
$(y \leq x < y + 2) \wedge (x \geq y + 3)$	$-y$	$x + y$
$(x \geq y + 2) \wedge (x < y + 1)$	$x + y$	x
$(x \geq y + 2) \wedge (y + 1 \leq x < y + 3)$	$x + y$	$-y$
$(x \geq y + 2) \wedge (x \geq y + 3)$	$x + y$	$x + y$

H^2 H^1 G

TABLE 1.8. m-D normalisation of Table 1.5

	$y < 0$	$y \geq 0$
$x < y$	x	x
$y \leq x < y + 1$	$-y$	x
$y + 1 \leq x < y + 2$	$-y$	$-y$
$y + 2 \leq x < y + 3$	$x + y$	$-y$
$x \geq y + 3$	$x + y$	$x + y$

H^2 H^1 G

TABLE 1.9. Simplification of Table 1.8

Remarks 1 (Comparison of the two normalisation algorithms). (*a*) Although the first step of this transformation (to Table 1.8) is algorithmic, and can be implemented, the second step (to Table 1.9) is not. Therefore m-D (unlike 1-D) normalisation can only be partially implemented (in general).

(*b*) The difference between the 1-D and the m-D normalisations can be summarised by saying that 1-D normalisation involves combining condition headers, while m-D normalisation involves refining the partitioning of the function domain.

We will also have occasion to consider algorithms which *flatten*, *i.e.*, lower the dimensionality of, a table. For example, the 1-D flattening of the normal Table 1.3 is the (unexciting) Table 1.10, and the 1-D flattening of the inverted Table 1.4 is essentially the same as its 1-D normalisation (Table 1.7), except that the header H^1 and grid G are interchanged.

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

H^1 G

TABLE 1.10. 1-D flattening of Table 1.3

Remark 2 (Applications of table transformations). The point of this investigation (as stated above) is that different kinds of tables are suitable for different purposes, and mechanical table transformations are essential for converting tables from one form to another, more useful, form. One real-life example of this (with the SRC project) has been described in the introduction. Other typical examples, involving on the kinds of tables introduced above, are the following:

(*a*) It may happen that an inverted table is more straightforward to construct than a normal table, because of the way the data presents itself. However, tables in normal form are often useful for inspection, *i.e.*, checking that no cases have been forgotten. This is a case where the normalisation algorithm could be useful.

(*b*) Flat (*i.e.*, 1-dimensional) tables are convenient for transforming tables to program code. The flattening algorithm mentioned above would therefore be useful as an intermediate step for generating program code from normal (or inverted!) tables.

Remark 3 Deterministic vs. nondeterministic tables. Nondeterministic tables may have a role, particularly in specifications (with "don't-care" cases, etc.). However our investigation applies only to *deterministic functions*, and tables defining such functions. Much of the theory would probably generalise satisfactorily to a nondeterministic setting, but we have not investigated this.

1.3 Background: Previous work by the present investigators

In [Zuc96] normal and inverted tables were considered with the formalism of the present paper. The transformations of inversion and 1-D normalisation between them were studied, as well as their interrelationships.

In [She95] the implementation of these transformations was described within the framework of the Table Tool System developed by the Software Engineering Research Group at McMaster University [SERG97]. In addition, the m-D normalising transformation considered above was defined, and its relationship with the inverting transformation studied, (but not implemented: see Remark 1(a) in §1.2).

The present paper combines and extends these two investigations.

1.4 Outline

Section 2 contains preliminary material on the formalism of tabular notations, based on many-sorted signatures, and the syntax and semantics of normal and inverted function tables.

The important semantic conditions of *properness* and *strict properness* are defined.

We will deal with three notions of *equivalence* between tables (exact definitions will be given later):

(1) *semantic equivalence*: the relation between two tables which define the same function;

(2) *elementary equivalence*: the relation between two tables where (roughly) one is obtained from the other by permuting, repeating and combining rows and columns;

(3) *elementary semantic equivalence*: this extends elementary equivalence by combining it with a special case of semantic equivalence, namely semantic equivalence between corresponding pairs of condition cells (*i.e.*, cells containing boolean conditions).

Of these, only elementary equivalence is (in general) a decidable relation. Also, elementary equivalence is (in general) a strict refinement of elementary semantic equivalence, which is in turn a strict refinement of semantic equivalence.

Transformations φ on tables should satisfy two properties: they should be *(properness and) semantics preserving*, and they should be *effective* or algorithmic. (In fact, as we have seen, the dimensionality-preserving normalising transformation described in Section 4 does not fully satisfy the effectivity requirement.)

In Section 3 various algorithms for tabular transformations are reviewed. the *inverting*, *normalising* and the *flattening* algorithms considered above.

In Section 4 the interrelationships between these transformations are studied. In particular, we have the following theorems:

(1a) inversion of a normal table followed by 1-D normalisation is elementarily equivalent to 1-D flattening;

(1b) conversely, 1-D normalisation of an inverted table followed by inversion is elementarily equivalent to 1-D flattening;

(2a) inversion of a normal table followed by m-D normalisation is elementarily semantically equivalent to the original table;

(2b) conversely, m-D normalisation of an inverted table followed by inversion is elementarily semantically equivalent to the original table;

(3a) inversion of a normal table commutes with slicing up to elementary equivalence;

(3b) m-D normalisation of an inverted table commutes with slicing up to elementary semantic equivalence.

Here *slicing* means constructing a "slice" of a table, *i.e.*, a subtable formed by holding certain dimensions fixed.

Proofs of these Theorems are omitted. They may be found in [ZS98].

Section 5 gives some information on the Table Transformation Tool, incorporating a number of table transformation algorithms, which was developed as part of the Table Tool System (TTS) at McMaster University [SERG97].

Section 6 contains ideas for future work.

2 Definitions and basic concepts

2.1 Many-sorted signatures and algebras

A *many-sorted signature* Σ consists of a finite set of *sorts* $s, \ldots$, and a finite set of *function symbols* of fixed type, say $F : s_1 \times \cdots \times s_m \to s$, with *arity* $m \geq 0$, *domain sorts* $s_1, \ldots, s_m$ and *range sort* s. (For $m = 0$, this gives a *constant* of sort s.)

As a typical example, consider the "real signature" $\Sigma_{\mathcal{R}}$, with two sorts: real and bool, and function symbols

$$\begin{aligned}
0,\ 1 &: \ \to \mathsf{real} \\
+,\ -,\ \cdot,\ \div &: \ \mathsf{real} \times \mathsf{real} \to \mathsf{real} \\
\sqrt{} &: \ \mathsf{real} \to \mathsf{real} \\
\wedge,\ \vee &: \ \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool} \\
\neg &: \ \mathsf{bool} \to \mathsf{bool} \\
\mathsf{true},\ \mathsf{false} &: \ \to \mathsf{bool}
\end{aligned}$$

This signature (or something like it) is the one most widely used in the practical work cited in this paper, as well as in the examples used throughout the paper.

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

For example, for the signature $\Sigma_{\mathcal{R}}$ given above, the $\Sigma_{\mathcal{R}}$-algebra $\mathcal{R}$, which we shall call the "standard real algebra", has the two carriers $\mathbb{R}$ (of reals) and $\mathbb{B}$ (of truth values tt and ff). Further, it has *zero* as a distinguished element of $\mathbb{R}$, the operations of *addition*, *subtraction*, *multiplication* and *division* and *square root extraction* on the reals.

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 or conventional values, defining, *e.g.*, $x \div 0 = 0$ for real x and $\sqrt{x} = 0$ for negative x.

Remark 1 (Total vs. partial functions; Improper tables; Error algebras). Use of default values as above can be dangerous, in that it may hide errors. A more refined treatment would involve *error values* instead of default values, brought about by *partial functions*, including functions "defined" by *improper tables*. This could be handled by some form of *three valued logic*, as in [TZ88], or by the two-valued logic of [Par93]. The work in this paper on transformations could presumably be adapted to such a framework. For the sake of simplicity, we wish to postpone such issues to a later paper. This topic is currently under investigation by the first author, in the framework of *error algebras*. For related work on this topic, we mention the references [BCJ84, KK94, Far95, GHM78] and the papers in the collection [Lam91]. (See also Remark on "quasi-properness" in §2.3.3.)

Remark 2. All the syntax below will be defined relative to Σ, which may be taken as the real signature $\Sigma_{\mathcal{R}}$ defined above. Also all the semantics will be defined relative to a Σ-algebra A, which may be taken as the corresponding standard real algebra $\mathcal{R}$, also defined above.

2.2 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* (as we will see) 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.

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

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

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

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

In clause (c), the symbol F may range over functions in the signature Σ, and also functions *defined* in some formalism over Σ, including, *e.g.*, *tabular functions* or functions defined by tables (as discussed below).

2.2.2 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.

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.

2.2.3 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, $\boldsymbol{var}(E_1, \ldots, E_n) = \bigcup_{i=1}^{n} \boldsymbol{var}(E_i)$.

(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 has sort 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 $F : s_1, \ldots, s_m \to s$ of Σ has an *interpretation* on A:

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

given by the definition of A. (For a defined function symbol, such a semantics has to be provided. In particular, the interpretation of a function symbol defined by a table will be discussed below.)

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 standard, by induction on the complexity of E. We omit the inductive clauses here.

We will often drop the superscript 'A', and not usually refer explicitly to the algebra A. Thus 'state' will mean 'state in A', etc. (As stated in Remark 2 in §2.1, we can take A to be the real algebra $\mathcal{R}$.) Finally:

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

2.3 Normal tables

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

2.3.1 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) 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* by 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

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

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.*

2.3.2 Syntax of normal tables

We will use Table 1.3 as a running example for the following definitions.

Definition 1. A *grid* G is an n-dimensional array (for some $n \geq 0$) with index set

$$I \ = \ \boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_n)$$

(for some n and some $l_1, \ldots, l_n$) where the cell entries are *expressions*. Thus, *e.g.*, Table 1.3 is a 2-dimensional grid. Further:

- $\boldsymbol{len}_i(G)$ is the *length* of G in its ith dimension (for $i = 1, \ldots, n$). (The length of Table 1.3 in dimensions 1 and 2 is 2 and 3 respectively.)
- The *shape* of G is the tuple $\boldsymbol{shape}(G) = (\boldsymbol{len}_1(G), \ldots, \boldsymbol{len}_n(G))$. (The shape of Table 1.3 is $(2, 3)$.)
- The *size* of G is the number of cells in G:

  $$\boldsymbol{size}(G) = \boldsymbol{len}_1(G) \times \cdots \times \boldsymbol{len}_n(G).$$

  (The size of Table 1.3 is 6.)
- The entry in the cell of G with index $\boldsymbol{i} = (i_1, \ldots, i_n)$ is written $G_{\boldsymbol{i}}$, for $\boldsymbol{i} \in I$. We also write the grid as

  $$G \ = \ \langle G_{\boldsymbol{i}} \mid \boldsymbol{i} \in I \rangle.$$

- A *k-strip of* G (for $1 \leq k \leq n$) is a tuple of entries of G in the k-th dimension of length $\boldsymbol{len}_k(G)$. (Thus in a 2-dimensional table, each row is a 2-strip and each column is a 1-strip.)
- A *cross-section*[2] of G is an $(n-1)$-dimensional subgrid, *i.e.*, a subgrid in which one of the dimensions is held fixed. (In a 2-dimensional table, each row or column is a cross-section as well as a strip. In a 3-dimensional table, each 2-dimensional subgrid is a cross-section, and so on.) Such a cross-section is said to be *orthogonal to* the fixed dimension, at a particular *level*. (In Table 1.3, the bottom row is a cross-section, orthogonal to dimension 1 at level 2, and the rightmost column is a cross-section, orthogonal to dimension 2 at level 3.)

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 the header of T in dimension i, namely a tuple of Σ*-conditions* of length $\boldsymbol{len}_i(G)$. The j-th entry of header H^k *corresponds to* or *labels* a particular cross-section of T at level j.

We write, *e.g.*, $T_{\boldsymbol{i}} \equiv G_{\boldsymbol{i}}$, the *entry* in the cell $\boldsymbol{i}$ of T (*i.e.*, cell $\boldsymbol{i}$ of G), and so on.

Note that for a 2-dimensional table T, and index (i,j), $T_{(i,j)}$ (also written $T_{i,j}$) is the entry in *row* i and *column* j. This is consistent with the well-known convention for matrices.

2.3.3 Properness. We want 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* $C_1, \ldots, C_n$.

[2] called a (-1)-*slice* in [ZS98]

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 (for example, $\mathcal{R}$). On the other hand, the tuple $(x < 0,\ x \not< 0)$ is proper in all algebras (of the appropriate signature).

Definition 3. A table T is *proper* if all its headers are proper.

Table 1.3 is an example of a proper normal table. Note again that its properness is relative to the Σ-algebra A — in this case, $\mathcal{R}$.

Remark (Quasi-properness). A looser definition of properness of normal tables, which we call "quasi-properness", is possible: we still require a *universality condition* for the headers, but replace *disjointness* by a weaker condition of *consistency*, *i.e.*, that the output agrees on "overlaps", so that the tables still define a deterministic function. Table 2.1 is a very simple example of a quasi-proper (1-dimensional) table.

$x \leq 0$
$x \geq 0$

H^1

0
x

G

TABLE 2.1. A quasi-proper table

We have not considered quasi-properness for a practical as well as a theoretical reason: practically, the consistency condition is generally harder to check than disjointness; and theoretically, the proofs of preservation of semantics under the transformations considered here seem to be much harder (although we believe the theorems still hold). However quasi-properness is an important concept which should not be ignored; we intend to investigate this in the future, in the context of an of error algebras (see Remark 1 in §2.1).

2.3.4 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 $H^j \equiv (C^j_1, \ldots, C^j_{l_j})$, where the C^j_i 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 $C^j_{i_j}$ 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 $[\![T]\!]\sigma = [\![t_{\boldsymbol{i}}]\!]\sigma$.

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):

$$\boldsymbol{term}(T) \;\equiv\; \bigwedge_{i_1=1}^{l_1} \cdots \bigwedge_{i_n=1}^{l_n} \big(C^1_{i_1} \wedge \ldots \wedge C^n_{i_n} \;\to\; t_{i_1,\ldots,i_n}\big). \tag{2.3.1}$$

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

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

The problems connected with the semantics of improper tables are indicated in [Zuc96]. (See also Remark 1 in §2.1.) This issue is deferred to a later publication.

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

Definition 1. A list $\boldsymbol{x}$ of variables is said to *cover* T 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 of type s_i for $i = 1, \ldots, m$. Then relative to $\boldsymbol{x}$, T *names* or *defines* a *tabular function symbol*

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

with interpretation on A

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

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 (using λ-notation):

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

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, $[\![T_1]\!]^A\sigma = [\![T_2]\!]^A\sigma$.

Note that semantic equivalence is defined here only as a relation between *proper tables.*

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

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

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

The questions of the *decidability* (or *computability*) of properness of tables, and of semantic eqivalence between proper tables, are discussed in [Zuc96]

2.4 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:

(i) One of the dimensions of T, say k, is called the *value dimension* or *principal dimension* of T. The corresponding headers H^k is called the *value header* or *principal header*. It contains *terms*, all of the same sort, instead of conditions. The other headers are called *condition headers*, and contain conditions as before.

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

In the rest of §*2.4, k will denote the principal dimension of the table.*

A k-strip of T is called a *principal strip* of T, and a cross-section of T orthogonal to the principal dimension k is called a *principal cross-section* of T.

The *operational semantics* for T is as follows. For a given state σ over T, search along the condition 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 condition headers are proper, and

(ii) every principal strip in T is proper *relative to the conjunction of the entries in the condition headers corresponding to that strip.*

Table 1.4 is an example of a proper inverted table. Here the value header is the row header H^1, so that the principal dimension is 1, and the 1-strips are the columns. Notice that these columns are *not* proper, but *are* proper relative to the corresponding entries in the column header H^2.

Suppose $T \equiv (G, H^1, \ldots, H^n)$, with shape $(l_1, \ldots, l_n)$ and principal dimension k. So it has an index set $I = \boldsymbol{seg}(l_1) \times \cdots \times \boldsymbol{seg}(l_m)$, grid $G \equiv \langle G_{\boldsymbol{i}} \mid \boldsymbol{i} \in I\rangle$ with conditions $G_{\boldsymbol{i}}$, value header $H^k \equiv (t_1, \ldots, t_{l_k})$ with terms t_i, all of the same sort, and condition 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 through cell $G_{\boldsymbol{i}}$ 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 principal 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) $\sigma \models G_{i_1,\ldots,i_n}$.

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 condition headers are proper, and

(ii) every principal strip in T is proper.

Table 1.5 is an example of a strictly proper inverted table. Notice that the columns of this table are proper (in contrast to Table 1.4).

Proposition 2. *(With T as above, k the principal 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 4.2(b)).

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

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

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

$$\bigwedge_{i_1=1}^{l_1} \bigwedge_{i_2=1}^{l_2} \cdots \bigwedge_{i_n=1}^{l_n} (G_{i_1 i_2 \ldots i_n} \wedge C^2_{i_2} \wedge \ldots \wedge C^n_{i_n} \ \rightarrow\ t_{i_1}). \tag{2.4.1}$$

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

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

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 (using λ-notation again) 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 (§2.3.4 , Definition 2).

Note that in the formula for $\boldsymbol{term}(T)$ above, we assumed that the principal dimension was 1. Likewise, in the examples involving 2-dimensional inverted tables below, we will always assume that the principal 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.

3 Algorithms for transforming tables

We are interested in transforming tables to other, semantically equivalent tables, for the reasons discussed in section 1. We will define transformations

$$\varphi : \mathcal{C} \rightarrow \mathcal{C}'$$

of tables from one class $\mathcal{C}$ to another class $\mathcal{C}'$. These transformations should 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 $\varphi(T)$, and $\varphi(T) \approx T$.

(2) φ is *effective* or *computable.*

If $\phi(T) = T'$, then T' is the *transform* of T under φ. In this section we will consider four algorithms for transforming tables: one for lowering the dimensionality of a table (§3.1), one for transforming a normal to an inverted table (§3.2), and two for transforming an inverted to a normal table: the "1-dimensional" and "multidimensional" algorithms (§3.3 and §3.4 resp.).

In connection with property (2) above, we should however point out that the multidimensional normalisation algorithm is not completely effective, as noted earlier (Remark 1(a) in §1.2)

3.1 Changing the dimensionality; Flattening 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 $\boldsymbol{enum}$ function of §2.3.1.

More precisely, consider a normal table $T = (G, H^1, \ldots, H^n)$ $(n \geq 2)$ with shape $(l_1, \ldots, l_n)$, grid entries $G_{i_1,\ldots,i_n}$ and headers $H^j \equiv (C_1^j, \ldots, C_{l_j}^j)$. This is transformed to a table $T' \equiv (G', H^1, \ldots, H^{n-2}, H'^{n-1})$ with shape $(l_1, \ldots, \ldots, l_{n-2},\ l'_{n-1})$, where $l'_{n-1} = l_{n-1} \cdot l_n$, a new header $H'^{n-1} \equiv (C_1'^{n-1}, \ldots, C_{l'_{n-1}}'^{n-1})$, where

$$C_{[i,j]}'^{n-1} \equiv C_i^{n-1} \wedge C_j^n, \tag{3.1.1}$$

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

$$G'_{i_1,\ldots,i_{n-2},[i_{n-1},i_n]} \equiv G_{i_1,\ldots,i_{n-2},i_{n-1},i_n}. \tag{3.1.2}$$

(Recall the bracket notation for ***enum*** in §2.3.1.) 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.

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

For example, the 2-dimensional normal Table 1.3 can be flattened to the 1-dimensional Table 1.10.

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

Finally, we can transform a 2-dimensional inverted table down to 1 dimension, called the *1-dimensional flattening* of the original table, as follows. Consider (for example) a 2-dimensional inverted table, such as Table 3.1, with value header H_1.

	C^2_1	C^2_2
	H^2	

H^1		G	
t_1		G_{11}	G_{12}
t_2		G_{21}	G_{22}
t_2		G_{31}	G_{32}

TABLE 3.1. 2-dimensional inverted table

Remark 1 (Notation for conditions). We will generally write 'C' (with super- and subscripts) for conditions in headers, and 'G' (with subscripts) for conditions in the grid.

Table 3.1 can be flattened to the 1-dimensional inverted table shown as Table 3.2.

H^1	G'
t_1	$(C^2_1 \wedge G_{11}) \vee (C^2_2 \wedge G_{12})$
t_2	$(C^2_1 \wedge G_{21}) \vee (C^2_2 \wedge G_{22})$
t_3	$(C^2_1 \wedge G_{31}) \vee (C^2_2 \wedge G_{32})$

TABLE 3.2. 1-dimensional flattening of Table 3.1 (Short version)

More generally, a 2-dimensional inverted table $T = (G, H^1, H^2)$ with shape (l, m), value header $H^1 = (t_1, \ldots, t_l)$, condition header $H^2 = (C_1^2, \ldots, C_m^2)$, and grid cells G_{ij} $(1 \le i \le l, 1 \le j \le m)$, can be flattened into a 1-dimensional inverted table $T' = (G', H^1)$ with value header H^1 as before, and grid cells $G'_i\,(1 \le i \le l)$ where $G'_i = \bigvee_{j=1}^{m}(C_j^2 \wedge C_{ij})$.

Note that the header H^1 of the transformed Table 3.2 still has length l. However the conditions in the grid are relatively complicated (large disjunctions). It may happen that these disjunctions can be simplified by inspection. Otherwise we can effect a tradeoff between complexity of conditions and header length by *"splitting disjunctions"*, as in Table 3.3. This increases the header length to $l \cdot m$. We call Tables 3.2 and 3.3 *short* and *long* flattenings of Table 3.1. (We will normally work with the short version, but our results are easily adaptable to the long version.)

H^1	G''
t_1	$C_1^2 \wedge G_{11}$
t_1	$C_2^2 \wedge G_{12}$
t_2	$C_1^2 \wedge G_{21}$
t_2	$C_2^2 \wedge G_{22}$
t_3	$C_1^2 \wedge G_{31}$
t_3	$C_1^2 \wedge G_{31}$

TABLE 3.3. 1-dimensional flattening of Table 3.1 (Long version)

Remarks. (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 $D \wedge \neg E$.

(3) This transformation (from Table 3.1 to Table 3.2) will be considered again from a different perspective below (see the Remark in §3.3).

(4) Conversely, we can view the relationship between Tables 3.3 and 3.2 as the result of the elementary transformation of *combining* principal cross-sections in Table 3.3 which have the same entries in the value header. (This will be considered more carefully in §3.2.)

For all the flattening transformations considered above, the following holds.

Theorem 3.1. *Suppose T is proper, and T' is formed by flattening T. Then*

(a) T' is proper;

(b) $T' \approx T$.

Proof: This follows from the construction of T'. In the case where T is normal, or inverted of dimensionality > 2, parts (a) and (b) follow from formulas (3.1.1) and (3.1.2) respectively. $\square$

3.2 Inverting a normal table

Inverting a normal table was illustrated in §2.1, with the inversion of Table 1.3 to Table 1.6. Consider, more generally, the case of a 2-dimensional 3×3 normal table, such as Table 3.4, which can be "inverted along dimension 1" (say), to produce Table 3.5.

C_1^2	C_2^2	C_3^2

H^2

C_1^1
C_2^1
C_3^1

H^1

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

G

TABLE 3.4. A normal table

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 3.5. Inversion of Table 3.4 (Standard version)

We call this transformation the *standard version* of the inversion algorithm, or the *standard inversion* (in contrast to the compact version considered below). To derive the general formula for a table produced by such an inversion, consider a normal table $T \equiv (G, H^1, \dots, H^n)$ with shape $(l_1, \dots, l_n)$, grid entries $t_{i_1,\dots,i_n}$ and headers $H^j \equiv$

$(C^j_1, \ldots, C^j_{l_j})$. This is inverted along dimension 1 (say) to form a table $\tilde{T} \equiv (\tilde{G}, \tilde{H}^1, H^2, \ldots$ $\ldots, H^n)$ with shape $(\tilde{l}_1, l_2, \ldots, l_n)$, where $\tilde{l}_1 = \textbf{\textit{size}}(T) = l_1 \cdot \ldots \cdot l_n$, value header $\tilde{H}^1 \equiv (\tilde{t}_1, \ldots \tilde{t}_{\tilde{l}_1})$, with $\tilde{t}_{[i_1,\ldots,i_n]} \equiv t_{i_1,\ldots,i_n}$, condition headers $H^2, \ldots, H^n$ unchanged from T, and grid entries $\tilde{G}_{j_1,j_2,\ldots,j_n}$, where

$$\tilde{G}_{[i_1,\ldots,i_n],j_2,\ldots,j_n} \equiv \begin{cases} C^1_{i_1} & \text{if } i_2 = j_2, \ldots, i_n = j_n \\ \mathsf{false} & \text{otherwise.} \end{cases} \tag{3.2.1}$$

Theorem 3.2. *Suppose T is a proper normal table, and $\tilde{T}$ is an inversion of T. Then*

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

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

Proof: Part (a), which amounts to strict properness of the principal strips of $\tilde{T}$, follows from formula (3.2.1) and the properness of T. Part (b) follows from a comparison of $\textbf{\textit{term}}(T)$ (formula (2.3.1)) and $\textbf{\textit{term}}(\tilde{T})$, which is constructed by applying formula (2.4.1) to the table $\tilde{T}$ as defined above. $\square$

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!)

It is, however, often possible to make the inverted table much more compact by performing the following *elementary transformation* on it. Suppose two or more entries of the value header are identical. Then we *combine* the associated principal cross-sections into one cross-section by forming pointwise disjunctions of corresponding cells. (An example is given by the transformation of Table 3.3 into Table 3.2; see Remark 4 in §3.1.)

As an example, suppose the grid in Table 3.4 contains only 2 distinct terms, say $t_{11} \equiv t_{12} \equiv t_{22} \equiv t_{31} \equiv t_1$, and $t_{21} \equiv t_{32} \equiv t_{13} \equiv t_{23} \equiv t_{33} \equiv t_2$, as shown in Table 3.6.

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

TABLE 3.6. A special case of Table 3.4

	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	$C^1_1 \vee C^1_2 \vee C^1_3$

H^2 $\tilde{\mathsf{H}}^1$ $\tilde{\mathsf{G}}$

TABLE 3.7. Inversion of Table 3.6 (Compact version)

Then some or all of the principal cross-sections in the inverted Table 3.5 corresponding to a single term can be combined, to produce (in the extreme case) the inverted Table 3.7. Here the grid cell entries have been constructed by forming disjunctions of corresponding cells in Table 3.5, and replacing '$C \vee \mathsf{false}$' and '$\mathsf{false} \vee C$' by 'C'. We call this a *compact version* of the inversion algorithm or a *compact inversion.*

We prefer to think of these as two versions of the same algorithm, rather than two distinct inversion algorithms, since the one is derivable from the other by an elementrary transformation (combining principal cross-sections, as discussed above).

Theorem 3.2 applies to compact as well as standard inversions. The reason is that the transformation of combining principal cross-sections preserves strict properness and semantic equivalence, as can easily be seen.

3.3 Normalising an inverted table: 1-dimensional algorithm

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

This is illustrated in §2.1, with the 1-D "normalisation" of Table 1.4 to Table 1.7. Consider, more generally, the 2-dimensional 3×2 inverted table shown as Table 3.1, with value header H^1. This can be normalised to the 1-dimensional Table 3.8.

$\hat{\mathsf{H}}^1$	$\hat{\mathsf{G}}$
$(C^2_1 \wedge G_{11}) \vee (C^2_2 \wedge G_{12})$	t_1
$(C^2_1 \wedge G_{21}) \vee (C^2_2 \wedge G_{22})$	t_2
$(C^2_1 \wedge G_{31}) \vee (C^2_2 \wedge G_{32})$	t_3

TABLE 3.8. 1-D normalisation of Table 3.1 (Short version)

Note that the header $\hat{H}^1$ of the transformed Table 3.8 still has length $\boldsymbol{len}_1(T)$. However the conditions in this header are relatively complicated (large disjunctions). Again, we can effect a tradeoff between complexity of conditions and header length by splitting

$C_1^2 \wedge G_{11}$	t_1
$C_2^2 \wedge G_{12}$	t_1
$C_1^2 \wedge G_{21}$	t_2
$C_2^2 \wedge G_{22}$	t_2
$C_1^2 \wedge G_{31}$	t_3
$C_2^2 \wedge G_{32}$	t_3
$\hat{H}^1$	$\hat{G}$

TABLE 3.9. 1-D normalisation of Table 3.1 (Long version)

disjunctions, as in Table 3.9. This will increase the header length to ***size*** (T). We call Tables 3.8 and 3.9 *short* and *long* versions of each other.

Remark. The (1-dimensional) normal Table 3.8 is essentially the same as the (1-dimensional) inverted Table 3.2, which was formed by flattening Table 3.1 (see Remark 3 in §3.1). These two 1-dimensional tables are transformed to each other simply by interchanging their row header and grid. Similarly, their respective long versions, namely Tables 3.3 and 3.9, are essentially the same.

We call this transformation a *1-dimensional (*or *1-D) normalisation* (with *short* and *long* versions). The general formula for a table produced by this algorithm (using the short version) is derived as follows. Consider an inverted table $T \equiv (G, H^1, \ldots, H^n)$ with shape $(l_1, \ldots, l_n)$, grid entries $G_{i_1,\ldots,i_n}$, value header $H^1 \equiv (t_1, \ldots, t_{l_1})$, and condition 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 $1 \le i \le l_1$

$$\hat{C}_i \;\equiv\; \bigvee_{i_2=1}^{l_2} \cdots \bigvee_{i_n=1}^{l_n} (G_{i,i_2,\ldots,i_n} \wedge C_{i_2}^2 \wedge \ldots \wedge C_{i_n}^n). \tag{3.3.1}$$

For this transformation (short or long version) we have

Theorem 3.3. *Suppose T is a proper inverted table, and $\hat{T}$ is a 1-D normalisation of T. Then*

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

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

Proof: Part (a), which amounts to the properness of the header $\hat{H}^1$, follows from (3.3.1) above and the properness of T. Part (b) (for the short version of normalisation) follows again from a comparison of ***term***(T) (formula 2.4.1) and ***term***$(\hat{T})$, which is constructed by applying formula 2.3.1 to $\hat{T}$ as defined above. This proof can easily be adapted to the long version of normalisation. $\square$

3.4 Normalising an inverted table: Multidimensional algorithm

We present a second, more complex, normalisation method for inverted tables. This method, unlike the first, preserves dimensionality. However it only applies to *strictly proper* inverted tables. (In practice, this is not a big drawback, since most inverted tables used in program documentation are strictly proper.)

This was illustrated in §2.1, with the multidimensional normalisation of Table 1.5 to Table 1.8. Consider, more generally, the 2-dimensional 2×3 inverted Table 3.10, with value header H^1, which is assumed to be strictly proper. This can be normalised to the 2-dimensional Table 3.11.

C_1^2	C_2^2	C_3^2

H^2

t_1
t_2

H^1

G_{11}	G_{12}	G_{13}
G_{21}	G_{22}	G_{23}

G

TABLE 3.10. An inverted table

C_1^2	C_2^2	C_3^2

H^2

$G_{11} \wedge G_{12} \wedge G_{13}$
$G_{11} \wedge G_{12} \wedge G_{23}$
$G_{11} \wedge G_{22} \wedge G_{13}$
$G_{11} \wedge G_{22} \wedge G_{23}$
$G_{21} \wedge G_{12} \wedge G_{13}$
$G_{21} \wedge G_{12} \wedge G_{23}$
$G_{21} \wedge G_{22} \wedge G_{13}$
$G_{21} \wedge G_{22} \wedge G_{23}$

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

t_1	t_1	t_1
t_1	t_1	t_2
t_1	t_2	t_1
t_1	t_2	t_2
t_2	t_1	t_1
t_2	t_1	t_2
t_2	t_2	t_1
t_2	t_2	t_2

$\hat{\mathsf{G}}$

TABLE 3.11. m-D normalisation of Table 3.10

We show that (a) Table 3.11 is proper, and (b) it is semantically equivalent to Table 3.10. For (b): consider, *e.g.*, the disjunction of the first 4 entries in $\hat{H}^1$, *i.e.*, all those conditions for which the corresponding term in column 1 is t_1:

$$\begin{aligned}
&(G_{11} \wedge G_{12} \wedge G_{13}) \vee (G_{11} \wedge G_{12} \wedge G_{23}) \vee (G_{11} \wedge G_{22} \wedge G_{13}) \vee (G_{11} \wedge G_{22} \wedge G_{23}) \\
&= (G_{11} \wedge G_{12} \wedge (G_{13} \vee G_{23})) \;\vee\; (G_{11} \wedge G_{22} \wedge (G_{13} \vee G_{23})) \\
&= (G_{11} \wedge G_{12}) \vee (G_{11} \wedge G_{22}) \\
&= G_{11} \wedge (G_{12} \vee G_{22}) \\
&= G_{11}.
\end{aligned} \tag{3.4.1}$$

Here we use the fact that $(G_{13} \vee G_{23}) = \mathsf{true}$ and $(G_{12} \vee G_{22}) = \mathsf{true}$, by strict properness of the columns in Table 3.10. This shows that the condition under which the function value is t_1 when C_1^2 holds is exactly the same in both tables. Similarly for the other cases.

For (a), we must show properness of $\hat{H}^1$. Universality follows from (3.4.1) above and the fact that similarly the disjunction of the last 4 entries in $\hat{H}^1$ is equivalent to G_{21}, and $(G_{11} \vee G_{21}) = \mathsf{true}$, again by strict properness of Table 3.10. The disjointness property holds because, for a conjunction

$$(G_{i_1 1} \wedge G_{i_2 2} \wedge G_{i_3 3}) \;\wedge\; (G_{j_1 1} \wedge G_{j_2 2} \wedge G_{j_3 3}) \tag{3.4.2}$$

of conditions from any two different cells of $\hat{H}^1$, $i_k \neq j_k$ (for $k = 1$, 2 or 3) implies $G_{i_k k} \wedge G_{j_k k} = \mathsf{false}$, again by strict properness of the columns in Table 3.10, and so (3.4.2) evaluates to false.

We call this transformation *multi-dimensional (*or *m-D) normalisation*. The difference between this and 1-D normalisation can be summarised by saying that

1-D normalisation involves combining condition headers, and
m-D normalisation involves refining the partitioning of the function domain.

The general formula for a table produced by m-D normalisation is derived as follows. Consider again an inverted table $T \equiv (G, H^1, \ldots, H^n)$ with shape $(l_1, \ldots, l_n)$, grid entries $G_{i_1,\ldots,i_n}$, value header $H^1 \equiv (t_1, \ldots, t_{l_1})$, and condition headers $H^j \equiv (C_1^j, \ldots, C_{l_j}^j)$ $(j = 2, \ldots, n)$. This is transformed to an n-dimensional normal table $\hat{T} \equiv (\hat{G}, \hat{H}^1, H^2, \ldots, H^n)$ with shape $(\hat{l}_1, l_2, \ldots, l_n)$, where

$$\hat{l}_1 \;=\; l_1^{l_2 \cdot l_3 \cdot \ldots \cdot l_n}. \tag{3.4.3}$$

In order to describe the entries in the new header $\hat{H}^1$ and grid $\hat{G}$, we first give some notation for certain index sets. Define

$$\begin{aligned}
I &= \boldsymbol{seg}(l_1), \\
\bar{J} &= \boldsymbol{seg}(l_2) \times \cdots \times \boldsymbol{seg}(l_n), \\
K &= \boldsymbol{seg}(\hat{l}_1).
\end{aligned}$$

Note that $I^{\bar{J}}$ (the set of functions from $\bar{J}$ to I) has cardinality

$$\boldsymbol{card}\,(I)^{\boldsymbol{card}\,(\bar{J})} \;=\; \hat{l}_1 \;=\; \boldsymbol{card}\,(K).$$

A typical element of $I^{\bar{J}}$ is itself a grid of indices

$$\langle i_{\bar{\jmath}} \mid \bar{\jmath} \in \bar{J} \rangle$$

indexed by $\bar{J}$, where for all $\bar{\jmath} = (i_2, \ldots, i_n) \in J$, $1 \le i_{\bar{\jmath}} \le l_1$. Now think of K as an "enumerated" version of $I^{\bar{J}}$ under the ***enum*** function (§2.3.1), with the corresponding element $k \in K$ denoted by

$$k \;=\; [i_{\bar{\jmath}} \mid \bar{\jmath} \in \bar{J}] \;=_{df}\; \boldsymbol{enum}(\langle i_{\bar{\jmath}} \mid \bar{\jmath} \in \bar{J} \rangle).$$

Now the entries in the new header $\hat{H}^1$ are indexed by K; they are given by

$$\hat{C}^1_{[i_{\bar{\jmath}} \mid \bar{\jmath} \in \bar{J}]} \;\equiv\; \bigwedge_{\bar{\jmath} \in \bar{J}} G_{(i_{\bar{\jmath}},\, \bar{\jmath})} \tag{3.4.4}$$

where for each index $[i_{\bar{\jmath}} \mid \bar{\jmath} \in \bar{J}]$, the rhs of (3.4.4) is a conjunction of a list of conditions of length $\boldsymbol{card}\,(\bar{J}) = l_2 \cdot \ldots \cdot l_n$, and for $\bar{\jmath} = (i_2, \ldots, i_n) \in \bar{J}$,

$$(i_{\bar{\jmath}}, \bar{\jmath}) \;=\; (i_{\bar{\jmath}}, i_2, \ldots, i_n) \in \boldsymbol{seg}\,(l_1) \times \boldsymbol{seg}\,(l_2) \times \cdots \times \boldsymbol{seg}\,(l_n).$$

Finally, the entries in the grid $\hat{G}$ are indexed by $K \times \bar{J}$; they are given by

$$\hat{t}_{[i_{\bar{\jmath}} \mid \bar{\jmath} \in \bar{J}],\bar{\jmath}'} \;\equiv\; t_{i_{\bar{\jmath}'}}.$$

Theorem 3.4. *Suppose T is a strictly proper inverted table, and $\hat{T}$ is an m-D normalisation of T. Then*

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

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

We have given the proof in the special case that T is Table 3.10 and $\hat{T}$ is Table 3.11. The proof in the general case can easily be constructed from this.

Notice that although the dimensionality is preserved by this method, the length of the changed header is increased considerably. (For convenience we will continue to assume that H^1 is the value header in the original table.) It turns out, however, that typically many of the entries in this new header evaluate to false, and the corresponding cross-sections can be eliminated. (An example of this was shown in §1.2, with the simplification of Table 1.8 to Table 1.9.)

Since the conditions in the new header $\hat{H}_1$ are conjunctions of length $l_2 \cdot \ldots \cdot l_n$, we would prefer to combine this transformation with *evaluations* of these conditions, *i.e.*, finding simpler semantically eqivalent formulae, and, more especially, identify those conditions which evaluate to false, since the corresponding cross-sections can then be deleted.

Note that there are two ways in which such a conjunction (3.4.4) can evaluate to false: either (a) one of the conjuncts, *i.e.*, a grid entry in the given table, is itself (equivalent to) false, or (b) a conjunction of some *sublist* of the list of conjuncts in the rhs of (3.4.4) is (equivalent to) false. Identification and elimination of either of these results in a considerable reduction in length for $\hat{H}^1$: for (a), by a factor of $1/l_1$, and for (b), by a factor of $1/l_1^k$, where k is the size of the sublist whose conjunction evaluates to false, since the conjunction of any list which includes a false sublist must itself be false.

There are two problems with this method, in connection with its complexity:

(i) The *mechanical evaluation* of conjunctions of conditions as in (3.4.4), especially the testing of equivalence to false. In general this is an undecidable problem, even for unquantified conditions [Zuc96, §10]; but one can try to identify special cases in which it is decidable, and feasible.

(ii) The *large number* of such evaluations ($\hat{l}_1$: see (3.4.3)) of conjunctions for entries in $\hat{H}^1$. This number can be reduced by testing conjunctions of initial sublists of the lists of conditions on the rhs of (3.4.4) for equivalence to false (and generating such sublists in a suitable order), since then (as noted above) all conjunctions which extend such a false conjunction must themselves be false, and can be removed from further consideration.

Thus a solution for (ii) depends essentially on a solution for (i), *i.e.*, evaluation of conjunctions. This problem is a subject for ongoing research. It has not been solved satisfactorily yet, and so this method has not yet been implemented.

4 Interrelationships between transformations

We explore the relationships between some of the transformations and operations on tables considered in this paper. In §4.1 we find connections between inversion, 1-D normalisation and flattening. In §4.2 we consider the composition of an inversion and m-D normalisation. In both these cases, we will show that the composition of certain transformations on a given table is "elementarily equivalent" (in a sense to be made precise) to the table itself, or to some (other) transformation on it. We will use two notions of "elementary equivalence": a "syntactic" notion (in §4.1) and a "semantic" notion (in §4.2), both stricter than semantic equivalence. The difference between the two is discussed in §4.2. In §4.3 we show that the operations of *slicing* and *inverting* commute, as do the operations of *slicing* and *m-D normalising*.

Proofs of the theorems in this section are fairly routine but notationally forbidding; they are therefore omitted. Full proofs can be found in [ZS98].

4.1 Inversion, 1-D normalisation and 1-D flattening

Definition 1. An *elementary transformation* of a table is any one of the following:

(a) a *structural transformation*:

- (i) permuting two cross-sections together with their corresponding header entries;
- (ii) deleting a cross-section with 'false' in the corresponding header entry;
- (iii) (in an inverted table) deleting a principal cross-section with only 'false' entries;
- (iv) (in an inverted table) splitting a principal cross-section by "splitting a disjunction" in the corresponding header entry;
- (v) (in an inverted table) combining two or more principal cross-sections with the same value header entry into a single cross-section.

(b) a *transformation of a condition cell entry*, as follows:

- (i) permuting components of a conjunction or disjunction
- (ii) distributing a conjunction over a disjunction,
- (iii) simplifying conditions '$C \wedge \mathsf{false}$' to 'false',
- (iv) simplifying conditions '$C \vee \mathsf{false}$' to 'C',

Remarks. (1) In two dimensions the structural transformations (a) amount to permuting, deleting, repeating and combining rows or columns.

(2) Another reasonable candidate for a structural elementary transformation would be:

- (vi) removing a trivial dimension, *i.e.*, a dimension (and corresponding header) of length 1.

However this turns out not to be needed for the proof of Theorem 4.1 below.

Definition 2. *Elementary equivalence* of tables is the equivalence relation on $\boldsymbol{Tab}_N \cup \boldsymbol{Tab}_I$ 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.

We write $T_1 \approx_e T_2$ to denote that T_1 and T_2 are elementarily equivalent.

Remarks. (1) Elementary equivalence between tables is *decidable*.

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

(3) The long and short flattenings (§3.1) of a given table are elementarily equivalent to each other, as are the standard and compact inversions (§3.2) of a normal table, and the long and short 1-D normalisations (§3.3) of an inverted table.

(4) Warning: Properness (in particular, disjointness) is not necessarily preserved under elementary transformation $(a)(iv)$: splitting disjunctions. (This point was overlooked in

[Zuc96] and [She95].) The relation of elementary equivalence must therefore be *restricted to pairs of proper tables.* In any case, properness is not violated in splitting disjunctions to transform short to long variants of the flattening and 1-D normalising algorithms, as used in Theorem 4.1 below.

Elementary equivalence between tables is preserved by all the transformations considered in Section 3. In other words:

Lemma 4.1. *If φ is any of the table transformations considered in Section 3, then for any two proper tables T_1 and T_2 in the domain of φ,*

$$T_1 \approx_e T_2 \Longrightarrow \phi(T_1) \approx_e \phi(T_2).$$

This is straightforward but tedious to check.

Now, given a normal table T, *inversion* followed by *1-D normalisation* is essentially the same as the 1-D flattening of T; and conversely, starting with an inverted table T, *1-D normalisation* followed by *inversion* is essentially the same as the 1-D flattening of T. This is made precise in the following theorem.

Theorem 4.1. *(a) If T is a proper normal table, then the result of an inversion of T followed by a 1-D normalisation is elementarily equivalent to a 1-D flattening of T.*

(b) If T is a proper inverted table, then the result of a 1-D normalisation followed by an inversion is elementarily equivalent to a 1-D flattening of T.

4.2 Inversion and m-D normalisation

First we must modify the notion of elementary transformation and elementary equivalence used in §4.1:

Definition 1. An *elementary semantic transformation* of a table is *either*

(a) a *structural transformation* as in part (a) of Definition 1 in §4.1, *or*

(b) a *semantic transformation* of a condition cell entry, namely the replacement of a condition by a semantically equivalent condition.

Definition 2. *Elementary semantic equivalence* of tables is the equivalence relation on $\boldsymbol{Tab}_N \cup \boldsymbol{Tab}_I$ generated by the class of elementary semantic transformations.

We write $T_1 \approx_{es} T_2$ to denote that T_1 and T_2 are elementarily sematically equivalent.

Remarks. (1) The semantic transformation (b) above incorporates all the transformations (b) listed in Definition 1 of §4.1, and much more. Thus elementary equivalence is a refinement of elementary semantic equivalence.

(2) Elementary semantic equivalence is, in turn, a refinement of semantic equivalence (for proper tables).

(3) Unlike elementary equivalence, elementary semantic equivalence is, in general, *undecidable*.

Again, elementary semantic equivalence between tables is preserved by all the transformations considered in Section 3. In other words:

Lemma 4.2. *If φ is any of the table transformations considered in Section 3, then for any two proper tables T_1 and T_2 in the domain of φ,*

$$T_1 \approx_{es} T_2 \Longrightarrow \phi(T_1) \approx_{es} \phi(T_2).$$

Now, *inversion* composed with *m-D normalisation* (in either order) is essentially the identity transformation, in the following sense.

Theorem 4.2. *(a) If T is a proper normal table, then the result of an inversion of T followed by an m-D normalisation is elementarily semantically equivalent to T.*

(b) If T is a strictly proper inverted table, then the result of an m-D normalisation of T followed by an inversion is elementarily semantically equivalent to T.

4.3 Slicing and inverting; Slicing and m-D normalising

Here *slicing* means constructing a "slice" of a table, *i.e.*, a subtable formed by holding certain dimensions fixed.

Note first that for the slicing of an inverted table to make sense, we must assume that it is not sliced across the principal dimension, *i.e.*, the principal dimension is not one of the fixed dimensions.

It is easy to see that *properness is preserved* by slicing a *normal* table, but *not* (necessarily) by slicing an inverted table. For *inverted* tables, it is *strict properness* which is preserved:

Lemma 4.3. *(a) A slice of a proper normal table is proper.*

(b) A slice of a strictly proper inverted table is strictly proper.

Theorem 4.3. *(a) The operations of slicing and inverting a proper normal table commute up to elementary equivalence.*

(b) The operations of slicing and m-D normalising a strictly proper inverted table commute up to semantic elementary equivalence.

5 Table Transformation Tool

To facilitate the use of tabular notation, the Table Tool System (TTS) described in [SERG97] has been developed by a number of workers at the Software Engineering Research Group (SERG), McMaster University. The goal of the TTS project is to develop an integrated, extensible system of tools — that is, a set of tools that work together, so that a designer or programmer can manipulate tables easily in computer systems documentation.

As one of the tools in the TTS, the Table Transformation Tool (TTT) implemented the table transformation algorithms proposed in Section 3, specifically:

(*i*) inverting a normal table,

(*ii*) 1-D normalisation of an inverted table,

(*iii*) flattening a table.

Note that no algorithm for m-D normalisation has been implemented, because (as stated in section 3) the testing of the equivalence of conditions to false at one stage of the "algorithm" is not decidable.

The TTS provides two modules: the Table Holder, which hides the representation of the data structures used to implement the Σ-algebra, and the Information Module, which stores type and other information associated with each identifier or symbol.

By using these two modules, the implementation of the transformation algorithms was greatly simplified, since we need concern ourselves only with the structure of the algorithms, rather than the specific representation of the data and syntax.

General steps of the algorithm implementations include: (a) checking whether the given expression is a so called "well-formed" table, i.e., syntactically correct for the algorithm; (b) constructing the shape of the transformed table, each grid and the length along each dimension of the table; (c) copying the unchanged headers if any; (d) calculating the changed headers; and (e) calculating the main grid entries.

The code is written following the programming guidelines proposed in [PMI94]. A set of *displays*, which specify the relations the programs should satisfy, is provided along with the code. The display method facilitates integration of the table transformation algorithms described in this paper with the TTS to form the TTT.

Recent improvements on the TTT include: (1) a graphic user interface, consistent with the TTS interface, and (2) a utility to facilitate the interpretation of tables [Abr97].

6 Future work

Some of the following ideas are being planned in the Software Engineering Research Group at McMaster University (SERG); others are more speculative at this stage. An asterisk indicates that a start has already been made on that topic (typically as a Master's project).

(1)* *Simplification* of tables (in any format) is clearly important. We refer here both to simplifying the global structure if the table, and the terms and formulas in the cells. Note that in the m-D normalisation algorithm (§3.4), the second, or simplifying, step presents problems of implementation. Simplification can best be done interactively, in conjunction with a symbolic computation system such as Maple [CGG+91]. A start in this direction has been made in [Ras98].

(2)* It is important to develop tools for (partial) testing of *properness* of tables, at least in special cases, since this problem is not decidable or feasible for many common signatures

[Zuc96]. We should note that progress has been made by other research groups on partial testing of properness of tables [HL96, HJL96].

(3)* Similarly, it is important to develop (partial) tests for *semantic equivalence* of tables, or of terms occurring within tables. Some progress has been made in this direction in [Nad99].

(4) An important special case of (3) is the evaluation of *conditions*, *i.e.*, testing whether they are equivalent to true or false. This is useful in two ways: (*a*) to test for properness as in (2); (*b*) to simplify the table resulting from the m-D normalisation algorithm (§3.4).

(5)* Continuing the development in §6.1(2), a utility should be developed for the *type checking* of input tables.

(6) The main reason for table transformations is to simplify tables (in some sense). *Heuristic guidelines* should therefore be developed for deciding which version of a given table is likely to be simplest.

(7)*Finally, tools should be developed for transformations from, and to, other kinds of tables than the normal and inverted function tables considered here. Some work in this direction is currently being done by SERG with "decision tables".

References

[Abr97] R.F. Abraham. Evaluating generalized tabular expressions in software documentation. M.Eng. Thesis, Department of Electrical & Computer Engineering, McMaster University, 1997. CRL (SERG) Report 346, Software Engineering Research Group, McMaster University.

[AFB+92] T.A. Alspaugh, S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, and J.E. Shore. Software requirements for the A-7E Aircraft. NRL Report NRL/FR/5530-92-9194, U.S. Naval Research Laboratory, Washington, DC, 1992.

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[CGG+91] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V Reference Manual.* Springer-Verlag, 1991.

[Far95] W.M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43:279–294, 1995.

[GHM78] J.V. Guttag, E. Horowitz, and D.R. Musser. The design of data type specifications. In R.T. Yeh, editor, *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, pages 60–79. Prentice Hall, 1978.

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

[HJL96] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5:231–261, 1996.

[HKPS78] K.L. Heninger, J. Kallander, D.L. Parnas, and J.E. Shore. Software requirements for the A-7E Aircraft. NRL Memorandum Report 3876, U.S. Naval Research Laboratory, Washington, DC, 1978.

[HL96] M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22:363–377, 1996.

[KK94] M. Kerber and M. Kohlhase. A mechanization of strong kleene logic for partial functions. In A. Bundy, editor, *CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 371–385. Springer-Verlag, 1994.

[Lam91] K. Lambert, editor. *Philosophical Applications of Free Logic*. Oxford University Press, 1991.

[Nad99] K. Nadarajah. Semantical equality of tabular expressions. M.Eng. Thesis, Department of Electrical & Computer Engineering, McMaster University, 1999. SERG Report 378, Software Engineering Research Group, McMaster University.

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

[Par92] D.L. Parnas. Tabular representation of relations. CRL (SERG) Report 260, Software Engineering Research Group, McMaster University, 1992.

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

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

[PMI94] D.L. Parnas, J. Madey, and M. Iglewski. Formal documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20:948–976, 1994.

[Ras98] P. Rastogi. Specialization: An approach to simplifying tables in software documentation. M.eng. thesis, Department of Electrical & Computer Engineering, McMaster University, 1998. CRL (SERG) Report 360, Software Engineering Research Group, McMaster University.

[SERG97] McMaster University Software Engineering Research Group. Table tool system developer's guide. CRL (SERG) Reports 339 & 340, Software Engineering Research Group, McMaster University, 1997.

[She95] H. Shen. Implementation of table inversion algorithms. M.Eng. Thesis, Department of Electrical & Computer Engineering, McMaster University, 1995.

CRL (SERG) Report 315, Software Engineering Research Group, McMaster University.

[TZ88] J.V. Tucker and J.I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, volume 6 of *CWI Monographs*. North Holland, 1988.

[ZS98] J.I. Zucker and H. Shen. Table transformation: Theory and tools. Report CAS 98-01, Department of Computing & Software, McMaster University, 1998. Also CRL Report 363, Communications Research Laboratory, McMaster University.

[Zuc96] J.I. Zucker. Transformations of normal and inverted function tables. *Formal Aspects of Computing*, 8:679–705, 1996.