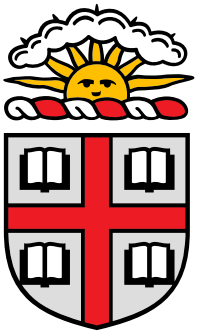
Dependently Typed Tables

Joseph Rotella

Advisor: Robert Y. Lewis

Reader: Shriram Krishnamurthi



A thesis submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science with Honors in Mathematics–Computer Science at Brown University

Providence, Rhode Island
May 2024

Acknowledgments

Thank you to my advisor, Rob Lewis. I am deeply grateful and very fortunate to have benefited from Rob's mentorship and expertise throughout my time at Brown.

Thank you also to Shriram Krishnamurthi for his guidance as this thesis' reader and for suggesting typed tabular programming as its topic, to Kuang-Chen Lu and Ben Greenman for their insightful correspondence, and to the members of the Spring 2022 Lean independent study group at Brown for many thought-provoking conversations in the early days of this project.

Abstract

Tabular data arises frequently in multiple areas of computer science, often in contexts in which the compile-time guarantees of static typing would offer significant benefits. However, tables pose challenges for many type systems due to their heterogeneous structure and the nontrivial type-level computations required to express many common table operations. This thesis investigates the realizability of rich static typing for tabular programming by implementing a tabular-data library in Lean, a dependently typed functional programming language. We provide a nearly complete formal verification of our implementation with respect to the Brown Benchmark for Table Types. We also present several mechanisms for improving the ergonomics of our library through metaprogramming and syntax extensions that occlude the type system's technical machinery from users.

Contents

1 Introduction

Tabular data arises frequently in data science, database systems, and other areas of computer science. Data sets, relational databases, and spreadsheets all frequently represent data as rows of entries containing values for each of a collection of labeled columns.

Many popular libraries for interacting with tabular data are implemented in dynamically typed languages [6, 11, 20]. By contrast, there exist comparatively few tools for manipulating, or indeed even representing, tables in statically typed languages. Yet static typing offers several potential benefits over dynamic typing for working with tabular data. Sufficiently expressive type systems can detect many classes of errors at compile time, providing stronger safety and correctness guarantees. In the domains in which tabular data most frequently appears, such guarantees are particularly valuable. Forestalling would-be runtime errors provides greater assurance of database integrity by preventing invariant-violating code from even compiling. When executing computations on large data sets, runtime errors may not be detected until well into a long-running query; compile-time error detection thus offers a significant time savings in addition to increased confidence in the correctness of the final results.

Whence, then, the dearth of statically typed tabular-data libraries? Apart from the popularity of dynamically typed languages, this is likely at least partially attributable to the significant type-level book-keeping requirements of tables, which are infeasible or impossible to realize in some type systems. Because each column of a table may contain a distinct type of data, and a table may contain arbitrarily many columns, it is difficult even to encode an adequately general datatype for representing tables in many type systems. For instance, the type systems of many ML dialects do not allow data types to be parameterized over an indeterminate number of type arguments. Moreover, in order to describe the type of their output, many common table operations require nontrivial type-level computations whose complexity can exceed the expressive capabilities of many type systems. Even the seemingly straightforward operation of horizontally appending one table to another (so as to form a new table with the columns of both inputs) requires a type-level notion of concatenation that some type systems are incapable of expressing.

This thesis investigates the realizability of rich static typing for tabular programming by implementing a tabular-data library in Lean, a dependently typed functional programming language. We take advantage of Lean's facilities for formal proof to verify various properties of our table API. In addition to demonstrating the feasibility of a dependently typed tabular programming library, we present several mechanisms for enhancing the ergonomics of such a library and occluding the type system's technical machinery from the end user. We discuss some of the ways in which our library depends on behavior for which languages like Lean may not be optimized and methods for circumventing these challenges. Finally, we consider several areas for further refinement to increase both the type-safety and usability of this table library or others like it. The code for this thesis can be found at https://github.com/jrr6/lean-tables/tree/thesis.

2 Background

2.1 Definitions

Our definition of a table is inspired by the definition used by the Brown Benchmark for Table Types (B2T2) [12], although we modify some terminology to better suit the representation

used in this thesis. A table consists of a *schema*, which specifies the name of each column and the type of data each contains, and a *body* containing rectangularly arranged data. We refer to the name–type pairs in a schema as *headers*. The table body is organized into *rows* and *columns*. There is precisely one column, identified by the name s and containing data of type τ, corresponding to each header (s, τ) in the schema. A table may contain arbitrarily many rows, each of which contains exactly one possibly-empty *cell* for each column of the table. Symmetrically, we may consider each column as containing one cell for every row in the table, though we will generally prefer to consider tables row-wise.

2.2 The Brown Benchmark for Table Types (B2T2)

Many tools exist for programming with tabular data [6, 11, 14, 16, 20]. Rather than attempt to replicate the functionality of any one of these tools, many of which are specifically tailored to the domains of data science or database programming, we follow the specification provided by B2T2. The B2T2 specification is intended to serve as a "fixed reference point" for table libraries of this sort. Drawing on popular tabular programming tools like the foregoing, B2T2 specifies a realistic suite of table functionality and also—of particular interest for this thesis—aims to exercise the type systems within which tabular-data libraries are implemented. The specification provides a general definition of the table data structure, an API of common table operations, and contracts for each function in the API. It contains a collection of test cases and example programs to assess the correctness and expressive capacity of an implementation. It also supplies several invalid programs that test whether errors are detected at compile time as well as the quality of error messages generated by the type system [12].

As of this writing, two implementations of the B2T2 specification exist [13]. The B2T2 authors provide a reference implementation written in TypeScript. The second implementation is written in Empirical, a data analysis-focused language with support for statically typed data frames [19]. We focus on the TypeScript implementation as a basis of comparison for our own because it is more comprehensive and because, unlike Empirical, TypeScript and Lean are general-purpose programming languages whose type systems and built-in language functionality are not explicitly tailored to tabular programming applications.

The TypeScript implementation of B2T2 employs a row-wise table representation with a combination of type- and data-level mechanisms for encoding schemata. A table body is represented by an array of rows, each of which is represented by a dictionary whose fields are column names and whose values are cells. A type-level dictionary parameterizing the table type maps column names to the type contained in the corresponding column. This allows the type-correctness of rows with respect to a table's schema to be statically checked: incorrectly typed cells, invalid column names, and missing column names all raise compile-time errors. However, this dictionary representation alone does not enforce an ordering on columns in a schema; to accomplish this, the ordering of columns is maintained as a `header` field[1] of table objects. Functions on tables must therefore independently specify their behavior with respect to both column names and types (at the type level) and column ordering (in the `header` data field).

Using this combination of type- and data-level specification, the TypeScript implementation is able to characterize the schematic behavior of all the table operations in the B2T2 specification. However, due to limitations of TypeScript, it is possible to define invalid values

[1] B2T2 uses "header" to mean "the sequence of column names of a table."

of the table type which may not be caught at compile time or, in some cases, even at runtime. For instance, the `header` field is constrained only to contain keys of the type-level dictionary, but it cannot enforce the requirement that each name appear exactly once. Therefore, it is possible to define the `header` to contain more entries than the table has columns (by repeating names) or to omit columns entirely. Moreover, the implementation makes considerable use of type-casting to coerce values to types that TypeScript fails to recognize them to have. The compile-time guarantees provided by the type system for this implementation are thus somewhat limited, and there is still some onus on the programmer to maintain invariants that cannot be enforced by the type system. Because, unlike TypeScript's type system, the type theory of Lean is sound [5] and supports true dependent types, we see this as an area in which Lean can offer benefits over existing implementations.

Perhaps the most significant advantage of the TypeScript implementation is its ease of use. The types of tables and functions thereupon are relatively straightforwardly expressible, and table operations appear to the user much as would an analogous untyped implementation in pure JavaScript. The benefits of the library's type-level guarantees thus do not appear to incur any meaningful ergonomic cost. In contrast, dependent typing adds considerable complexity to our implementation, though we manage to ameliorate at least some of these ergonomic challenges through the use of syntactic extensions and metaprogramming (see section 6).

2.3 Motivating Dependently Typed Tables

Lean, as noted, is a dependently typed functional programming language. In this and the subsequent section, we discuss the key properties of Lean's type theory of which our table implementation makes use. Because work on this project began in spring of 2022, at which time many features and definitions now present in Lean 4 and its library did not yet exist, some of this project's design choices may not match current conventions, or take advantage of more recent developments, in Lean. Importantly, our code does not make use of either Std4 [10] or Mathlib [8], two Lean 4 libraries which now provide some of the auxiliary definitions, especially for lists and integers, that we implement in our library.

To motivate the utility of dependent types, we first consider in greater detail why simpler type theories are inadequate for representing tables in full generality. Type theories with support for parametric polymorphism, such as those of most ML dialects, allow for the declaration of inductive data types parameterized over some index type: the declared type depends on the type-level argument it is given. A canonical example is the type family of lists, which are parameterized over the type of their elements:

```
inductive List₀ (α : Type) : Type
  | nil  : List₀ α
  | cons : α → List₀ α → List₀ α
```

In such a type theory, one might represent a table as a list of lists, where each list represents one row of data, together with a list of strings containing the column names:

```
def Table₀ (α : Type) : Type :=
  List₀ String × List₀ (List₀ α)
```

Such a representation severely limits the kinds of tables we can represent, since every column of a table of type `Table₀ α` is required to have the same type `α`. Moreover, it is possible to construct ill-formed values of type `Table₀ α`, since the type carries a well-formedness invariant that is inexpressible in the type itself: namely, that every row list is of equal length.

Dependent types allow us to remedy both of these shortcomings. In dependent type theories such as Lean's, one can specify not just a type but data (such as a natural number, string, or list) as the index of a type constructor. A *heterogeneous list* is a common example of such a data type. It is parameterized over a list of types, each of which specifies the type of the corresponding entry at the data level:

```
inductive HList : List Type → Type 1
  | nil : HList []
  | cons {α : Type} {αs : List Type} :
    α → HList αs → HList (α :: αs)
```

(Note that arguments specified in braces are implicit: they are not written when applying the constructor and are instead inferred by Lean. The notation `Type 1` indicates that heterogeneous lists are elements of a higher type universe than the types over which they are parameterized; we discuss type universes further in section 2.4.)

Heterogeneous lists facilitate the representation of differently typed columns in tables. A heterogeneous list parameterized over a list of types `αs` corresponds to a row in a table whose columns, from left to right, contain the types specified by `αs`. The body of a table with column types `αs`, then, can be represented as a list of type `List (HList αs)`. This also remedies the second shortcoming of our simply typed approach: since heterogeneous lists must contain exactly one element for each element of the type-level list `αs`, their lengths are fully determined by their type, allowing us to enforce the consistency of row lengths at the type level.

While the above encoding suggests a general approach for a dependently typed table representation, it does not fully capture our definition from section 2.1. In particular, it does not allow us to refer to columns by name, nor does it enable the representation of empty cells. We describe the modifications required to accommodate these features in the discussion of our table representation in section 3.1.

2.4 The Type Theory of Lean

Lean's type theory is based on the Calculus of Inductive Constructions [5]. It contains a non-cumulative hierarchy of type universes, at the base of which is the universe `Prop` of propositions. Above `Prop` are "data" universes `Type n` for each natural-number universe level `n` (`Type 0` is also abbreviated as `Type`). A definition in Lean can be made universe-polymorphic by parameterizing the level of a universe over a level variable `u`. For instance, a universe-polymorphic variant of `HList` would have type `List (Type u) → Type (u + 1)`, where `u` is a universe-level variable, and the types of the `α` and `αs` arguments to the `cons` constructor would change accordingly.

The universe `Prop` of propositions has several features that differ from the `Type` universes above it. One such feature is proof irrelevance: for any type `P : Prop`, any two values of type `P` are equal. To avoid inconsistency with proof irrelevance, Lean disallows large elimination from `Prop`: values computed by recursion on a proof value in `Prop` must themselves be propositions, not data in a higher type universe. That is, the motive for a propositional recursor must always be a proposition [2].

An exception to this prohibition of large elimination is subsingleton elimination: a propositional inductive type—such as an equality type—with at most one constructor, all of whose arguments are `Prop`-valued or appear as indices of the proposition, can be eliminated into higher type universes [3]. This enables casting a data value in a non-propositional type universe along a proof of equality: given a data value of type `T a` and a proof that `a = b`, one

can cast the data value to type `T b` by recursion on the proof of equality. Such casting is necessary if one wishes to supply a value whose type is provably but not *definitionally* equal to the type Lean expects—that is, the two types cannot be shown to be equal merely by unfolding definitions [4].

Lean's type system also includes support for type classes. Type classes provide a form of ad-hoc polymorphism, allowing a programmer to define type-specific instances of a specified class of operations. Instances of type classes are registered in a database and can be automatically synthesized by Lean. Type-class arguments to functions are marked as *instance-implicit* using brackets, indicating to Lean that they should be automatically synthesized [18]. During type-class resolution, Lean will expand defined symbols which are marked as "reducible" in the type for which a class is being synthesized. However, definitions that are not marked as such (as is the case by default) will not be reduced. Since type-class resolution will fail if a type cannot be reduced to one for which a type class is registered, it is often desirable that definitions appearing in types be reducible. For instance, while there exists a `ToString` instance for the type `Nat`, resolution of the `ToString` class for the type `Nat'` defined by `def Nat' := Nat` would fail because the definition is not reducible. A type class of particular interest for our table implementation is `DecidableEq`, the class of types with decidable equality.[2] This type class provides a means of specifying a decision procedure for equality of values of a given type.

3 Data Types and API Structure

3.1 Table Representation

As discussed in section 2.3, a list of heterogeneous lists provides a natural dependently typed representation of the structure of a table body. We adapt this basic structure to support two desirable features of tables in practice. First, columns in tables should be identifiable by names rather than merely by their indices within each heterogeneous list. Second, many real-world applications in which tabular data appears must accommodate the absence of data; accordingly, we follow B2T2 in taking possible emptiness of entries as a primitive notion in our table representation. While this second constraint may be undesirable in certain applications, it is relatively straightforward to remove without substantially affecting the representation we adopt or the operations we define thereupon.

We define our table body representation in three stages: a table comprises a list of rows; a row is, roughly, a heterogeneous list of cells; a cell is a possibly-empty container of a value. We do not explicitly encode columns as part of our representation, but column c of a table can be obtained by taking the entry for c from each row of the table.

At the type level, we parameterize a table over its schema. A schema is a (homogeneous) list of headers, which are pairs consisting of the name of a column and the type of data it contains. These and the preceding definitions are shown in figure 1.

Rows are also parameterized over schemata. The second component in each entry in a schema determines the type of the element at the corresponding index in the row. As seen in figure 1, the schema replaces the `List Type` argument to `HList` seen in section 2.3; we have simply added column name information to that list.

[2] For a programmatic use case such as ours, we would now prefer to use the `LawfulBEq` ("lawful Boolean equality") class, which was added to Lean 4 after work on this project began. Instances of this type class are automatically generated for types that instantiate `DecidableEq`.

```
def Header {η : Type u_η} := (η × Type u)
def Schema {η : Type u_η} := List (@Header η)

structure Table {η : Type u_η} [DecidableEq η] (hs : @Schema η) where
  rows : List (Row hs)

inductive Row {η : Type u_η} [DecidableEq η]
  : @Schema η → Type (max u_η (u + 1))
  | nil : Row []
  | cons {name : η} {τ : Type u} {hs : Schema} :
    Cell name τ → Row hs → Row ((name, τ) :: hs)

inductive Cell {η : Type u_η} [DecidableEq η]
  (name : η) (τ : Type u) : Type (max u u_η)
  | emp : Cell name τ
  | val : τ → Cell name τ
```

Figure 1: Core data types for representing tables

name	age
Bob	
Alice	17
Eve	13

```
def students : Table [("name", String), ("age", Nat)] :=
  Table.mk [
    Row.cons (Cell.val "Bob") (Row.cons Cell.emp Row.nil),
    Row.cons (Cell.val "Alice") (Row.cons (Cell.val 17) Row.nil),
    Row.cons (Cell.val "Eve") (Row.cons (Cell.val 13) Row.nil)
  ]
```

Figure 2: A sample table and its explicit representation. (A more convenient notation for table values is shown in section 6.1.)

Rows do not, however, directly store elements of the types specified by their schemata. Instead, each entry in a row is a cell parameterized over the name and type of the column in which it appears. Since cells may be empty, this builds the possibility of emptiness into our table representation. Just as rows enrich standard heterogeneous lists with column names, so does a cell enrich a standard option type with column-name information. Figure 2 shows the encoding of a simple table using this representation.

We enforce one further constraint on our tables: while we do not require that columns be named by strings, we do require that equality of the column-name type be decidable. This is not strictly required by all table operations, and much of our representation would be relatively unchanged if this requirement were removed, since individual table functions can still require a decidable-equality type-class argument. However, given the identifying role column names are intended to play, enforcing decidable equality seems to be a reasonable constraint and prevents the possibility of constructing tables on which a significant number of our table operations fail to work.

Constructors
- emptyTable
- addRows
- addColumn
- buildColumn
- vcat
- hcat
- values
- crossJoin
- leftJoin

Properties
- nrows
- ncols
- header

Access Subcomponents
- getRow
- getValue
- getColumn1
- getColumn2

Subtable
- selectRows1
- selectRows2
- selectColumns1
- selectColumns2
- selectColumns3
- head
- distinct
- dropColumn
- dropColumns
- tfilter

Ordering
- tsort
- sortByColumns
- orderBy

Aggregate
- count
- bin
- pivotTable
- groupBy

Missing Values
- completeCases
- dropna
- fillna

Data Cleaning
- pivotLonger
- pivotWider

Utilities
- flatten
- transformColumn
- renameColumns
- find
- groupByRetentive
- groupBySubtractive
- update
- select
- selectMany
- groupJoin
- join

Figure 3: B2T2 API functions (see [13] for documentation)

3.2 Table API

Our implementation exposes an API comprising the 49 common table operations specified by B2T2 (see figure 3). These are intended to represent "idiomatic tasks" when working with tabular data and were selected with the goal of "reveal[ing] challenges for type systems," but they are intended to compose neither a completely minimized set of core table functionality nor a fully robust table library [12]. Nevertheless, they represent a range of common kinds of table operations, including joins, sorts, pivots, and functional combinators. We describe in this and the following sections several benefits of dependent types for these operations as well as some drawbacks encountered in their implementation.

By structural analogy to our table representation, most table operations correspond to functions at multiple levels of our table encoding. First, we state the type of an operation by defining a function on schemata that describes how an operation transforms the schema(ta) of its input(s). Second, the "horizontal" functionality of an operation—that is, its manipulation of columns in a given row—is implemented as a function on rows (these are, effectively, heterogeneous-list operations). Third, the operation's "vertical" functionality—its manipulation of rows in the table body—is implemented as a function on lists.

As an example, we consider `selectColumns3`, a function which forms a new table by selecting columns `cs` from an input table `t`. We first describe the schema of the resulting table. We represent our selected columns `cs` as a list of (dependent) pairs, each consisting of a header together with a proof of its membership in the schema (we refer to these as *certified* headers). Consequently, the schema of the resulting table is simply the list of headers (without their proofs): that is, `Schema.map Sigma.fst cs` (where `Schema.map` is provably equal to `List.map`). Our row-wise operation is simply a list map: we do not add, remove,

or reorder rows in our output; we merely select columns from each. Finally, we define a per-row operation `Row.pick` that selects each of the columns `cs` from a row. Since rows, like tables, are parameterized over schemata, the type of `Row.pick` also invokes our schema-level function on `cs`: it maps a `Row schema` to a `Row (Schema.map Sigma.fst cs)`. Combining these three operations, we can define `selectColumns3` as follows:

```
def selectColumns3 (t : Table schema) (cs : List (CertifiedHeader schema)) :
    Table (Schema.map Sigma.fst cs) :=
  {rows := List.map (λ r => Row.pick r cs) t.rows}
```

While a simple example, this three-tiered structure is representative of many table functions we define in this library.

An important consequence of our choice of table representation is that we can define the type of such a function so that it is definitionally equal to the type of its body without type-casting. This simplifies our code and greatly eases its verification. Such considerations motivated in part the use of lists rather than a more efficient data structure—such as a dictionary, as the TypeScript implementation uses—for representing schemata. Because of the similarity in structure between our schemata and table bodies, recursive type-level operations naturally match the recursive structure of data-level operations. Indeed, we avoid the need for type-casting in nearly all API functions (we mention one exception in section 4.2).

Moreover, changing the representation of schemata alone would likely offer little runtime performance benefit, since schematic look-ups occur principally at compile time. As we will describe in section 3.3, these look-ups produce proofs that effectively serve as indices for runtime look-ups in table bodies. These table accesses, not schematic ones, principally impact runtime performance. Therefore, to realize runtime performance benefits from the use of more efficient data structures, we would need to modify our table body representation as well. We discuss this possibility further in section 8.2.

3.3 Proof Types

The example of `selectColumns3` highlights a common feature of many table functions: they require the user to specify one or more columns on which to act. (Other examples include specifying the name of a column by which to sort the table or the name of a column to drop.) If the column name provided to such an operation is invalid, however, the operation cannot be completed. Some means must therefore be provided to check the validity of a column name. In many table libraries, this check occurs at runtime, but the type-level schema associated with a table in our representation allows this to be verified at compile time. We implement this checking by requiring all functions that take a column name as an argument to also take a *proof* of that name's membership in the schema of the provided table. We define two proof types, `Schema.HasName` and `Schema.HasHeader`, for this purpose. (A `CertifiedName` or `CertifiedHeader` is a pair of a name or header together with its proof.) The desirability of multiple predicates arises from the differing informational needs of different table functions: some require only the name of a column on which to act, while others additionally require the corresponding type of data that column stores. For instance, to drop (delete) a column, we need only know the name of the column to drop; to sort by a column's contents, however, we must also have access to the type of the column's data (to ensure, for example, that an appropriate ordering class has been registered for that type).

Values of these proof types not only provide correctness guarantees but also, because they are constructive proofs, serve as indices into a schema for the table. Table operations that

accept these proofs can recursively deconstruct them to effectively obtain an index into the schema (and the data it parameterizes) corresponding to the requested column. Defining such functions by structural recursion on proofs, rather than performing equality checks between the requested column name and each successive name in the schema, makes them easier to verify (by straightforward induction on the proof) and improves performance by exploiting the schema information captured at compile time to avoid redundant equality checks at runtime. In some suitably optimized programming languages, this technique can even be used to avoid the need to store column names at all at runtime, since column accesses are mediated solely by the corresponding proof objects [22].

Crucially, these are proof *types*, not propositions. Because Lean disallows large elimination from `Prop`, data-producing table functions cannot be defined by recursion on proofs in `Prop`. Accordingly, `Schema.HasName` and `Schema.HasCol` are (universe-polymorphically) `Type`-valued. We further discuss the tensions between the need to compute with proofs and the properties of Lean's propositional universe in section 7.

4 Implementation Challenges

4.1 Type-Level Computation and Reducibility

A benefit—and challenge—of our table representation is the ability to precisely encode the manner in which table operations modify tables' schemata. One of the simplest examples is that discussed in section 1: the `hcat` operation, which horizontally concatenates two tables together, producing a table with the columns of both (empty cells are inserted in each row for the columns from the opposite table). Since a `Schema` is merely a `List`, a natural type for this function would be `Table s₁ → Table s₂ → Table (List.append s₁ s₂)`. However, built-in `List` functions are only semireducible, meaning that certain type-level operations and type-class synthesis will not unfold their definitions. This poses a range of challenges for the usability of the library. For instance, simple equality checks between tables may fail because Lean is unable to synthesize the appropriate decidable-equality type-class instance. Moreover, some table functions require the synthesis of decidable equality instances for rows or tables with a given schema; if the output of a function whose return type includes a semireducible function is passed to one of these functions, the relevant type class will not be synthesized and the function will fail to evaluate. While Lean 3 provided a mechanism for adjusting the reducibility of any definition after its declaration [1], Lean 4 disallows such modification for imported definitions. It is therefore not possible to adjust the reducibility of built-in `List` functions. It is also not clear that such an adjustment would be desirable, since, outside of this library, `List` functions are generally not used in a manner that requires reducibility, and needlessly marking functions as reducible may incur performance penalties during unification [3].

In the initial development of this library, we used a custom testing framework to force the unfolding of these definitions so that the type classes necessary to perform equality checks on test cases could be synthesized. Ultimately, however, such workarounds scaled and generalized poorly. Consequently, the table library re-implements the subset of built-in `List` functions used in type specifications as reducible functions in the `Schema` namespace. These functions are identical to their `List` counterparts but are tagged with the `@[reducible]` attribute, making them fully reducible and thereby circumventing the challenges discussed above. The type of `hcat`, for example, is thus `Table s₁ → Table s₂ →`

Name	Age	Quiz 1	Quiz 2
Alice	17	6	8
Bob	12	8	9

(a) Table before applying `pivotLonger` (or after applying `pivotWider` to columns "Test" and "Score" in figure 4b)

`pivotLonger` ⇄ `pivotWider`

Name	Age	Test	Score
Alice	17	Quiz 1	6
Alice	17	Quiz 2	8
Bob	12	Quiz 1	8
Bob	12	Quiz 2	9

(b) Table after applying `pivotLonger` to the "Quiz 1" and "Quiz 2" columns in figure 4a with new column names "Test" and "Score"

Figure 4: An example of the `pivotLonger` and `pivotWider` operations

`Table (Schema.append s₁ s₂)`. The `Schema` namespace provides lemmas proving equality between corresponding schema and list functions, which we use in our verification of table functions that depend on these `Schema` definitions to take advantage of existing lemmas proved for their `List` counterparts.

4.2 Several Functions of Interest

Several functions in the B2T2 specification posed interesting challenges for our Lean implementation. Here, we highlight two cases of note: "pivot" functions, which require significant type-level computation and transfer of values between the type and data levels; and "grouping" functions, which produce tables containing tables. We then discuss another class of functions of particular interest in section 5.

The `pivotWider` function is arguably the most complex in the B2T2 specification. It is the inverse of `pivotLonger`, which moves the data stored in a specified collection of columns into two new columns, one containing the name of the former column and the other the value it formerly held, duplicating each row of the input table for each column to be recorded. An example is shown in figure 4.

As this example illustrates, `pivotWider` performs complex computations not only at the data level but at the type level as well. Given a table `t` with schema `sch` and column names `c1` and `c2` therein, the schema of `pivotWider t c1 c2` is computed by first removing `c1` and `c2` from `sch`, then appending to the resulting schema the list of all the unique entries in column `c1` each paired with the type contained in column `c2`. Moreover, the computations done at the data level must be written such that the resulting table's schema definitionally equals (or else must be cast to) this value. While we nearly universally manage to avoid the use of casting elsewhere in our implementation, we end up relying on non-definitional type-casts in `pivotWider`, and moreover resort to Lean's tactic mode—generally reserved for generating proofs rather than data—to construct one particularly complicated proof term (though we use only a small number of relatively simple tactics).

The most significant challenge for `pivotWider` as currently written is that its type relies on several semireducible functions, meaning that type-class resolution and other operations which depend upon type-level reduction may fail on expressions involving this function. This is not, however, a theoretical limitation; though we do not do so, we see no impediment to implementing reducible versions of each of the functions used in the type of `pivotWider` and updating its data-level computation and proofs to match these new type-level functions. Despite these challenges, our success in implementing `pivotWider`, and in doing so in a

Dept	Code	Title
CS	150	Programming
Math	101	Calculus
CS	200	Algorithms
Math	202	Analysis

(a) Table before grouping

Key	Groups		
	Dept	Code	Title
CS	CS	150	Programming
	CS	200	Algorithms
	Dept	Code	Title
Math	Math	101	Calculus
	Math	202	Analysis

(b) Table after (retentive) grouping by "Dept"

Figure 5: An example of the `groupByRetentive` operation's behavior

manner that occludes its internal complexity from the user, suggests that our representation is sufficiently robust as to represent real-world table operations that do not neatly align with traditional type–data distinctions.

Two "grouping" operations in the API, `groupBySubtractive` and `groupByRetentive`, have the interesting property of producing tables that themselves contain tables. These functions produce a two-column output: the first column is a requested column from the input table, while the second contains tables consisting of all rows in the input that contained the corresponding entry in the first column. (The subtractive variant removes the selected column from the subtables in the output.) An example is depicted in figure 5.

As the types in figure 1 indicate, tables belong to a higher type universe than the values they contain, all of which must belong to the same type universe. This poses a challenge for representing a table like that depicted in figure 5b: the tables in the right-hand column contain strings and natural numbers, both of which are in the universe `Type`, meaning that the tables themselves have types in `Type 1`. However, the "Key" column consists of strings, which, as just noted, are in `Type`. Since Lean does not have a cumulative hierarchy of types, we must *lift* the elements of the left-hand column to `Type 1` in order for the table's contents to belong to a consistent type universe. While not technically difficult, this makes working with nested tables and tables containing data from multiple type universes somewhat tedious, as all values of types from lower universes must be prefixed with `ULift.up` to lift them to the appropriate universe.

5 Iterated Table Operations

A particularly challenging class of functions for our table representation comprises those operations which modify a table's schema by acting on multiple columns simultaneously. Such functions complicate the notion of a certified column name or header because, as we will show, the schema for which such a value must be certified may turn out not to be the schema of the original table argument passed to a function.

To motivate these difficulties, we consider the simplest function of this kind in the B2T2 specification, `renameColumns`, which renames multiple specified columns in a table. A more straightforward version of this function (not part of the B2T2 specification) is `renameColumn` below, which renames just one column and is comparatively easy to implement. It takes as arguments a table, a name that belongs to that table's schema, and a new name for the corresponding column. Its effect on its output's schema is described by `Schema.renameColumn`, which replaces the corresponding name in the schema with the new value:

```
def renameColumn (t : Table schema) (cnm : CertifiedName schema) (nm' : η) :
    Table (Schema.renameColumn schema cnm.snd nm') :=
  { rows := t.rows.map (λ r => Row.renameCell r cnm.snd nm') }
```

(Recall that a `CertifiedName` is a dependent pair of a column name and a proof of schema membership; `Schema.renameColumn` and `Row.renameCell` require only the latter.)

Naïvely, it would seem that `renameColumns` could simply take as an argument a *list* of pairs each containin an (existing) certified name and a (new) name. It could then fold over this list, invoking `renameColumn` for each pair and maintaining the updated table as an accumulator:

```
def renameColumnsNaive (t : Table schema)
                       (nms : List (CertifiedName schema × η)) :=
  nms.foldl (λ acc (cnm, nm') => renameColumn acc cnm nm') t
```

But this approach does not work. Consider a table `t` containing a single column named `"ID"`. Suppose we apply `renameColumnsNaive` to `t` with the name-list argument `[(⟨"ID", _⟩, "name"), (⟨"ID", _⟩, "ident")]` (we elide proof terms for clarity). While each entry supplies a valid name (`"ID"`) for the initial schema in its first component, this name is *not* valid for the table updated at the second iteration of the fold, in which the column formerly named `"ID"` is now named `"name"`. In other words, `renameColumn` is called in the second iteration with a name (`"ID"`) that is not among those in the schema (namely, `"name"`). Indeed, the application of `renameColumn` in the above code does not type-check precisely because `acc` may not have the schema `schema` for which `cnm` is a certified name. To remedy this issue, each entry of the sequence `nms` must store in its first component a `CertifiedName` not for the initial schema but rather for the schema that results after updating the initial schema according to all preceding entries in `nms`. (Thus, the second entry in the sequence would need to provide a certified name for the schema containing only `"name"`.)

We write "sequence" in the preceding sentence because a simply typed list is inadequate to represent such a collection. Since the schema for which a name is certified is a type-level argument, each entry in such a sequence has a (possibly) distinct type; lists, however, cannot store heterogeneously typed data. Nor are heterogeneous lists, as introduced previously, adequate: the type of each entry of the sequence must *depend* on the entries that precede it, but `HLists` do not allow for type dependencies between elements. Instead, we must introduce a new data structure—a form of heterogeneous list that allows the type of each element to depend on prior elements—to represent the arguments to a function like `renameColumns`. Each entry of such a sequence can be thought of as representing an "action" that modifies the schema of a table. Each entry acts on the schema that results from all prior entries' actions on an initial schema. Accordingly, we term this data structure an *action list*:

```
inductive ActionList {η : Type u_η} [DecidableEq η]
                     {κ : @Schema η → Type u}
                     (f : ∀ (s : @Schema η), κ s → @Schema η) :
                     @Schema η → Type _
  | nil  {schema} : ActionList f schema
  | cons {schema} : (entry : κ schema) →
                    ActionList f (f schema entry) →
                    ActionList f schema
```

An action list is parameterized over a function `f` on schemata. Each entry in the action list is an argument to this function. Specifically, each entry is the second argument to `f` for the case where its first argument is the schema that results from successively applying `f`, with

action arguments given by the preceding entries of the action list, to the schema over which the head of the action list is parameterized. (Notice that the type of the action argument `κ schema` to `f` depends on the schema argument to `f`. This is because the "action" argument to `f` must be allowed to depend on the schema on which it is acting: for instance, it may contain a certified name or header for the schema. Indeed, this is the very dependence that motivated the introduction of action lists.)

In the case of `renameColumns`, the schema-level function `f` is `Schema.renameColumnCN`, which updates a schema for an individual column renaming; its type is

```
Schema.renameColumnCN :
    (s : @Schema η) → (nmAndNew : η × η) × Schema.HasName nmAndNew.fst s →
    @Schema η
```

Accordingly, given a schema `schema`, the elements of an action list of type `ActionList Schema.renameColumnCN schema` are dependent tuples containing pairs of existing and new names along with proofs that the existing names are in the schema to which the action-list entry will be applied—*not* merely the original schema `schema`. Using the `name` tactic (discussed in section 6) for generating name-membership proofs, the following is an explicit representation of such an action list:

```
ActionList.cons ⟨("ID", "id"), by name⟩
  (ActionList.cons ⟨("Age", "age"), by name⟩ ActionList.nil)
: ActionList Schema.renameColumnCN [("ID", String), ("Age", Nat)])
```

It is then possible to define a function that updates schemata analogously to folding over the sequence of inputs as in the naïve example:

```
def Schema.renameColumns {η : Type u_η} [DecidableEq η] :
    (s : @Schema η) → ActionList renameColumnCN s → @Schema η
  | s, ActionList.nil => s
  | s, ActionList.cons c ccs => renameColumns (renameColumnCN s c) ccs
```

The table function `renameColumns` is defined similarly on cells within each row; we specify its type using `Schema.renameColumns`:

```
renameColumns : Table schema → (ccs : ActionList Schema.renameColumnCN schema) →
    Table (Schema.renameColumns schema ccs)
```

As `Schema.renameColumns` illustrates, action lists make implementing this kind of function relatively straightforward: because we recursively perform corresponding "actions" at both the type and data levels according to the entries of our action list, it is not, for example, necessary to cast proofs or values to a type other than that to which they definitionally reduce. Action lists also provide more fine-grained guarantees for operations such as `flatten`, which "flattens" columns containing sequential data into separate rows, as shown in figure 6. Using action lists, a column containing type `List (List α)` can successfully be flattened twice, while attempting to do the same to a column containing type `List α` (for some `α` not an instance of `List`) will fail. By contrast, only checking action arguments with respect to the initial schema (as does, for instance, the TypeScript implementation [13]) will allow the user to attempt to flatten a column of non-nested lists twice, causing a runtime error.

However, action lists present two important drawbacks. First, as shown previously, they are unwieldy to write and often require explicitly specifying arguments (such as membership proofs or, in more complicated functions, type-class instances or column-type arguments) that could be inferred from context. We make use of Lean's metaprogramming and syntax-extension features to hide much of this complexity from the user; this is discussed in section

Name	Age	Exams Taken	Scores
Alice	17	[midterm, final]	[88, 85]
Bob	12	[midterm, final]	[77, 87]

(a) Table before flattening

Name	Age	Exams Taken	Scores
Alice	17	midterm	88
Alice	17	final	85
Bob	12	midterm	77
Bob	12	final	87

(b) Table after flattening "Exams Taken" and "Scores"

Figure 6: An example of the `flatten` operation's behavior

6. Secondly, action lists and the behavior of functions defined thereupon pose a significant challenge for specification and verification, which we discuss in section 7.

6 Notation and Metaprogramming

6.1 Basic Notation and Proof Tactics

Lean provides extensive facilities for custom syntax definitions, metaprogramming, and in-editor widgets for displaying custom UIs [15]. We take advantage of these features to simplify the notation of table values, provide more robust rendering of tables, and automatically generate arguments to table functions that can be inferred from context.

We define custom syntax for notating `Row` values analogous to Lean's built-in notation for lists. This syntax is flexible, allowing a user to specify or omit cell names and to denote empty cells. Figure 7 demonstrates its use. We also provide a user widget for rendering tables as HTML, which improves table rendering in the variable-width Lean Infoview sidebar and greatly reduces the difficulty of rendering tables within tables, as in the examples from section 4.2.

Lean's metaprogramming features allow us to simplify functions that require proof arguments by automating proof search and making these arguments fully invisible in many situations. While requiring proofs of schema membership for arguments to table functions provides strong correctness guarantees, it is tedious to supply these proofs in every function call. This problem is substantially worse with action lists, the elements of which may contain proofs, type-class instances, and other ancillary arguments in addition to the column name or other "action" being requested. Using syntax extensions and metaprograms, however, we can automatically synthesize such arguments at compile time, allowing users to interact with the library without confronting this machinery and the complexity it entails.

Our library provides two proof tactics, `name` and `header`, which perform naïve proof search for `Schema.HasName` and `Schema.HasCol` proofs, respectively. Functions that require a proof of schema membership for a name or header take this proof argument as an *automatic parameter*: unless manually specified by the caller, such a parameter is automatically generated by Lean for a given function application using either `name` or `header`, inferring the required proof type from the explicitly provided arguments. A user can thus write `dropColumn students "name"`, omitting the `dropColumn` function's third argument (a proof of schema membership), and Lean will evaluate this as if the user had supplied the proof term: `dropColumn students "name" (by name)`. Therefore, in many cases, callers need not generate such membership proofs themselves, and indeed can omit the argument entirely when writing a function application.

```
/- Definitions using explicit constructors: -/
def studentsMissingEx :
    Table [("name", String), ("age", Nat), ("favorite color", String)] :=
  Table.mk [
    Row.cons (Cell.val "Bob")   (Row.cons Cell.emp
      (Row.cons (Cell.val "blue") Row.nil)),
    Row.cons (Cell.val "Alice") (Row.cons (Cell.val 17)
      (Row.cons (Cell.val "green") Row.nil)),
    Row.cons (Cell.val "Eve")   (Row.cons (Cell.val 13)
      (Row.cons Cell.emp Row.nil))
  ]
def newStudentRowEx : Row [("name", String), ("age", Nat), ("favorite color",
    String)] :=
  Row.cons (Cell.val "Dave") (Row.cons (Cell.val 14) (Row.cons (Cell.val "red")
    Row.nil))

/- Definitions using syntax extensions: -/
def studentsMissing :
    Table [("name", String), ("age", Nat), ("favorite color", String)] :=
  Table.mk [
    /["Bob"  , EMP, "blue" ],
    /["Alice", 17 , "green"],
    /["Eve"  , 13 , EMP    ]
  ]
-- Column-name annotations allow Lean to automatically infer the row's schema
def newStudentRow := /["name" := "Dave", "age" := 14, "favorite color" := "red"]
```

Figure 7: A comparison of the explicit notation of a table and row and the corresponding definitions using custom row syntax. Empty cells are denoted by `EMP`.

Furthermore, by performing a compile-time look-up in a table's schema, the type of a column can also be inferred from its name. Accordingly, table functions that in fact require a full header as an argument only require that the caller supply the name, allowing the type argument to be inferred during proof search. In many instances, this allows our library functions to be no more syntactically complex than those in the TypeScript implementation despite Lean's stronger typing. An example from the B2T2 specification of sorting a table using `tsort`, for instance, is written as `tsort students "age" true` in Lean and `tsort(students, "age", true)` in TypeScript—our library automatically generates the schema-membership proof and associated column type for `"age"` and synthesizes the relevant ordering type-class instance.

6.2 Notation for (Action) Lists

Functions that take lists or action lists of arguments require more extensive metaprogrammatic features and syntactic extensions. In addition to auto-generating proofs, we must also provide a means of inferring or synthesizing arguments—such as inferrable type arguments or type-class instances—for which we would rely on Lean's built-in inference and synthesis facilities in the single-argument case. (Lean does not, for instance, provide a mechanism to automatically resolve a list of type-class instances.)

Consider again the function `flatten`. The collection of columns to be flattened is provided as an action list containing dependent pairs of a name `c`, a type `τ`, and a proof that the header `(c, List τ)` appears in the table's schema. Thus, the operation depicted in figure 6 might be written in the following way:

```
flatten courses (ActionList.cons ⟨"Exams Taken", String, by header⟩
                  (ActionList.cons ⟨"Scores", Nat, by header⟩
                    ActionList.nil))
```

As this example shows, action lists add significant complexity to an operation that is straightforwardly expressible in TypeScript: `flatten(courses, ["Exams Taken", "Scores"])`. Furthermore, it can be difficult for a user even to determine the expected type of each entry in the action list, since the type hint provided by Lean simply indicates that an action list for `Schema.flattenList` is expected; the user must then look up the definition of `Schema.flattenList` to determine what the appropriate action argument to that function is. However, as in the single-argument case, all the elements of each dependent pair can be inferred from just the column name. Exploiting this fact, and adding a list-like syntax for action lists, we can simplify the above to the following:

```
flatten courses A["Exams Taken", "Scores"]
```

The `A` preceding the sequence of column names invokes a syntax extension provided by our library that automatically constructs pairs and dependent pairs, synthesizes type-class instances, performs schema-proof search, and generates proofs of decidable propositions (which is of use for functions that require, for example, proofs that arguments in an action list are nonnegative). The full algorithm can be found in the file `Notation.lean` in the project repository.

This algorithm is context-sensitive. The function `pivotWider`, for instance, takes as one of its arguments an action list containing column names together with proofs that, for some fixed type `τ` and each name `nm`, the header `(nm, τ)` appears in the input table's schema (since, as can be seen in figure 4, the types of all pivoted columns must be the same in order for the resulting "value" column to be homogeneous). Accordingly, the action-list expression appearing in

```
pivotLonger courses A["Exams Taken", "Scores"] "Key" "Value"
```

evaluates to the same expression as does

```
ActionList.cons ⟨"Exams Taken", by header⟩
  (ActionList.cons ⟨"Scores", by header⟩ ActionList.nil)
```

rather than the string–type–proof tuples generated by the same syntax in the `flatten` example.

We also extend this notation to simply typed lists: the argument `[⟨("age", Nat), by header, inferInstance⟩]` to the `find` function, for instance, becomes `A["age"]`. This provides the added benefit that casual users of the library need not be concerned with the technical distinction between simply typed lists and action lists, since the same notation can be used for both.

One limitation of our proof search (for both action lists and automatic parameters) is that schemata with repeated column names may cause unexpected behavior. This is due to the fact that the proof-search tactics simply select the first column matching the specified name (and satisfying any present restrictions on the corresponding type, such as having certain type-class instances). One step toward resolving this issue would be to enforce that all

column names in a table's schema be unique. It bears noting, however, that certain functions can break schema uniqueness even if the original schema had unique column names: for instance, a test from the B2T2 suite attempts to swap the names of columns `("midterm", Nat)` and `("final", Nat)` (appearing in that relative order) in a table `gradebook` as follows:

```
renameColumns gradebook A[("midterm", "final"), ("final", "midterm")]
```

However, this test returns `gradebook` unaltered. The column initially named `"midterm"` is renamed `"final"` by the first renaming, making it the first column with the name `"final"`—and thus the column renamed `"midterm"`—when the second renaming is performed. In order to perform the desired swap, the arguments must be passed in the opposite order:

```
renameColumns gradebook A[("final", "midterm"), ("midterm", "final")]
```

This case is emblematic of a more foundational class of considerations regarding uniqueness of names in schemata, which we discuss in section 7.2.

6.3 Ergonomics in B2T2 Examples

We assessed the ergonomics of our library and its notational conveniences in part through the implementation of B2T2's unit tests, example programs, and sample "buggy programs." For straightforward programs like those contained in the B2T2 unit tests—which each involve a limited number of operations performed on fixed tables—proof arguments need rarely be explicitly provided, and we can make extensive use of our custom row and action-list notation. Most of the added complexity in these tests compared to their TypeScript implementation is due to more careful type distinctions, such as explicitly accounting for empty cells in various computations.

B2T2's example programs, which require more complex computations and type-level constraints, required greater explicit interaction with the underlying type-level machinery, though we were still able to take advantage of notational and metaprogrammatic conveniences in some cases. Two of these programs—one that performs p-hacking on Boolean data, and one that extracts quiz scores from a grade book—require enforcing certain homogeneity constraints on a table's schema. The p-hacking operations require that every column of a table contain Boolean data. We enforce this using a data-valued predicate `Schema.Homogeneous`, using which we can cast a certified header `(nm, τ)` for a homogeneously Boolean schema to a certified header `(nm, Bool)`. Applying this casting is straightforward, and homogeneity proofs are invisibly generated by naïve proof search as automatic parameters to ease notation. However, the grade book operations require a more sophisticated homogeneity constraint: all columns whose names begin with "quiz" must contain numerical data. While we believe that it should be possible to employ an approach similar to the one we use for the p-hacking examples—and, indeed, [23] realizes such an approach—we were unable to do so due to an apparent Lean bug that prevented us from defining a `Type`-valued predicate enforcing the prefix condition. We instead resorted to a manual approach using casting along (propositional) equality proofs, which required proving a nontrivial proposition by hand and did not lend itself to the generality or automation of our scheme in the p-hacking case.

An additional benefit of having metaprograms mediate interaction with our library is that we can display custom compile-time error messages when an invalid program is entered, allowing us to display more relevant information than would ordinarily be generated by Lean. By default, the naïve proof search procedure used by our `name` and `header` tactics

would display the following error after failing to prove that the name `"Name"` appears in the schema `[("ID", String)]`:

```
unsolved goals
case a
⊢ Schema.HasName "Name" []
```

While the error suggests a failure related to generating a `HasName` proof for `"Name"`, the empty-list argument in the goal (resulting from the final stage of our proof search's traversal of the original schema) obscures the reason for the proof search's failure. This may be especially confusing because these proofs are often generated as automatic parameters, meaning that a user might not even be aware of the existence of the function argument responsible for the error. Even if one recognizes the meaning of this error message, one must still identify the schema for which a `HasName` proof is required before once can debug the error, since the original schema does not appear in the message. In our `name` and `header` tactics, we therefore record the initial expected proof type to display a more informative error message if our proof search fails, allowing the user to immediately identify both the nature of the failure and the schema in which the unsuccessful look-up was performed:

```
Could not prove that name "Name" is in schema [("ID", String)]
```

Using context at the metaprogram level to display informative error messages like these is especially important in a library where the underlying cause of an error can be many layers removed from the code responsible for generating the corresponding error message. One notable example in the B2T2 buggy-code tests involves an out-of-range row index for a table, which produces the inscrutable error message "failed to reduce to 'true': false." The message is specified by a proof tactic invoked by an automatic parameter that evaluates a Boolean-valued decision procedure, but two layers of context—the fact that the index must be in bounds for the provided table as well as the corresponding numerical inequality that the decision procedure is attempting to prove—are lost by the time the error message is displayed.

7 Verification

7.1 Trends

In addition to implementing the functions specified by B2T2, we have also, with a few exceptions, formalized their specifications in Lean. The B2T2 specification focuses heavily on "structural," type-level properties of tables; accordingly, most of the specifications concern properties of schemata rather than the data contained in tables. Many proofs—including many of the more complex ones—involve verifying that a certain name appears in the schema resulting from a function application or that a certain column contains data of a desired type. For instance, one specification for `groupBySubtractive` ensures that each subtable's schema is a sublist of the original table's schema:

```
theorem groupBySubtractive_spec6 :
  ∀ (t : Table sch) (c : η) (hc : sch.HasCol (c, τ)),
  ∀ t', t' ∈ (getColumn2 (groupBySubtractive t c hc) "groups" (.tl .hd)).somes →
  List.Sublist (schema t') sch
```

Some of the B2T2 specifications do place structural constraints on the data in table bodies; however, they generally under-specify the behavior of functions. Common examples of

such specifications are those requiring that tables maintain the same number of rows before and after certain operations. Since these specifications generally apply to functions implemented by mapping some computation over every row of the table, their proofs are generally direct applications of the length-preservation property of `List.map`. A more interesting example of a data-level specification is the fourth specification of `groupBySubtractive`, which requires that its "key" column contain no duplicates:

```
theorem groupBySubtractive_spec4 [inst : DecidableEq τ] :
  ∀ (t : Table sch) (c : η) (hc : sch.HasCol (c, τ)),
  List.NoDuplicates (getColumn2 (groupBySubtractive t c hc) "key")
```

Because of the considerable interplay between proofs and data in our implementation, our verification at times challenged or was challenged by Lean's proof-writing infrastructure and the distinctive properties of Lean's propositional universe. A number of our proofs are actually data—values of type `Schema.HasCol` or `Schema.HasName`—which prevented us from using Lean's built-in propositional types due to restrictions on large elimination or other type-universe constraints. Moreover, many of our proofs proceed by induction on our data-level proof types, which is not currently supported by Lean's `induction` tactic due to the complexity of the types' indices. Many such proofs were instead written using pattern-matching, though here too we encountered issues with the `cases` tactic employed by Lean's pattern-matcher. As a result of such issues, we occasionally had to encode in auxiliary functions or lemmas reasoning by cases that might have been more concisely captured by pattern matching. (See, for example, the function `Schema.hasNameOfNthsHasName`, used in the third specification for `selectColumns3`.)

7.2 Divergence from B2T2

One key way in which our implementation diverges from the B2T2 specification—which has important implications for our verification—is that our table representation does not enforce the requirement that all names in a schema be unique. Instead, we define a predicate `Schema.Unique` that can be used to optionally enforce column-name uniqueness of schemata. We made this decision for three reasons. First, it is common in Lean to decouple data from proofs in cases where it is possible to do so, rather than embedding the proofs in the data representation itself [7]. Second, most B2T2 specifications can be stated or closely approximated without requiring uniqueness. For instance, some specifications of the form "the column `c` has type `τ`" are formalized as `Schema.HasCol (c, τ) schema`; without uniqueness, this more accurately formalizes "*a* column `c` has type `τ`," but adding a uniqueness constraint recovers the intended specification. Since specifications that do not require schematic uniqueness are often amenable to "translation" in this manner by simply adding an additional uniqueness requirement, proving the more general case was usually preferable. Finally, adding uniqueness requirements may add considerable complexity to some API functions, further motivating the separation of core data operations from uniqueness proofs. As discussed in section 6.2, the conditions under which functions like `renameColumns` preserve schematic uniqueness can be subtle and difficult to state or encode. A full account of schematic uniqueness would have to provide uniqueness-preservation proofs for each of the functions in the API. We do not provide such proofs in our implementation, but a precise account of uniqueness preservation is an important next step to accurately realize not only the B2T2 specification but also the desiderata for representing real-world tabular data.

In other instances, we diverge from B2T2's specifications because they simply fail to hold

of our implementation. This is the case for several of the specifications for `renameColumns` and `flatten`. One of the former, for instance, requires that if some entry in the action list renames a column `c` to `c'`, then the type contained in the column named `c'` in the output must be equal to that contained in column `c` in the input. However, because action list-based functions act iteratively, it is possible that the column `c'` is again renamed by a subsequent entry in the action list, so that `c'` does not even appear in the final schema. (A similar issue arises for `flatten`, whose specification in B2T2 stipulates that the type stored in a "flattened" column should be the element type of the list type it stored in the input; a column that is flattened twice does not obey this specification.) These deviations arise because B2T2 seems to envision functions like `renameColumns` and `flatten` as executing on their arguments effectively in parallel: all arguments refer to the schema of the original table, and operations on previously-modified columns are disallowed. By contrast, action lists—and functions which proceed by recursion thereupon—are fundamentally sequential in nature. It is, moreover, significantly more challenging even to state specifications analogous to B2T2's for our action list-based implementations of `renameColumns` and `flatten`. For the former, for instance, in order to define the name `c'` to which a name `c` in the initial schema maps, we must account for all intermediate names to which `c` is mapped in the course of traversing the action list.

While it seems theoretically feasible to implement `renameColumns` or `flatten` in the manner B2T2 specifies using our table representation, specifying the type of such a function would likely be significantly more complex. We would need additional proof arguments to prohibit invalid but well typed inputs (such as action lists that attempt to flatten the same column multiple times) and would likely require non-definitional type casting at the data level. Ultimately, we felt that the programmatic benefits of an action list-based approach outweighed the losses in our realization of the B2T2 specification.

8 Conclusion

8.1 Summary

In this thesis, we have presented a dependently typed library for tabular programming in Lean based on the Brown Benchmark for Table Types. It provides strong static guarantees and contains formal proofs of the subset of the B2T2 specifications which it satisfies. Although we deviate from the B2T2 specification in a few instances, we have successfully implemented every function (or a close approximation thereof) contained in the specification. For complex iterated table operations, we introduce the *action list* data structure to facilitate relatively straightforward typing of these functions and to avoid the need for non-definitional type-casting. While type-safety mechanisms like action lists contribute significant ostensible complexity to our API, we employ Lean's syntax-extension and metaprogramming features to simplify interaction with the library and occlude from the user much of the complexity required to realize type-correctness.

Most of the challenges our implementation encountered were due to the significant type-level computation required by many table functions. Such computations created difficulties for type-class resolution and revealed cases in which full reducibility of many data-level functions is not only desirable but necessary. Additionally, the need to compute with proofs as first-class data objects (at both the type and data levels) prevented us from taking full advantage of the sophisticated tooling that exists for proof-writing in Lean's propositional

universe. Lastly, we found that some assumptions made by the B2T2 specification—which was inspired largely by dynamically typed libraries [12]—did not easily lend themselves to type-safe realizations, especially those concerning the behavior of multi-step table operations that the specification conceives of as occurring in parallel. Even our modified versions of these operations—those involving action lists—required significant type-level machinery and highlighted the importance of metaprogrammatic notational simplification to avoid incurring significant ergonomic costs compared to their counterparts in dynamically typed languages.

8.2 Related Work

Our table representation was motivated by existing approaches to encoding common data structures with dependent types. In particular, we were influenced by Weirich's encoding of dictionaries in [21], which, although implemented using Haskell GADTs rather than true dependent types, uses a similar type-level mechanism for storing name-indexed heterogeneous data.

Another dependently typed tabular-data library, Idris2-Table [22], was developed while work on this thesis was ongoing. Idris2-Table uses a table representation very similar to ours, though it encodes schemata and table bodies using Idris' built-in `SnocList` type, which is optimized for tail- rather than head-concatenation. This optimization likely improves performance for most table operations: functions that perform type-level schematic concatenation typically append a small number of columns on the right to an arbitrarily large schema on the left, and row insertions generally occur at the bottom rather than the top of the table. The Idris implementation also elects—as we do—not to enforce uniqueness of column names in schemata [23].

Another approach to table typing, developed in [17], uses record types to encode a row-wise representation of tables. This is similar to the approach used by the TypeScript implementation of B2T2: tables are represented as sequences of record-valued rows, which store column-named entries in an unordered key-value mapping. This approach naturally extends Hindley–Milner-style type systems that support record types and is amenable to use in non-dependently typed settings. Several Haskell libraries, such as Frames [9], use record types in conjunction with Template Haskell metaprograms to represent tabular data. Like the TypeScript implementation, the approach in [17] has the advantage of allowing column name validity to be checked by the type system directly in the form of first-class record labels, dispensing with the complexity of requiring proof values for every column access. However, the record-based approach did not prove amenable to representing more complex functions in the B2T2 specification, such as the two pivot functions, and is unable to fix a column ordering in its table representation [17]. Ideally, a tabular programming framework would combine the ease of use of record-based approaches with the robustness of a dependently-typed encoding.

8.3 Future Work

One aspect of tabular programming that we have largely ignored in our implementation is performance. Optimization of the data structures used in our encoding may be one avenue for improvement. As we previously noted, Idris2-Table uses `SnocLists` instead of `Lists` for representing schemata and table bodies, which are better optimized for the most common forms of schematic and row concatenation that occur in the B2T2 API. Alternatively, as mentioned in section 3.2, search trees or hash maps could be used instead of lists for

storing schemata and table bodies. These would likely require greater type-level overhead (especially given that such a scheme would likely require separate storage of column order) and may entail a loss of the definitional type-level reduction our list-based approach enjoys. However, a sufficiently robust scheme might be able to enjoy the attendant performance benefits while relying on either careful type definitions or type-casting infrastructure to prevent issues at the type level.

We also see room for further improvement of the ergonomics of dependently typed tabular programming tools like ours and for minimizing the complexity added by proofs and other type-level machinery. Even the most elegant dependently typed table encodings inevitably incur a nontrivial amount of such typing-induced overhead. As we demonstrated in section 6, notational and metaprogrammatic resources can serve as important tools to limit the extent to which users are exposed to this overhead. Another important ergonomic consideration, discussed in section 6.3, is the quality of error messages presented to a user when interacting with the library. In our testing using B2T2's sample buggy programs, the error messages provided by Lean when presented with malformed tables or invalid table-function applications ranged from reasonably illuminating to nearly indecipherable. As we noted, the use of metaprograms to scaffold interaction with our library provides an opportunity to supersede Lean's built-in error messages with more informative ones of our own, which we demonstrated in the cases of the `name` and `header` tactics. While we have explored such functionality to a limited extent in this thesis, this may be a fruitful path for providing more meaningful compile-time error messages to users, especially when syntactic conveniences may occlude the actual values (like proof terms) responsible for static errors.

References

[1] Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich. *The Lean Reference Manual*. Release 3.3.0. 2018. URL: https://leanprover.github.io/reference/lean_reference.pdf.

[2] Jeremy Avigad et al. *Theorem Proving in Lean 4*. 2024. URL: https://leanprover.github.io/theorem_proving_in_lean4/.

[3] Anne Baanen et al. *The Hitchhiker's Guide to Logical Verification*. 2022. URL: https://github.com/blanchette/logical_verification_2022/blob/9ac0bf7/hitchhikers_guide.pdf.

[4] Kevin Buzzard. *Formalising Mathematics*. 2024. URL: https://www.ma.imperial.ac.uk/~buzzard/xena/formalising-mathematics-2024.

[5] Mario Carneiro. "The Type Theory of Lean". Master's thesis. Pittsburgh, PA: Carnegie Mellon University, Apr. 2019. URL: https://github.com/digama0/lean-type-theory/releases/tag/v1.0.

[6] JuliaData Collaborators. *DataFrames.jl*. URL: https://github.com/JuliaData/DataFrames.jl.

[7] The mathlib Community. "The lean mathematical library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. ISBN: 9781450370974. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.

[8] The mathlib community. *leanprover/mathilb4*. 2024. URL: https://github.com/leanprover-community/mathlib4/tree/22092a2aaa2dc2a.

[9] Anthony Cowley. *Frames Tutorial*. URL: https://acowley.github.io/Frames/.

[10] Std4 Developers. *leanprover/std4*. 2024. URL: https://github.com/leanprover/std4/tree/59e9660.

[11] The R Foundation. *R: The R Project for Statistical Computing*. URL: https://www.r-project.org/.

[12] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. "Types for Tables: A Language Design Benchmark". In: *The Art, Science, and Engineering of Programming* 6.2 (2022), 8:1–8:30.

[13] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. *brownplt/B2T2*. 2023. URL: https://github.com/brownplt/B2T2/tree/a7ee6ee.

[14] Erik Meijer. "The world according to LINQ". In: *Commun. ACM* 54.10 (Oct. 2011), pp. 45–51. ISSN: 0001-0782. DOI: 10.1145/2001269.2001285. URL: https://doi.org/10.1145/2001269.2001285.

[15] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. "An Extensible User Interface for Lean 4". In: *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 24:1–24:20. ISBN: 978-3-95977-284-6. DOI: 10.4230/LIPIcs.ITP.2023.24. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.24.

[16] Oracle. *MySQL 8.0 Reference Manual*. 2024. URL: https://dev.mysql.com/doc/refman/8.0/en/.

[17] Adam Paszke and Ningning Xie. "Infix-Extensible Record Types for Tabular Data". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2023. , Seattle, WA, USA, Association for Computing Machinery, 2023, pp. 29–43. DOI: 10.1145/3609027.3609406. URL: https://doi.org/10.1145/3609027.3609406.

[18] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. *Tabled Typeclass Resolution*. 2020. arXiv: 2001.04301 [cs.PL].

[19] Empirical Software Solutions. *A language for time-series analysis*. URL: https://www.empirical-soft.com/.

[20] The pandas development team. *pandas-dev/pandas: Pandas*. URL: https://doi.org/10.5281/zenodo.3509134.

[21] Stephanie Weirich. "The Influence of Dependent Types (Keynote)". In: *SIGPLAN Not.* 52.1 (Jan. 2017), p. 1. ISSN: 0362-1340. DOI: 10.1145/3093333.3009923. URL: https://doi.org/10.1145/3093333.3009923.

[22] Robert Wright, Michel Steuwer, and Ohad Kammar. "Idris2-Table: evaluating dependently-typed tables with the Brown Benchmark for Table Types (Extended Abstract)". In: TyDe 2022. Association for Computing Machinery, 2022. URL: https://tydeworkshop.org/2022-abstracts/paper6.pdf.

[23] Robert Wright, Michel Steuwer, and Ohad Kammar. *madman-bob/idris2-table*. 2022. URL: https://github.com/madman-bob/idris2-table/tree/d8fe047.