Typed Table Transformations

Martin Erwig
Oregon State University
erwig@oregonstate.edu

***Abstract*—Spreadsheet tables are often labeled, and these labels effectively constitute types for the data in the table. In such cases tables can be considered to be built from typed data where the placement of values within the table is controlled by the types used for rows and columns. We present a new approach to the transformations of spreadsheet tables that is based on transformations of row and column types. We illustrate the basic idea of type-based table construction and transformation and lay out a series of research questions that should be addressed in future work.**

I. Introduction

Spreadsheets present data and computation with those data in tabular form. As reported by Harris and Gulwani [1], Excel users often face the problem of transforming tables. Consider, for example, the table in Figure 1(a), which is adapted from [1] and based on the actual transformation needs by an Excel user[1]. In our version the table shows the earnings of three companies for three different quarters. Suppose we want to transform this table into a list that shows individual earnings in separate rows for each company and quarter, ignoring empty cells for non-existing data. The result should look like the table shown in Figure 1(b).

Harris and Gulwani describe in their paper an algorithm that can infer transformations of tables such as from (a) to (b) from input/output examples. In this example, as in many others, we can observe that the rows and columns of tables contain labels that explain the (purpose of the) data in the table. These labels can be interpreted as type information for the data in the table [2], [3]. In this paper we present an approach that exploits this fact and lets users describe table transformations based on transformation of their row and column types.

Based on the concepts of row and column types and typed tables, we describe in Section II how tables can be systematically constructed from attributed data driven by types. In Section III we then show how table transformations can be expressed through transformations and operations on their types. We discuss some related work in Section IV and present conclusions in Section V where we also lay out a plan for future work. Since this is a short paper reporting on work in progress, the focus is on explaining the major ideas and identifying research questions to be addressed in the future.

II. Typed Tables

The data items in a two-dimensional table are uniquely identified by their row and column positions. When the rows and columns are labeled, these labels can serve as names for the table positions, which provides a more high-level, domain-centered way for talking about data placement in tables. For example, to find the value 2.9 we can look into the second row and third column of the table 1(a), or we can look up the value for company A and quarter Q2. In the following we formalize this idea.

	Q1	Q2	Q3
A	3.5	2.9	4.0
B	3.2		4.3
C		4.9	

(a) Original Table

A	Q1	3.5
A	Q2	2.9
A	Q3	4.0
B	Q1	3.2
B	Q3	4.3
C	Q2	4.9

(b) Linearized

A	4.0
B	4.3
C	4.9

(c) Aggregated

Fig. 1. (a) A table showing the earnings of three companies in three different quarters. (b) The same data in linearized form. (c) The data for each company aggregated over all quarters.

A. Values and Types

In the context of spreadsheets, values ($v \in V$) include simple data types such as numbers, dates, or strings. We use the metavariable n to range over those values (typically strings) that are used as type names.

A *domain type* ($\delta \in \Delta$) consists of a name and a finite set of values. A domain type is different from predefined types such as *Int* or *String* ordinarily found in programming languages: It is defined by a user and is used to indicate the nature, origin, or purpose of other values. A simple example of a domain type is $\mathit{Company}\langle \mathrm{A}, \mathrm{B}, \mathrm{C} \rangle$ where *Company* is the name of the type and A, B, and C are its values. Domain types such as *Company* that contain only plain values are called *plain*. Otherwise, they are called *refined*. We will see types with refined values later in Section III-B.

An *attribute* is a value, such as Joe, associated with a type name, such *Name*, and is written as *Name*=Joe. Note that attributes can be formed arbitrarily; in particular, the value does not have to be an element of the associated type. A set of attributes is called a *record*, and a value with an associated record is called an *attributed value*.

Finally, a *table type* ($\tau \in \delta \times \delta$) consists of a pair of domain types, the first representing the column type and the second representing the row type, and a table $t \in T$ is a mapping from addresses, represented as pairs of natural numbers, to values. The syntax of values and types is summarized in Figure 2. The attentive reader will notice that we do not consider formulas in this model.

[1] http://www.excelforum.com/excel-programming-vba-macros/698490-using-a-macro-to-extract-and-rearrange-data.html

Values & Type Names	$v, n \in V$
Refined Values	$\hat{v} \in \hat{V} ::= v \mid v \prec \delta$
Domain Types	$\delta, \gamma, \rho \in \Delta ::= n\langle \hat{v}^* \rangle$
Table Types	$\tau \in \delta \times \delta$
Attributes	$a \in A ::= n{=}v$
Records	$r \in R ::= \{a^*\}$
Attributed Values	$\bar{v} \in \bar{V} ::= v^r$
Attributed Data	$d \in D = 2^{\bar{V}}$
Tables	$t \in T = \mathbb{N} \times \mathbb{N} \to V$

Fig. 2. Syntax of values and types

B. Tables

Our approach to table transformations is based on the premise that tables are the result of the systematic presentation of attributed values. Specifically, the construction of tables is driven by types that are associated with their rows and columns. For example, the table in Figure 1(a) is the result of creating a table with column type $\mathit{Quarter}\langle \mathrm{Q1}, \mathrm{Q2}, \mathrm{Q3} \rangle$ and row type $\mathit{Company}\langle \mathrm{A}, \mathrm{B}, \mathrm{C} \rangle$ from a set of attributed values such as the following:

$$\{2.9^{\{\mathit{Company}=\mathrm{A}, \mathit{Quarter}=\mathrm{Q2}, \ldots\}}, 3.2^{\{\mathit{Company}=\mathrm{B}, \mathit{Quarter}=\mathrm{Q1}, \ldots\}}, \ldots\}$$

For example, the first attributed value in this set is 2.9, which has (at least) the two attribute values A of type *Company* and Q2 of type *Quarter*; it may have further attributes, but these two are relevant for the proper placement of the value in the $\mathit{Quarter} \times \mathit{Company}$ table, which works by looking up the position of the two attributes in their respective types. Since Q2 is the second element of the type *Quarter* and A is the first element of the type *Company*, the value 2.9 is placed in column 2 and row 1 of the core table.[2] In the same fashion all the other values will be positioned based on their attributes.

The locations of empty cells in the table correspond to attribute combinations for *Quarter* and *Company* attributes that do not occur with values in the data set.

This approach to the construction of tables can be formalized by a function $\boxplus : \Delta \times \Delta \times D \to T$. The definition employs the auxiliary lookup function $\delta{\uparrow}r$, which searches for an attribute $n{=}v$ in the record r given domain type $\delta = n\langle \hat{v}_1, \ldots, \hat{v}_k \rangle$ and, if found, determines v's position among δ's values $\hat{v}_1, \ldots, \hat{v}_k$, which then provides the row or column for v in the constructed table. The definition for $\uparrow$ is obvious if all the $\hat{v}_i$ are plain values. The case for types with refined values is more involved and will be discussed later. The definition of $\boxplus$ is now straightforward. Note that we generally employ the metavariable γ for column types and ρ for row types.

$$\tfrac{\gamma}{\rho}\boxplus D = \{((x, y), v) \mid v^r \in D \wedge r{\uparrow}\gamma = x \wedge r{\uparrow}\rho = y\}$$

Note that the function $\boxplus$ only builds the core part of the table consisting of the data values. In addition, we need to add the column and row headers. Since the corresponding definitions are not very interesting, we omit them here for brevity.

Assuming that the data source of attributed values is D, we can now construct the table shown in Figure 1(a) with the following expression.

$$t = \tfrac{\mathit{Quarter}}{\mathit{Company}}\boxplus D$$

From the definition of $\boxplus$ we can immediately infer the following properties of table construction.

First, as already mentioned, values whose attributes do not contain values of both row and column type will be not placed, that is, data with insufficient attributes are simply ignored.

Second, when data items have the same attribute values for the row and column types, the set definition does not produce a function and is effectively undefined in terms of its result type (which is $T = \mathbb{N} \times \mathbb{N} \to V$). In this case the table construction simply fails, since different values would be mapped to the same locations, resulting in an ambiguity. This interpretation is probably too strict, since many application scenarios could benefit from constructing tables with underspecified type information. These cases can be handled in two different ways: (1) We can preserve multiple values by mapping a row/column combination not just to a single cell but to a group of cells. This complicates the computation of cell locations, but is otherwise not difficult to achieve in principle. (2) We can apply aggregating functions such as SUM to aggregate a set of values into one value.

Third, it is easy to see that tables built with $\boxplus$ can be transposed by simply exchanging the column and row type, that is, we know the following identity holds.

$$\left(\tfrac{\gamma}{\rho}\boxplus D\right)^{\mathrm{T}} = \tfrac{\rho}{\gamma}\boxplus D$$

III. Table Transformations Through Type Operators

Since the structure of tables built with $\boxplus$ depends on the row and column types, it is not surprising that changes to these types result in corresponding changes for the constructed tables. In this section we present different kinds of type transformations and discuss how they give rise to corresponding table transformations.

A. Transforming of Row and Column Types

Consider again the table shown in Figure 1(a) and defined as t in the previous section. Suppose now that we want this table to show the data only for companies A and C and also only for the first and third quarter. We can apply the selection criteria to the row and column types using a selection operation $\sigma_P(\delta)$ to filter out all elements from type δ that do not satisfy the predicate P. By using filtered domain types in the $\boxplus$ operation we can build the correspondingly amended table.[3]

$$\tfrac{\sigma_{\_\neq \mathrm{Q2}}(\mathit{Quarter})}{\sigma_{\_\neq \mathrm{B}}(\mathit{Company})}\boxplus D$$

[2]The table row and column headers that are given by the values of the corresponding row and value types are later added to the core table and are ignored in this calculation.

[3]We use the Purescript notation for partial function application in which a binary operation applied to one of its arguments denotes a function of its other remaining argument, for example, $\_{\neq}x \equiv \lambda y.\, y{\neq}x$.

Especially for bigger tables and with more complicated selection criteria, reconstructing the table in this way is probably faster and less error-prone than directly editing the table in Excel by repeatedly cutting and pasting rows and columns.

Note that we can achieve the same effect by filtering the data source D directly with a conjunction of the two predicates.

$$\overset{Quarter}{Company}\boxplus(\sigma_{Company\neq \mathrm{B}\wedge Quarter\neq \mathrm{Q2}}(D))$$

In general, when $\gamma = n\langle\ldots\rangle$ and $\rho = m\langle\ldots\rangle$, we can observe the following equivalence between type and data selection.

$$\begin{matrix}\sigma_Q(\gamma)\\ \sigma_P(\rho)\end{matrix}\boxplus D \;=\; \overset{\gamma}{\rho}\boxplus(\sigma_{P(n)\wedge Q(m)}(D))$$

Why then should we bother about the manipulation of row and column types? Having selection available for table types has several potential advantages. First, the row and column type selections show more directly the effect of the selection on the structure and shape of the table than does the selection of the data. Second, and more importantly, some effects on table restructuring cannot be easily achieved by manipulating the data. Consider, for example, the task of reordering the rows of table t. This can be accomplished through the following expression.

$$\overset{Quarter}{rev(Company)}\boxplus D$$

However, it is not clear how to achieve this effect through a transformation of D.

We can envision a number of other operations on row and column types that can be put to use in the manipulating tables. For example, dually to filtering types we can also extend types by new values, which amounts to growing tables by row or columns. Of course, this produces results only if the underlying data also contain correspondingly attributed values.

B. Type Refinement

A close look at the definition of δ in Figure 2 reveals that a type is essentially a tree structure with a type name as its root, values as the root's children, and potentially other types as subtrees of children that are refined. For plain types these trees are rather trivial and have height 2, and consequently, table construction with plain types covers only rather simple, albeit useful, application scenarios.

Refined types correspond to trees with a more complex structure, and so more sophisticated table constructions and transformations can be achieved with type operations that create or modify types containing refined values.

The first such operation is type refinement $\delta' \otimes \delta$ that refines a domain type δ' by another domain type δ which means to attach the whole type δ to every leaf value in δ'. The formal definition is as follows.

$$\begin{aligned} n\langle\hat{v}_1,\ldots,\hat{v}_k\rangle \otimes \delta &= n\langle\hat{v}_1\otimes\delta,\ldots,\hat{v}_k\otimes\delta\rangle\\ &\text{where } (v\prec\delta')\otimes\delta = v\prec(\delta'\otimes\delta)\\ &\qquad\quad v\otimes\delta = v\prec\delta \end{aligned}$$

The notion of type membership gets a bit more interesting for types with refined values, since values occur at multiple levels. In fact, each path from the root to a leaf represents a record of (n, v) pairs. In the following we define a function $\lfloor\_\rfloor$ that computes for each type the sequence of records it represents. In the definition we use $\cdot$ to denote the concatenation of sequences and $\cup$ for computing the union of two records.

$$\begin{aligned} \lfloor n\langle\,\rangle\rfloor &= \langle\,\rangle\\ \lfloor n\langle v,\hat{v}^*\rangle\rfloor &= \langle\{n{=}v\}\rangle\cdot\lfloor n\langle\hat{v}^*\rangle\rfloor\\ \lfloor n\langle v\prec\delta,\hat{v}^*\rangle\rfloor &= \langle\{n{=}v\}\cup r \mid r\in\lfloor\delta\rfloor\rangle\cdot\lfloor n\langle\hat{v}^*\rangle\rfloor \end{aligned}$$

For a type without any refined values, the records contain only single attributes.

$$\lfloor Company\langle \mathrm{A},\mathrm{B}\rangle\rfloor = \langle\{Company{=}\mathrm{A}\},\{Company{=}\mathrm{B}\}\rangle$$

The following example illustrates how refined values expand records into multiple attributes.

$$\begin{aligned}\lfloor Company\langle \mathrm{A}\prec Quarter\langle \mathrm{Q1},\mathrm{Q2}\rangle,\mathrm{B}\rangle\rfloor = \langle&\{Company{=}\mathrm{A},\ Quarter{=}\mathrm{Q1}\},\\ &\{Company{=}\mathrm{A},\ Quarter{=}\mathrm{Q2}\},\\ &\{Company{=}\mathrm{B}\}\rangle\end{aligned}$$

With this extended semantics of types, the placement of attributed values in tables has to be adapted. Specifically, instead of locating the position of a value, we now locate the position of a record that is subsumed by the record of the attributed value to be placed. We write s_i for selecting the ith element from the sequence s. As a special case, we consider the type *Unit*, which consists of just one single element and which is used to specify untyped single rows or columns of tables: The position of any value is always 1 with respect to the type *Unit*.

$$r{\uparrow}\delta \;=\; \begin{cases}1 & \text{if } \delta = Unit\\ i \text{ such that } r\supseteq\lfloor\delta\rfloor_i & \text{otherwise}\end{cases}$$

If domain types do not contain duplicates, there will be at most one record that r can subsume, and i is unambiguously defined. With this amended definition of value lookup, the previous definition for $\boxplus$ works now for types with and without refined values.

We can now try to construct the table shown in Figure 1(b) with the following expression.

$$\overset{Unit}{Company\otimes Quarter}\boxplus D$$

However, the result of this expression is not quite the table shown in Figure 1(b), since it still contains empty cells just as the original table in Figure 1(a) does. We can remove those by using a table filtering function that eliminates empty rows (and columns) from a table (using a predicate *empty* that is true for an empty data row or column).

$$\sigma_{\neg empty}\left(\overset{Unit}{Company\otimes Quarter}\boxplus D\right)$$

If this behavior is needed frequently or maybe even the expected default, we can easily define a corresponding version of $\boxplus$ that applies the filter by default.

Note that we can put such filter functionality to a much wider use. For example, we can produce a list of all companies and quarters for which *no* data is available.

$$\sigma_{empty}\left(\overset{Unit}{Company\otimes Quarter}\boxplus D\right)$$

Since the table construction is driven exclusively by the row and column types, interesting table variations can be achieved by simple type transformations. Suppose, for example, that we want to show quarters first and companies nested inside of quarters. We can accomplish this transformation by changing the order of type refinement.

$$\overset{Unit}{Quarter \otimes Company} \boxplus D$$

This flexibility of type refinement is due to its *not* being a commutative operations, that is:

$$\delta' \neq \delta \implies \delta' \otimes \delta \neq \delta \otimes \delta'$$

Type refinement is very similar to cartesian product. The difference is the tree-shape representation in which the refining type is attached to each value of the type that is being refined. This provides additional flexibility for further type operations. For example, we can define operations for selectively removing or adding refinements for individual values. In this sense, types with refined values bear some similarity to dependent sum types in type theory.

C. Other Type Operations

We can envision several other interesting type operations that can be exploited for new forms of table constructions or transformations. One such operation is *type coarsening*, which is the inverse of refinement and extracts a type from refined values. This operation can be used to transform nested list structures into tables. We could employ coarsening to transform tables of the kind shown in Figure 1(b) into tables of the from shown in Figure 1(a). In other words, inverse type transformations define inverse table transformations.

More interestingly, though, coarsening can be used in connection with aggregating functions to produce summary tables. For example, if we coarsen the type *Company*⊗*Quarter* back to *Company*, we have multiple values in the data source matching each company, which makes the table construction ambiguous. By aggregating a collection of values into a single value with a binary function, we can get back a well-defined behavior. For example, the following construction produces the table shown in Figure 1(c).

$$\overset{Unit}{Company} \boxplus \textsc{max}(D)$$

Note that the binary function does *not* have to be commutative or associative, since the values to be aggregated are ordered. The function doesn't need an identity element either, since empty cells can simply be kept and don't need to be aggregated.

IV. Related Work

This work is inspired by the work of Harris and Gulwani [1] in which they present a programming-by-example approach to infer table transformations from input/output example tables. They describe an algorithm ProgFromEx that represents inferred table transformations in a language called TableProg. We haven't performed a detail comparison of the two approaches yet, but the obvious advantages of ProgFromEx are that it needs no programming at all by end users and can probably deal easier with some more complicated, unstructured cases. The main advantage of our approach is that the transformation results are better predictable because transformations are based on a simple and clear semantics instead of a complicated inference algorithm. Moreover, our table transformations are highly reusable and composable, which promises better scalability. However, a detailed comparison of the two approaches is subject of future work.

Pivot tables as found, for example, in Excel can also be used to transform tabular data. While the input table basically represents an attribute data set (like a relation in a relational database), constructing Pivot tables is primarily an interactive process and not based on explicitly applying table transformation operations. Since Excel Pivot tables don't have any associated notion of types, they obviously cannot provide operations for type transformations. However, given the strong similarities, typed table transformation could be used as an underlying formal model for *typed* Pivot tables.

The table transformations we have considered here are ad hoc in the sense that they are based on arbitrary type transformations. However, a large class of systematic table transformations are the result of controlled evolution of table data. Several approaches have been proposed to capture such evolution-based table transformations [4], [5], [6], [7]. An interesting topic for future work is to identify type transformations that can represent evolution-based table transformations.

V. Conclusion and Future Work

We have demonstrated that table types and transformations can be an effective basis for the systematic construction and manipulation of spreadsheet tables. However, the presented approach is incomplete and requires several additional components to become a versatile and widely applicable table manipulation tool. In particular, in future work we plan to address the following questions.

- How do we obtain labeled data sources in the first place? We can have users mark areas of spreadsheets as data sources. By using an interactive tool or label inference [8], [9], values can be turned into attributed values.
- How do we transform tables with formulas? To be able to place formulas arbitrarily, we have to base cell references on labels instead of addresses, just as we did in [10] or [11]. For generating concrete references when constructing tables, we also have to distinguish between aggregating and iterating operations [12].
- We need a comprehensive language definition that includes operations for turning tables into attributed data as well as combining different tables.

Tables are the fundamental data structure of spreadsheets. Investigating the properties of tables as well as their transformations should thus be a priority of spreadsheet research.

Acknowledgments

This work is partially supported by the National Science Foundation under the grant CCF-1717300.

References

[1] W. R. Harris and S. Gulwani, "Spreadsheet Table Transformations from Examples," in *ACM Conf. on Programming Languages Design and Implementation*, 2011, pp. 317–328.

[2] M. Erwig and M. M. Burnett, "Adding Apples and Oranges," in *4th Int. Symp. on Practical Aspects of Declarative Languages*, ser. LNCS 2257, 2002, pp. 173–191.

[3] R. Abraham, M. Erwig, and S. Andrew, "A Type System Based on End-User Vocabulary," in *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, 2007, pp. 215–222.

[4] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger, "Automatic Generation and Maintenance of Correct Spreadsheets," in *27th IEEE Int. Conf. on Software Engineering*, 2005, pp. 136–145.

[5] R. Abraham and M. Erwig, "Inferring Templates from Spreadsheets," in *28th IEEE Int. Conf. on Software Engineering*, 2006, pp. 182–191.

[6] G. Engels and M. Erwig, "ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications," in *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, 2005, pp. 124–133.

[7] J. Cunha, M. Erwig, J. Mendes, and J. Saraiva, "Automatically Inferring Models from Spreadsheets," *Automated Software Engineering*, no. 3, pp. 361–392, 2016.

[8] R. Abraham and M. Erwig, "Header and Unit Inference for Spreadsheets Through Spatial Analyses," in *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, 2004, pp. 165–172.

[9] ——, "UCheck: A Spreadsheet Unit Checker for End Users," *Journal of Visual Languages and Computing*, vol. 18, no. 1, pp. 71–95, 2007.

[10] M. Luckey, M. Erwig, and G. Engels, "Systematic Evolution of Model-Based Spreadsheet Applications," *Journal of Visual Languages and Computing*, vol. 23, no. 5, pp. 267–286, 2012.

[11] J. Cunha, M. Dan, M. Erwig, D. Fedorin, and A. Grejuc, "Explaining Spreadsheets," in *ACM SIGPLAN Conf. on Generative Programming: Concepts & Experiences*, 2018.

[12] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein, "Gencel – A Program Generator for Correct Spreadsheets," *Journal of Functional Programming*, vol. 16, no. 3, pp. 293–325, 2006.