Compositional Syntax and Semantics of Tables

Wolfram Kahl

Software Quality Research Laboratory
Department of Computing and Software
McMaster University

URL: http://www.cas.mcmaster.ca/~kahl/

October 12, 2003

Abstract

Parnas together with a number of colleagues established the systematic use of certain kinds of tables as a useful tool in software documentation and inspection with an accessible, multi-dimensional syntax and intuitive semantics.

Previous approaches to formalisation of table semantics based their definitions on the multi-dimensional array structure of tables and thus achieved close correspondence with the intuitive understanding of tables.

In this paper, we argue that a different view, supporting a compositional semantics, is more advantageous for tool support and for reasoning about tables. For this purpose, we also need a compositional table syntax, and we perform an analysis of table syntax that leads us to a particular compositional view of table structure.

This simple, inductive view of the structure of tables allows us to provide highly flexible tools for defining the semantics of tabular expressions. The straight-forward compositional formalisation of table semantics on the one hand yields very general table transformation theorems and enables us to perform fully formal proofs for these theorems in a mechanised theorem prover, and on the other hand also may serve as basis for the implementation of semantics-aware table support tools.

SQRL

Software Quality Research Laboratory

McMaster University

SQRL Report No. 15

Contents

1 Introduction

Starting 1977, Parnas *et al.* introduced the use of two-dimensional expressions called *tables* into software requirements specification documents as a means of organising the presentation of complex relations [HKPS78, HPU81, PAM91, Par94]. While in the beginning, the meaning of these tables was simple and intuitively clear, over time, more table kinds were introduced and the need of a more formal definition of their meaning was felt.

Parnas formally defined the meaning of ten different kinds of tables in [Par92], using separate definitions to assign relations, functions, or predicates as meaning to tables of each kind. Although he does not discuss syntactic composition of tables, Parnas' definitions of table semantics are *compositional* in the sense that he first defines the meaning of a cell under the influence of the relevant headers as a relation, (partial) function, or predicate, and then lets the meaning of the table be the union, or disjunction, of all the meanings of the cells of the table's grid. Parnas called his tables also *tabular expressions* to emphasise that they still should be considered as mathematical expressions, which justifies their use for example in recursive definitions. Parnas' disjunctive definitions for normal and inverted tables were reformulated in a conjunctive shape by Zucker for discussing table transformation [Zuc96, ZS98].

Janicki proposed a first unified framework for table semantics [Jan95, JK01]; the term *tabular expression* was now defined to mean a table together with a *semantic rule* that defines how the relational meaning of the table is derived from its entries, and semantic rules were provided for the table kinds of [Par92].

Abstracting this approach to a more algebraic flavour, Desharnais, Khedri, and Mili, starting from understanding grids and headers as *separate* arrays of relations [DKM01], proposed to interpret tables using relation algebraic operations mixed with array reduction operations inspired by the programming language APL. This results in a very elegant formalism that offers an alternative, much more concise way to define semantic rules for tables — it is not demonstrated how the separate definitions for the different table kinds can be systematically derived from the semantics rules of tabular expressions *á la* [JK01].

In some sense, we may consider the approaches to table semantics by Parnas, Janicki, and Zucker [Par92, Jan95, Zuc96, JPZ97, JK01] as geared towards "table users", in particular, (software) engineers writing, reading, or inspecting such tables and needing to understand exactly what the tables they interact with mean.

The relation-algebraic semantics of [DKM01] can be seen as geared more towards facilitation of algebraic manipulation and mechanised reasoning about tables.

In this paper, we present a new framework of defining semantic rules for tables in a general and flexible way. This new table semantics framework is explicitly motivated by the desire to have an understanding of tables that can be used for reasoning about tables and table transformation, and also as a basis for machine-support of table manipulation and transformation. It may therefore appear less "direct" to the table user, but the compositional approach we take has advantages for reasoning and mechanisation. To demonstrate this we include in the appendices the basics of a table library in the purely functional programming language Haskell [PJ^+03]; along the same lines, a theory has been developed in the mechanised theorem proving system Isabelle/HOL [NPW02], including proofs of the presented table transformation theorems.[1]

In the remainder of this section, we provide some guidance to the theoretical principles behind our analysis of tables and an overview of the body of this paper.

[1] The Haskell library and Isabelle theories are available at URL: http://www.cas.mcmaster.ca/~kahl/Tables/.

A typical table, as for example the table T_f drawn in Fig. 1 (taken from [JPZ97]), consists of a two-dimensional grid and headers; these are subdivided into cells containing (predicate) expressions.

		$y = 10$	$y > 10$	$y < 10$	H_1
H_2	$x \geq 0$	0	y^2	$-y^2$	G
	$x < 0$	x	$x + y$	$x - y$	

Figure 1: The table T_f

The whole table can again be used as an expression, and frequently its meaning is, at least intuitively, explained as a conventional expression — the following two presentations for the meaning of T_f may be found in [JPZ97], first one in liberal mathematical notation:

$$f(x, y) = \begin{cases} 0 & \text{if } x \geq 0 \wedge y = 10 \\ x & \text{if } x < 0 \wedge y = 10 \\ y^2 & \text{if } x \geq 0 \wedge y > 10 \\ -y^2 & \text{if } x \geq 0 \wedge y < 10 \\ x + y & \text{if } x < 0 \wedge y > 10 \\ x - y & \text{if } x < 0 \wedge y < 10 \end{cases}$$

Then one in classical predicate logic notation:

$$\begin{aligned} &(\forall\, x.(\forall\, y.((x \geq 0 \wedge y = 10) \Rightarrow f(x, y) = 0) \wedge ((x < 0 \wedge y = 10) \Rightarrow f(x, y) = x) \wedge \\ &\qquad ((x \geq 0 \wedge y > 10) \Rightarrow f(x, y) = y^2) \wedge ((x \geq 0 \wedge y < 10) \Rightarrow f(x, y) = -y^2) \wedge \\ &\qquad ((x < 0 \wedge y > 10) \Rightarrow f(x, y) = x + y) \wedge ((x < 0 \wedge y < 10) \Rightarrow f(x, y) = x - y))). \end{aligned}$$

With this expression-building nature, tables are a syntactic device that is in a certain sense similar to the syntactic devices of infix operators like “$\_ + \_$”, mix-fix operators like “$(\_, \_)$”, or “arrangement operators” like subscripts or superscripts.

All these syntactic devices by themselves are essentially meaningless, but only acquire meaning when some semantic rule is provided. Nevertheless, we may still discuss the *structure* of the syntactic constructions produced by these syntactic devices. For example, for infix operator application, this structure is that of an ordered pair. We analyse the structure of tables in Sect. 2, and arrive at a simple, compositional structure of a generalised class of tables. This table structure is equipped with an intuitive typing system in Sect. 3.

The structure of syntactic constructions induces the ways how semantics may be defined for such a syntactic device. For example, since an infix operator application results in an ordered pair, the semantics of an infix operator is given by a function that has an ordered pair as arguments, usually presented as a binary function. Some operators, like the pairing operator “$(\_, \_)$”, are only intended for structure building (in this case, for building pair structures), and thus induce *container data types* (in this case Cartesian product types). In fact, Cartesian products are the container data type associated with the structure of binary infix operator applications, so knowledge about ways to define functions over Cartesian products is necessary for being able to define the semantics of binary infix operators. In Sect. 4, we define the general mechanisms of structure-oriented table processing, and thus show how to define functions over tables considered as container data types, exploiting the inductive structure elicited in Sect. 2. In particular, we propose a homogeneous treatment of all dimensions in tables with arbitrary dimensionality, and do not limit ourselves to relations as results of such functions, but permit arbitrary interpretations.

Some infix operators are considered as "derived" — they are defined by rules that allow rewriting any expression containing them into an equivalent expression not containing them. To a large extent, this is also perceived to be the case for tables. For example, in the context of permitting tables for defining recursive functions, Parnas writes [Par92]: "Each such table will be equivalent to a more conventional mathematical expression; the rules governing recursive definitions are unchanged." The rules defining the translation of a table into a conventional mathematical expression can of course be defined using the mechanisms introduced in Sect. 4; in Sect. 5 we discuss how this view affects the organisation of tabular expressions and their semantics. Finally, we present a collection of table transformation theorems in Sect. 6.

2 A Compositional View of Table Syntax

The attractiveness of tables is due in large part to their two-dimensional *concrete* syntax, which has proven to be a very intuitive and accessible device for stringent documentation.

For most conventional programming and specification languages, the concrete syntax is one-dimensional — such a language is a set of character strings over some alphabet, and the concrete syntax of such languages is defined by string grammars. The definition of the semantics of such languages, however, usually does not start from the productions of the concrete syntax, but from the *abstract* syntax, which defines the abstract view of such a language as a set of *terms* which are essentially a certain kind of labelled directed trees — these can be understood as parse trees liberated from details such as white space or parentheses. As such, these trees represent a direct view of how larger, more complex language elements (e.g., programs) are composed from simpler language elements. The characteristic feature of semantics defined in this way is *compositionality*, i.e., the property that the semantics of a larger program depends only on the semantics of its immediate constituents and on the way it has been composed from these. Powerful algebraic properties of the semantic rules associated with simple syntactic composition mechanisms are particularly useful in supporting reasoning about programs and program transformation.

Previous approaches to table semantics [Par92, Jan95, Zuc96, DKM01, JK01] all started from the multi-dimensional structure of the concrete syntax, and from there defined a more or less monolithic semantics. Proving table transformation theorems as contained in [ZS98] (see also [Zuc96, SZP96]) is therefore always burdened (at least notationally) with the full semantics over all dimensions, even if the transformation itself only affects one or two dimensions.

In order to achieve more modular table transformation theorems and proofs, we need compositionally defined table semantics. For this, however, we first of all need a compositional view of table syntax, i.e., we need to investigate how larger tables can be constructed from smaller tables. The remainder of this section develops such a compositional view of table syntax.

In all previous formalisations of table semantics, tables of arbitrary dimensionality are allowed: an n-dimensional table has, as its central component, a *grid*, which is an n-dimensional array of *cells*, and it also has n headers, which are one-dimensional arrays of header cells; the length of each header is the same as that of the corresponding dimension of the grid.

This system of arrays is used in the previous literature as the abstract syntax of a table, and table semantics and transformation involve the use of sometimes quite sophisticated indexing mechanisms. Although such mechanisms allow us to talk about, e.g., "table slices", it appears that no considerations of table composition and decomposition have been published so far.

One possible reason behind this is that in most of the literature, attention is focused on "complete" tables, i.e., tables that, in different ways for different table kinds, completely cover a relevant domain. Since fragments of a "complete" table will typically not be "complete", table decomposition is not natural in such contexts.

However, "completeness" is a semantic aspect, and we decided to find a compositional syntax of tables in order to be able to define compositional semantics. Therefore, no semantic inhibitions can keep us from syntactically decomposing tables, and we shall do so in the framework of the traditional abstract syntax, which considers tables as systems of arrays, and we illustrate our efforts using the traditional concrete syntax, which depicts tables as boxed grid arrangements.

One natural approach to answering the question "what is a table" is to investigate what the simplest tables are, and how more complex tables can be built from simpler tables.

However, it turns out that there are several problems with determining the precise nature of the simplest tables. Therefore, we first investigate how known tables can be considered as composed from constituent structures.

2.1 Horizontal Table Concatenation

Let us first consider again the table T_f drawn in Fig. 1. This table is two-dimensional, as made obvious by its graphical presentation. It has two *headers*, H_1 and H_2, and the grid G.

Such a table can be split along any grid line — for the time being we consider only those splits that split the first header, H_1, for example into the tables $T_{f,\mathrm{a}}$ and $T_{f,\mathrm{b}}$ drawn in Fig. 2.

		$y = 10$	$y > 10$	$H_{1\mathrm{a}}$
H_2	$x \geq 0$	0	y^2	G_a
	$x < 0$	x	$x + y$	

		$y < 10$	$H_{1\mathrm{b}}$
H_2	$x \geq 0$	$-y^2$	G_b
	$x < 0$	$x - y$	

Figure 2: The tables $T_{f,\mathrm{a}}$ and $T_{f,\mathrm{b}}$

Both of these tables are still two-dimensional, and both have the same header in the second dimension. We call the operation that joins $T_{f,\mathrm{a}}$ and $T_{f,\mathrm{b}}$ into T_f *table concatenation*:

Notation 1 We use the infix operator $|||$ for concatenation of tables of the same dimension. □

Therefore we have:

$$T_f = T_{f,\mathrm{a}} \mathbin{|||} T_{f,\mathrm{b}}$$

The sequencing of the dimensions of graphically displayed tables will usually be indicated by header labels such as H_1, H_2; in addition, the first header will usually be at the top, such that the first dimension will be arranged in a horizontal way, so we also call $|||$ *horizontal table concatenation*.

There is only one further obvious decomposition via $|||$, namely $T_{f,\mathrm{a}} = T_{f,\mathrm{aa}} \mathbin{|||} T_{f,\mathrm{ab}}$, with the two component tables drawn in Fig. 3.

Substituting this new decomposition in the above decomposition of T_f, we obtain

$$T_f = (T_{f,\mathrm{aa}} \mathbin{|||} T_{f,\mathrm{ab}}) \mathbin{|||} T_{f,\mathrm{b}} \quad .$$

		$y = 10$	$H_{1\mathrm{aa}}$
H_2	$x \geq 0$	0	G_{aa}
	$x < 0$	x	

		$y > 10$	$H_{1\mathrm{ab}}$
H_2	$x \geq 0$	y^2	G_{ab}
	$x < 0$	$x + y$	

Figure 3: The tables $T_{f,\mathrm{aa}}$ and $T_{f,\mathrm{ab}}$

However, there is nothing in the displayed or perceived table structure of T_f that indicates that this decomposition is preferable to the differently parenthesised alternative

$$T_f = T_{f,\mathrm{aa}} \,|||\, (T_{f,\mathrm{ab}} \,|||\, T_{f,\mathrm{b}}) \ .$$

Therefore, it is natural to demand the following:

Requirement 1 Table concatenation $|||$ is associative,

$$(t_1 \,|||\, t_2) \,|||\, t_3 = t_1 \,|||\, (t_2 \,|||\, t_3) \ ,$$

and if one of the two sides is defined, then so is the other. □

Table concatenation $|||$ is *not* commutative, since, for example, the table $T_{f,\mathrm{a}'} := T_{f,\mathrm{ab}} \,|||\, T_{f,\mathrm{aa}}$, drawn in Fig. 4, is obviously "graphically" different from the table $T_{f,\mathrm{a}} = T_{f,\mathrm{aa}} \,|||\, T_{f,\mathrm{ab}}$ from Fig. 2.

		$y > 10$	$y = 10$	$H_{1\mathrm{a}'}$
H_2	$x \geq 0$	y^2	0	$G_{\mathrm{a}'}$
	$x < 0$	$x + y$	x	

Figure 4: The table $T_{f,\mathrm{a}'} := T_{f,\mathrm{ab}} \,|||\, T_{f,\mathrm{aa}}$

This derives from the fact that the sequence of the presentation is an important part of tabular notation — swapping columns obviously produces a *different* (albeit possibly equivalent) table. Note that the only argument for this non-commutativity is that it appears as natural considering the standard graphical presentation of tables. If one decides to consider $|||$ as commutative, then its interpretations (see Def. 4.1.1) need to be commutative, too. In fact, from a semantic point of view, demanding commutativity of the interpretations of $|||$ appears to be preferable.

An important question is whether further decomposition of $T_{f,\mathrm{aa}}$, $T_{f,\mathrm{ab}}$, and $T_{f,\mathrm{b}}$ via $|||$ is possible — this would yield empty tables that are (partial) units for concatenation.

- In [Par92, Zuc96], dimensions are explicitly restricted to length at least one, so empty tables are not allowed.
- In [JK01], empty tables are not explicitly forbidden, but seem to be not intended.
- In [DKM01], empty tables are explicitly allowed, but not motivated. Since in that paper, only relational meet and join are used for array reduction, and these operations have unit elements, empty tables do not lead to any problems.

Since it appears that the only motivation for allowing empty tables would be to obtain, in some sense, a “nicer algebra”, we will disallow empty tables, but remember this as a design decision:

Issue 1 Should there be empty tables, i.e., (partial) units for $|||$? $\square$

2.2 Adding New Dimensions

What is the further decomposition of $T_{f,\mathrm{aa}}$? Its obvious components are the header “$y = 10$” (let’s call it $H_{1\mathrm{aa}}$) and another, one-dimensional table that we call T_j, drawn in Fig. 5.

$x \geq 0$	$x < 0$	H_2
0	x	G_{J}

Figure 5: The one-dimensional table T_j

We have “turned this table around”, since now H_2 is the header in its first dimension. To obtain the two-dimensional table $T_{f,\mathrm{aa}}$ from the header $H_{1\mathrm{aa}}$ and the one-dimensional table T_j, we need a new operation for equipping a table with a header in a new dimension:

Notation 2 The infix operator $\triangleright$ combines a header h and an n-dimensional table t into an $(n + 1)$-dimensional table $h \triangleright t$.

In $h \triangleright t$, the header h will be the single header of the *first* dimension; dimensions of t will accordingly be shifted in $h \triangleright t$. $\square$

Then we have $T_{f,\mathrm{aa}} = H_{1\mathrm{aa}} \triangleright T_j$, and $H_{1\mathrm{aa}}$ is the only header of $T_{f,\mathrm{aa}}$ in its first dimension, while the first dimension of T_j is the second dimension of $T_{f,\mathrm{aa}}$.

2.3 Cells

We can decompose T_j, too, yielding $T_j = T_{j,\mathrm{a}} \mathbin{|||} T_{j,\mathrm{b}}$ with the components drawn in Fig. 6.

$x \geq 0$	$H_{2\mathrm{a}}$
0	G_{Ja}

$x < 0$	$H_{2\mathrm{b}}$
x	G_{Jb}

Figure 6: The one-dimensional tables $T_{j,\mathrm{a}}$ and $T_{j,\mathrm{b}}$

If we decompose $T_{j,\mathrm{a}}$ via $\triangleright$, we obtain a header $H_{j,\mathrm{a}}$ and the object drawn in Fig. 7, which, by analogy to the above, has to be a zero-dimensional table.

0	G_{Ja}

Figure 7: The cell $[\,0\,]$

Since this table is constructed from an expression in a single grid cell, we need a way to produce a zero-dimensional table from such an expression:

Notation 3 For some cell content element e, the zero-dimensional table consisting of a single *cell* containing e is written $[\,e\,]$. $\square$

For most conventional tables, decomposition by $|||$ and $\triangleright$ finally leads to such single-cell grids.

2.4 A Short Detour: Indexed Tables

The definition of table syntax by Janicki *et al.* starts from table skeletons that are defined using "cell connection graphs" [Jan95, JPZ97, JK01]. Under the assumption that there are no empty tables, we also arrive at cells as the simplest tables via an argument based on this structure: Since the simplest non-empty cell connection graph consists only of the grid, and since the simplest non-empty grid has one element, the simplest table has to be a single cell, i.e., a simple one-element grid containing an expression. Furthermore, tables have dimensions, and with conventional table skeletons, the dimension of a table is the number of header nodes connected to the grid in the cell connection graph. Since a cell has no headers, it is a zero-dimensional table.

This argument does not contribute to the converse implication:

Issue 2 Are all zero-dimensional tables cells? □

At first sight, the literature seems to be unanimous: An n-dimensional grid is a indexed set, indexed by an n-dimensional Cartesian product; since there is only one zero-dimensional Cartesian product, and this contains only a single element, namely the zero-tuple, a zero-dimensional grid contains exactly one element.

However, if we consider zero-dimensional tables to be any "grids without headers" according to the approach to table decomposition we followed above, then we encounter some tables where, after decomposition and stripping of all headers, the remaining grids contain more than one cell.

For example, in some applications, so-called *indexed tables* have been used. There, single-cell headers appear to belong to a multi-cell grid, and the interpretation of the headers provides *indices* (starting at zero) into the grid for retrieval of the result, as for example in the table drawn in Fig. 8 ("%" stands for the "modulo" operation producing the remainder of integer division).

<table>
<tr><td></td><td></td><td colspan="3">x % 3</td><td>H_1</td></tr>
<tr><td rowspan="2">H_2</td><td rowspan="2">$y \% 2$</td><td>0</td><td>y^2</td><td>$-y^2$</td><td rowspan="2">G</td></tr>
<tr><td>x</td><td>$x + y$</td><td>$x - y$</td></tr>
</table>

Figure 8: An "indexed table"

Note that this indexed table is still considered as a two-dimensional table, and that even the grid G without the headers would have to be considered as being two-dimensional, and decomposable via special "grid concatenation" operators that would be different from the table concatenation operators like $\|\|$.

In our opinion, however, it appears more natural to consider a tabular expression based on an indexed table to be an abbreviation:

- It could be seen as abbreviating a tabular expression that contains the index-generating functions in its semantic rule (we show later how to present this). The table would then be composed from single cells with headers containing just the indices, as can be seen in Fig. 9 — for indexed tables with more than about five elements in any one dimension, this also greatly improves readability.
- Alternatively, the "indexed table" itself could be seen as abbreviating a conventional table where every header cell explicitly relates its index with the contents of the corresponding "long header" of the indexed table, as in Fig. 10.

		0	1	2	H_1
H_2	0	0	y^2	$-y^2$	G
	1	x	$x+y$	$x-y$	

Figure 9: The table T_i underlying the "indexed table" of Fig. 8

		$x\%3 = 0$	$x\%3 = 1$	$x\%3 = 2$	H_1
H_2	$y\%2 = 0$	0	y^2	$-y^2$	G
	$y\%2 = 1$	x	$x+y$	$x-y$	

Figure 10: The "indexed table" of Fig. 8 made explicit

(This can be seen as the result of expanding "abbreviated grids", see 4.7.)

For either view of indexed tables, no grid concatenation operations are required.

One might argue also for other reasons that a grid without headers need not be a singleton grid. However, whenever the table composition rule takes into account grid positions, then the coordinates of these positions can be considered as implicit headers, and we would demand them to be made explicit, as in the above discussion of indexed tables.

Therefore, we take the stance that header-less grids should be singleton expressions.

2.5 Regular Tables and Table Skeletons

Given an n-dimensional table, the next dimension is reached by adding a header element using "$\triangleright$". A table $h \triangleright t$ therefore has one dimension more than t, and in that dimension only the single header h.

Conventionally, two tables can be concatenated into a single table only if they coincide in their lower dimensions, i.e., in all dimensions except the first. This ensures that in the concatenated table, every dimension is described by a *single* system of headers. Tables where this is the case will be called *regular tables*.

For the time being let us ignore nested headers (see 4.7); the system of headers for any dimension can then adequately be described as a list of headers. Therefore, the whole table structure of a regular table can be described by a list containing one header list for every dimension. We will call such a list of header lists a *regular table skeleton*.

For example, the table T_f from Fig. 1 on page 4 has the following skeleton:[2]

$$\mathsf{regSkel}\ T_f = \langle\ \langle y = 10, y > 10, y < 10 \rangle,\ \langle x \geq 0, x < 0 \rangle\ \rangle$$

If concatenation of two regular tables T_1 and T_2 is to yield a regular table $T_1 \mathbin{|||} T_2$, then the tails of the skeletons of T_1 and T_2 have to coincide.

Assuming $\mathsf{regSkel}\ T_1 = d_1 :: ds_1$ and $\mathsf{regSkel}\ T_2 = d_2 :: ds_2$, then T_1 and T_2 are called *compatible* iff $ds_1 = ds_2$.

[2] We display a list with elements $x_1, \ldots, x_n$ using the notation $\langle x_1, \ldots, x_n \rangle$; we use the infix operator $\frown$ for list concatenation, and also employ the ML list construction notation $x :: xs := \langle x \rangle \frown xs$. The usage of variable names like "xs" for lists is best understood by reading such names as plurals, in this case the plural of "x". The operator "$::$" associates to the right: $x :: y :: ys = x :: (y :: ys)$.

Concatenation of T_1 and T_2 to a regular table $T_1 \mathbin{|||} T_2$ is only possible if they are compatible, and then we have:

$$\mathsf{regSkel}\ (T_1 \mathbin{|||} T_2) = (d_1 \frown d_2) :: ds_1$$

So concatenation of compatible regular tables produces a regular table that shares all lower dimensions with the arguments, and the highest dimension is the concatenation of the highest dimensions of the arguments.

Recently, Parnas suggested to consider "ragged tables", too; for these, concatenation would be restricted only to tables of the same dimension. (Dropping even that restriction would be possible, but would make typing much more complicated.)

If one wanted to define skeletons for ragged tables, too, these would be tree-shaped. However, since the main use of regular table skeletons is for determining legality of concatenation, skeletons are not so important for possibly ragged tables.

Issue 3 Should attention be restricted to regular tables? □

In this paper, we shall allow ragged tables, but we keep the requirement that only tables of equal dimension can be concatenated, and we consider regular tables as a special case of our general table concept. Accordingly, we consider $\mathsf{regSkel}$ as a partial function from tables into regular table skeletons; $\mathsf{regSkel}\ T$ is defined if and only if T is regular.

3 Table Types as Initial Algebras

The compositional table syntax we elaborated in the previous section will be used in sections 4 and 5 to define compositional semantics; the central step in such a development is replacing syntactic constructors with semantic operators. In the case of tables, this means that we replace the table constructors with functions operating on the domains chosen for semantics of tables. Since those semantic operators will be parameters of the overall semantics functions, we will be dealing with *higher-order functions*, i.e., functions that accept functions as arguments. We also will want to "partially apply" multiple-argument functions to only selected arguments, and use the result as a function again.

For all this, a typed setting is useful in order to avoid confusion, and a *functional mathematical language* eases presentation considerably. Such languages are for example polymorphic simple type theory, or polymorphic simply-typed λ-calculus, which is the basis of Higher-Order Logic (HOL) [GM93, NPW02], and typed set theories such as those used in the specification languages Z [Spi89, ISO02] and B [Abr96]. The essential equivalence of the two approaches is shown in [San98]; see also for example [Far03] for arguments in support of using such a language for mathematical exposition. The syntax and type system of these mathematical languages is closely mirrored in functional programming languages like ML [MT91] and Haskell [PJ$^+$03].

In 3.1 we give a short introduction to the polymorphic simply typed λ-calculus we shall be using as our mathematical language.

Both as semantics of tables, and, later, in the formalism defining semantics of tabular expressions, we shall also use relations. Therefore, we include as 3.2 an overview of the relational operators we shall be using; this may be skipped on a first reading.

For more information about typed λ-calculi see for example [Mit96]; for relations, [SS93, BKS97].

Finally, in 3.3, we show how to use the typing framework of 3.1 to give a formal definition of table types as initial algebras in such a way that all grid cells have to have the same type, and for each dimension, all header entries have to have the same type, but the type of grid cells and the types of headers cells in the different dimensions may all be different.

3.1 Functions and Types

In standard polymorphic simply typed λ-calculus, *types* can be

- *base types*, such as $\mathbb{N}$ for the natural numbers, $\mathbb{Z}$ for the integers, $\mathbb{R}$ for the real numbers, or $\mathbb{B}$ for the truth values,
- *type variables* α, β, β_i, ...
- *constructed types*, using type constructors with fixed arity, including
  - *function types* constructed using the binary infix type constructor "$\rightarrow$",
  - *product types* constructed using the binary infix type constructor "$\times$", and
  - *sum types* constructed using the binary infix type constructor "$+$".

If C is an n-ary type constructor without any special fixity, and $t_1, \ldots, t_n$ are types, then the resulting constructed type will be written "$C\ t_1\ \ldots\ t_n$".

The function type constructor associates to the right, i.e., the type of functions with two arguments of types t_1 and t_2 and results of type t_3 may be written in either of the following two ways:

$$t_1 \rightarrow t_2 \rightarrow t_3 \qquad = \qquad t_1 \rightarrow (t_2 \rightarrow t_3)\ .$$

If τ is a *type substitution*, i.e., a partial function from type variables to types, then we write application of τ to some type t as $\tau\ t$.

Well-typed terms are formed in the following ways:

- Variables $x, y, z, \ldots$ may be used at arbitrary types $x : t$, ..., but in a common scope only at one type. (Formally, typing of variables should be handled using so-called contexts, which we omit for the sake of simplicity.)
- For a constant c defined with type t, i.e., for $c : t$, any instance $c : \tau\ t$ is a well-typed term.
- For two well-typed terms $f : t_1 \rightarrow t_2$ and $a : t_1$, *function application* $(f\ \ a) : t_2$ is a well-typed term again.
- For a well-typed term $b : t_2$ and a variable x, that, if it occurs in b, occurs there at type t_1, the λ-abstraction $(\lambda\ x : t_1 \,.\, b) : t_1 \rightarrow t_2$ is a well-typed term again.

λ-abstractions will be needed only occasionally in this paper. They are used to denote "nameless functions"; they also enable conversion of implicit definitions such as "$f(x) = 3 \cdot x + 1$" into explicit definitions "$f = \lambda\ x : \mathbb{R} \,.\, 3 \cdot x + 1$". The parentheses around λ-abstractions are frequently omitted; by convention, the scope of the variable bound by λ extends "as far right as possible", that is, usually to the end of the term or to a closing parenthesis for which the opening parenthesis preceded the λ. Where it is clear from the context, we may omit the type of the bound variable in λ-abstractions.

The parentheses around function applications may be omitted, too; the convention is that function application associates to the left, i.e., the application of a function $f : t_1 \rightarrow t_2 \rightarrow t_3$ to two arguments $a : t_1$ and $b : t_2$ may be written in either of the following two ways:

$$f\ a\ b \qquad = \qquad (f\ \ a)\ b\ .$$

The type $t_1 \rightarrow t_2 \rightarrow t_3$ of functions that take their two arguments separately is the *curried* version of the type of functions that take two arguments in a pair. The two isomorphisms between these types are:

$$\begin{array}{ll} \mathsf{curry} & : (t_1 \times t_2 \rightarrow t_3) \rightarrow (t_1 \rightarrow t_2 \rightarrow t_3) \\ \mathsf{uncurry} & : (t_1 \rightarrow t_2 \rightarrow t_3) \rightarrow (t_1 \times t_2 \rightarrow t_3) \end{array}$$

with

$$\begin{array}{lll} \mathsf{curry} & = & \lambda\, f : t_1 \times t_2 \rightarrow t_3 \;.\; \lambda\, x : t_1 \;.\; \lambda\, y : t_2 \;.\; f\ (x, y) \\ \mathsf{uncurry} & = & \lambda\, f : t_1 \rightarrow t_2 \rightarrow t_3 \;.\; \lambda\, p : t_1 \times t_2 \;.\; f\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \end{array}$$

where $\mathsf{fst} : \alpha \times \beta \rightarrow \alpha$ and $\mathsf{snd} : \alpha \times \beta \rightarrow \beta$ are the product projections.

Infix operators will have curried types: For an operator

$$\otimes : \alpha \rightarrow \beta \rightarrow \gamma \ ,$$

and terms $e_1 : \alpha$, and $e_2 : \beta$, we write $e_1 \otimes e_2$ for the application $(\otimes)\ e_1\ e_2$ of the operator (turned into a function by the parentheses) to its two arguments.

Function application has *higher priority* than any infix operator, i.e., for arbitrary infix operators $\otimes$,

$$g\ a \otimes h\ b \quad = \quad (g\ a) \otimes (h\ b) \ .$$

The central calculation rule of λ-calculi defines how to simplify application of a λ-abstraction:

$$(\lambda\ x\ .\ b)\ a \quad = \quad b[x \leftarrow a] \ ,$$

where $b[x \leftarrow a]$ means the result of substituting a for x in b — substitution may need to rename bound variables in b. (This rule, as a directed rewriting rule, is traditionally called β-reduction.)

Function composition is the infix operator

$$\circ : (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$

with $(f \circ g)\ x = f\ (g\ x)$.

For table typing in 3.3, we shall need an extension to this standard calculus: We add the syntactic category of *type lists*, where

- a *type list variable* $bs, bs_i, \ldots$ is a type list,
- $\langle\,\rangle$ is a type list, and
- if t is a type, and ts is a type list, then $t :: ts$ is a type list,

and we also use list display notation like for object-level lists, for example:

$$\langle t_1, t_2, t_3 \rangle \quad = \quad t_1 :: t_2 :: t_3 :: \langle\,\rangle$$

Then, we allow arguments to type constructors in selected positions to be such type lists, instead of just types. Information about which argument positions take lists of types and not types is part of the extended arity information, which, instead of just a natural number, now is a list of flags, either $\mathbf{t}$ for types, or $\mathbf{l}$ for type lists.

The arity of the function type constructor "$\rightarrow$" therefore is $\langle \mathbf{t}, \mathbf{t} \rangle$.

3.2 Relations and Set Comprehensions

Besides $\rightarrow$ as the function type constructor, we also admit $\leftrightarrow$ as the relation type constructor. A relation $R : A \leftrightarrow B$ can also be seen as a subset of $A \times B$, so set operations become available for relations, and we shall use intersection $\cap$ and union $\cup$ in this way.

For a relation $R : A \leftrightarrow B$, we use the conventional relational infix notation

$$xRy \quad :\Leftrightarrow \quad (x, y) \in R \ ,$$

and for pairs that are elements of relations, we often use an alternative pair notation:

$$x \mapsto y \quad := \quad (x, y) \ .$$

Since the identity relation $\mathbb{I}_A : A \leftrightarrow A$ with $\mathbb{I}_A := \{x : A \bullet x \mapsto x\}$ is total and univalent, it can also be used as a function $\mathbb{I}_A : A \rightarrow A$.

For relation construction, we shall frequently use set comprehension, denoted following the pattern

$$\{\ \mathit{declaration} \mid \mathit{predicate} \bullet \mathit{term}\ \} \ ,$$

meaning the set of all values of *term* under bindings for the locally bound variables from *declaration* that satisfy the *predicate*, for example,

$$\{k : \mathbb{N} \mid k < 4 \bullet k^2\} \quad = \quad \{0, 1, 4, 9\}$$

There are two short forms for special cases: If the *predicate* is true, it can be omitted:

$$\{\ \mathit{declaration} \bullet \mathit{term}\ \} \qquad = \qquad \{\ \mathit{declaration} \mid \mathsf{true} \bullet \mathit{term}\ \} \ ,$$

and if the *term* is just the tuple of all variables in the same sequence as introduced in the *declaration*, then the *term* may be omitted, for example:

$$\begin{aligned} \{\ k : \mathbb{N} \quad &\mid k < 4 \quad \} = \ \{0, 1, 2, 3\} \\ \{\ k : \mathbb{N}, q : \mathbb{Z} &\mid q^2 + k = 1\ \} = \ \{(0, -1), (0, 1), (1, 0)\} \end{aligned}$$

Quantifiers are used with the same pattern as set comprehensions:

$$\begin{array}{lcl} \forall\ \mathit{declaration} \mid \mathit{predicate}_1 \bullet \mathit{predicate}_2 & \Leftrightarrow & \forall\ \mathit{declaration} \bullet \mathit{predicate}_1 \Rightarrow \mathit{predicate}_2 \\ \exists\ \mathit{declaration} \mid \mathit{predicate}_1 \bullet \mathit{predicate}_2 & \Leftrightarrow & \exists\ \mathit{declaration} \bullet \mathit{predicate}_1 \wedge \mathit{predicate}_2 \end{array}$$

We will be using the following additional relational operations:

- *Conversion:* every relation $R : A \leftrightarrow B$ has a *converse* $R^{\smile} : B \leftrightarrow A$, with

$$R^{\smile} = \{x : A;\ y : B \mid xRy \bullet y \mapsto x\} \ .$$

- *(Sequential) composition:* For two relations $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$, their composition $R \mathbin{;} S : A \leftrightarrow C$ is defined as follows:

$$R \mathbin{;} S := \{x : A;\ z : C \mid (\exists\, y : B \bullet xRy \wedge ySz)\}$$

- *Range:* For a relation $R : A \leftrightarrow B$, its range $\mathsf{ran}\ R$ is the set of second components of the pairs in R:

$$\mathsf{ran}\ R = \{y : B \mid (\exists\, x : A \bullet xRy)\}$$

- *Parallel composition:* For two relations $R : A \leftrightarrow B$ and $S : C \leftrightarrow D$, their parallel composition $(R \parallel S) : A \times C \leftrightarrow B \times D$ is defined as follows:

$$(R \parallel S) := \{x_1 : A;\ y_1 : B;\ x_2 : C;\ y_2 : D \mid x_1 R y_1 \wedge x_2 S y_2 \bullet (x_1, x_2) \mapsto (y_1, y_2)\}$$

3.3 Formal Definition and Typing of Tables and Table Constructors

When considering typing for tables, many different approaches are possible.

Here, we propose a very simple typing scheme for tables that is intended first of all to serve as a clarifying instrument for the development of our approach towards table evaluation and semantics in the next few sections.

For a first, intuitive table typing it is rather obvious that for every dimension, all header elements of that dimension should have the same type. Since every cell of the grid is, at the lowest level, combined with a header element of the lowest dimension, all grid cells should have the same type, too.

Therefore, the type of n-dimensional tables may be parameterised by $n+1$ types: one for the grid cells, and, for each dimension, one for the headers of that dimension.

We organise the header types into a type list:

Definition 3.3.1 The *table type constructor* is written $\mathbb{T}$ and has arity $\langle \mathbf{l}, \mathbf{t} \rangle$.

For each type α, the type $\mathbb{T}\ \langle\rangle\ \alpha$ of *zero-dimensional tables*, or *cells, with content type* α is the isomorphic image of α via the cell constructor

$$[\_] \quad : \quad \alpha \quad \rightarrow \quad \mathbb{T}\ \langle\rangle\ \alpha \ .$$

(I.e., $\mathbb{T}\ \langle\rangle\ \alpha$ is the free algebra over the signature containing only the one constructor $[\_]$.)

For each type $\mathbb{T}\ bs\ \alpha$ of n-dimensional tables and each type β, the type $\mathbb{T}\ (\beta :: bs)\ \alpha$ of $(n+1)$-*dimensional tables with header types* $\beta :: bs$ *and cell type* α is the initial algebra generated by the two constructors

$$\begin{array}{lcrclcl} \_ \rhd \_ & : & \beta & \rightarrow & \mathbb{T}\ bs\ \alpha & \rightarrow & \mathbb{T}\ (\beta :: bs)\ \alpha \\ \_ \parallel\!| \_ & : & \mathbb{T}\ (\beta :: bs)\ \alpha & \rightarrow & \mathbb{T}\ (\beta :: bs)\ \alpha & \rightarrow & \mathbb{T}\ (\beta :: bs)\ \alpha \end{array}$$

and the associativity law for $\parallel\!|$. □

This definition introduces the table constructors as constants of the indicated types.

For $\parallel\!|$, one might consider to choose the simpler type $\mathbb{T}\ bs\ \alpha \rightarrow \mathbb{T}\ bs\ \alpha \rightarrow \mathbb{T}\ bs\ \alpha$; however, with that choice, a legal instance of that type would be $\mathbb{T}\ \langle\rangle\ \alpha \rightarrow \mathbb{T}\ \langle\rangle\ \alpha \rightarrow \mathbb{T}\ bs\ \alpha$, and therewith horizontal concatenation of cells would become possible. For this reason we reject this for the body of this paper.

As a result of the above definition, we denote the type of n-dimensional tables with grid cells of type α and headers in the i-th dimension of type β_i for every $i \in \{1, \ldots, n\}$ as

$$\mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha \ ,$$

and have the following bidirectional typing rules for the table constructors:

$$\begin{array}{lcrcl} & & c : \alpha & \Leftrightarrow & [\,c\,] : \mathbb{T}\ \langle\rangle\ \alpha \\ h : \beta_1 & \wedge & t : \mathbb{T}\ \langle \beta_2, \ldots, \beta_n \rangle\ \alpha & \Leftrightarrow & (h \rhd t) : \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha \\ n > 0 & \wedge & t_1, t_2 : \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha & \Leftrightarrow & (t_1 \parallel\!| t_2) : \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha \end{array}$$

Since in most programming languages, parameterisation by a list of types is not possible, we show in the appendix a way to achieve a strongly-typed interface to tables in the functional programming language Haskell that is closely guided by the exposition in this section, but has to accommodate the restrictions of Haskell's type system. Our Isabelle/HOL formalisation uses essentially the same approach.

In the approach followed in the appendix, greater technical elegance is achieved by treating cells as headers over "no information". This makes cells naturally concatenable, but essentially the same method as above can be used to prevent this: imposing a stronger-than-principal typing on operations such as table concatenation easily can restrict those operations to at least one-dimensional tables. However, since there are technical advantages to a homogeneous treatment of cells, we adopt these restrictions neither in the Haskell formalisation in the appendix, nor in the Isabelle/HOL formalisation.

4 Tables as Container Data Structures

The discussion in Sect. 2 showed that n-dimensional tables with $n > 0$ can be seen as non-empty lists of header-subtable pairs, constructed from singletons built via $\triangleright$ and the associative concatenation operator $\|\|$. In the following, we use this view of a table dimension as the basis for defining functions with tables as domain by structural induction.

A different view would consider non-empty lists as constructed from singletons and a "cons" operation adding a single element to a non-empty list. Defining structural induction based on this view would have the advantage that operations replacing "cons" need not be associative, but it would also have the disadvantage that in most cases, the effort for composing the two components of singletons would have to be duplicated for the first-element component of the treatment of the "cons".

That disadvantage would disappear if we started from empty tables, i.e., from empty lists in the individual dimensions. However, the necessity to provide an image for the empty list would exclude (or make unsafe) some functions that can safely be defined on non-empty lists, for example maximum of natural or real numbers. Since we consider this as an important argument against introducing empty tables, we shall stick with the restriction to non-empty tables for the remainder of this paper.

It is actually not hard to convert between the "cons view" of tables and the view starting from $\triangleright$ and $\|\|$, see the appendix section A.2.2 for conversions in the Haskell implementation; the corresponding development has also been performed in the Isabelle formalisation, with all proofs. In terms of expressivity, the two views are therefore completely equivalent. Since the view starting from $\triangleright$ and $\|\|$ is technically simpler and more elegant, we shall not consider the "cons view" any further.

We now consider tables with the structure based on $\triangleright$ and the associative concatenation $\|\|$ as elicited in Sect. 2 and with the typing as defined in Sect. 3 as inductively defined *container data structures*. So a table of type $\mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha$ contains cell elements of type α in the grid and header elements of type β_i in the header of the i-th dimension, just as a list of type $\mathsf{seq}\ \alpha$ contains list elements of type α and is inductively defined over the empty list and list construction "::".

This perspective allows us to define functions over tables by structural induction similar to the way list folding functions (formerly often called "reduce" functions) are defined over lists.

In 4.1 we show how to define such *table folding functions* for single dimensions, and then give some example applications in 4.2. In 4.3 we introduce *table evaluation structures* that allow folding whole tables. In 4.4 and 4.5 we discuss different ways to define table evaluation structures for the semantics of normal tables; both ways are transferred to inverted tables in 4.6. Finally, in 4.7, we discuss the integration of "abbreviated grids", or *nested headers*, into our framework.

4.1 Defining Functions over Tables

Functions mapping tables to values are most naturally defined via structural induction over the structure of tables (this may also be called primitive recursion; in APL and other early functional programming languages, primitive recursion functions over recursive data structures are frequently called "reduce", while in current functional programming languages, "fold" is more common). For now, we consider only the case of n-dimensional tables with $n > 0$: Such a table $t : \mathbb{T}\ (\beta :: bs)\ \alpha$ is constructed from headers of type β and $(n-1)$-dimensional subtables of type $\mathbb{T}\ bs\ \alpha$ using the constructors $\triangleright$ and $|||$ at their original types

$$\begin{array}{lcccccl} \_\triangleright\_ & : & \beta & \to & \mathbb{T}\ bs\ \alpha & \to & \mathbb{T}\ (\beta :: bs)\ \alpha \quad \text{and} \\ \_|||\_ & : & \mathbb{T}\ (\beta :: bs)\ \alpha & \to & \mathbb{T}\ (\beta :: bs)\ \alpha & \to & \mathbb{T}\ (\beta :: bs)\ \alpha \quad . \end{array}$$

For structural induction, the subtable type $\mathbb{T}\ bs\ \alpha$ is replaced by a type σ of subtable results, and the table type $\mathbb{T}\ (\beta :: bs)\ \alpha$ is replaced by the result type γ; the constructors are replaced by arbitrary functions $\otimes$ and $\oplus$ of appropriate types, where, since $|||$ is associative, $\oplus$ needs to be associative, too. The gap between $(n-1)$-dimensional subtables and subtable results is bridged by an additional function q:

Definition 4.1.1 *Table folding* is a second order function

$$\digamma : (\beta \to \sigma \to \gamma) \times (\gamma \to \gamma \to \gamma) \to (\mathbb{T}\ bs\ \alpha \to \sigma) \to (\mathbb{T}\ (\beta :: bs)\ \alpha \to \gamma)\ .$$

Given three functions

- $\otimes : \beta \to \sigma \to \gamma$, the *header combinator*,
- $\oplus : \gamma \to \gamma \to \gamma$, the *combining function*, which has to be associative, and
- $q : \mathbb{T}\ bs\ \alpha \to \sigma$, the *subtable mapping*,

the *table folding function* $(\digamma\ (\otimes, \oplus)\ q) : \mathbb{T}\ (\beta :: bs)\ \alpha \to \gamma$ is defined by

- $\digamma\ (\otimes, \oplus)\ q\ (h \triangleright t) = h \otimes q\ t$,
- $\digamma\ (\otimes, \oplus)\ q\ (t_1 ||| t_2) = \digamma\ (\otimes, \oplus)\ q\ t_1 \oplus \digamma\ (\otimes, \oplus)\ q\ t_2$.

The pair $(\otimes, \oplus)$ will also be called the *combinator pair* of this folding. □

Associativity of $\oplus$ is necessary to make $F := \digamma\ (\otimes, \oplus)\ q$ well-defined, since the second clause of the definition of $\digamma\ (\otimes, \oplus)\ q$ and associativity of $|||$ imply:

$$\begin{aligned} (F\ t_1 \oplus F\ t_2) \oplus F\ t_3 &= F\ (t_1 ||| t_2) \oplus F\ t_3 = F\ ((t_1 ||| t_2) ||| t_3) \\ &= F\ (t_1 |||(t_2 ||| t_3)) = F\ t_1 \oplus F\ (t_2 ||| t_3) = F\ t_1 \oplus (F\ t_2 \oplus F\ t_3) \end{aligned}$$

4.2 Example Applications of Table Folding

As a first example of the use of table folding, we define a single combinator pair $(\otimes_c, \oplus_c)$ for checking both completeness and absence of overlap in the headers of the first dimension of a table where these are Boolean conditions:

$$\begin{array}{lrcl} \otimes_c : \mathbb{B} \to \sigma \to (\mathbb{B} \times \mathbb{B}) & h \otimes_c t & := & (h, \mathsf{false}) \\ \oplus_c : (\mathbb{B} \times \mathbb{B}) \to (\mathbb{B} \times \mathbb{B}) \to (\mathbb{B} \times \mathbb{B}) & (d_1, o_1) \oplus_c (d_2, o_2) & := & (d_1 \vee d_2, o_1 \vee o_2 \vee (d_1 \wedge d_2)) \end{array}$$

One easily checks that $\oplus_c$ is associative; the folding $\digamma\ (\otimes_c, \oplus_c)\ \mathbb{I}$ calculates a pair (d, o) where

- d is the disjunction of all headers of the first dimension; if d is equivalent to true, then the headers of the first dimension are complete;
- o is a Boolean expression that is true for every overlap between headers; if o is equivalent to false, then the headers of the first dimension are non-overlapping.

Results of this folding depend only on the headers of the first dimension and are completely independent of other headers and the cells, which is reflected in the fact that $h \otimes_{\mathrm{c}} t$ does not depend on t.

Many useful auxiliary functions can be defined directly as table folding functions:

- $\mathsf{headers}$ returning the header list of the first dimension of a table:

$$\mathsf{headers} := \digamma\ ((\lambda\ h\ .\ \lambda\ t\ .\ \langle h\rangle), (^\frown))\ \mathbb{I}$$

- hMap and tMap are second-order functions that apply their argument functions to each header, respectively to each n-dimensional subtable, of their $(n+1)$-dimensional argument tables:

$$\begin{array}{lcl} \mathsf{hMap} & : & (\beta \to \beta') \to \mathbb{T}\ (\beta :: bs)\ \alpha \to \mathbb{T}\ (\beta' :: bs)\ \alpha \\ \mathsf{hMap}\ f & = & \digamma\ (\ (\lambda\ h\ t\ .\ f\ h\ \triangleright\ t)\ ,\ |||\ )\ \mathbb{I} = \digamma\ (\triangleright, |||)\ f \\ \mathsf{tMap} & : & (\mathbb{T}\ bs\ \alpha \to \mathbb{T}\ bs'\ \alpha') \to \mathbb{T}\ (\beta :: bs)\ \alpha \to \mathbb{T}\ (\beta :: bs')\ \alpha' \\ \mathsf{tMap}\ g & = & \digamma\ (\ (\lambda\ h\ t\ .\ h\ \triangleright\ g\ t)\ ,\ |||\ )\ \mathbb{I} \end{array}$$

Using tMap, we can lift many functions that act on the first table dimension to higher table dimensions, for example:

- $\mathsf{addH2}$ for adding a single *second-dimension* header to an $(n+1)$-dimensional table, yielding an $(n+2)$-dimensional table, and corresponding functions for adding headers at deeper dimensions:

$$\begin{array}{lll} \mathsf{addH2} : \beta_2 \to \mathbb{T}\ (\beta_1 :: bs)\ \alpha & \to \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha & \mathsf{addH2} := \mathsf{tMap} \circ (\triangleright) \\ \mathsf{addH3} : \beta_3 \to \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha & \to \mathbb{T}\ (\beta_1 :: \beta_2 :: \beta_3 :: bs)\ \alpha & \mathsf{addH3} := \mathsf{tMap} \circ \mathsf{addH2} \end{array}$$

  For examples, we continue to follow the convention (adhered to throughout Sect. 2) that the first-dimension header is drawn on the top of the table, and the second-dimension header to the left. We show an example application of $\mathsf{addH2}$ in Fig. 11 — compare this to $T_{f,\mathrm{aa}} = (y = 10) \triangleright T_j$ in Fig. 3.

$x \geq 0$	$x < 0$
$y + x$	$y - x$

	$x \geq 0$	$x < 0$
$y > 5$	$y + x$	$y - x$

Figure 11: The tables T_k and $\mathsf{addH2}\ (y > 5)\ T_k$

- $\mathsf{delH1} : \mathbb{T}\ (\beta :: bs)\ \alpha \to \mathbb{T}\ bs\ \alpha$ for deleting the first-dimension headers from an $(n+1)$-dimensional table, returning only the first n-dimensional subtable:

$$\mathsf{delH1} := \digamma\ ((\lambda\ h\ .\ \lambda\ t\ .\ t), (\lambda\ t_1\ .\ \lambda\ t_2\ .\ t_1))\ \mathbb{I}$$

  The corresponding functions for higher dimensions are easily defined from this:

$$\begin{array}{lll} \mathsf{delH2} : \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha & \to \mathbb{T}\ (\beta_1 :: bs)\ \alpha & \mathsf{delH2} := \mathsf{tMap}\ \mathsf{delH1} \\ \mathsf{delH3} : \mathbb{T}\ (\beta_1 :: \beta_2 :: \beta_3 :: bs)\ \alpha & \to \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha & \mathsf{delH3} := \mathsf{tMap}\ \mathsf{delH2} \\ \mathsf{delH4} : \mathbb{T}\ (\beta_1 :: \beta_2 :: \beta_3 :: \beta_4 :: bs)\ \alpha & \to \mathbb{T}\ (\beta_1 :: \beta_2 :: \beta_3 :: bs)\ \alpha & \mathsf{delH4} := \mathsf{tMap}\ \mathsf{delH3} \end{array}$$

- $\mathsf{updC0} : \alpha' \to \mathbb{T}\ \langle\,\rangle\ \alpha \to \mathbb{T}\ \langle\,\rangle\ \alpha'$ updates cells in zero-dimensional tables:

$$\mathsf{updC0}\ u\ [\,c\,] = u$$

For higher dimensions, the corresponding functions are again obtained via tMap:

$$\begin{array}{lll} \mathsf{updC1} : \alpha' \to \mathbb{T}\ \langle\beta_1\rangle\ \alpha & \to \mathbb{T}\ \langle\beta_1\rangle\ \alpha' & \mathsf{updC1} := \mathsf{tMap} \circ \mathsf{updC0} \\ \mathsf{updC2} : \alpha' \to \mathbb{T}\ \langle\beta_1, \beta_2\rangle\ \alpha & \to \mathbb{T}\ \langle\beta_1, \beta_2\rangle\ \alpha' & \mathsf{updC2} := \mathsf{tMap} \circ \mathsf{updC1} \\ \mathsf{updC3} : \alpha' \to \mathbb{T}\ \langle\beta_1, \beta_2, \beta_3\rangle\ \alpha & \to \mathbb{T}\ \langle\beta_1, \beta_2, \beta_3\rangle\ \alpha' & \mathsf{updC3} := \mathsf{tMap} \circ \mathsf{updC2} \end{array}$$

In the appendix A.2 we also show how, among others, the following functions can be implemented using standard functional programming techniques in addition to table folding:

- $\mathsf{regSkel}_i$ returns a regular table skeleton for the i outer dimensions of tables that are regular at least in their i outer dimensions — we have $\mathsf{regSkel}_1\ t = \langle\,\mathsf{headers}\ t\,\rangle$,
- *Vertical concatenation* $///$ accepts two regular $(n+2)$-dimensional tables coinciding in the headers of the *first* dimension, and produces a regular result table with the same first-dimension headers, and in the second dimension the concatenation of the second-dimension headers of the argument tables — an example is given in Fig. 12.

	$x \geq 0$	$x < 0$
$y \leq 5$	0	x

	$x \geq 0$	$x < 0$
$y \leq 5$	0	x
$y > 5$	$y + x$	$y - x$

Figure 12: The tables $\mathsf{addH2}\ (y \leq 5)\ T_j$ and $(\mathsf{addH2}\ (y \leq 5)\ T_j) /// (\mathsf{addH2}\ (y > 5)\ T_k)$

The two functions $\mathsf{addH2}$ and $///$ can be used as a combinator pair for the table folding function that directly defines *table transposition* for at least two-dimensional tables that are regular in their two outer dimensions:

$$\begin{array}{lcl} \mathsf{transpose} & : & \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha \to \mathbb{T}\ (\beta_2 :: \beta_1 :: bs)\ \alpha \\ \mathsf{transpose} & := & \mathcal{F}\ (\mathsf{addH2}, (///))\ \mathbb{I} \end{array}$$

Using the formalisation in Isabelle/HOL, we have performed a fully formalised proof of the following:

Lemma 4.2.1 If T is regular in two dimensions with $\mathsf{regSkel}_2\ T = \langle hs_1, hs_2\rangle$, then $\mathsf{transpose}\ T$ is regular, too, and $\mathsf{regSkel}_2\ (\mathsf{transpose}\ T) = \langle hs_2, hs_1\rangle$. □

Higher-dimensional transpositions are easily obtained with the help of tMap:

$$\begin{array}{lcl} \mathsf{transpose3} & : & \mathbb{T}\ (\beta_1 :: \beta_2 :: \beta_3 :: bs)\ \alpha \to \mathbb{T}\ (\beta_3 :: \beta_2 :: \beta_1 :: bs)\ \alpha \\ \mathsf{transpose3} & := & \mathsf{transpose} \circ \mathsf{tMap}\ \mathsf{transpose} \circ \mathsf{transpose} \\ \mathsf{transpose4} & : & \mathbb{T}\ (\beta_1 :: \beta_2 :: \beta_3 :: \beta_4 :: bs)\ \alpha \to \mathbb{T}\ (\beta_4 :: \beta_2 :: \beta_3 :: \beta_1 :: bs)\ \alpha \\ \mathsf{transpose4} & := & \mathsf{transpose} \circ \mathsf{tMap}\ \mathsf{transpose3} \circ \mathsf{transpose} \end{array}$$

4.3 Table Evaluation Structures

For evaluating a whole table, we usually need to replace the subtable mapping q with another table evaluation function; for zero-dimensional tables, that could be induced by an arbitrary function $f : \alpha \to \sigma$ taking cells, and for higher dimensions we would use table folding again.

For example, for fully evaluating a two-dimensional table $t : \mathbb{T}\ \langle \beta_1, \beta_2 \rangle\ \alpha$, we need:

$$
\begin{array}{lll}
f : \alpha \to \gamma_3 & \otimes_2 : \beta_2 \to \gamma_3 \to \gamma_2 & \otimes_1 : \beta_1 \to \gamma_2 \to \gamma_1 \\
 & \oplus_2 : \gamma_2 \to \gamma_2 \to \gamma_2 & \oplus_1 : \gamma_1 \to \gamma_1 \to \gamma_1
\end{array}
$$

to define $\mathcal{F}\ (\otimes_1, \oplus_1)\ (\mathcal{F}\ (\otimes_2, \oplus_2)\ f')\ t$, where $f'\ [\,c\,] = f\ \ c$.

We organise this material into *table evaluation structures*:

Definition 4.3.1 A *table evaluation structure (TES)* for tables of type $\mathbb{T}\ \langle \beta_1, \dots, \beta_n \rangle\ \alpha$ is written

$$w \llparenthesis \langle (\otimes_1, \oplus_1), \dots, (\otimes_n, \oplus_n) \rangle \rrparenthesis f$$

and consists of

- a sequence $\langle \gamma_0, \gamma_1, \dots, \gamma_{n+1} \rangle$ (not explicitly listed, but left implicit) of types,
- a *wrapper* function $w : \gamma_1 \to \gamma_0$,
- a *cell embedding* function $f : \alpha \to \gamma_{n+1}$, and
- a sequence $\langle (\otimes_1, \oplus_1), \dots, (\otimes_n, \oplus_n) \rangle$ of combinator pairs, with

$$
\begin{array}{l}
\otimes_i : \beta_i \to \gamma_{i+1} \to \gamma_i \\
\oplus_i : \gamma_i \to \gamma_i \to \gamma_i \\
\oplus_i \ \ \text{associative}
\end{array}
$$

  for every $i \in \{1, \dots, n\}$.

The type constructor $\mathbb{S}$ for table evaluation structures has arity $\langle \mathbf{l}, \mathbf{t}, \mathbf{t} \rangle$; the type of all table evaluation structures for tables of type $\mathbb{T}\ \langle \beta_1, \dots, \beta_n \rangle\ \alpha$ and with final result type γ_0 is $\mathbb{S}\ \langle \beta_1, \dots, \beta_n \rangle\ \alpha\ \gamma_0$.

TES application $\mathbb{F}$ has the following type:

$$\mathbb{F} :: \mathbb{S}\ bs\ \alpha\ \gamma_0 \to \mathbb{T}\ bs\ \alpha \to \gamma_0\ \ ,$$

and is defined by the following:

- For $S : \mathbb{S}\ \langle\,\rangle\ \alpha\ \gamma_0$ and a table $t : \mathbb{T}\ \langle\,\rangle\ \alpha$, S is of the shape $S = w \llparenthesis \langle\,\rangle \rrparenthesis f$ for some $f : \alpha \to \gamma_1$ and some $w : \gamma_1 \to \gamma_0$, and t is of the shape $t = [\,c\,]$, and we have

$$\mathbb{F}\ \ (w \llparenthesis \langle\,\rangle \rrparenthesis f)\ [\,c\,] = w\ (f\ \ c)$$

- For $n > 0$, a TES $S : \mathbb{S}\ \langle \beta_1, \dots, \beta_n \rangle\ \alpha\ \gamma_0$ and a table $t : \mathbb{T}\ \langle \beta_1, \dots, \beta_n \rangle\ \alpha$, S is of the shape $S = w \llparenthesis (\otimes_1, \oplus_1) :: ps \rrparenthesis f$, and we define:

$$\mathbb{F}\ \ (w \llparenthesis (\otimes_1, \oplus_1) :: ps \rrparenthesis f)\ \ t = w\ \ (\mathcal{F}\ (\otimes_1, \oplus_1)\ \ (\mathbb{F}\ \ (\mathbb{I} \llparenthesis ps \rrparenthesis f)\ \ t)) \qquad \square$$

One might be tempted to achieve a simpler definition by omitting f, i.e., replacing it with the identity on α. However, this would frequently require special treatment for the last $\otimes$ (in the two-dimensional case this would be $\otimes_2$), and such special treatment would make the structure of the whole evaluation function less transparent.

This is nicely demonstrated by a TES for the completeness test for two-dimensional "normal" tables:

$$S_{\text{complete}} := \mathbb{I} \,⦇\, \langle(\wedge,\vee),(\wedge,\vee)\rangle \,⦈\, (\lambda\ x\ .\ \mathsf{true})$$

Here we used a constant function for defining the cell embedding to always return true, which is the neutral element for the header attachment function $\wedge$. Application of S_{complete} to a two-dimensional table with Boolean expressions in all headers therefore returns a Boolean expression denoting the coverage of the whole table — if this is equivalent to true, the table is complete.

Besides the cell embedding f, also the wrapper w could be seen as superfluous in the definition of TESs, but, as we shall see below, it is frequently convenient to think about top-level table composition as separate from whole-table semantics.

4.4 Table Evaluation Structures for Semantics of Normal Tables

As a first exploration of the issues involved in defining table semantics, we now discuss TESs for calculating the meaning of *normal tables*. Normal tables contain predicate expressions in all headers, and terms ("value expressions") in all grid cells. A normal table denotes a function, composed from one partially defined function for every cell, where the domain of that latter function is defined by all the headers governing that cell, and the values in that domain are defined by the cell expression.

The predicate logic formula given for T_f in the introduction can be considered as derived from the table T_f via application of the following TES:

$$S_{\mathsf{N}} \quad := \quad (\forall\, x\, .\, (\forall\, y\, .\, \_))\,⦇\, \langle\ (\Rightarrow,\wedge),\ (\Rightarrow,\wedge)\ \rangle \,⦈\, (f(x,y) = \_)\ \ .$$

Here, we used the underscore "_" in an informal way to let "$(f(x,y) = \_)$" denote a function that maps any expression "E" to the formula (i.e., expression of Boolean type) "$f(x,y) = E$", and similarly for the wrapper: "$(\forall\, x\, .\, (\forall\, y\, .\, \_))$" denotes a function that maps any formula "F" to the formula "$(\forall\, x\, .\, (\forall\, y\, .\, F))$".

However, $⊮\ S_{\mathsf{N}}\ T_f$ is just a formula equivalent to that shown in the introduction; it can *be used to define* a function, but it does not *denote* a function, nor a relation.

For obtaining a function (or relation) from table evaluation, we obviously have to change at least the wrapper. For improved flexibility, we also replace the cell mapping function and define the TES S_{NC}, where "NC" stands for *normal* tables with *conjunctive* composition; the wrapper w_{NC} will be defined below:

$$S_{\mathsf{NC}} \quad := \quad w_{\mathsf{NC}} \,⦇\, \langle\ (\Rightarrow,\wedge),\ (\Rightarrow,\wedge)\ \rangle \,⦈\, (z = \_)\ \ .$$

There is also a different, disjunctive interpretation of normal tables:

$$S_{\mathsf{ND}} \quad := \quad w_{\mathsf{ND}} \,⦇\, \langle\ (\wedge,\vee),\ (\wedge,\vee)\ \rangle \,⦈\, (z = \_)\ \ .$$

This is equivalent to S_{NC} for *complete tables with non-overlapping headers.* If this condition is not satisfied, the two interpretations produce different results — for simplicity, assume wrapper and cell mapping as in S_{N}:

- For *incomplete tables*, the second version could be understood to specify undefinedness outside the coverage of the headers, while the first version makes no statement about the value of f there.

- In the case of *ambiguously overlapping tables*, i.e., existence of differently-valued cell entries for overlaps between headers, the first version gives rise to a contradiction, while the second allows both results.

In either case, one still needs to be careful how the relation induced by the table T_f really is defined, since, when setting the wrapper w_{NC} to the identity, the result of applying the TES S_{NC} to a table like T_f is just a formula F involving the free variables x, y, and z.

Therefore, the task of the wrapper is to convert this formula into a relation. Note that the different wrapper functions w_i defined below are to be understood as functions on *first-order syntax*, which implies that F may be instantiated by an arbitrary formula that may contain free variables, and no bound variables will be renamed on the right-hand sides when F is instantiated. The different wrapper functions given in the following in particular allow us to formalise the discussion of the problematic of giving semantics to "improper" tables in [Zuc96]:

i) $w_1(F) = \{x, y, z : \mathbb{R} \mid F \bullet (x, y) \mapsto z\}$

   (a) With S_{NC}: If T is incomplete, then the result is not univalent; if T is ambiguously overlapping, the result is not total.

   (b) With S_{ND}: If T is incomplete, then the result is not total; if T is ambiguously overlapping, the result is not univalent.

ii) $w_2(F) = \mathsf{any}\{f : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \mid (\forall\, x, y, z : \mathbb{R} \mid F \bullet f(x, y) = z)\}$

   (a) With S_{NC}: If T is incomplete or ambiguously overlapping, then several such f exist.

   (b) With S_{ND}: If T is incomplete, then several such f exist; if T is ambiguously overlapping, no such f exists.

iii) $w_3(F) = \mathsf{any}\{f : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \mid \forall\, x, y : \mathbb{R} \bullet (\exists\, z : \mathbb{R} \mid F \bullet f(x, y) = z)\}$

   (a) With S_{NC}: If T is incomplete, then several such f exist; if T is ambiguously overlapping, no such f exists.

   (b) With S_{ND}: If T is incomplete, then no such f exists (since f is specified to be total); if T is ambiguously overlapping, then several such f exist.

More variants can be considered, including allowing f to be a partial function, or replacing the cell embedding function $(z = \_)$ with $(f(x, y) = \_)$.

This wide array of different possibilities for interpreting incomplete and overlapping tables on the one hand reinforces the old observation that it is safer to restrict oneself to complete and non-overlapping tables; on the other hand it also shows that the function of the wrapper as the final mapping from some formula to the real meaning of the table is important and deserves to be made explicit.

Finally, we would suggest to use w_1 for both w_{NC} and w_{ND} — this way, both S_{NC} and S_{ND}, when applied, always return a relation.

4.5 Table Evaluation Structures for Relational Semantics

In the previous section, all TESs first produced, as result of their folding functions, linear expressions (or formulae) that would then be used by the wrappers to define the "real" semantics of normal tables.

As an alternative, we may follow [DKM01] and directly provide a relational interpretation, for example using the following TES:

$$S_{\mathrm{R1}} \quad := \quad \mathbb{I} \, \langle\!| \, \langle \, ((\mathcal{R}\ \_) \cap \_, \ \cup), \ ((\mathcal{R}\ \_) \cap \_, \ \cup) \, \rangle \, |\!\rangle \, \{x, y : \mathbb{R} \bullet (x, y) \mapsto \_\} \ ,$$

where $\mathcal{R}\ E := \{x, y, z : \mathbb{R} \mid E \bullet (x, y) \mapsto z\}$, and the notation "$(\mathcal{R}\ \_) \cap \_$" stands for the function that maps two arguments h and t to $(\mathcal{R}\ h) \cap t$.

It is immediately obvious how this corresponds to the version given in [DKM01]; our version profits from the homogeneous treatment of both dimensions (and both versions illustrate the need for meta-level function abstraction).

Note that in this version, the variables x and y (and z) are different locally bound variables in every header and grid cell, and the intermediate return types γ_i of the TES S_{R1} all are the relation type $\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{R}$. Furthermore, in the absence of ambiguous overlaps, this produces univalent relations, and for complete tables also total relations — this semantics is equivalent to case (i)(b) from 4.4.

For reconstructing case (i)(a) from 4.4 in the relational semantics, we just have to translate S_{NC} into relational operations, using Boolean implication, which can be defined using relational complement: $R \rightarrow S := \overline{R} \cup S.$,

$$S_{\mathrm{R2}} \quad := \quad \mathbb{I} \, \langle\!| \, \langle \, ((\mathcal{R}\ \_) \rightarrow \_, \ \cap), \ ((\mathcal{R}\ \_) \rightarrow \_, \ \cap) \, \rangle \, |\!\rangle \, \{x, y : \mathbb{R} \bullet (x, y) \mapsto \_\} \ ,$$

With this TES, the defined relation is exactly as in (i)(a) in 4.4.

4.6 Table Evaluation Structures for Semantics of Inverted Tables

In some applications, a table describes a function with only a few different result terms, each valid under different combinations of conditions. For such cases, *inverted tables* have been introduced.

		$x + y$	$x - y$	$y - x$	H_1
H_2	$y \geq 0$	$x < 0$	$0 \leq x < y$	$x \geq y$	G
	$y < 0$	$x < y$	$y \leq x < 0$	$x \geq 0$	

Figure 13: The inverted table T_g

For example, [JPZ97] explains the inverted table T_g drawn in Fig. 13 as corresponding to the following conventional definition:

$$g(x, y) = \begin{cases} x + y & \text{if} \quad (x < 0 \wedge y \geq 0) \qquad \vee (x < y \wedge y < 0) \\ x - y & \text{if} \quad (0 \leq x < y \wedge y \geq 0) \ \vee (y \leq x < 0 \wedge y < 0) \\ y - x & \text{if} \quad (x \geq y \wedge y \geq 0) \qquad \vee (x \geq 0 \wedge y < 0) \end{cases}$$

Comparing this with the semantics of normal tables, we see that, what is inverted here, is the relation between grid and first header in the implication of the first combinator pair (the λ-abstraction here is meta-level, binding variables h and t standing for syntactic expressions):

$$S_{\mathsf{IC}} \quad := \quad w_1 \, \langle\!| \, \langle \, ((\lambda\ \mathsf{h}\ \mathsf{t} \,.\, \mathsf{t} \Rightarrow (z = \mathsf{h})), \wedge), \ (\wedge, \vee) \, \rangle \, |\!\rangle \, \mathbb{I} \ .$$

The combinator pair of the second dimension, however, appears to be taken not from S_{NC}, but from S_{ND} — it is easy to see that this closely corresponds to the above interpretation.

The disjunctive interpretation turns out to be much more naturally related to its version for normal tables — only the position of result extraction changes:

$$S_{\mathsf{ID}} \quad := \quad w_1 \between \langle\ (((z = \_) \wedge \_), \vee),\ (\wedge, \vee)\ \rangle \between \mathbb{I}\ .$$

It is possible to achieve a purely conjunctive interpretation, but this involves second-order functions $\otimes_1$ and $\oplus_2$, and γ_2 is the type of functions from formulae to formulae:

$$S_{\mathsf{ICC}} \quad := \quad w_1 \between \langle\ (\otimes_1, \oplus_1),\ (\otimes_2, \oplus_2)\ \rangle \between \mathbb{I}\ ,$$

with the following combinators:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{e} \otimes_1 \mathsf{c} & = & \mathsf{c}(z = \mathsf{e}) & \mathsf{h} \otimes_2 \mathsf{g} & = & \lambda\, \mathsf{r}\ .\ \mathsf{h} \Rightarrow (\mathsf{g} \Rightarrow \mathsf{r}) \\ \oplus_1 & = & \wedge & \mathsf{p} \oplus_2 \mathsf{q} & = & \lambda\, \mathsf{r}\ .\ \mathsf{p}\ \mathsf{r} \wedge \mathsf{q}\ \mathsf{r} \end{array}$$

For obtaining a direct relation-algebraic interpretation, it is easiest to convert the disjunctive interpretation:

$$S_{\mathsf{IR}} \quad := \quad \mathbb{I} \between \langle\ ((\{x, y : \mathbb{R} \bullet (x, y) \mapsto \_\} \cap \_),\ \cup),\ ((\mathcal{R}\ \_) \cap \_,\ \cup)\ \rangle \between \mathcal{R}\ .$$

In comparison with S_{R1}, the structure is again unchanged, as in [DKM01], but in our approach we had to make it explicit that the result extraction is now in the first header, and the grid cells are interpreted by the vector-building meta-function $\mathcal{R}$.

4.7 Nested Headers

From practical use, tables emerged in which some headers are not just sequences of header elements, but nested, tree-like structures, where only the leaves are associated with indexes into the grid. Such headers are called *abbreviated grids* in [Par92, Sect. 5]; it appears that abbreviated grids have not yet been discussed in any of the other approaches to table semantics.

In this section, we discuss two ways how abbreviated grids could be treated as extensions to our table typing and table folding systems. The first way is inflexible in several respects and does not fit in easily with our typing system, but achieves a clean separation between syntax and semantics. In the second way, semantic information is moved into the headers, but the resulting construction fits easily into our typing and folding systems and will be useful as an extension of table syntax at the user interface.

The structure of a table header constructed using abbreviated grids is that of a node-labelled ordered forest, i.e., a sequence of node-labelled trees where outgoing edges are linearly ordered. We shall use the name *nested headers* for table headers that are potentially constructed from "abbreviated grids".

Among other uses, nested headers provide a simple and intuitive explanation for "indexed tables" — the example of Fig. 8 can be considered as abbreviating the nested-header table from Fig. 14. This, in turn, can be seen as an abbreviation of the conventional table of Fig. 10.

From Parnas' explanation we can derive the following degrees of freedom for headers constructed using abbreviated grids:

i) Obviously, the tree structure resulting from grid abbreviation need not be balanced. This is extensively used in practice in the shape of "decision tree headers".

ii) Less obviously, different abbreviation mechanisms appear to be allowed within a single header.

			$x\%3\ =$			H_1
			0	1	2	
H_2	$y\,\%\,2\ =$	0	0	y^2	$-y^2$	G
		1	x	$x+y$	$x-y$	

Figure 14: A nested-header table corresponding to the "indexed table" of Fig. 8

The first item is unproblematic — in fact, enforcing balanced trees would require additional technical effort.

To realise (ii), however, it is necessary to equip each branch in the tree with information about the abbreviation mechanisms used. This means that information about table interpretation has to be *part of the table itself*, and can no longer be kept separately.

Abbreviated grids fit well into the traditional view that considers a table to consist of n headers and an n-dimensional array of cells. They are, however, not compatible with our inductive view of table construction from cells and header elements via $\rhd$ and $|||$. Therefore, we need a different way to construct tables with nested headers.

Let us first consider free homogeneously abbreviated grids, i.e., header forests without any balancing requirement, and with the *same* abbreviation mechanism for every branch — this rules out the substitution grids described in [Par92, Sect. 5.3]. Also, let us assume that tables constructed via $\rhd$ and $|||$ are considered as special cases of tables with nested headers.

Then, a table $h \rhd t$ has the simplest shape of abbreviated grid header, namely a one-node forest. Concatenation of such tables creates headers that are forests consisting only of one-node trees.

For moving to a second level of trees, we introduce a new table constructor, the *header nesting constructor*, with the following first attempt of a type:

$$\mathrel{\rhd\!\!\!\!\mid} : \eta_1 \to \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha \to \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha$$

Obviously, η_1 should be part of the result table type — one possibility is to generalise table types accordingly. If we allow two type parameters to be included for dimensions with nested headers; then the typing is as follows:

$$\mathrel{\rhd\!\!\!\!\mid} : \eta_1 \to \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha \to \mathbb{T}\ \langle (\beta_1, \eta_1), \ldots, \beta_n \rangle\ \alpha$$

This allows a first level of branching. However, it does not allow further levels — for those we need to be able to use $\mathrel{\rhd\!\!\!\!\mid}$ at a different type:

$$\mathrel{\rhd\!\!\!\!\mid} : \eta_1 \to \mathbb{T}\ \langle (\beta_1, \eta_1), \ldots, \beta_n \rangle\ \alpha \to \mathbb{T}\ \langle (\beta_1, \eta_1), \ldots, \beta_n \rangle\ \alpha$$

If we restricted $\mathrel{\rhd\!\!\!\!\mid}$ to this last type, then the first level of branching can never be achieved, because of the typing of $\rhd$:

$$\_ \rhd \_ : \beta_1 \to \mathbb{T}\ \langle \beta_2, \ldots, \beta_n \rangle\ \alpha \to \mathbb{T}\ \langle \beta_1, \beta_2 \ldots, \beta_n \rangle\ \alpha$$

So either a further constructor needs to be introduced, or at least one of the constructors $\rhd$ and $\mathrel{\rhd\!\!\!\!\mid}$ needs to be overloaded.

Another possibility is to demand $\eta_1 = \beta_1$; this would resolve the typing problems, but would also be quite a strong restriction.

With respect to table folding and table evaluation, this second possibility does not save any effort: in general, no combination of the already supplied functions has the right type to serve as interpretation of $\mathrel{\rhd\!\!\!\!|}$, so for dimensions with nested headers, combinator pairs turn into combinator triples. For direct interpretation of $\mathrel{\rhd\!\!\!\!|}$, we would need a new element of type $\eta_i \rightarrow \gamma_i \rightarrow \gamma_i$.

However, this direct interpretation does not correspond to the original intuition and construction of abbreviated grids. Achieving that structure requires complete redefinition of table folding:

Definition 4.7.1 Given five functions

- $q : \mathbb{T}\ bs\ \alpha \rightarrow \sigma$, the *subtable mapping*,
- $\otimes : \beta \rightarrow \sigma \rightarrow \gamma$, the *header combinator*,
- $\oslash : \eta \rightarrow \beta \rightarrow \beta$, the *header nesting combinator*,
- $\oplus : \gamma \rightarrow \gamma \rightarrow \gamma$, the *combining function*, which has to be associative, and
- $f : \beta \rightarrow \beta$, the *header modification* accumulator function,

the *nested-header table folding function* $\mathsf{tFoldN}_f(\oslash, \otimes, \oplus, q) : \mathbb{T}\ ((\beta, \eta) :: bs)\ \alpha \rightarrow \gamma$ is defined by

- $\mathsf{tFoldN}_f(\oslash, \otimes, \oplus, q)\ (h \rhd t) = f(h) \otimes q(t)$,
- $\mathsf{tFoldN}_f(\oslash, \otimes, \oplus, q)\ (e \mathrel{\rhd\!\!\!\!|} t) = \mathsf{tFoldN}_{\lambda\ b\,.\,f(e \oslash b)}(\oslash, \otimes, \oplus, q)\ t$,
- $\mathsf{tFoldN}_f(\oslash, \otimes, \oplus, q)\ (t_1 \mathbin{|||} t_2) = \mathsf{tFoldN}_f(\oslash, \otimes, \oplus, q)\ (t_1) \oplus \mathsf{tFoldN}_f(\oslash, \otimes, \oplus, q)\ (t_2)$.

The triple $(\oslash, \otimes, \oplus)$ will also be called the *combinator triple* of this folding. □

For table evaluation, tFoldN would be invoked with $f = \mathbb{I}$ at every new dimension, and the adaptation of the definition is straightforward.

Summarising, we see that, although we started out with rather restrictive assumptions about nested headers constructed with $\mathrel{\rhd\!\!\!\!|}$, this combinator still requires very unsatisfactory kludges in the typing system, and adaptations of the whole folding machinery. Its advantage is that it allows a complete separation of syntax and semantics — header nesting in any dimension has an interpretation that is provided by the TES and constant over the whole table.

As indicated above, a different approach would be to let every branch come equipped with information how to combine the headers. This would allow, for instance, to combine conjunction grids with substitution grids within a single header — a very plausible application of that possibility.

In this case, we have a parameterised header nesting constructor — the first argument will be written as a subscript:

$$\rhd\!|_{\_} : (\beta_1' \rightarrow \beta_1'' \rightarrow \beta_1) \rightarrow \beta_1' \rightarrow \mathbb{T}\ \langle \beta_1'', \ldots, \beta_n \rangle\ \alpha \rightarrow \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha$$

For folding over tables constructed with $\rhd\!|$, the original combinator pairs are sufficient again, but we still need the accumulator function:

Definition 4.7.2 Given five functions

- $q : \mathbb{T}\ bs\ \alpha \rightarrow \sigma$, the *subtable mapping*,
- $\otimes : \beta \rightarrow \sigma \rightarrow \gamma$, the *header combinator*,
- $\oplus : \gamma \rightarrow \gamma \rightarrow \gamma$, the *combining function*, which has to be associative, and
- $f : \beta' \rightarrow \beta$, the *header modification* accumulator function,

the *nested-header table folding function* $\mathsf{tFoldH}_f(\otimes, \oplus, q) : \mathbb{T}\ (\beta' :: bs)\ \alpha \rightarrow \gamma$ is defined by

- $\mathsf{tFoldH}_f(\otimes, \oplus, q)\ (h \rhd t) = f(h) \otimes q(t)$,
- $\mathsf{tFoldH}_f(\otimes, \oplus, q)\ (e \rhd\!|_g\ t) = \mathsf{tFoldH}_{\lambda\ b\ .\ f\ (g\ e\ b)}(\otimes, \oplus, q)\ t$,
- $\mathsf{tFoldH}_f(\otimes, \oplus, q)\ (t_1 \mathbin{|||} t_2) = \mathsf{tFoldH}_f(\otimes, \oplus, q)\ (t_1) \oplus \mathsf{tFoldH}_f(\otimes, \oplus, q)\ (t_2)$. $\square$

With this approach, table evaluation needs no adaptation at all, since we may regain the original table folding interface by defining

$$\ulcorner (\otimes, \oplus)\ q := \mathsf{tFoldH}_{\mathbb{I}}(\otimes, \oplus, q)\ .$$

However, it is not hard to see that this equation can also be guaranteed by defining $\rhd\!|$ as a *derived table constructor* from the other table constructors via the following equations:

$$\begin{array}{lcl} e \rhd\!|_g (h \rhd t) & = & g\ e\ h\ \rhd t \\ e \rhd\!|_g (t_1 \mathbin{|||} t_2) & = & (e \rhd\!|_g\ t_1) \mathbin{|||} (e \rhd\!|_g\ t_2) \end{array}$$

This obviously is an instance of header mapping:

$$e \rhd\!|_g\ t = \mathsf{hMap}\ (g\ e)\ t \qquad (*)$$

This also clarifies that the decomposition of the function $g\ e$ into g and e is at least technically completely arbitrary — pragmatically, it will be prudent to choose g from a small set of functions that are easily denoted at branching points in tables.

Instead of considering $\rhd\!|$ as a derived table constructor, the definition $(*)$ can also be considered as basis for the definition of a *table translation* function from tables with nested headers into tables without. This point of view is important when considering tool support and reasoning about tables:

- Tool support for tables should also provide support for abbreviated grids, since these are a widely accepted tool for intuitive table organisation and presentation. Table storage formats and table presentation interfaces therefore need to work with a table data type that provides a separate, primitive header nesting constructor.
- For reasoning about and with tables, and also for inspections that move from grid cell to grid cell, the header nesting structure is irrelevant, and only the expanded headers in each dimension need to be known. Tools supporting these activities will therefore usually work on a table data structure without nested headers.
- Table transformation will usually either work only within an abbreviated grid, or it will consider the abbreviated grid as expanded and will then work over larger parts of the table including the expanded grid. Translation between the two formats makes transformation algorithms easier to formulate.

5 Tabular Expressions

Usually, tables are considered not as data structures containing data elements, but as containing expressions. In addition, a table together with a semantic rule is considered to be an expression again, called *tabular expression*.

The view that tabular expressions sometimes have to be considered as equivalent to *expanded* conventional expressions requires attention to fine nuances in the ways how semantics of tabular expressions is defined; this will be discussed in 5.1. In 5.2, we show how the functional table folding definitions of 4.1 and 4.3 can be generalised to a relational setting. We then introduce two different kinds of tabular expressions in 5.3, employing for one the functional, and for the other the relational table folding mechanisms.

5.1 Tables as Syntactic Devices

Sometimes, tabular expressions are considered to directly denote some semantic object, usually a relation, obtained via a TES $S : \mathbb{S}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha\ \gamma_0$ where the intermediate types $\gamma_0, \ldots, \gamma_{n+1}$ are all semantic domains (for example, domains of relations) and the combinators $\oplus_i : \gamma_i \to \gamma_i \to \gamma_i$ are associative.

Sometimes, however, tabular expressions are considered as *expanding* to equivalent expressions. Let us assume typed expressions, and let $\mathcal{T}_\alpha$ denote the set of terms (or expressions) of type α. The table contained in such a tabular expression then has to be considered as of type $\mathbb{T}\ \langle \mathcal{T}_{\beta_1}, \ldots, \mathcal{T}_{\beta_n} \rangle\ \mathcal{T}_\alpha$. The expansion of such a tabular expression would also most naturally be defined via a TES, now of type $\mathbb{S}\ \langle \mathcal{T}_{\beta_1}, \ldots, \mathcal{T}_{\beta_n} \rangle\ \mathcal{T}_\alpha\ \mathcal{T}_{\gamma_0}$, with combinators $\oplus_i : \mathcal{T}_{\gamma_i} \to \mathcal{T}_{\gamma_i} \to \mathcal{T}_{\gamma_i}$. For example, consider the TES S_{N} (from 4.4) for normal tables. There, $\oplus_1 = \oplus_2 = \wedge$. If we consider $\wedge$ as binary operator on expressions of Boolean type, then it is *not* associative, since $(x \wedge y) \wedge z$ and $x \wedge (y \wedge z)$ are obviously *different expressions*. Although this difference disappears in the semantics, it is relevant on the syntactic level: for software tools, truly associative binary operators, such that e.g. $(x \wedge y) \wedge z = x \wedge y \wedge z = x \wedge (y \wedge z)$, need to be supported by list structures. However, mechanised syntax frequently only supports operators with fixed arity, and these, as constructors of (usually free) expression types, are by definition *not* associative.

Therefore, tabular expressions that expand into conventional expressions need the following:

- a semantics of the expression language,
- a relaxed table folding definition that allows combinators that are only *semantically associative*,
- an accordingly relaxed TES that defines the expansion into equivalent expressions.

In some sense one might consider the semantics as only a part of the wrapper of the TES. However, since the semantics justifies the combinators on all levels of the TES, its rôle is more pervasive.

Since the semantics is usually "understood" throughout any particular context of table use, this issue can be ignored by most table users, and tabular expressions only need to indicate the TES (and, for TESs with different combinator pairs for different dimensions, the sequence of the headers).

Such tabular expressions then can be used as subexpressions inside larger expressions, as for example in Fig. 15 — note that the tabular expression there denotes a universally quantified formula since it uses the TES S_{N} (page 21), and not one of the TESs S_{NC} (page 21) or S_{R2} (page 23) that would make it denote a relation.

$\boxed{S_{\mathsf{N}}}$	$y = 10$	$y > 10$	$y < 10$	H_1
H_2: $x \geq 0$	0	y^2	$-y^2$	$\Rightarrow \quad f(-3, 45) = 42$
H_2: $x < 0$	x	$x + y$	$x - y$	

Figure 15: A tabular expression involving T_f and the TES S_{N} in context

The universally quantified formula quoted from [JPZ97] on page 4 as one presentation of the meaning of T_f uses $\wedge$ as a "syntactically associative" operator, without parentheses around any parts of the six-element conjunction there.

5.2 Relational Table Folding

If we wish to provide mechanisations of tables supported by tools that do not provide syntactically associative operators, we have two options:

- We could prefer one structuring into binary operator applications over all others. As essentially an instance of this, we might switch to the "cons view" discussed on page 16.
- We can leave open the choice of structuring into binary operator applications.

Since it is less restrictive, we adapt the latter approach; this turns the folding functions of Def. 4.1.1 into *folding relations* relating each table with possibly multiple results, according to the different ways to decompose it via $|||$. In the relation algebraic formulae in the following, we use the uncurried versions of the table constructors which are functions (i.e., total and univalent relations) of the following types:

$$\begin{array}{lclcl} \mathsf{uncurry}\ (\triangleright) & =: & \mathsf{addH} & : & \beta \times \mathbb{T}\ bs\ \alpha \to \mathbb{T}\ (\beta :: bs)\ \alpha \\ \mathsf{uncurry}\ (|||) & =: & \mathsf{hConc} & : & \mathbb{T}\ (\beta :: bs)\ \alpha \times \mathbb{T}\ (\beta :: bs)\ \alpha \to \mathbb{T}\ (\beta :: bs)\ \alpha \end{array}$$

We also know that addH is injective, but hConc is not, because of its associativity, and the two constructors are jointly surjective, but with non-overlapping ranges, i.e.,

$$\mathsf{ran\ addH} \cap \mathsf{ran\ hConc} = \varnothing \qquad \text{and} \qquad \mathsf{ran\ addH} \cup \mathsf{ran\ hConc} = \mathbb{T}\ (\beta :: bs)\ \alpha\ .$$

Definition 5.2.1 Given three relations

- $Q : \mathbb{T}\ bs\ \alpha \leftrightarrow \sigma$, the *subtable interpretation*,
- $H : \beta \times \sigma \leftrightarrow \gamma$, the *header combinator*, and
- $C : \gamma \times \gamma \leftrightarrow \gamma$, the *combining relation*, which has to be total,

the *table folding relation* $\mathsf{tFoldR}\ (H, C)\ Q : \mathbb{T}\ (\beta :: bs)\ \alpha \leftrightarrow \gamma$ is defined by

- $(h \triangleright t, v) \in \mathsf{tFoldR}\ (H, C)\ Q$ iff there exist a $s : \sigma$ and a $c : \gamma$ such that $(t, s) \in Q$ and $((h, s), v) \in H$, or, equivalently,

$$\mathsf{addH} \mathbin{;} \mathsf{tFoldR}\ (H, C)\ Q = (\mathbb{I}\ \|\ Q) \mathbin{;} H$$

- $(t_1 ||| t_2, v) \in \mathsf{tFoldR}\ (H, C)\ Q$ iff there exist $t_1', t_2' : \mathbb{T}\ (\beta :: bs)\ \alpha$ and $v_1, v_2 : \gamma$ such that $(t_1 ||| t_2) = (t_1' ||| t_2')$ and $(t_1', v_1) \in \mathsf{tFoldR}\ (H, C)\ Q$ and $(t_2', v_2) \in \mathsf{tFoldR}\ (H, C)\ Q$ and $((v_1, v_2), v) \in C$, or, equivalently,

$$\mathsf{hConc} \mathbin{;} \mathsf{tFoldR}\ (H, C)\ Q = \mathsf{hConc} \mathbin{;} \mathsf{hConc}^{\smile} \mathbin{;} (\mathsf{tFoldR}\ (H, C)\ Q\ \|\ \mathsf{tFoldR}\ (H, C)\ Q) \mathbin{;} C$$

The pair (H, C) will also be called the *combinator pair* of this folding. □

Due to the above-mentioned properties of addH and hConc, the two equations can be merged into the following:

$$\begin{array}{lcl} \mathsf{tFoldR}\ (H, C)\ Q & = & \mathsf{addH}^{\smile} \mathbin{;} (\mathbb{I}\ \|\ Q) \mathbin{;} H\ \cup \\ & & \mathsf{hConc}^{\smile} \mathbin{;} (\mathsf{tFoldR}\ (H, C)\ Q\ \|\ \mathsf{tFoldR}\ (H, C)\ Q) \mathbin{;} C \end{array}$$

Standard reasoning, using the facts that C is total and that $\mathsf{hConc}^{\smile} \mathbin{;} \mathsf{fst}$ and $\mathsf{hConc}^{\smile} \mathbin{;} \mathsf{snd}$ are both Noetherian, shows that the equation

$$X = \mathsf{addH}^{\smile} \mathbin{;} (\mathbb{I}\ \|\ Q) \mathbin{;} H\ \cup\ \mathsf{hConc}^{\smile} \mathbin{;} (X\ \|\ X) \mathbin{;} C$$

has a unique solution, so $\mathsf{tFoldR}\ (H, C)\ Q$ is well-defined as the solution to this equation. (If we allowed C to be non-total, we would use the least solution.)

Note that by allowing H, C, and Q to be possibly non-univalent relations we only performed a natural generalisation; the move to relations is motivated by the necessity to allow non-associative C, which, in general, destroys univalence of folding over the associative table concatenation $|||$.

Lemma 5.2.2 If H, C, and Q are univalent and C is associative, then $\mathsf{tFoldR}\ (H, C)\ Q$ is univalent. □

This relational approach is easily extended to TESs:

Definition 5.2.3 A *relational table evaluation structure (RTES)* for n-dimensional tables of type $\mathbb{T}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha$ is written

$$W \Lbag \langle(H_1, C_1), \ldots, (H_n, C_n)\rangle \Rbag F$$

and consists of

- a sequence $\langle\gamma_0, \gamma_1, \ldots, \gamma_{n+1}\rangle$ (not explicitly listed, but left implicit) of types,
- a *wrapper* relation $W : \gamma_1 \leftrightarrow \gamma_0$,
- a *cell embedding* relation $F : \alpha \leftrightarrow \gamma_{n+1}$, and
- a sequence $\langle(H_1, C_1), \ldots, (H_n, C_n)\rangle$ of relational combinator pairs, with

  $$H_i : \beta_i \times \gamma_{i+1} \leftrightarrow \gamma_i$$
  $$C_i : \gamma_i \times \gamma_i \leftrightarrow \gamma_i$$

  for every $i \in \{1, \ldots, n\}$.

The set of all relational table evaluation structures for tables of type $\mathbb{T}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha$ and with first result type γ_0 is written $\mathbb{S}_{\mathsf{R}}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha\ \gamma_0$.

Application of an RTES $S : \mathbb{S}_{\mathsf{R}}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha\ \gamma_0$ is written $\mathsf{RTeval}\ S$ and is a relation from tables to results:

$$(\mathsf{RTeval}\ S) : \mathbb{T}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha \leftrightarrow \gamma_0\ \ .$$

RTES application is defined by the following:

- For $S : \mathbb{S}_{\mathsf{R}}\ \langle\,\rangle\ \alpha\ \gamma_0$, S is of the shape $S = W \Lbag \langle\,\rangle \Rbag F$ for some $F : \alpha \leftrightarrow \gamma_1$ and some $W : \gamma_1 \leftrightarrow \gamma_0$; defining $\mathsf{cell} : \alpha \to \mathbb{T}\ \langle\,\rangle\ \alpha$, by $\mathsf{cell}(c) = [\,c\,]$, we have

  $$\mathsf{RTeval}\ (W \Lbag \langle\,\rangle \Rbag F) = \mathsf{cell}^{\smile} ; F ; W$$

- For $n > 0$, a TES $S : \mathbb{S}_{\mathsf{R}}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha\ \gamma_0$ and a table $t : \mathbb{T}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha$, S is of the shape $S = W \Lbag (H_1, C_1) :: ps \Rbag F$, and we define:

  $$\mathsf{RTeval}\ (W \Lbag (H_1, C_1) :: ps \Rbag F) = \ (\mathsf{tFoldR}\ (H_1, C_1)\ (\mathsf{RTeval}\ (\mathbb{I} \Lbag ps \Rbag F))) ; W$$

An RTES $W \Lbag \langle(H_1, C_1), \ldots, (H_n, C_n)\rangle \Rbag F$ is called *associative* if all the C_i are associative, and *univalent* if in addition W and F and all H_i and C_i are univalent relations; it is called *total* if W and F and all H_i and C_i are total relations. □

Here we see that allowing Q to be non-univalent in $\mathsf{tFoldR}\ (H, C)\ Q$ is actually essential to enable multi-dimensional relational table folding.

5.3 Tabular Expressions

Motivated by the discussion in 5.1, we introduce two kinds of tabular expressions, which will be defined formally below:

- An *expanding tabular expression*

  $$\boxed{S, V}\ulcorner T \lrcorner$$

  consists of an RTES S, a "semantics" V, and a table T. Such an expanding tabular expression has to be understood as standing for any one of the elements associated with T by $\mathsf{RTeval}\ S$ — this usually will be a syntactic expression — in a context where this element will be interpreted via the semantics V, which ensures that all those elements are equivalent.

- A *tabular expression with direct semantics*

  $$\boxed{\boxed{S}}\ulcorner T \lrcorner$$

  consists of a TES S and a table T. Such a tabular expression directly stands for $\mathbb{F}\ S\ T$, which normally will be an abstract mathematical object and not a syntactic expression, since otherwise the necessary associativity of the combining functions in the TES S can usually not be guaranteed.

To illustrate the difference, let $\mathsf{FV}(E)$ denote the set of free variables of the expression E, and consider the following:

$$\mathsf{FV}\left(\boxed{S_{\mathsf{N}}, (\models \_)}\ulcorner T_f \lrcorner\right) = \{f\}$$

since the argument of FV here is (a rearranged and more parenthesised version of) the formula on page 4.

The tabular expression here is *expected* to be used in a context that ultimately views this formulae as argument of a validity judgement via the provided semantics function $(\models \_)$; Fig. 15 should be understood to contain this tabular expression in such a context. The context it occurs in here, however, is $\mathsf{FV}(\_)$, so for the statement above it is necessary to show that each formula related with T_f via $\mathsf{RTeval}\ S_{\mathsf{N}}$ has the same set of free variables.

With a different RTES, we obtain:

$$\mathsf{FV}\left(\boxed{S_{\mathsf{NC}}, \llbracket \_ \rrbracket}\ulcorner T_f \lrcorner\right) = \{\}\ ,$$

since the set comprehension expressions resulting from applying S_{NC} (considered as an RTES producing expressions) to the table T_f have no free variables.

With respect to FV and the supplied semantics, the same applies as above — here, we supplied a "standard semantics" $\llbracket \_ \rrbracket$ that will map each set comprehension expression to a set (which would be, in this case, a relation).

Finally,

$$\mathsf{FV}\left(\boxed{\boxed{S_{\mathsf{NC}}}}\ulcorner T_f \lrcorner\right)$$

is *nonsense*, since here, S_{NC} is understood as a TES producing a relation, not an RTES producing expressions, and therefore FV is applied to a relation, and not an expression.

For evaluation structures S that, like S_{NC}, can be interpreted both as TES and as RTES, with the two views linked by the supplied standard semantics, this standard semantics also establishes the link between the two kinds of tabular expressions:

$$\boxed{\boxed{S}}\ulcorner T \lrcorner = \llbracket\ \boxed{S, \llbracket \_ \rrbracket}\ulcorner T \lrcorner\ \rrbracket$$

Definition 5.3.1 A *tabular expression with direct semantics*

$$\boxed{\boxed{S}}\,\ulcorner T \urcorner$$

consists of:

- a TES $S : \mathbb{S}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha\ \gamma_0$, and
- a table $T : \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha$.

Its value is $\mathbb{F}\ \ S\ \ T$, and it is of type γ_0. □

As an example of an occurrence of a tabular expression with direct semantics in tabular notation we show the table T_f together with the TES S_{NC} for standard normal table semantics in Fig. 16.

$f :=$		$\boxed{\boxed{S_{\mathsf{NC}}}}$	$y = 10$	$y > 10$	$y < 10$	H_1
	H_2	$x \geq 0$	0	y^2	$-y^2$	
		$x < 0$	x	$x + y$	$x - y$	

Figure 16: Using a tabular expression with direct semantics

Definition 5.3.2 An *expanding tabular expression*

$$\boxed{S, V}\,\ulcorner T \urcorner$$

consists of:

- an RTES $S : \mathbb{S}_{\mathsf{R}}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha\ \gamma_0$,
- a total *semantics function* $V : \gamma_0 \rightarrow \gamma$ such that $\mathsf{RTeval}\ S \mathbin{;} V$ is univalent, and
- a table $T : \mathbb{T}\ \langle \beta_1, \ldots, \beta_n \rangle\ \alpha$.

Such an expanding tabular expression should only occur in contexts where it is intended to be interpreted by V.

It is of type γ_0 and can be replaced by any $E : \gamma_0$ for which $(T, E) \in \mathsf{RTeval}\ S$.

Its *canonic value* is $(\mathsf{RTeval}\ S \mathbin{;} V)\ (T)$.

Where V is understood from the context, we just write $\boxed{S}\,\ulcorner T \urcorner$. □

An expanding tabular expression in tabular notation has already been shown in Fig. 15.

6 Table Transformation

Transformations of tabular expressions can now be proven correct by showing equations about values of tabular expressions involving tables related by syntactic transformation functions.

For all theorems in this section we have performed fully formal proofs of the corresponding statements in our Isabelle formalisation. In fact, we used the interactive theorem prover Isabelle to extract the preconditions for the combinators in each case by incremental proof attempts.

For each of the three cases transposition, collapsing, and inversion we first present a general theorem that contains very weak assumptions about the combinator pairs for the TESs before and after transformation, and then a more restricted theorem with stronger assumptions, including stronger typing assumptions. The table transformation theorems from the literature are special cases of those restricted theorems, instantiated with appropriate TESs for, mostly, normal and inverted tables.

6.1 Table Transposition

The function $\mathsf{transpose} : \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha \to \mathbb{T}\ (\beta_2 :: \beta_1 :: bs)\ \alpha$ for transposition of the top two dimensions of regular tables has been introduced on page 19. We first show

Theorem 6.1.1 (Transposition of Tabular Expressions)
Let an identity-wrapped table evaluation structure $(\mathbb{I}_{\gamma_3} \llparenthesis\, ps \,\rrparenthesis\, q) : \mathbb{S}\ \gamma_3\ bs\ \alpha$, and combinators with the following types be given:

$$\begin{array}{lll}
 & \otimes_1 : \beta_1 \to \gamma_2 \to \gamma_1 & \otimes_2 : \beta_2 \to \gamma_3 \to \gamma_2 \\
w_1 : \gamma_1 \to \gamma_0 & \oplus_1 : \gamma_1 \to \gamma_1 \to \gamma_1 & \oplus_2 : \gamma_2 \to \gamma_2 \to \gamma_2 \\
 & \otimes_4 : \alpha \to \gamma_5 \to \gamma_4 & \otimes_5 : \beta_2 \to \gamma_6 \to \gamma_5 \\
w_2 : \gamma_4 \to \gamma_0 & \oplus_4 : \gamma_4 \to \gamma_4 \to \gamma_4 & \oplus_5 : \gamma_5 \to \gamma_5 \to \gamma_5
\end{array}$$

Assume further that there exists an associative operation $\odot : \gamma_0 \to \gamma_0 \to \gamma_0$, and that the following laws are satisfied (for all $h_1 : \beta_1;\ h_2 : \beta_2;\ x_1, x_2 : \gamma_1;\ y_1, y_2 : \gamma_2;\ z : \gamma_3;\ u_1, u_2 : \gamma_3;\ v_1, v_2 : \gamma_4;\ r_1, r_2, r_3, r_4 : \gamma_0$):

$$\begin{array}{rcl}
w_1\ (h_1 \otimes_1 (h_2 \otimes_2 z)) & = & w_2\ (h_2 \otimes_3 (h_1 \otimes_4 z)) \\
w_1\ (x_1 \oplus_1 x_2) & = & (w_1\ x_1) \odot (w_1\ x_2) \\
w_2\ (u_1 \oplus_1 u_2) & = & (w_2\ u_1) \odot (w_2\ u_2) \\
w_1\ (h_1 \otimes_1 (y_1 \oplus_2 y_2)) & = & (w_1\ (h_1 \otimes_1 y_1)) \odot (w_1\ (h_1 \otimes_1 y_2)) \\
w_2\ (h_2 \otimes_3 (v_1 \oplus_4 v_2)) & = & (w_2\ (h_2 \otimes_3 v_1)) \odot (w_2\ (h_2 \otimes_3 v_2)) \\
(r_1 \odot r_2) \odot (r_3 \odot r_4) & = & (r_1 \odot r_3) \odot (r_2 \odot r_4)
\end{array}$$

Then for every table $T : \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha$, transposition is value-preserving in the following way:

$$\boxed{\boxed{w_1 \llparenthesis\, (\otimes_1, \oplus_1) :: (\otimes_2, \oplus_2) :: ps \,\rrparenthesis\, q}} \llbracket T \rrbracket \quad = \quad \boxed{\boxed{w_2 \llparenthesis\, (\otimes_3, \oplus_3) :: (\otimes_4, \oplus_4) :: ps \,\rrparenthesis\, q}} \llbracket \mathsf{transpose}\ T \rrbracket \qquad \square$$

If transposition is expected to be reflected directly in the TESs before and after evaluation, i.e., if only the top two combinator pairs should be swapped, then the top three intermediate types coincide and the relationship between the two combinator pairs needs to be very close:

Theorem 6.1.2 (Restricted Transposition of Tabular Expressions)
Let a TES $(w \llparenthesis \langle\ (\otimes_1, \oplus_1),\ (\otimes_2, \oplus_2)\ \rangle \rrparenthesis f) : \mathbb{S}\ \gamma_0\ \langle \beta_1, \beta_2 \rangle\ \alpha$, consisting of the following combinators and functions be given:

$$\begin{array}{lll}
\otimes_1 : \beta_1 \to \gamma_1 \to \gamma_1 & \otimes_2 : \beta_2 \to \gamma_1 \to \gamma_1 & w : \gamma_1 \to \gamma_0 \\
\oplus_1 : \gamma_1 \to \gamma_1 \to \gamma_1 & \oplus_2 : \gamma_1 \to \gamma_1 \to \gamma_1 & f : \alpha \to \gamma_1
\end{array}$$

If these combinators satisfy the following laws:

$$\begin{array}{rcl}
x \otimes_1 (y \otimes_2 z) & = & y \otimes_2 (x \otimes_1 z) \\
x \otimes_1 (y \oplus_2 z) & = & (x \otimes_1 y) \oplus_2 (x \otimes_1 z) \\
x \otimes_2 (y \oplus_1 z) & = & (x \otimes_2 y) \oplus_1 (x \otimes_2 z) \\
(x_1 \oplus_1 y_1) \oplus_2 (x_2 \oplus_1 y_2) & = & (x_1 \oplus_2 x_2) \oplus_1 (y_1 \oplus_2 y_2)
\end{array}$$

then transposition of tabular expressions built from tables $T : \mathbb{T}\ \langle\beta_1, \beta_2\rangle\ \alpha$ and this TES is value-preserving in the following way:

$$\boxed{\boxed{w \llparenthesis \langle\ (\otimes_1, \oplus_1),\ (\otimes_2, \oplus_2)\ \rangle \rrparenthesis f}} \ulcorner T \urcorner = \boxed{\boxed{w \llparenthesis \langle\ (\otimes_2, \oplus_2),\ (\otimes_1, \oplus_1)\ \rangle \rrparenthesis f}} \ulcorner \mathsf{transpose}\ T \urcorner \qquad \square$$

6.2 Collapsing of Table Dimensions

We define another primitive table transformation $\mathsf{collapse}$ that collapses the two outermost dimensions of its argument table, combining the two headers associated with each sub-subtable together into a single header using a given combining function f:

$$\begin{array}{lcl} \mathsf{collapse} & : & (\beta_1 \rightarrow \beta_2 \rightarrow \beta) \rightarrow \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha \rightarrow \mathbb{T}\ (\beta :: bs)\ \alpha \\ \mathsf{collapse}\ f & = & \digamma\ (\mathsf{hMap} \circ f, |||)\ \mathbb{I} \end{array}$$

This gives again rise to a value-preserving table transformation:

Theorem 6.2.1 (Collapsing two Dimensions of Tabular Expressions)
Let an identity-wrapped table evaluation structure $\mathbb{I} \llparenthesis ps \rrparenthesis q : \mathbb{S}\ \gamma_3\ bs\ \alpha$ and combinators with the following types be given:

$$\begin{array}{lll} \otimes_1 : \beta_1 \rightarrow \gamma_2 \rightarrow \gamma_1 & \otimes_2 : \beta_2 \rightarrow \gamma_3 \rightarrow \gamma_2 & \otimes_3 : \beta \rightarrow \gamma_3 \rightarrow \delta_1 \\ \oplus_1 : \gamma_1 \rightarrow \gamma_1 \rightarrow \gamma_1 & \oplus_2 : \gamma_2 \rightarrow \gamma_2 \rightarrow \gamma_2 & \oplus_3 : \delta_1 \rightarrow \delta_1 \rightarrow \delta_1 \\ w_1 : \gamma_1 \rightarrow \gamma_0 & w_2 : \delta_1 \rightarrow \gamma_0 & \odot : \beta_1 \rightarrow \beta_2 \rightarrow \beta \end{array}$$

Assume further that there exist additional combinators

$$\oplus_4 : \gamma_0 \rightarrow \gamma_0 \rightarrow \gamma_0 \qquad \oplus_5 : \gamma_0 \rightarrow \gamma_0 \rightarrow \gamma_0$$

and that the following laws are satisfied:

$$\begin{array}{rcl} w_1(h_1 \otimes_1 (h_2 \otimes_2 x)) & = & w_2((h_1 \odot h_2) \otimes_3 x) \\ w_1(h \otimes_1 (x \oplus_2 y)) & = & (w_1(h \otimes_1 x)) \oplus_4 (w_1(h \otimes_1 y)) \\ w_1(x \oplus_1 y) & = & (w_1\ x) \oplus_5 (w_1\ y) \\ w_2(x \oplus_3 y) & = & (w_2\ x) \oplus_4 (w_2\ y) \\ w_2(x \oplus_3 y) & = & (w_2\ x) \oplus_5 (w_2\ y) \end{array}$$

(The last two laws strongly suggest that $\oplus_4$ and $\oplus_5$ will coincide in most cases, but technically this is not necessary.)
Then for every table $T : \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha$, collapsing is value-preserving in the following way:

$$\boxed{\boxed{w_1 \llparenthesis (\otimes_1, \oplus_1) :: (\otimes_2, \oplus_2) :: ps \rrparenthesis q}} \ulcorner T \urcorner = \boxed{\boxed{w_2 \llparenthesis (\otimes_3, \oplus_3) :: ps \rrparenthesis q}} \ulcorner \mathsf{collapse}\ (\odot)\ T \urcorner \qquad \square$$

With identity wrappers, the outer combinations $\oplus_1$ and $\oplus_3$ have to coincide; the following theorem is further simplified by considering, for the table that is to be collapsed. an only two-dimensional TES with identity as cell embedding:

Theorem 6.2.2 (Collapsing two Dimensions of Id-Wrapped Tabular Expressions) Let combinators with the following types be given:

$$\begin{array}{lll} \otimes_1 : \beta_1 \to \gamma_2 \to \gamma_1 & \otimes_2 : \beta_2 \to \mathbb{T}\ bs\ \alpha \to \gamma_2 & \otimes_3 : \beta \to \mathbb{T}\ bs\ \alpha \to \gamma_1 \\ \oplus_1 : \gamma_1 \to \gamma_1 \to \gamma_1 & \oplus_2 : \gamma_2 \to \gamma_2 \to \gamma_2 & \odot : \beta_1 \to \beta_2 \to \beta \end{array}$$

If these combinators satisfy the following laws:

$$\begin{array}{rcl} h_1 \otimes_1 (h_2 \otimes_2 x) & = & (h_1 \odot h_2) \otimes_3 x \\ h \otimes_1 (x \oplus_2 y) & = & (h \otimes_1 x) \oplus_1 (h \otimes_1 y) \end{array}$$

then for every table $T : \mathbb{T}\ (\beta_1 :: \beta_2 :: bs)\ \alpha$, collapsing is value-preserving in the following way:

$$\boxed{\mathbb{I} \langle\!\!| \langle (\otimes_1, \oplus_1), (\otimes_2, \oplus_2) \rangle |\!\!\rangle \mathbb{I}} \ulcorner T \lrcorner \quad = \quad \boxed{\mathbb{I} \langle\!\!| \langle (\otimes_3, \oplus_1) \rangle |\!\!\rangle \mathbb{I}} \ulcorner \mathsf{collapse}\ (\odot)\ T \lrcorner \qquad \square$$

6.3 Table Inversion

Table inversion is a transformation that converts normal tables into inverted tables; for a nice motivational example see [SZP96].

On the type level, inversion functions swap some header type with the cell type; without loss of generality, we restrict ourselves here to swapping the *first* header type:

$$\begin{array}{llll} \mathsf{inverse1} : & & \mathbb{T}\ \langle \beta_1 \rangle\ \alpha & \to \mathbb{T}\ \langle \alpha \rangle\ \beta_1 \\ \mathsf{inverse2} : & \beta_1 \to & \mathbb{T}\ \langle \beta_1, \beta_2 \rangle\ \alpha & \to \mathbb{T}\ \langle \alpha, \beta_2 \rangle\ \beta_1 \\ \mathsf{inverse3} : & \beta_1 \to & \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3 \rangle\ \alpha & \to \mathbb{T}\ \langle \alpha, \beta_2, \beta_3 \rangle\ \beta_1 \\ \mathsf{inverse4} : & \beta_1 \to & \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3, \beta_4 \rangle\ \alpha & \to \mathbb{T}\ \langle \alpha, \beta_2, \beta_3, \beta_4 \rangle\ \beta_1 \\ \vdots & : & & \end{array}$$

Since in previous work, only regular tables are considered, the result of inversion as considered in the literature is also "automatically" constrained to be regular. With the usual semantics of normal and inverted tables, an inversion producing regular tables can only be semantics-preserving for arbitrary regular normal tables if for each original grid cell from the normal table, the subtable governed by the contents of that cell in the inverted table has only a single entry corresponding to its original first-dimension header cell, and all other entries are "empty" in a semantic sense to be made precise below. With the usual semantics of normal and inverted tables, these "empty" cells are filled with "false". Since we define the syntactic transformation of inversion independent of semantics, we need to supply those "empty" entries as an explicit argument to the inversion functions, except for inversion of one-dimensional tables, where the respective subtables contain only a one-element grid.

These first-header inversion functions can be transformed into the other cases by composition with table transpositions, for example:

$$\begin{array}{llll} \mathsf{inverse2\_2} : & \beta_2 \to & \mathbb{T}\ \langle \beta_1, \beta_2 \rangle\ \alpha & \to \mathbb{T}\ \langle \beta_1, \alpha \rangle\ \beta_2 \\ \mathsf{inverse3\_2} : & \beta_2 \to & \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3 \rangle\ \alpha & \to \mathbb{T}\ \langle \beta_1, \alpha, \beta_3 \rangle\ \beta_2 \\ \mathsf{inverse3\_3} : & \beta_3 \to & \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3 \rangle\ \alpha & \to \mathbb{T}\ \langle \beta_1, \beta_2, \alpha \rangle\ \beta_3 \\ \mathsf{inverse4\_2} : & \beta_2 \to & \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3, \beta_4 \rangle\ \alpha & \to \mathbb{T}\ \langle \beta_1, \alpha, \beta_3, \beta_4 \rangle\ \beta_2 \\ \mathsf{inverse4\_3} : & \beta_3 \to & \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3, \beta_4 \rangle\ \alpha & \to \mathbb{T}\ \langle \beta_1, \beta_2, \alpha, \beta_4 \rangle\ \beta_3 \\ \mathsf{inverse4\_4} : & \beta_4 \to & \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3, \beta_4 \rangle\ \alpha & \to \mathbb{T}\ \langle \beta_1, \beta_2, \beta_3, \alpha \rangle\ \beta_4 \end{array}$$

with

$$\begin{aligned}
\mathsf{inverse2_2} &:= \lambda\ u\ .\ \mathsf{transpose} \circ \mathsf{inverse2}\ u \circ \mathsf{transpose}\\
\mathsf{inverse3_2} &:= \lambda\ u\ .\ \mathsf{transpose} \circ \mathsf{inverse3}\ u \circ \mathsf{transpose}\\
\mathsf{inverse3_3} &:= \lambda\ u\ .\ \mathsf{transpose3} \circ \mathsf{inverse3}\ u \circ \mathsf{transpose3}\\
\mathsf{inverse4_2} &:= \lambda\ u\ .\ \mathsf{transpose} \circ \mathsf{inverse4}\ u \circ \mathsf{transpose}\\
\mathsf{inverse4_3} &:= \lambda\ u\ .\ \mathsf{transpose3} \circ \mathsf{inverse4}\ u \circ \mathsf{transpose3}\\
\mathsf{inverse4_4} &:= \lambda\ u\ .\ \mathsf{transpose4} \circ \mathsf{inverse4}\ u \circ \mathsf{transpose4}
\end{aligned}$$

Inversion of one-dimensional tables is easily defined:

$$\mathsf{inverse1} := \mathcal{F}\ (\mathsf{addH2}, |||)\ \mathbb{I}$$

For two-dimensional tables, let us first consider the case where there is a single header in the first dimension, for example $T_{f,\mathrm{b}}$, which can be considered as resulting from vertical concatenation of two two-dimensional one-cell tables $T_{f,\mathrm{b}} = T_{f,\mathrm{ba}} /\!/\!/ T_{f,\mathrm{bb}}$, see Fig. 17.

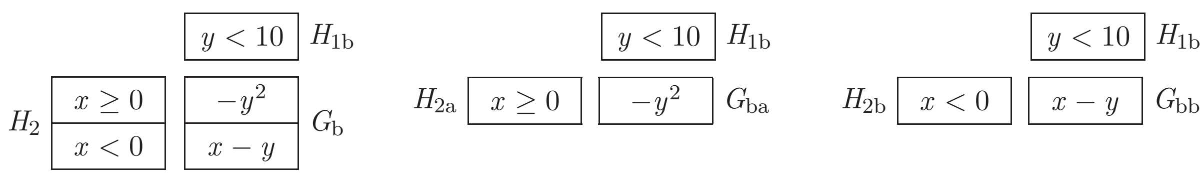

Figure 17: The tables $T_{f,\mathrm{b}}$, $T_{f,\mathrm{ba}}$, and $T_{f,\mathrm{bb}}$

According to the explanation above, the inverses of these one-cell tables are obtained by swapping their cell contents with their first header entries; the results are shown in Fig. 18.

Figure 18: The tables $T_{f,\mathrm{bai}}$, and $T_{f,\mathrm{bbi}}$

It is easy to check that these are inverted tables with preserved semantics, i.e.,

$$\boxed{S_{\mathsf{ND}}}\llbracket T_{f,\mathrm{ba}} \rrbracket = \boxed{S_{\mathsf{ID}}}\llbracket T_{f,\mathrm{bai}} \rrbracket \qquad \text{and} \qquad \boxed{S_{\mathsf{ND}}}\llbracket T_{f,\mathrm{bb}} \rrbracket = \boxed{S_{\mathsf{ID}}}\llbracket T_{f,\mathrm{bbi}} \rrbracket\ .$$

Combining these two tables via horizontal concatenation produces the *ragged table* $T_{f,\mathrm{bai}} \mathbin{|||} T_{f,\mathrm{bbi}}$; it is an inverted table equivalent to $T_{f,\mathrm{b}}$, i.e.,

$$\boxed{S_{\mathsf{ND}}}\llbracket T_{f,\mathrm{b}} \rrbracket = \boxed{S_{\mathsf{ID}}}\llbracket T_{f,\mathrm{bai}} \mathbin{|||} T_{f,\mathrm{bbi}} \rrbracket\ .$$

If one accepts ragged tables as results of inversion, then

$$\mathcal{F}\ ((\lambda\ h_1\ .\ \mathcal{F}\ (\mathsf{addH2}, |||)\ (\mathsf{addH2}\ h_1)), |||)\ \mathbb{I}$$

can be used as inversion function for two-dimensional tables. With this "ragged inversion", the theorems below still hold, and can be stripped from the requirements regarding the existence of units e_0, e_1 and "empty values" u — for systematic lifting of "ragged inversion" to higher dimensions see A.3.

If a regular table is desired as result of inversion, as is the case throughout the literature, then $T_{f,\mathrm{bai}}$ and $T_{f,\mathrm{bbi}}$ need to be combined via *diagonal concatenation* $\natural$ to form the inverted table corresponding to $T_{f,\mathrm{b}}$, shown in Fig. 19; for this diagonal concatenation, the non-diagonal blocks need to be filled with the "empty" element mentioned above, which in this case is false — this is supplied to $\natural$ as a subscript.

		$-y^2$	$x - y$	$H_{1\mathrm{bi}}$
H_2	$x \geq 0$	$y < 10$	false	G_{bi}
	$x < 0$	false	$y < 10$	

Figure 19: The table $\mathsf{inverse2}\ \mathsf{false}\ T_{f,\mathrm{b}}$

From these drawings, it is easily seen that

$$\boxed{\boxed{S_{\mathsf{ND}}}}\left\ulcorner T_{f,\mathrm{b}} \right\lrcorner = \boxed{\boxed{S_{\mathsf{ID}}}}\left\ulcorner \mathsf{inverse2}\ \mathsf{false}\ T_{f,\mathrm{b}} \right\lrcorner .$$

Furthermore, vertical concatenation of two-dimensional tables with a single header in the first dimension (like $T_{f,\mathrm{b}}$) translates into diagonal concatenation of larger blocks, and horizontal concatenation translates into horizontal concatenation, so inversion of two-dimensional tables is defined by the following:

$$\begin{array}{lcl}
\mathsf{inverse2}\ u\ (h_1 \rhd (h_2 \rhd c)) & = & c \rhd (h_2 \rhd h_1) \\
\mathsf{inverse2}\ u\ ((h_1 \rhd u_1) /\!\!/\!\!/ (h_1 \rhd u_2)) & = & \mathsf{inverse2}\ u\ (h_1 \rhd u_1) \natural_u \mathsf{inverse2}\ u\ (h_1 \rhd u_2) \\
\mathsf{inverse2}\ u\ (t_1 \,|\!|\!|\, t_2) & = & \mathsf{inverse2}\ u\ t_1 \,|\!|\!|\, \mathsf{inverse2}\ u\ t_2
\end{array}$$

Diagonal concatenation of regular tables is defined quite directly; if $\mathsf{regSkel}_2\ (t_1) = (h_1, h_2)$ and $\mathsf{regSkel}_2\ (t_2) = (k_1, k_2)$, then $\mathsf{tMap}\ (\lambda\ t\ .\ \mathsf{updC1}\ u\ (\mathsf{delH1}\ t_2))\ t_1$ produces a table v_1 filled with empty cells u and with $\mathsf{regSkel}_2\ (v_1) = (h_1, k_2)$, and *vice versa* for the opposite corner:

$$t_1 \natural_u t_2 := (t_1 /\!\!/\!\!/ \mathsf{fill}\ u\ t_1\ t_2) \,|\!|\!|\, (\mathsf{fill}\ u\ t_2\ t_1 /\!\!/\!\!/ t_2)$$

We used an auxiliary function

$$\begin{array}{lcl}
\mathsf{fill} & :: & \alpha \to \mathbb{T}\ \langle \beta_1, \beta_2' \rangle\ \alpha' \to \mathbb{T}\ \langle \beta_1', \beta_2 \rangle\ \alpha'' \to \mathbb{T}\ \langle \beta_1, \beta_2 \rangle\ \alpha \\
\mathsf{fill}\ u\ t_1\ t_2 & := & \mathsf{tMap}\ (\lambda\ t\ .\ \mathsf{updC1}\ u\ (\mathsf{delH1}\ t_2))\ t_1
\end{array}$$

that produces a table where all cells are filled with u and the headers are appropriate for the concatenation in the diagonal concatenation:

$$\mathsf{regSkel}_2\ (\mathsf{fill}\ u\ t_1\ t_2) = (\mathsf{fst}\ (\mathsf{regSkel}_2\ t_1),\ \mathsf{snd}\ (\mathsf{regSkel}_2\ t_2))$$

In comparison with the definition of table inversion used in [SZP96], our definition produces a permutation of the top-level concatenations of the result table, since our definition groups by original cell, while that of [SZP96] groups by what are the entries of the second header in our setting. However, this permutation is semantically justified by the laws that need to hold for inversion to be applicable, see below.

In the appendix we show how parameterising diagonal concatenations with $\mathsf{updC1}\ u$ instead of with u leads to a definition that can equally be used for higher-dimensional table inversions. Further parameterisation of the definition of $\mathsf{inverse2}$ leads to an "inversion lifting combinator" that allows the sequence $\mathsf{inverse2}$, $\mathsf{inverse3}$, $\mathsf{inverse4}$, ... to be constructed in a simple and systematic way, see page 52.

For these formal definitions we then obtain the following general theorem about semantics-preservation of syntactic inversion:

Theorem 6.3.1 (Inversion of Two-Dimensional Tabular Expressions)
Let two table evaluation structures

$$\begin{aligned} S &= (w_1 \llparenthesis \langle(\otimes_1, \oplus_1), (\otimes_2, \oplus_2)\rangle \rrparenthesis f_3) : \mathbb{S}\ \langle\beta_1, \beta_2\rangle\ \alpha\ \gamma_0 \qquad \text{and} \\ S_{\mathsf{inv}} &= (w_2 \llparenthesis \langle(\otimes_4, \oplus_4), (\otimes_5, \oplus_5)\rangle \rrparenthesis f_6) : \mathbb{S}\ \langle\alpha, \beta_2\rangle\ \beta_1\ \gamma_0 \end{aligned}$$

be given, with the following combinator types:

$$\begin{array}{llll} & \otimes_1 : \beta_1 \to \gamma_2 \to \gamma_1 & \otimes_2 : \beta_2 \to \gamma_3 \to \gamma_2 & f_3 : \alpha \to \gamma_3 \\ w_1 : \gamma_1 \to \gamma_0 & \oplus_1 : \gamma_1 \to \gamma_1 \to \gamma_1 & \oplus_2 : \gamma_2 \to \gamma_2 \to \gamma_2 & \\ & \otimes_4 : \alpha \to \gamma_5 \to \gamma_4 & \otimes_5 : \beta_2 \to \gamma_6 \to \gamma_5 & f_6 : \beta_1 \to \gamma_6 \\ w_2 : \gamma_4 \to \gamma_0 & \oplus_4 : \gamma_4 \to \gamma_4 \to \gamma_4 & \oplus_5 : \gamma_5 \to \gamma_5 \to \gamma_5 & \end{array}$$

In addition, let γ_0 have a monoid structure, that is, there is an associative operation $\odot : \gamma_0 \to \gamma_0 \to \gamma_0$ together with a neutral element $e_0 : \gamma_0$.
Assume further an "empty element" $u : \beta_1$, and the following laws (for all $c : \alpha$; $h_1 : \beta_1$; $h_2 : \beta_2$; $x_1, x_2 : \gamma_1$; $y, y_1, y_2 : \gamma_2$; $z_2, z_2, z_3, z_4 : \gamma_4$; $v_1, v_2 : \gamma_5$):

$$\begin{array}{lcl} w_1\ (u \otimes_1 y) & = & e_0 \\ w_1\ (x_1 \oplus_1 x_2) & = & (w_1\ x_1) \odot (w_1\ x_2) \\ w_1\ (h_1 \otimes_1 (y_1 \oplus_2 y_2)) & = & (w_1\ (h_1 \otimes_1 y_1)) \odot (w_1\ (h_1 \otimes_1 y_2)) \\ w_1\ (h_1 \otimes_1 (h_2 \otimes_2 (f_3\ c))) & = & w_2\ (c \otimes_4 (h_2 \otimes_5 (f_6\ h_1))) \\ w_2\ (z_1 \oplus_4 z_2) & = & (w_2\ z_1) \odot (w_2\ z_2) \\ w_2\ (c \otimes_4 (v_1 \oplus_5 v_2)) & = & (w_2\ (c \otimes_4 v_1)) \odot (w_2\ (c \otimes_4 v_2)) \\ w_2\ (z_1 \oplus_4 z_2) \oplus_4 (z_3 \oplus_4 z_4) & = & (w_2\ (z_1 \oplus_4 z_3)) \odot (w_2\ (z_2 \oplus_4 z_4)) \end{array}$$

Then for every table $T : \mathbb{T}\ \langle\beta_1, \beta_2\rangle\ \alpha$, inversion is value-preserving in the following way:

$$\boxed{\boxed{w_1 \llparenthesis \langle(\otimes_1, \oplus_1), (\otimes_2, \oplus_2)\rangle \rrparenthesis f_3}} \llbracket T \rrbracket = \boxed{\boxed{w_2 \llparenthesis \langle(\otimes_4, \oplus_4), (\otimes_5, \oplus_5)\rangle \rrparenthesis f_6}} \llbracket \mathsf{inverse2}\ u\ T \rrbracket \qquad \square$$

In the identity-wrapped case, γ_1 and γ_4 coincide with γ_0, and, by the wrapper distributivity laws, $\oplus_1$ and $\oplus_4$ have to coincide with $\odot$.

Theorem 6.3.2 (Inversion of Id-Wrapped Two-Dimensional Tabular Expressions)
Let two identity-wrapped table evaluation structures

$$\begin{aligned} S &= (\mathbb{I} \llparenthesis \langle(\otimes_1, \oplus_1), (\otimes_2, \oplus_2)\rangle \rrparenthesis f_3) : \mathbb{S}\ \langle\beta_1, \beta_2\rangle\ \alpha\ \gamma_0 \ \text{ and} \\ S_{\mathsf{inv}} &= (\mathbb{I} \llparenthesis \langle(\otimes_4, \oplus_1), (\otimes_5, \oplus_5)\rangle \rrparenthesis f_6) : \mathbb{S}\ \langle\alpha, \beta_2\rangle\ \beta_1\ \gamma_0 \end{aligned}$$

be given, with the following combinator types:

$$\begin{array}{lll} \otimes_1 : \beta_1 \to \gamma_2 \to \gamma_1 & \otimes_2 : \beta_2 \to \gamma_3 \to \gamma_2 & f_3 : \alpha \to \gamma_3 \\ \oplus_1 : \gamma_1 \to \gamma_1 \to \gamma_1 & \oplus_2 : \gamma_2 \to \gamma_2 \to \gamma_2 & \\ \otimes_4 : \alpha \to \gamma_5 \to \gamma_1 & \otimes_5 : \beta_2 \to \gamma_6 \to \gamma_5 & f_6 : \beta_1 \to \gamma_6 \\ & \oplus_5 : \gamma_5 \to \gamma_5 \to \gamma_5 & \end{array}$$

In addition, assume a neutral element $e_1 : \gamma_0$ for $\oplus_1$, an "empty element" $u : \beta_1$, and the following laws (for all $c : \alpha$; $h_1 : \beta_1$; $h_2 : \beta_2$; $x_1, x_2, x_3, x_4 : \gamma_1$; $y, y_1, y_2 : \gamma_2$; $z_1, z_2 : \gamma_5$):

$$\begin{array}{lcl}
u \otimes_1 y & = & e_1 \\
h_1 \otimes_1 (y_1 \oplus_2 y_2) & = & (h_1 \otimes_1 y_1) \oplus_1 (h_1 \otimes_1 y_2) \\
h_1 \otimes_1 (h_2 \otimes_2 (f_3\ c)) & = & c \otimes_4 (h_2 \otimes_5 (f_6\ h_1)) \\
c \otimes_4 (z_1 \oplus_5 z_2) & = & (c \otimes_4 z_1) \oplus_1 (c \otimes_4 z_2) \\
(x_1 \oplus_1 x_2) \oplus_1 (x_3 \oplus_1 x_4) & = & (x_1 \oplus_1 x_3) \oplus_1 (x_2 \oplus_1 x_4)
\end{array}$$

Then for every table $T : \mathbb{T}\ \langle \beta_1, \beta_2 \rangle\ \alpha$, inversion is value-preserving in the following way:

$$\boxed{\boxed{\mathbb{I} \Lbag \langle (\otimes_1, \oplus_1), (\otimes_2, \oplus_2) \rangle \Rbag f_3}} \ulcorner T \urcorner = \boxed{\boxed{\mathbb{I} \Lbag \langle (\otimes_4, \oplus_1), (\otimes_5, \oplus_5) \rangle \Rbag f_6}} \ulcorner \mathsf{inverse2}\ u\ T \urcorner \qquad \square$$

7 Conclusion

Starting with an investigation how tables can be considered as produced by a compositional syntax, we identified a small number of combinators that can be used as basis of arbitrary table construction, and that form a rich table algebra.

Constructing an intuitively accessible type system on this table algebra helped us to design *table folding* as a natural way to perform calculations on the first-dimension structure of a table, and extend this to *table evaluation structures* (TESs), the application of which allows evaluation of the whole table in full depth.

These mechanisms enable modular and transparent definitions of table semantics, and are simpler and more homogenous than previous approaches.

The fact that tables are not always used to denote some semantic value, but often just as subexpressions, abbreviating large syntactic expressions, prompted us to carefully look into this distinction, and accordingly design two different concepts of *tabular expressions* that embody the two different uses in a clear and consistent manner.

Furthermore, by performing the Haskell formalisation given in the appendix, we have shown how the compositional view can be used to obtain a simple and comprehensible basis for implementation, that is in addition supported by the correctness proofs in the Isabelle/HOL formalisation, all available from http://www.cas.mcmaster.ca/~kahl/Tables/.

In our investigation of table structure in Sect. 2, we identified several issues; let us now revisit those:

1. **Empty tables as (partial) units for horizontal concatenation?**

   Throughout our whole development, we have not encountered any strong motivation for permitting empty tables.

   One possible use would be, in the presence of concatenable cells as in the appendix, to use empty zero-dimensional tables instead of the externally supplied "empty entry" u for table inversion.

   However, since the presence of empty tables would make an additional argument to table folding necessary, and thus would slightly complicate the evaluation machinery, we propose to work without empty tables as far as possible, until a stronger case is made, in particular since folding *with* unit does not eliminate the need for folding *without* unit.

2. Are all zero-dimensional tables cells?

From a technical point of view, answering a qualified "no" and allowing one-dimensional concatenation of cells (as in the appendix) makes table evaluation more homogenous. However, we are not yet aware of any practical uses.

3. Should ragged tables be allowed?

From the point of view of table evaluation and semantics, ragged tables introduce no problems. In fact, with our compositional approach to table syntax, ragged tables arise naturally as results of horizontal concatenation with no restrictions imposed except well-typedness.

We have shown how regular tables can be considered as a (useful) special case, and some transformations, for example transposition, are only available for regular tables.

On the other hand, inversion becomes much simpler if ragged tables are allowed as results.

Therefore, we propose that for tools based on our framework, ragged tables should be permitted as naturally arising, and special support for regular tables added where appropriate.

As exemplified by the discussions throughout this paper, our systematic view of compositional table syntax and semantics is a useful basis for investigating and solving problems involving structure and meaning of tables.

Our next endeavour will be to base a new generation of table manipulation and transformation tools on this theoretical foundation.

Acknowledgements

I am grateful to Dave Parnas for early support of the central ideas of this paper, and to him and Ridha Khedri for continued discussion about tables and valuable information concerning "table history". Dave Parnas and Ridha Khedri also provided valuable comments on previous versions of this paper; thanks for careful reading also go to Millie de Guzman and Jan Scheffczyk.

A Haskell Modules for Tables and Tabular Expressions

This appendix presents a formalisation of tables in the purely functional programming language Haskell [PJ+03]. Haskell notation is mostly quite intuitive and largely similar to the notation used in the body of this paper, so we will not explain Haskell features here; tutorials, language definition, and implementations can be found on the Haskell web site http://haskell.org/.

We present the essential data types of tables, TESs, and tabular expressions, and also the central table transformation functions, but concentrate on the generic table framework that is completely free of any decision about the contents of grid and header cells. In the future, we plan to continue this "HTables" project, available at http://www.cas.mcmaster.ca/~kahl/Tables/HTables/, to build user-friendly table tools on top of the framework presented here.

The Haskell code is presented as a *literate program* [Knu84, Knu92], with code interspersed between explanatory documentation, just like in traditional mathematical presentations. The code seen by the Haskell system is typeset as complete paragraphs in `normal typewriter font`, while tentative code, alternative versions, laws, etc. are typeset in *`slanted typewriter font`*.

Haskell is statically strongly typed with a type system encompassing that of HOL, but lacking parameterisation by type lists. Due mainly to this limitation in the type system, some aspects of tables and tabular expression receive a slightly different treatment here than in the body of this paper. We took a rather conservative approach to Haskell extensions — the basic table modules, up to inversion, are all in the standardised Haskell version "Haskell98" [PJ+03]. For obtaining an elegant but still very general treatment of TESs (and tabular expressions), we employ multi-parameter type classes, a widely supported Haskell extension.

An additional use of the Haskell formalisation presented here is as an introduction to and executable experimentation tool for our formalisation in the mechanised proof assistant Isabelle/HOL [NPW02]. That formalisation closely follows the Haskell version, but has to pay additional attention to issues such as the use of non-empty lists, making it technically more involved and less accessible. Therefore, we chose to include only the Haskell code here, but in a presentation that can also serve as a guide to the formalisation in Isabelle/HOL, which is available at http://www.cas.mcmaster.ca/~kahl/Tables/Isabelle/.

For an efficient table system implementation, some data type and function implementations would be organised differently — the Isabelle/HOL theories include theorems justifying such more efficient choices.

For simplicity, we restrict ourselves to a single table data type with no nested headers.

A.1 Tables as Abstract Data Type

This module exports the following items:

- The binary table type constructor `T` as an *abstract* (i.e., not supporting pattern matching) type constructor of kind `* -> * -> *`, further explained below.
- The table construction function `cell` and the infix table construction operators `>||` and `|||` (the latter is associative):

```
cell  :: c               -> T c ()
(>||) :: h      -> t     -> T h t
(|||) :: T h t -> T h t -> T h t
```

  This typing allows horizontal composition of cells, but has the advantage that many definitions become simpler and more homogeneous.

A way to ensure statically that cells are never horizontally composed would be to use the following type declaration for horizontal composition (leaving the definition unchanged):

```
(|||) :: T h1 (T h2 t) -> T h1 (T h2 t) -> T h1 (T h2 t)
```

- The table folding function `tFold`:

```
tFold :: (h -> t -> e) -> (e -> e -> e) -> T h t -> e
```

This satisfies the following laws if `comp` is associative:

```
tFold addH comp (cell c)    = c `addH` ()
tFold addH comp (h  >|| t ) = h `addH` t
tFold addH comp (t1 ||| t2) = tFold addH comp t1 `comp` tFold addH comp t2
```

The export list of this module guarantees that all table analysis can only be performed via `tFold`.

- The type class `Table`, explained below.

```
module Table(T(), cell, (>||), (|||), tFold, Table()) where

infixr 7 >||
infixr 8 |||
```

In Sect. 3, we introduced $\mathbb{T}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha$ as the general type of n-dimensional tables — the general type of tables is therefore parameterised by the cell type and the *list of header types*.

Since Haskell does not allow parameterisation by lists of types, we essentially have the following choices:

i) introduce separate table types for different dimensions,

ii) restrict the tables accommodated by the table data type to those fitting into a simpler type scheme, for example with $\beta_1 = \ldots = \beta_n$,

iii) relax the typing of tables, generalising the data type for tables so that it can (potentially) contain some non-table objects.

We chose a combination of the first and the third approach by introducing a binary type constructor `T` such that for a header type `h` and a type `t` of n-dimensional tables the type expression `T h t` stands for the resulting $(n+1)$-dimensional tables.

For achieving greater homogeneity, we let cells of type $\mathbb{T}\ \langle\rangle\ \alpha$ populate the type `T` α `()`, so that n-dimensional tables of type $\mathbb{T}\ \langle\beta_1, \ldots, \beta_n\rangle\ \alpha$ will have a Haskell representation of type `T` β_1 `(` $\cdots$ `(T` β_n `(T` α `()))`$\cdots$`)`.

This does of course also make possible types `T` β_1 `(` $\cdots$ `(T` β_n `(T` α γ`))`$\cdots$`)` for arbitrary γ. However, Haskell's type class system can be used to restrict the table constructors to well-formed tables only; we can force γ to be `()` for types in the class `Table` via the following definitions:

```
class Table t
instance Table ()
instance Table t => Table (T h t)
```

Then, the exported functions can be restricted to operate on valid `Table`s only by replacing the type declarations given above by the following:

```
(>||) :: Table t => h -> t -> T h t
(|||) :: Table t => T h t -> T h t -> T h t
tFold :: Table t => (h -> t -> e) -> (e -> e -> e) -> T h t -> e
```

A few technical points should be noted:

- Here, the one-element type `()` is considered as a table, too. By dropping the limitations of Haskell 98 and additionally enabling overlapping instance declarations as supported by some implementations, this can be avoided. The interface functions would then require "`Table (T h t)`" as type constraint instead of "`Table t`" to achieve the same generality.

- If the class `Table` is exported, then importing modules can define their own instances of this class, and the static type safety that was the main motivation of introducing this class is lost.

- If the class `Table` is not exported, then we do have type safety for tables (i.e., the guarantee that the γ from above is always `()` in the types of arguments to the functions exported from this module. However, it is then not possible to give explicit type declarations for any function that has tables in its type, since those tables will be restricted to the class `Table` that cannot be referred to from importing modules.

  (Not being able to provide type declarations will incur as further complication the impossibility to define these functions in pattern bindings, i.e., without explicit reference to their arguments; this is due to Haskell's monomorphism restriction.)

In addition, from a pragmatic point of view, different types γ would not amount to much more than different ways to factor the information contained in cells.

On the whole, not much is gained by introducing the class `Table` and using it to constrain the functions exported from this module. We chose not to constrain the interface functions, but we export the `Table` class so that users may use it to constrain their own functions if so desired.

Implementation

We implement a table as a list of header-subtable-pairs; the abstract interface guarantees that these lists are always non-empty.

```
newtype T h t = T [(h,t)] deriving Eq

(>||) :: h -> t -> T h t
(>||) h t = T [(h,t)]

(|||) :: T h t -> T h t -> T h t
T ps1 ||| T ps2  = T (ps1 ++ ps2)

cell :: c -> T c ()
cell x = x >|| ()

tFold :: (h -> t -> e) -> (e -> e -> e) -> T h t -> e
tFold addH comp (T ps) = foldr1 comp (map (uncurry addH) ps)
```

The laws are easily verified:

```
  tFold addH comp (h >|| t)
= tFold addH comp (T [(x,t)])
= foldr1 comp (map (uncurry addH) [(x,t)])
= foldr1 comp [uncurry addH (x,t)]
= uncurry addH (x,t)
= addH x t
= x `addH` t

  tFold addH comp (T ps1 ||| T ps2)  -- T is surjective
= tFold addH comp (T (ps1 ++ ps2))
= foldr1 comp (map (uncurry addH) (ps1 ++ ps2))
= foldr1 comp (map (uncurry addH) ps1 ++ map (uncurry addH) ps2)
=   {-  ps1 and ps2 are non-empty  -}
  foldr1 comp (map (uncurry addH) ps1) `comp`
   foldr1 comp (map (uncurry addH) ps2)
= tFold addH comp (T ps1) `comp` tFold addH comp (T ps2)

  tFold addH comp (cell c)
= tFold addH comp (c >|| ())
= c `addH` ()
```

A.2 Table Manipulation and Advanced Construction

Here we define utility functions for table construction and manipulation on top of the interface of the module `Table`. That module is also re-exported, so that there is no need to ever explicitly import `Table`.

All functions here are defined in a direct, mathematical way, and no new datatypes are introduced, so that a separate interface definition is unnecessary.

```
module Tables(module Table, module Tables) where

import Table
```

A.2.1 Simple Table Manipulation and Access

The table datatype constructor `T` is a bifunctor, and the arrow parts of the individual functors are easily defined via `tFold`:

```
hMap :: (h1 -> h2) -> T h1 t -> T h2 t
hMap f = tFold ((>||) . f) (|||)

tMap :: (t1 -> t2) -> T h t1 -> T h t2
tMap f = tFold (\ h t -> h >|| f t)  (|||)
```

The header list of the first dimension is obtained through another simple application of `tFold`:

```
headers :: T h t -> [h]
headers = tFold (\ h t -> [h]) (++)
```

Sometimes, a three-argument folding function like F from Def. 4.1.1 is more appropriate; we can easily compose this from `tFold` and `tMap`:

```
tFoldM a c f = tFold a c . tMap f
```

A definition via just `tFold` avoids the construction of the intermediate table:

```
tFoldM :: (h -> s -> r) -> (r -> r -> r) -> (t -> s) -> T h t -> r
tFoldM a c f = tFold (\ h t -> a h (f t)) c
```

A.2.2 The "Cons View" of Tables

Adding a first element to a table assembles the header-subtable pair and concatenates it to the given tail:

```
hCons :: h -> t -> T h t -> T h t
hCons h t0 t = (h >|| t0) ||| t
```

The case distinction between singleton and longer tables is implemented as

```
hUnCons :: T h t -> Either (h,t) ((h,t), T h t)
```

with the following specification:

```
hUnCons (h >|| t) = Left (h,t)
hUnCons (hCons h t0 t) = Right ((h,t0), t)
```

This can be implemented via `tFold` — note that `q` is associative:

```
hUnCons = fst . tFold (\ h t -> (Left (h, t), (h >|| t))) q
  where q (r,t1) (_, t2) = (case r of
                               Left p -> Right (p,t2)
                               Right (p,t1') -> Right (p,t1' ||| t2)
                            , t1 ||| t2)
```

This analysis function can now be used to define structural induction over the "cons view" of tables:

```
tFoldr :: (h -> t -> r) -> (h -> t -> r -> r) -> T h t -> r
tFoldr a f t = case hUnCons t of
  Left (h,t) -> a h t
  Right ((h,t),t') -> f h t (tFoldr a f t')
```

Note that no laws are required to hold here, in particular, no variant of associativity is necessary for `f`.

Also note that this arrangement makes it possible to let the header and subtable of the last component receive a treatment (via `a`) different from what the others receive (via `f`).

The "cons view" of tables could be used as an alternative basis; concatenation `|||` and `tFold` can be regained via the following:

```
t1 ||| t2 = tFoldr (\ h t -> hCons h t t2) hCons t1
tFold a c = tFoldr a (\ h t r -> a h t `c` r)
```

A.2.3 Regular Table Skeletons

We now define functions `regSkel`N that map each N-dimensional table to `Nothing` if it is not regular, and to `Just s` if it is regular, where `s` is a nested tuple of lists representing the table's skeleton — since the headers of different dimensions can belong to different types, we cannot in general represent a regular table skeleton as a list of header lists in Haskell.

For regular zero-dimensional tables, we have to deal with the possibility of concatenated cells — since we consider tables containing concatenated cells as not regular, we return `Nothing` for concatenations at the cell level. A single cell is always regular, and has no header structure that would influence concatenability of higher dimensions, so we return the zero-tuple `()` as its skeleton.

```
regSkel0 :: T c () -> Maybe ()
regSkel0 = tFold (\ _ _ -> Just ()) (\ _ _ -> Nothing)
```

For an $(n+1)$-dimensional table t to be regular, all its n-dimensional constituent tables need to be regular and need to have the same skeletons; the skeleton of t is then that of the constituent tables with the header list of the first dimension added (we employ Haskell's monadic `do` notation in the `Maybe` monad):

```
regSkelStep :: Eq s => (t -> Maybe s) -> T h t -> Maybe ([h],s)
regSkelStep reg t = do s <- tFold (\ h t -> reg t) mEq t
                       return (headers t, s)
```

The skeleton equality and propagation herein is achieved via the following (associative and idempotent) "equality folding combinator":

```
mEq :: Eq a => Maybe a -> Maybe a -> Maybe a
mEq (Just x) (Just y) = if x == y then Just x else Nothing
mEq  _ _ = Nothing
```

Note that a one-dimensional table can only fail to be regular by containing concatenated cells — without that, one-dimensional tables cannot be ragged.

```
regSkel1 :: T h (T c ()) -> Maybe ([h],())
regSkel1 t = regSkelStep regSkel0 t

regSkel2 :: Eq h2 => T h1 (T h2 (T c ())) -> Maybe ([h1],([h2],()))
regSkel2 t = regSkelStep regSkel1 t

regSkel3 :: (Eq h2, Eq h3) =>
            T h1 (T h2 (T h3 (T c ()))) -> Maybe ([h1],([h2],([h3],())))
regSkel3 t = regSkelStep regSkel2 t
```

If one is only interested in the regularity of the N outermost dimensions, then regularity to zero dimensions would return a trivial skeleton for arbitrary objects, and regularity to one dimension always succeeds, only constructing the header list:

```
regSkelOuter0 :: t -> Maybe ()
regSkelOuter0 _ =  Just ()

regSkelOuter1 :: T h t -> Maybe ([h],())
regSkelOuter1 t = regSkelStep regSkelOuter0 t
                = Just (headers t, ())
```

Therefore, we simplify both definition and typing of `regSkelOuter1`:

```
regSkelOuter1 :: T h t -> Maybe [h]
regSkelOuter1 t = Just (headers t)

regSkelOuter2 :: Eq h2 => T h1 (T h2 c) -> Maybe ([h1],[h2])
regSkelOuter2 t = regSkelStep regSkelOuter1 t

regSkelOuter3 :: (Eq h2, Eq h3) =>
                 T h1 (T h2 (T h3 t)) -> Maybe ([h1],([h2],[h3]))
regSkelOuter3 t = regSkelStep regSkelOuter2 t
```

If `t` is regular, its first header list equals `headers t`; we introduce a corresponding function for the second dimension — this is partial, and defined only on tables that are regular in their two outermost dimensions:

```
reg2dimen2 t = case regSkelOuter2 t of
  Nothing -> error "reg2dimen2: not a regular table"
  Just (_,hs2) -> hs2
```

A.2.4 Table Construction Functions

Since an $(n+1)$-dimensional table is the concatenation (via $|||$) of a positive number of n-dimensional tables, each equipped with a header via $\triangleright$, this construction of an $(n+1)$-dimensional table can be considered to start from a list of pairs, where each pair consists of a header and an n-dimensional table.

The construction is then easily implemented via list folding — the use of `foldr1` is justified by the absence of empty tables and turns `tOfList` into a partial function which is undefined on the empty list:

```
tOfList :: [(h,t)] -> T h t
tOfList = foldr1 (|||) . map (uncurry (>||))
```

This partial function is surjective and injective, and therefore has a total right-inverse. This is a simple application of `tFold`, turning an at least one-dimensional table into a list of header-subtable-pairs:

```
tList :: T h t -> [(h,t)]
tList = tFold (\ h t -> [(h,t)]) (++)
```

The following holds:

`tOfList . tList = id :: T h t -> T h t`

Frequently, a zipping variant of `tOfList` is easier to apply:

```
zipT :: [h] -> [t] -> T h t
```

`zipT hs = tOfList . zip hs`

This can also be directly defined by applying the list constructors via list functions from Haskell's prelude:

```
zipT hs = foldr1 (|||) . zipWith (>||) hs
```

We use this to define functions that are intended for constructing regular one- and two-dimensional tables from header lists and a grid of cells — for `table2`, regularity has to be guaranteed by either supplying a rectangular cell grid or a header list `hs2` for the second dimension that is at most as long as the shortest element list of the grid. (Both functions are also partial since they are undefined for empty lists.)

```
table1 :: [h] -> [c] -> T h (T c ())
table1 hs = zipT hs . map cell

table2 :: [h1] -> [h2] -> [[c]] -> T h1 (T h2 (T c ()))
table2 hs1 hs2 = zipT hs1 . map (table1 hs2)
```

The sharing semantics of Haskell implies that headers in tables constructed by `table2` will not be duplicated, but shared. Although our implementation also allows for ragged tables, the space overhead for regular tables, if they are constructed for example via `table2`, is therefore limited to one pair constructor per cell, which will mostly be negligible in comparison with the cell contents.

A.2.5 Table Transposition

The list interface to tables that is exposed by `tList` and `tOfList` makes list functions available for table manipulation; here we use `zipWith` to create a corresponding function for at least one-dimensional tables:

```
tZipWith :: ((h1,t1) -> (h2,t2) -> (h,t)) -> T h1 t1 -> T h2 t2 -> T h t
tZipWith f t1 t2 = foldr1 (|||) $ map (uncurry (>||)) $
                   zipWith f (tList t1) (tList t2)
```

This function does of course inherit from `zipWith` the property that if the arguments are of different length, then the result has the length of the shorter argument.

Using `tZipWith`, we can define "vertical" concatenation of (at least) two-dimensional tables (since the header of the second table is never used, the given type is not principal):

```
(///) :: T h1 (T h2 c) -> T h1 (T h2 c) -> T h1 (T h2 c)
t1 /// t2 = tZipWith vConc0 t1 t2
 where
  vConc0 (h1,t1) (h2,t2) = (h1, t1 ||| t2)
```

This can be seen as a variant of `|||` that operates on the second dimension instead of on the first. For vertical concatenation of two tables `t1` and `t2` to make sense, the first-dimension headers have to coincide, i.e., `headers t1 = headers t2` has to hold.

We introduce an analogous variant of `>||` that preserves the first dimension and adds a new singleton second dimension. This operator `addH2` interacts with `>||` by interspersing its first operand between the two arguments of `>||` into a double application of the latter, and it distributes over `|||`:

```
h2 `addH2` (h1 >|| t ) = h1 >|| (h2 >|| t)
h2 `addH2` (t1 ||| t2) = (h2 `addH2` t1) ||| (h2 `addH2` t2)
```

This obviously is a simple instance of `tMap`:

```
addH2 :: h2 -> T h1 t -> T h1 (T h2 t)
addH2 h2 = tMap (\ t -> h2 >|| t)
```

We can further simplify this definition, and systematically continue into higher dimensions:

```
addH1 :: h1 -> t -> T h1 t
addH1 = (>||)

addH2 :: h2 -> T h1 t -> T h1 (T h2 t)
addH2 = tMap . addH1

addH3 :: h3 -> T h1 (T h2 t) -> T h1 (T h2 (T h3 t))
addH3 = tMap . addH2

addH4 :: h4 -> T h1 (T h2 (T h3 t)) -> T h1 (T h2 (T h3 (T h4 t)))
addH4 = tMap . addH3
```

Transposition is only defined for tables with at least two dimensions, and replaces the standard constructors `|||` and `>||` with those acting on the second dimension:

```
tTranspose :: T h1 (T h2 t) -> T h2 (T h1 t)
tTranspose = tFold addH2 (///)
```

Higher-dimensional transpositions are easily constructed from this, e.g.:

```
tTranspose3 :: T a (T b (T c d)) -> T c (T b (T a d))
tTranspose3 = tTranspose . tMap tTranspose . tTranspose

tTranspose4 :: T a (T b (T c (T d e))) -> T d (T b (T c (T a e)))
tTranspose4 = tTranspose . tMap tTranspose3 . tTranspose
```

Another primitive table transformation collapses the two outermost dimensions, combining the two headers together in a single header. We define this as an instance of a generalised concatenation function:

```
collapse :: (h1 -> t1 -> T h2 t2) -> T h1 t1 -> T h2 t2
collapse f = tFold f (|||)

collapse2 :: (h1 -> h2 -> h) -> T h1 (T h2 t) -> T h t
collapse2 f = collapse (hMap . f)
```

A.3 Table Inversion

```
module Inversion where

import Tables
```

Inversion of tables with one-cell grids is easy, since only the grid cell content needs to be "pushed up" by appropriate deep-dimensional header insertions: the following functions take two arguments h and t_0, and deliver the inversion (in the respective dimension) of table $h \triangleright t_0$ (we use the type variable `u` where normally the unit type `()` would be substituted):

```
invC1 :: h1 -> T                  c u    -> T c (T                  h1 u   )
invC2 :: h1 -> T h2 (T            c u  ) -> T c (T h2 (T            h1 u  ))
invC3 :: h1 -> T h2 (T h3 (T      c u )) -> T c (T h2 (T h3 (T      h1 u )))
invC4 :: h1 -> T h2 (T h3 (T h4 (T c u))) -> T c (T h2 (T h3 (T h4 (T h1 u))))
```

```
invC1 = addH2

liftInvC :: (h1 -> t -> T c t') -> h1 -> T h2 t -> T c (T h2 t')
liftInvC g h1 = tFoldM addH2 (|||) (g h1)

invC2 = liftInvC invC1
invC3 = liftInvC invC2
invC4 = liftInvC invC3
```

Table inversion as presented in the literature is complicated by the requirement that the resulting tables be regular.
If we allow the result table to be ragged, we only need to embed the appropriate cell inversion function into a simple table folding:

```
raggedInv2 :: T h1 (T h2 (T             c u  )) -> T c (T h2 (T             h1 u  ))
raggedInv3 :: T h1 (T h2 (T h3 (T       c u ))) -> T c (T h2 (T h3 (T       h1 u )))
raggedInv4 :: T h1 (T h2 (T h3 (T h4 (T c u)))) -> T c (T h2 (T h3 (T h4 (T h1 u))))

raggedInv2 = tFold invC2 (|||)
raggedInv3 = tFold invC3 (|||)
raggedInv4 = tFold invC4 (|||)
```

In the following, we let inversion always be targeted to production of *regular* tables. The essential change necessary to achieve this is replacing the horizontal concatenation `|||` in `liftInvC` with a *diagonal concatenation* that pads the remaining "grid space" with regular tables filled with "empty values" and carefully adapted skeletons.
Inversion of higher-dimensional normal tables needs some auxiliary functions, first of all for eliminating individual dimensions (in which case only one of each list of lower-dimensional subtables can be preserved):

```
delH1 :: T a b -> b
delH1 = tFold (\ h t -> t) const

delH2 :: T a (T b c) -> T a c
delH2 = tMap delH1
delH3 = tMap delH2
delH4 = tMap delH3
```

Using these, we can define "header slimming functions" which slim down the header of the selected dimension to a singleton containing the supplied entry:

```
slimH1 :: a -> T b                u    -> T                a u
slimH2 :: a -> T b (T c           u  ) -> T b (T           a u  )
slimH3 :: a -> T b (T c (T d      u )) -> T b (T c (T      a u ))
slimH4 :: a -> T b (T c (T d (T e u))) -> T b (T c (T d (T a u)))

slimH1 u = (>||) u . delH1
slimH2 = tMap . slimH1           -- = addH2 u . delH2
slimH3 = tMap . slimH2           -- = addH3 u . delH3
slimH4 = tMap . slimH3           -- = addH4 u . delH4
```

Since cells are headers, too, we do not need separate cell update functions like `updC`n from page 19; these are instances (obtained by setting `u = ()` in the types) of the header slimming functions:

```
updC0 = slimH1 :: a -> T b                  ()    -> T                a ()
updC1 = slimH2 :: a -> T b (T c             ()  ) -> T b (T           a ()  )
updC2 = slimH3 :: a -> T b (T c (T d        () )) -> T b (T c (T      a () ))
updC3 = slimH4 :: a -> T b (T c (T d (T e ()))) -> T b (T c (T d (T a ())))
```

The "slimming" effect is lost on the cell level as long as we do not permit cell concatenation. the Isabelle proofs for semantics preservation of table inversion include the possibility of cell concatenation; for Theorem 6.3.1, this results only in one additional distributivity law that is required to hold for the cell-level concatenation semantics.

Diagonal concatenation of regular tables uses horizontal and vertical concatenations to arrange the two argument tables as blocks on the main diagonal of the top two dimensions, and uses "empty" tables of appropriate skeletons to fill the whole concatenation up to be a regular table again:

```
dConc :: (T a b -> T a b) -> T c (T a b) -> T c (T a b) -> T c (T a b)
dConc h t1 t2 = (t1 /// fill0 h t1 t2)  |||  (fill0 h t2 t1 /// t2)
```

The fillers are parameterised by an "emptying" function `h` that acts on subtables below the first-dimension headers. The filling function preserves of its first argument table only the headers of the first dimension, and of the second argument table the headers of all other dimensions; the "emptying function" will usually update the cells, which may be at an arbitrarily deep dimension.

```
fill0 :: (t -> t') -> T h t0 -> T h0 t -> T h t'
fill0 h t1 t2 = tMap (const $ h $ delH1 t2) t1
```

In the special case of two dimensional tables, the emptying function will be `updC1`, as on page 37:

```
fill :: a -> T b1 t' -> T b1' (T b2 (T a' ())) -> T b1 (T b2 (T a ()))
fill u = fill0 (updC1 u)
```

Here, however, we will not need that function any further since we subsume the two-dimension inversion into a more general approach. This approach involves *inversion operators* that perform inversion for tables of the shape `h1 >|| t`; inversion of arbitrary tables is obtained from these inversion operators via folding with `(|||)`:

```
inverse1 ::     T a (T b c)
            -> T b (T a c)
inverse1 = tFold invOp1 (|||)

inverse2 :: a -> T a (T b (T c d))
              -> T c (T b (T a d))
inverse2 u = tFold (invOp2 u) (|||)

inverse3 :: a -> T a (T b (T c (T d e)))
              -> T d (T b (T c (T a e)))
inverse3 u = tFold (invOp3 u) (|||)

inverse4 :: a -> T a (T b (T c (T d (T e f))))
              -> T e (T b (T c (T d (T a f))))
inverse4 u = tFold (invOp4 u) (|||)
```

The operators themselves are defined by induction over the dimension; for one-dimensional tables this is just `addH2`:

```
invOp1 ::      a -> T b c
         -> T b   (T a c)
invOp1 = addH2
```

Higher-dimensional inversion operators are obtained using an “inversion lifting” function that in addition needs to supply the emptying function corresponding to the dimensionality to diagonal concatenation:

```
invLift :: (T b c -> T b c) ->
           (d ->     e -> T f      c) ->
           (d -> T b e -> T f (T b c))
invLift h g = \ h1 -> tFoldM addH2 (dConc h) (g h1)
```

This allows to systematically build inversion operators for arbitrary dimensionality; the partially applied slimming functions (usually in their rôle as cell update functions) are supplied for being passed to diagonal concatenation:

```
invOp2 :: a ->   a -> T b (T c d)
           -> T c   (T b (T a d))
invOp2 u = invLift (slimH2 u) invOp1

invOp3 :: a ->   a -> T b (T c (T d e))
           -> T d   (T b (T c (T a e)))
invOp3 u = invLift (slimH3 u) (invOp2 u)

invOp4 :: a ->   a -> T b (T c (T d (T e f)))
           -> T e   (T b (T c (T d (T a f))))
invOp4 u = invLift (slimH4 u) (invOp3 u)
```

Composing with transposition functions, we obtain “inner” inversions, too:

```
inverse2_2 :: b -> T a (T b (T c d))
                -> T a (T c (T b d))

inverse3_2 :: b -> T a (T b (T c (T d e)))
                -> T a (T d (T c (T b e)))

inverse3_3 :: c -> T a (T b (T c (T d e)))
                -> T a (T b (T d (T c e)))

inverse4_2 :: b -> T a (T b (T c (T d (T e f))))
                -> T a (T e (T c (T d (T b f))))

inverse4_3 :: c -> T a (T b (T c (T d (T e f))))
                -> T a (T b (T e (T d (T c f))))

inverse4_4 :: d -> T a (T b (T c (T d (T e f))))
                -> T a (T b (T c (T e (T d f))))

inverse2_2 u = tTranspose  . inverse2 u . tTranspose
inverse3_2 u = tTranspose  . inverse3 u . tTranspose
inverse3_3 u = tTranspose3 . inverse3 u . tTranspose3
inverse4_2 u = tTranspose  . inverse4 u . tTranspose
inverse4_3 u = tTranspose3 . inverse4 u . tTranspose3
inverse4_4 u = tTranspose4 . inverse4 u . tTranspose4
```

A.4 Table Evaluation Structures

This module exports, besides auxiliary datatypes, abstract datatype constructors for wrapper-less table evaluation structures (TESs). Not including wrappers considerably simplifies the presentation of this module; if wrappers are needed, they can easily be added by an additional layer pairing a wrapper-less TES with a wrapper function (representation).

```
module TES( FRep1(..), FRep2(..), F2(..)
          , CPR(), mkCPR, hComb, cComb
          , TESnil(), mkTES0, tEval0
          , TEScons(), consTES
          , tEval1, TES1, mkTES1
          , tEval2, TES2, mkTES2
          , tEval3, TES3, mkTES3
          , TES(..)
          , module Tables
          ) where

import Tables
```

The ingredients of TESs are functions, mostly grouped in combinator pairs, which are used in table evaluation. In a table transformation support tool, however, there may be the need to, rather than applying them, *inspect* those functions, for example in order to base some transformation decision on the result of this inspection.

Therefore, we allow arbitrary representations of functions, as long as they offer an interface for application. Such an interface is naturally encoded as a type class, and we need representations of unary and binary functions:

```
class FRep1 f1 where
  apply1 :: f1 a b -> a -> b

class FRep2 f2 where
  apply2 :: f2 a b c -> a -> b -> c
```

Although many applications will not be able to provide fully polymorphic representation types, standard techniques like “dummy parameters” together with type constraints can still be used to produce the required interfaces. However, polymorphic representation types enable us to restrict larger TESs to use the same representation type at every dimension without compromising flexibility of the header types or type safety.

Functions themselves are of course the most obvious function representations:

```
instance FRep1 (->) where apply1 f x = f x
instance FRep2 F2   where apply2     = applyF2

newtype F2 a b c = F2 {applyF2 :: (a -> b -> c)}
```

Combinator pair representations are then pairs of binary function representations — we export their type as an abstract data type, enforcing their consistency by constraining the types of the construction and access functions:

```
newtype CPR f2 h s r = CPR (f2 h s r, f2 r r r)
mkCPR :: FRep2 f2 => f2 h s r -> f2 r r r -> CPR f2 h s r
mkCPR = curry CPR
```

```
hComb :: FRep2 f2 => CPR f2 h s r -> (h -> s -> r)
cComb :: FRep2 f2 => CPR f2 h s r -> (r -> r -> r)
hComb (CPR p) = apply2 (fst p)
cComb (CPR p) = apply2 (snd p)
```

The typical use of a combinator pair is in a `tFold`; so we define variants that take combinator pairs as arguments:

```
tFoldCPR :: FRep2 f2 => CPR f2 h s r -> T h s -> r
tFoldCPR cp = tFold (hComb cp) (cComb cp)

tFoldMCPR :: FRep2 f2 => CPR f2 h s r -> (t -> s) -> T h t -> r
tFoldMCPR cp = tFoldM (hComb cp) (cComb cp)
```

Combinator pair representations of course *can* be just pairs of functions:

```
type CPairF h s r = CPR F2 h s r
```

For table evaluation structures, we face the same problem as for tables, that Haskell does not support parameterisation of type constructors by type lists.

For tables, we solved this by a type family generated by two constructors: the zero-ary constructor `()` for "subtables below cells", and the parameterised (with parameter `h`) family of unary type constructors `T h` for producing the types of $(n+1)$-dimensional tables from types of n-dimensional tables.

Here, we follow the same approach for table evaluation structures (disregarding the wrappers):

- the type constructor `TESnil` produces types for TESs for sub-cell tables (i.e., tables z used to form cells containing c in the construction $c \triangleright z$) — which in standard cases have type `()`, and
- the type constructor `TEScons` takes a header type `h`, a type `tes` of TESs for n-dimensional tables, and some auxiliary type parameters, and constructs from them the types of TESs for $(n+1)$-dimensional tables.

Accommodating the full flexibility of our table types, we allow TES application to start from an arbitrary "sub-cell type" `t`, and evaluation of such an "unstructured" table can be an arbitrary function from `t` to some result type `r`:

```
newtype (FRep1 f1, FRep2 f2) => TESnil t r f1 f2 = TESnil (f1 t r)

mkTES0 :: (FRep1 f1, FRep2 f2) => f1 t r -> TESnil t r f1 f2
mkTES0 f = TESnil f
```

For obtaining a homogeneous system of TES types, we parameterise `TESnil` not only with the type `f1` for unary function representations that it needs, but also with `f2` for binary function representations.

The constraint for the `newtype` is necessary to enforce the correct kind `* -> * -> * -> *` for `f2`, which otherwise would be assigned the default kind `*`; this would lead to clashes later.

Applying such a trivial TES means applying the function it consists of:

```
tEval0 :: (FRep1 f1, FRep2 f2) => TESnil t r f1 f2 -> t -> r
tEval0 (TESnil f) = apply1 f
```

The step from dimension n to dimension $(n+1)$ in wrapper-less TESs is performed by adding to a TES for n-dimensional table evaluation, typed with a `tes` type, a combinator pair representation of type `cp` for the new header type `h`, the subtable result type `s`, and the new result type `r`:

```
data (FRep1 f1, FRep2 f2) => TEScons h tes s r f1 f2 =
                             TEScons (CPR f2 h s r) (tes f1 f2)
```

Here, the main task for the constraint is to supply the correct kind `* -> * -> *` for `f1`.

```
consTES :: (FRep1 f1, FRep2 f2) =>
           CPR f2 h s r -> tes f1 f2 -> TEScons h tes s r f1 f2
consTES = TEScons
```

Applying such a TES involves applying the sub-TES to all subtables (via `tMap`) and then fold the combinator pair over the top-level structure:

```
tEval1 (TEScons cp tes) = tFoldMCPR cp (tEval0 tes)
tEval2 (TEScons cp tes) = tFoldMCPR cp (tEval1 tes)
tEval3 (TEScons cp tes) = tFoldMCPR cp (tEval2 tes)
tEval4 (TEScons cp tes) = tFoldMCPR cp (tEval3 tes)
```

These evaluation functions have the following principal types:

```
tEval1 :: (FRep1 f1, FRep2 f2) =>
          TEScons h1 (TESnil t g2) g2 g1 f1 f2 -> T h1 t -> g1

tEval2 :: (FRep2 f2, FRep1 f1) =>
          TEScons h1 (TEScons h2 (TESnil t g3) g3 g2) g2 g1 f1 f2
          -> T h1 (T h2 t) -> g1

tEval3 :: (FRep1 f1, FRep2 f2) =>
          TEScons h1 (TEScons h2 (TEScons h3 (TESnil t g4) g4 g3) g3 g2) g2 g1 f1 f2
          -> T h1 (T h2 (T h3 t)) -> g1

tEval4 :: (FRep1 f1, FRep2 f2) =>
          TEScons h1 (TEScons h2 (TEScons h3 (TEScons h4 (TESnil t g5)
                                                   g5 g4) g4 g3) g3 g2) g2 g1 f1 f2
          -> T h1 (T h2 (T h3 (T h4 t))) -> g1
```

As abbreviations, we introduce type synonyms for the "well-founded" TES types that are the arguments of these TES application functions — the following type equations are all at kind `(* -> * -> *) -> (* -> * -> * -> *) -> *`, since the `TES`n type synonyms all accept as two additional arguments the type constructors `f1 : * -> * -> *` and `f2 : * -> * -> * -> *` of the respective function representations:

```
type TES1 h1          t          g2 g1 = TEScons h1 (TESnil        t          g2) g2 g1
type TES2 h1 h2       t       g3 g2 g1 = TEScons h1 (TES1 h2       t       g3 g2) g2 g1
type TES3 h1 h2 h3    t    g4 g3 g2 g1 = TEScons h1 (TES2 h2 h3    t    g4 g3 g2) g2 g1
type TES4 h1 h2 h3 h4 t g5 g4 g3 g2 g1 = TEScons h1 (TES3 h2 h3 h4 t g5 g4 g3 g2) g2 g1
```

Unlike in Def. 4.3.1, here we cannot suppress the list of intermediate types γ_i from the externally visible type unless we use existentially quantified types, a non-standard type system extension available in some Haskell implementations, but not in HOL as supported by Isabelle.

We define functions with specialised types that help to assemble TESs for specific dimensions:

```
mkTES1 :: (FRep1 f1, FRep2 f2) =>
          CPR f2 h1 g2 g1 -> f1 t g2 -> TES1 h1 t g2 g1 f1 f2
mkTES1 cp f = TEScons cp (TESnil f)
```

```
mkTES2 :: (FRep1 f1, FRep2 f2) =>
          CPR f2 h1 g2 g1 ->
          CPR f2 h2 g3 g2 ->
          f1 t g3 ->
          TES2 h1 h2 t g3 g2 g1 f1 f2
mkTES2 cp1 cp2 f =  TEScons cp1 (mkTES1 cp2 f)

mkTES3 :: (FRep1 f1, FRep2 f2) =>
          CPR f2 h1 g2 g1 ->
          CPR f2 h2 g3 g2 ->
          CPR f2 h3 g4 g3 ->
          f1 t g4 ->
          TES3 h1 h2 h3 t g4 g3 g2 g1 f1 f2
mkTES3 cp1 cp2 cp3 f =  TEScons cp1 (mkTES2 cp2 cp3 f)

mkTES4 :: (FRep1 f1, FRep2 f2) =>
          CPR f2 h1 g2 g1 ->
          CPR f2 h2 g3 g2 ->
          CPR f2 h3 g4 g3 ->
          CPR f2 h4 g5 g4 ->
          f1 t g5 ->
          TES4 h1 h2 h3 h4 t g5 g4 g3 g2 g1 f1 f2
mkTES4 cp1 cp2 cp3 cp4 f =  TEScons cp1 (mkTES3 cp2 cp3 cp4 f)
```

Similar to the type class `Table` in A.1, we can again use a multi-parameter type class to discern "well-founded" TES type constructors. Here we need a three-parameter class, and if the class predicate `TES tes t r` holds for a type constructor

```
tes :: (* -> * -> *) -> (* -> * -> * -> *) -> *
```

for TESs, and types `t` for tables, and `r` for results, we read this as saying:

> "TESs of types that are appropriate instances (via function representations `f1` and `f2`) of `tes` can be applied to tables of type `t` producing results of type `r`."

In addition, we will use this class predicate only in contexts where for a given TES type, the table and result type are uniquely determined; we record this fact by including *functional dependencies* in the class declaration (note that the table type does not depend on the function representation type constructors):

```
class TES tes t r | tes -> t, tes -> r where
  tEval :: (FRep1 f1, FRep2 f2) => tes f1 f2 -> t -> r
```

The class member `tEval` obviously provides the TES application function as "witness" for our interpretation of the class predicate.

Using this understanding, we can easily produce the instance declarations for our two TES datatype constructors:

```
instance TES (TESnil t r) t r where
  tEval (TESnil f) t = apply1 f t

instance TES tes t t'
      => TES (TEScons h tes t' r) (T h t) r
 where
  tEval (TEScons cp tes) = tFoldMCPR cp (tEval tes)
```

With these instance declarations, the individual evaluation functions defined above all become instances of `tEval` at the types derived from their direct definitions:

```
tEval0 = tEval ::
  (FRep1 f1, FRep2 f2) => TESnil                   t              g1 f1 f2
                       ->                          t           -> g1

tEval1 = tEval ::
  (FRep1 f1, FRep2 f2) => TES1 h1                  t           g2 g1 f1 f2
                       -> T    h1                  t           -> g1

tEval2 = tEval ::
  (FRep1 f1, FRep2 f2) => TES2 h1    h2            t        g3 g2 g1 f1 f2
                       -> T    h1 (T h2            t         ) -> g1

tEval3 = tEval ::
  (FRep1 f1, FRep2 f2) => TES3 h1    h2    h3      t     g4 g3 g2 g1 f1 f2
                       -> T    h1 (T h2 (T h3      t      )  ) -> g1

tEval4 = tEval ::
  (FRep1 f1, FRep2 f2) => TES4 h1    h2    h3    h4 t g5 g4 g3 g2 g1 f1 f2
                       -> T    h1 (T h2 (T h3 (T h4 t  )  )  ) -> g1
```

A.5 Tabular Expressions as Abstract Data Type

```
module TE(TE(), mkTE, tes, table, teEval, module TES) where

import TES
```

A tabular expression is a table together with a TES for that table:

```
data (TES tes t r, FRep1 f1, FRep2 f2) =>
     TE f1 f2 tes t r = TE (tes f1 f2) t
```

Here we let the function representation type constructors be the first arguments, since for any particular tool, they are most likely to be fixed. For the TES types in module `TES` they have to be the last arguments since only that way it was possible to keep them outside the `TES` class, and to enforce homogeneous representation type constructors across dimensions.

We export `TE` as an abstract data type; the exported constructors and access functions in this case all have their principal types:

```
mkTE :: (TES tes t r, FRep1 f1, FRep2 f2) =>
        tes f1 f2 -> t -> TE f1 f2 tes t r
mkTE = TE

tes   :: (TES tes t r, FRep1 f1, FRep2 f2) => TE f1 f2 tes t r -> tes f1 f2
tes (TE te_tes te_t) = te_tes

table :: (TES tes t r, FRep1 f1, FRep2 f2) => TE f1 f2 tes t r -> t
table (TE te_tes te_t) = te_t
```

Thanks to the functional dependencies in the declaration of class `TES`, the declared type of the tabular expression evaluation function is principal, too:

```
teEval :: (TES tes t r, FRep1 f1, FRep2 f2) => TE f1 f2 tes t r -> r
teEval te = tEval (tes te) (table te)
```

A.6 Interpreter-Supported Table Interaction

This module is intended primarily for use in an interpreter; it explicitly imports all maximal table modules and re-exports them; since the module chain `Table` → `Tables` → `TES` → `TE` also re-exports at every step, all their exports from all table modules are exported from `TTop` and therefore available interactively.

This module is not Haskell 98 because of the use of multi-parameter classes and functional dependencies in `TES`.

Use:

```
hugs -98 TTop.lhs
ghci -fglasgow-exts TTop.lhs
```

```
module TTop(module TE, module Inversion) where

import TE
import Inversion
```

References

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[BKS97] Chris Brink, Wolfram Kahl, and Gunther Schmidt, editors. *Relational Methods in Computer Science*. Advances in Computing Science. Springer, Wien, New York, 1997.

[DKM01] Jules Desharnais, Ridha Khedri, and Ali Mili. Interpretation of tabular expressions using arrays of relations. In E. Orlowska and A. Szalas, editors, *Relational Methods for Computer Science Applications*, volume 65 of *Studies in Fuzziness and Soft Computing*, pages 3–14, Heidelberg, 2001. Springer-Physica Verlag.

[Far03] William M. Farmer. The seven virtues of simple type theory. SQRL Report 14, Software Quality Research Laboratory, Department of Computing and Software, McMaster University, October 2003. available from http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html.

[GM93] Michael J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[HKPS78] K.L. Heninger, J. Kallander, David Lorge Parnas, and J.E. Shore. Software requirements for the A-7E aircraft. NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington DC, November 1978.

[HPU81] S.D. Hester, D.L. Parnas, and D.F. Utter. Using documentation as a software design medium. *Bell System Tech. J.*, 60(8):1941–1977, October 1981.

[ISO02] ISO/IEC 13568:2002. Information technology — Z formal specification notation — syntax, type system and semantics, 2002. International Standard.

[Jan95] Ryszard Janicki. Towards a formal semantics of Parnas tables. In *Proc. of the* 17^{th} *Internat. Conf. on Software Engineering, Seattle, WA*, pages 231–240, 1995.

[JK01] Ryszard Janicki and Ridha Khedri. On a formal semantics of tabular expressions. *Science of Computer Programming*, 39:189–213, 2001.

[JPZ97] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. In Brink et al. [BKS97], chapter 12, pages 184–196.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Knu92] Donald E. Knuth. *Literate Programming*, volume 27 of *CSLI Lecture Notes*. Center for the Study of Language and Information, 1992.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PAM91] David Lorge Parnas, G.J.K. Asmis, and Jan Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, 1991.

[Par92] David Lorge Parnas. Tabular representation of relations. Technical Report CRL Report 260, McMaster Univ., Communications Research Laboratory, TRIO (Telecommunications Research Inst. of Ontario), October 1992.

[Par94] David Lorge Parnas. Inspection of safety critical software using program-function tables. In K. Duncan and K. Krueger, editors, 13^{th} *World Computer Congress 1994, Vol. 3: Linkage and Developing Countries*, volume A-53 of *IFIP Transactions*, pages 270–277. North-Holland, August 1994. Invited paper.

[PJ+03] Simon Peyton Jones et al. *The Revised Haskell 98 Report.* Cambridge Univ. Press, 2003. Also on http://haskell.org/.

[San98] Thomas Santen. On the semantic relation of Z and HOL. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98, The Z Formal Specification Notation*, volume 1493 of *LNCS*, pages 98–115. Springer, 1998.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1989. Out of print; available via URL: http://spivey.oriel.ox.ac.uk/~mike/zrm/.

[SS93] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists.* EATCS-Monographs on Theoretical Computer Science. Springer, 1993.

[SZP96] Hong Shen, Jeffery I. Zucker, and David L. Parnas. Table transformation tools: Why and how. In *COMPASS '96: Proceedings of the Eleventh Annual Conference on Computer Assurance, Gaithersburg, Maryland, USA, June 1996*, pages 3–11. IEEE and National Institute of Standards and Technology, 1996.

[ZS98] Jeffery I. Zucker and Hong Shen. Table transformation: Theory and tools. Technical Report CAS 98-01, McMaster University, Dept of Computing and Software, 1998. also Communications Research Laboratory (CRL) Report 363.

[Zuc96] Jeffery I. Zucker. Transformations of normal and inverted function tables. *Formal Aspects of Computing*, 8:679–705, 1996. (Also as CRL Report No. 291, August 1994, McMaster University, Communications Research Laboratory and Telecommunications Research Inst. of Ontario.).

List of Figures

Index